# SPARC® JPS2: Common Specification

*Sun Microsystems, Inc. and Fujitsu Limited*
*JRC Contributed Material*

*Release 1.0, 18 Sep 2003*

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A. 1-800-555-9SUN

Fujitsu Limited
4-1-1 Kamikodanaka
Nakahara-ku, Kawasaki, 211-8588
Japan

Release 1.0, 18 Sep 2003

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California, 95054
U.S.A.
http://www.sun.com

Fujitsu Limited
4-1-1 Kamikodanaka
Nakahara-ku, Kawasaki, 211-8588
Japan
http://www.fujitsu.com/

Please
Recycle

Adobe PostScript

# Contents

**Preface** . . . . . **xv**

**1. Overview** . . . . . **1**

1.1. Navigating the *SPARC Joint Programming Specification 2* . . . . . 2

1.2. Fonts and Notational Conventions . . . . . 3

1.2.1 Implementation Dependencies . . . . . 4

1.2.2 Notation for Numbers . . . . . 4

1.2.3 Informational Notes . . . . . 4

1.3. SPARC V9 Architecture . . . . . 5

1.3.1 Features . . . . . 5

1.3.2 Attributes . . . . . 6

1.3.3 System Components . . . . . 6

1.3.4 Architectural Definition . . . . . 7

1.3.5 SPARC V9 Compliance . . . . . 8

**2. Definitions** . . . . . **9**

**3. Architectural Overview** . . . . . **21**

3.1. SPARC V9 Processor Architecture . . . . . 21

3.1.1 Integer Unit (IU) . . . . . 22

3.1.2 Floating-Point Unit (FPU) . . . . . 22

3.2. Instructions . . . . . 22

3.2.1 Memory Access . . . . . 23

3.2.2 Arithmetic / Logical / Shift Instructions . . . . . 25

3.2.3 Control Transfer . . . . . 25

3.2.4 State Register Access . . . . . 26

3.2.5 Floating-Point Operate . . . . . 27

3.2.6 Conditional Move . . . . . 27

3.2.7 Register Window Management . . . . . 27

3.3. Traps . . . . . 27
3.4. Multithreaded Processor (MTP) . . . . . 28

**4. Data Formats . . . . . 29**

4.1. Integer Data Formats . . . . . 30
4.1.1 Signed Integer Data Types . . . . . 31
4.1.2 Unsigned Integer Data Types . . . . . 32
4.1.3 Tagged Word . . . . . 33
4.2. Floating-Point Data Formats . . . . . 33
4.2.1 Floating Point, Single Precision . . . . . 34
4.2.2 Floating Point, Double Precision . . . . . 34
4.2.3 Floating Point, Quad Precision . . . . . 35
4.2.4 Floating-Point Data Alignment in Memory and Registers . . . . . 36
4.3. Graphics Data Formats . . . . . 37
4.3.1 Pixel Data Format . . . . . 37
4.3.2 Fixed-Point Data Formats . . . . . 37

**5. Registers . . . . . 39**

5.1. Registers Accessible in Nonprivileged Mode . . . . . 40
5.1.1 General-Purpose r Registers . . . . . 41
5.1.2 Floating-Point `f` Registers . . . . . 46
5.1.3 Floating-Point State Register (`FSR`) . . . . . 53
5.1.4 State Registers . . . . . 62
5.2. Registers Accessible in Privileged Mode Only . . . . . 71
5.2.1 Non-Register-Window PR State Registers . . . . . 71
5.2.2 Register Window PR State Registers . . . . . 83
5.2.3 State Registers . . . . . 86
5.2.4 ASI-Accessible Registers . . . . . 91

**6. Instructions . . . . . 99**

6.1. Instruction Execution . . . . . 99
6.2. Instruction Formats and Fields . . . . . 100
6.3. Instruction Categories . . . . . 104
6.3.1 Memory Access Instructions . . . . . 105
6.3.2 Integer Arithmetic and Logical Instructions . . . . . 111
6.3.3 Control-Transfer Instructions (CTIs) . . . . . 112
6.3.4 Register Window Management Instructions . . . . . 118
6.3.5 State Register Access . . . . . 121
6.3.6 Privileged Register Access . . . . . 122
6.3.7 Floating-Point Operate (FPop) Instructions . . . . . 122

6.3.8 Implementation-Dependent Instructions . . . . . 123
6.3.9 Reserved Opcodes and Instruction Fields. . . . . 123
6.3.10 Summary of Unimplemented Instructions . . . . . 124
6.4. Register Window Management . . . . . 125
6.4.1 Register Window State Definition . . . . . 125
6.4.2 Register Window Traps . . . . . 126
**7. Traps . . . . . 129**
7.1. Virtual Processor States, Normal and RED_state Traps . . . . . 130
7.1.1 RED_state . . . . . 131
7.1.2 error_state . . . . . 134
7.2. Trap Categories . . . . . 134
7.2.1 Precise Traps . . . . . 134
7.2.2 Deferred Traps . . . . . 135
7.2.3 Disrupting Traps . . . . . 135
7.2.4 Reset Traps . . . . . 136
7.2.5 Uses of the Trap Categories . . . . . 137
7.3. Trap Control . . . . . 138
7.3.1 PIL Control. . . . . 138
7.3.2 FSR.tem Control . . . . . 139
7.4. Trap-Table Entry Addresses . . . . . 139
7.4.1 Trap Table Organization . . . . . 140
7.4.2 Trap Type (TT) . . . . . 140
7.4.3 Trap Priorities . . . . . 145
7.5. Trap Processing . . . . . 146
7.5.1 Normal Trap Processing . . . . . 148
7.5.2 RED_state Trap Processing . . . . . 153
7.6. Exception and Interrupt Descriptions . . . . . 160
7.6.1 Traps Defined by SPARC V9 As Mandatory. . . . . 161
7.6.2 SPARC V9 Optional Traps That Are Mandatory in SPARC JPS2 164
7.6.3 SPARC V9 Optional Traps That Are Optional in SPARC JPS2 165
7.6.4 SPARC V9 Optional Traps Not Used in SPARC JPS2 . . . . . 165
7.6.5 SPARC V9 Implementation-Dependent, Optional Traps That Are Mandatory in SPARC JPS2 166
7.6.6 SPARC JPS2 Implementation-Dependent Traps. . . . . 167
**8. Memory Models . . . . . 169**
8.1. Overview . . . . . 170
8.2. Memory, Real Memory, and I/O Locations . . . . . 171
8.3. Addressing and Alternate Address Spaces . . . . . 173

8.4. SPARC V9 Memory Model . . . . . 175
8.4.1 SPARC V9 Program Execution Model . . . . . 175
8.4.2 Virtual Processor / Memory Interface Model . . . . . 177
8.4.3 MEMBAR Instruction . . . . . 178
8.4.4 Memory Models . . . . . 180
8.4.5 Mode Control . . . . . 181
8.4.6 Hardware Primitives for Mutual Exclusion . . . . . 182
8.4.7 Synchronizing Instruction and Data Memory . . . . . 183

**9. Multithreaded Processing (MTP) . . . . . 185**

9.1. Overview of MTP . . . . . 185
9.1.1 MTP Definition . . . . . 186
9.1.2 General MTP Behavior . . . . . 188
9.2. Accessing MTP Registers . . . . . 189
9.2.1 Types of MTP Registers . . . . . 189
9.2.2 Accessing MTP Registers Through ASI Interface . . . . . 189
9.3. MTP Identification . . . . . 191
9.3.1 Register Specification . . . . . 191
9.3.2 Core ID Register (ASI_CORE_ID) . . . . . 191
9.3.3 Core Interrupt ID Register (ASI_INTR_ID) . . . . . 192
9.3.4 Reserved ASI Range . . . . . 196
9.4. Disabling and Parking Virtual Processors . . . . . 196
9.4.1 Core Available Register (ASI_CORE_AVAILABLE) . . . . . 197
9.4.2 Enabling and Disabling Virtual Processors . . . . . 197
9.4.3 Parking and Unparking Virtual Processors . . . . . 199
9.5. Reset and Trap Handling . . . . . 204
9.5.1 Per-Core Resets (SIR and WDR Resets) . . . . . 204
9.5.2 Full-MTP Resets (System Reset) . . . . . 204
9.5.3 Partial MTP Resets (XIR Reset) . . . . . 205
9.5.4 Traps . . . . . 207
9.6. Error Handling in MTPs . . . . . 207
9.6.1 Core-Specific Error Reporting . . . . . 208
9.6.2 Non-Core-Specific Error Reporting . . . . . 208
9.7. Additional MTP Software Interface . . . . . 211
9.7.1 Diagnostic / RAS Registers . . . . . 211
9.7.2 Booting Support . . . . . 211
9.8. Recommended Subset of MTP Interface for Non-MTP Parts . . . . . 212
9.9. Machine State Summary . . . . . 213
9.10. MTP State Transition . . . . . 216

**A. Instruction Definitions . . . . . 217**

A.1. Add . . . . . 224
A.2. Alignment Instructions (VIS I) . . . . . 226
A.3. Three-Dimensional Array Addressing Instructions (VIS I) . . . . . 228
A.4. Block Load and Store (VIS I) . . . . . 231
A.5. Byte Mask and Shuffle Instructions (VIS II) . . . . . 235
A.6. Branch on Integer Register with Prediction (BPr) . . . . . 237
A.7. Branch on Floating-Point Condition Codes with Prediction (FBPfcc) . 239
A.8. Branch on Integer Condition Codes with Prediction (BPcc) . . . 242
A.9. Call and Link . . . . . 245
A.10. Compare and Swap . . . . . 246
A.11. DONE and RETRY . . . . . 249
A.12. Edge Handling Instructions (VIS I, II) . . . . . 250
A.13. Floating-Point Add and Subtract . . . . . 253
A.14. Floating-Point Compare . . . . . 255
A.15. Convert Floating-Point to Integer . . . . . 257
A.16. Convert Between Floating-Point Formats . . . . . 259
A.17. Convert Integer to Floating-Point . . . . . 261
A.18. Floating-Point Move . . . . . 263
A.19. Floating-Point Multiply and Divide . . . . . 265
A.20. Floating-Point Square Root . . . . . 267
A.21. Flush Instruction Memory . . . . . 268
A.22. Flush Register Windows . . . . . 270
A.23. Illegal Instruction Trap . . . . . 271
A.24. Implementation-Dependent Instructions . . . . . 272
A.25. Jump and Link . . . . . 273
A.26. Load Floating-Point . . . . . 274
A.27. Load Floating-Point from Alternate Space . . . . . 276
A.28. Load Integer . . . . . 279
A.29. Load Integer from Alternate Space . . . . . 281
A.30. Load Quadword, Atomic (VIS I) . . . . . 283
A.31. Load-Store Unsigned Byte . . . . . 285
A.32. Load-Store Unsigned Byte to Alternate Space . . . . . 286
A.33. Logical Operate Instructions (VIS I) . . . . . 288
A.34. Logical Operations . . . . . 291
A.35. Memory Barrier . . . . . 293
A.36. Move Floating-Point Register on Condition (FMOVcc) . . . . . 296
A.37. Move Floating-Point Register on Integer Register Condition (FMOVr) 302
A.38. Move Integer Register on Condition (MOVcc) . . . . . 304
A.39. Move Integer Register on Register Condition (MOVr) . . . . . 309
A.40. Multiply and Divide (64-bit) . . . . . 311

A.41. No Operation . . . 312
A.42. Partial Store (VIS I) . . . 313
A.43. Partitioned Add / Subtract Instructions (VIS I) . . . 315
A.44. Partitioned Multiply Instructions (VIS I) . . . 317
A.44.1. FMUL8x16 Instruction . . . 318
A.44.2. FMUL8x16AU Instruction . . . 319
A.44.3. FMUL8x16AL Instruction . . . 319
A.44.4. FMUL8SUx16 Instruction . . . 320
A.44.5. FMUL8ULx16 Instruction . . . 320
A.44.6. FMULD8SUx16 Instruction . . . 321
A.44.7. FMULD8ULx16 Instruction . . . 322
A.45. Pixel Compare (VIS I) . . . 323
A.46. Pixel Component Distance (PDIST) (VIS I) . . . 325
A.47. Pixel Formatting (VIS I) . . . 326
A.47.1. FPACK16 . . . 327
A.47.2. FPACK32 . . . 328
A.47.3. FPACKFIX . . . 329
A.47.4. FEXPAND . . . 330
A.47.5. FPMERGE . . . 331
A.48. Population Count . . . 332
A.49. Prefetch Data . . . 334
A.49.1. SPARC V9 Prefetch Variants . . . 336
A.49.2. SPARC JPS2 Prefetch Variants (fcn = 20–23) . . . 338
A.49.3. Implementation-Dependent Prefetch Variants (fcn = 16–19, 24–31) . . . 339
A.49.4. General Comments . . . 340
A.50. Read Privileged Register . . . 341
A.51. Read State Register . . . 343
A.52. RETURN . . . 346
A.53. SAVE and RESTORE . . . 348
A.54. SAVED and RESTORED . . . 351
A.55. Set Interval Arithmetic Mode (VIS II) . . . 352
A.56. SETHI . . . 353
A.57. Shift . . . 354
A.58. Short Floating-Point Load and Store (VIS I) . . . 356
A.59. SHUTDOWN (VIS I) . . . 358
A.60. Software-Initiated Reset . . . 359
A.61. Store Floating-Point . . . 360
A.62. Store Floating-Point into Alternate Space . . . 363
A.63. Store Integer . . . 366
A.64. Store Integer into Alternate Space . . . 368
A.65. Subtract . . . 370
A.66. Tagged Add . . . 372

A.67. Tagged Subtract . . . . . 373
A.68. Trap on Integer Condition Codes (Tcc) . . . . . 374
A.69. Write Privileged Register . . . . . 377
A.70. Write State Register . . . . . 380
A.71. Deprecated Instructions . . . . . 383
A.71.1. Branch on Floating-Point Condition Codes (FBfcc). . . . . 384
A.71.2. Branch on Integer Condition Codes (Bicc) . . . . . 387
A.71.3. Divide (64-bit / 32-bit) . . . . . 390
A.71.4. Load Floating-Point Status Register . . . . . 393
A.71.5. Load Integer Doubleword . . . . . 394
A.71.6. Load Integer Doubleword from Alternate Space. . . . . 396
A.71.7. Multiply (32-bit). . . . . 398
A.71.8. Multiply Step . . . . . 400
A.71.9. Read Y Register . . . . . 402
A.71.10.Store Barrier . . . . . 403
A.71.11.Store Floating-Point Status Register Lower . . . . . 404
A.71.12.Store Integer Doubleword . . . . . 406
A.71.13.Store Integer Doubleword into Alternate Space . . . . . 408
A.71.14.Swap Register with Memory . . . . . 410
A.71.15.Swap Register with Alternate Space Memory . . . . . 412
A.71.16.Tagged Add and Trap on Overflow. . . . . 414
A.71.17.Tagged Subtract and Trap on Overflow . . . . . 416
A.71.18.Write Y Register . . . . . 418

**B. IEEE Std 754-1985 Requirements for SPARC V9. . . . . 419**

B.1. Traps Inhibiting Results . . . . . 419
B.2. NaN Operand and Result Definitions. . . . . 420
B.2.1. Untrapped Result in Different Format from Operands. . . . . 420
B.2.2. Untrapped Result in Same Format as Operands . . . . . 421
B.3. Trapped Underflow Definition (ufm = 1) . . . . . 422
B.4. Untrapped Underflow Definition (ufm = 0). . . . . 422
B.5. Integer Overflow Definition. . . . . 423
B.6. Floating-Point Nonstandard Mode . . . . . 424

**C. Implementation Dependencies . . . . . 425**

C.1. Definition of an Implementation Dependency. . . . . 426
C.2. Hardware Characteristics . . . . . 426
C.3. Implementation Dependency Categories . . . . . 427
C.4. List of Implementation Dependencies . . . . . 427

**D. Formal Specification of the Memory Models . . . . . 443**

D.1. Virtual Processor and Memory. . . . . 443
D.2. Overview of the Memory Model Specification . . . . . 444
D.3. Memory Transactions . . . . . 445

D.3.1. Memory Transactions . . . 445
D.3.2. Program Order . . . 446
D.3.3. Dependence Order . . . 447
D.3.4. Memory Order . . . 448
D.4. Specification of Relaxed Memory Order (RMO) . . . 449
D.4.1. Value Atomicity . . . 449
D.4.2. Store Atomicity . . . 449
D.4.3. Atomic Memory Transactions . . . 449
D.4.4. Memory Order Constraints . . . 449
D.4.5. Value of Memory Transactions . . . 450
D.4.6. Termination of Memory Transactions . . . 450
D.4.7. Flush Memory Transaction . . . 450
D.5. Specification of Partial Store Order (PSO) . . . 451
D.6. Specification of Total Store Order (TSO) . . . 451
D.7. Examples of Program Executions . . . 451
D.7.1. Observation of Store Atomicity . . . 451
D.7.2. Dekker's Algorithm . . . 453
D.7.3. Indirection Through Virtual Processors . . . 454
D.7.4. PSO Behavior . . . 454
D.7.5. Application to Compilers . . . 454
D.7.6. Verifying Memory Models . . . 455

**E. Opcode Maps . . . 457**

**F. Memory Management Unit . . . 467**

F.1. Virtual Address Translation . . . 467
F.2. Translation Table Entry (TTE) . . . 470
F.3. Translation Storage Buffer . . . 473
F.3.1. TSB Indexing Support . . . 473
F.3.2. TSB Cacheability . . . 474
F.3.3. TSB Organization . . . 474
F.4. Hardware Support for TSB Access . . . 475
F.4.1. Typical TLB Miss / Refill Sequence . . . 475
F.4.2. TSB Pointer Formation . . . 475
F.4.3. Required TLB Conditions . . . 478
F.4.4. Required TSB Conditions . . . 478
F.4.5. MMU Global Registers Selection . . . 478
F.5. Faults and Traps . . . 479
F.6. MMU Operation Summary . . . 481
F.7. ASI Value, Context, and Endianness Selection for Translation . . . 483
F.8. Reset, Disable, and RED_state Behavior . . . 485
F.9. SPARC V9 "MMU Requirements" Annex . . . 487
F.10. Internal Registers and ASI Operations . . . 487
F.10.1. Accessing MMU Registers . . . 488

F.10.2. Context Registers . . . 489
F.10.3. Instruction / Data MMU TLB Tag Access Registers . . . 490
F.10.4. I / D TLB Data In, Data Access, and Tag Read Registers . . . 491
F.10.5. I / D TSB Tag Target Registers . . . 494
F.10.6. I / D TSB Base Registers . . . 494
F.10.7. I / D TSB Extension Registers . . . 496
F.10.8. I / D TSB 8-Kbyte and 64-Kbyte Pointer and Direct Pointer Registers . . . 496
F.10.9. I / D Synchronous Fault Status Registers (I-SFSR, D-SFSR) . . . 497
F.10.10. Synchronous Fault Addresses . . . 499
F.10.11. I / D MMU Demap . . . 500
F.11. MMU Bypass . . . 503
F.12. Translation Lookaside Buffer Hardware . . . 503
F.12.1. TLB Operations . . . 503
F.12.2. TLB Replacement Policy . . . 504
F.12.3. TSB Pointer Logic Hardware Description . . . 504

**G. Assembly Language Syntax . . . 505**

G.1. Notation Used . . . 505
G.1.1. Register Names . . . 506
G.1.2. Special Symbol Names . . . 507
G.1.3. Values . . . 511
G.1.4. Labels . . . 512
G.1.5. Other Operand Syntax . . . 512
G.1.6. Comments . . . 513
G.2. Syntax Design . . . 514
G.3. Synthetic Instructions . . . 514

**H. Software Considerations . . . 517**

H.1. Nonprivileged Software . . . 517
H.1.1. Registers . . . 517
H.1.2. Leaf-Procedure Optimization . . . 521
H.1.3. Example Code for a Procedure Call . . . 523
H.1.4. Register Allocation Within a Window . . . 523
H.1.5. Other Register-Window-Usage Models . . . 524
H.1.6. Self-Modifying Code . . . 525
H.1.7. Thread Management . . . 525
H.1.8. Minimizing Branch Latency . . . 526
H.1.9. Prefetch . . . 527
H.1.10. Nonfaulting Load . . . 530
H.2. Supervisor Software . . . 533
H.2.1. Trap Handling . . . 534
H.2.2. Example Code for Spill Handler . . . 534
H.2.3. Client-Server Model . . . 535

H.2.4. User Trap Handlers . . . 537
H.2.5. Scratchpad Register Usage . . . 538

**I. Extending the SPARC V9 Architecture. . . . 539**

I.1. Read / Write Ancillary State Registers (ASRs) . . . 539
I.2. Implementation-Dependent and Reserved Opcodes . . . 540

**J. Programming with the Memory Models . . . 541**

J.1. Memory Operations . . . 542
J.2. Memory Model Selection . . . 542
J.3. Virtual Processor and Processes . . . 543
J.4. Higher-Level Programming Languages and Memory Models . . . 543
J.5. Portability and Recommended Programming Style . . . 544
J.6. Spin Locks . . . 546
J.7. Producer-Consumer Relationship . . . 547
J.8. Process Switch Sequence . . . 549
J.9. Dekker's Algorithm . . . 550
J.10. Code Patching . . . 550
J.11. Fetch_and_Add . . . 552
J.12. Barrier Synchronization . . . 554
J.13. Linked List Insertion and Deletion . . . 555
J.14. Communicating with I / O Devices . . . 556
J.14.1. I / O Registers with Side Effects . . . 558
J.14.2. Control and Status Registers . . . 558
J.14.3. The Descriptor . . . 559
J.14.4. Lock-Controlled Access to a Device Register . . . 559

**K. Changes from SPARC V8 to SPARC V9 . . . 561**

K.1. Trap Model . . . 561
K.2. Data Formats . . . 562
K.3. Little-Endian Support . . . 562
K.4. Little-Endian Byte Order . . . 562
K.5. Registers . . . 562
K.6. Alternate Space Access . . . 564
K.7. Instruction Set . . . 564
K.8. Memory Model . . . 566

**L. Address Space Identifiers (ASIs) . . . 567**

L.1. Address Space Identifiers and Address Spaces . . . 567
L.2. ASI Values . . . 568
L.3. ASI Assignments . . . 568
L.3.1. Supported ASIs . . . 568
L.3.2. Special Memory Access ASIs . . . 580

**M. Caches and Cache Coherency . . . . . 585**

**N. Interrupt Handling . . . . . 587**

N.1. Core Interrupt ID Register (ASI_INTR_ID) . . . . . 588
N.2. Interrupt Vector Dispatch . . . . . 588
N.3. Interrupt Vector Receive . . . . . 589
N.4. Interrupt Global Registers . . . . . 590
N.5. Interrupt ASI Registers . . . . . 590
N.5.1. Outgoing Interrupt Vector Data<7:0> Register . . . . . 590
N.5.2. Interrupt Vector Dispatch Register . . . . . 591
N.5.3. Interrupt Vector Dispatch Extension Register . . . . . 592
N.5.4. Interrupt Vector Dispatch Status Register . . . . . 592
N.5.5. Incoming Interrupt Vector Data<7:0> . . . . . 593
N.5.6. Interrupt Vector Receive Register . . . . . 593
N.6. Software Interrupt Register (SOFTINT) . . . . . 594
N.6.1. Setting the Software Interrupt Register . . . . . 595
N.6.2. Clearing the Software Interrupt Register . . . . . 595

**O. Reset, RED_state, and error_state . . . . . 597**

O.1. RED_state Characteristics . . . . . 597
O.2. Resets . . . . . 598
O.2.1. Externally Initiated Reset (XIR) . . . . . 598
O.2.2. error_state and Watchdog Reset (WDR) . . . . . 599
O.2.3. Software-Initiated Reset (SIR) . . . . . 599
O.3. RED_state Trap Vector . . . . . 599
O.4. Machine States . . . . . 600
O.4.1. Machines States for MTP . . . . . 603

**P. Error Handling . . . . . 607**

P.1. Error Classes and Signalling . . . . . 608
P.1.1. Error Classes in Severity . . . . . 608
P.1.2. Errors Asynchronous to Instruction Execution . . . . . 608
P.2. Corrective Actions . . . . . 609
P.2.1. Reset-Inducing ERROR Signal . . . . . 611
P.2.2. Precise Traps . . . . . 612
P.2.3. Deferred Traps . . . . . 612
P.2.4. Disrupting Traps . . . . . 615
P.3. Related Traps . . . . . 616
P.4. Related Registers / Error Logging . . . . . 617
P.5. Signalling / Special ECC . . . . . 618
P.6. Error Handling in MTPs . . . . . 619

**Q. Performance Instrumentation . . . . . 621**

**Bibliography . . . . . 623**

# Preface

SPARC® V9 is the standard instruction set architecture developed by SPARC International for 64-bit SPARC processors. Although the standard serves the needs of application programmers, some processor functions that primarily affect system programmers are left uncovered or implementation dependent in the standard. Sun Microsystems, with its UltraSPARC™ IV implementation, and Fujitsu, with its SPARC64® VI implementation, jointly worked to increase the commonalities between their processors in the areas that SPARC V9 does not cover.

The *SPARC Joint Programming Specification 2* is based on SPARC V9. It first defines the programmer's model and the hardware behavior common to the processors from both companies. These aspects of the processors conform to the instruction set architecture, memory model, error and trap handling specified by *The SPARC Architecture Manual-Version 9* and also conform to additional feature conventions jointly established by Sun and Fujitsu. Some features, especially initialization, error detection, error recovery, etc., strongly depend on the specific implementation and cannot be common. Such features and specific implementation-dependent deviations from common definitions are detailed in JPS2 Extensions documents that are companions to this document.

## Who Should Use This Book

Programmers who write code for the UltraSPARC IV processor, the SPARC64 VI processor, and the successors of both processor lines will find this book, combined with JPS2 Extensions documents, the single depository of information that logic designers, operating system programmers, or application software programmers can share to gain a common understanding of the features of SPARC processors from both Sun Microsystems, Inc., and Fujitsu.

# How This Book Is Organized

The book is organized in major sections: **Common Specification**, which contains information that is common to all implementations, and **JPS2 Extensions documents**. At present, we describe two implementations: SPARC64 VI, the Fujitsu implementation of SPARC V9, and UltraSPARC IV, the Sun Microsystems implementation. Other implementations may be added in the future.

The **Common Specification** section and the **JPS2 Extensions documents** begin at Chapter 1, page 1, each supplement contains its own index, and all supplements in general follow the organization of the *The SPARC Architecture Manual-Version 9*, as follows.

Chapter 1, *Overview*, describes features, attributes, and components and provides a high-level view of SPARC V9 and the implementations.

Chapter 2, *Definitions*, defines terms you should know before reading the book or parts.

Chapter 3, *Architectural Overview*, describes processors and instructions.

Chapter 4, *Data Formats*, presents data types.

Chapter 5, *Registers*, discusses the two types of registers: general-purpose (working data) registers and control/status registers.

Chapter 6, *Instructions*, details nuts and bolts of instructions.

Chapter 7, *Traps*, describes types, behavior, control, and processing of traps.

Chapter 8, *Memory Models*, discusses three types of memory models: Total Store Order, Partial Store Order, and Relaxed Memory Order.

Chapter 9, *Multithreaded Processing (MTP)*, describes features to support multithreaded processing (MTP).

An extensive set of appendixes complements the chapters. Appendixes D, H, I, J, and K contain material from *The SPARC Architecture Manual-Version 9*.

*Appendix A, Instruction Definitions*
*Appendix B, IEEE Std 754-1985 Requirements for SPARC V9*
*Appendix C, Implementation Dependencies*
Appendix D, *Formal Specification of the Memory Models*
*Appendix E, Opcode Maps*
*Appendix F, Memory Management Unit*
*Appendix G, Assembly Language Syntax*
*Appendix H, Software Considerations (Informative)*

*Appendix I, Extending the SPARC V9 Architecture (Informative)*
*Appendix J, Programming with the Memory Models (Informative)*
*Appendix K, Changes from SPARC V8 to SPARC V9*
Appendix L, *Address Space Identifiers (ASIs)*
Appendix N, *Interrupt Handling*
Appendix O, *Reset, RED_state, and error_state*
Appendix P, *Error Handling*

The JPS2 Extensions documents accompanying the book contain additional appendixes on implementation-specific topics such as cache organization, performance instrumentation, and interconnect programming model.

For navigation suggestions, see Chapter 1, *Overview*.

# Editorial Conventions

For editorial conventions, see Chapter 1, *Overview*. Notational conventions of *SPARC Joint Programming Specification 2* generally follow those of *The SPARC Architecture Manual-Version 9* and differ slightly from the standard Sun Microsystems notational conventions.

# Related Reading

The *SPARC Joint Programming Specification 2* refers to these related books:

- *The SPARC Architecture Manual-Version 9*
- *UltraSPARC™ User's Manual*
- *SPARC64™ Processor User's Guide*
- *SPARC Joint Programming Specification (JPS1): Commonality*

See also the bibliography section of **Common Specification** and the **JPS2 Extensions documents**.

# Overview

The *SPARC Joint Programming Specification 2* (SPARC JPS2) specifies a particular subset of SPARC V9 implementations, including Fujitsu's SPARC64 VI, Sun Microsystem's UltraSPARC IV, and certain successors to those processors.

SPARC JPS2 was derived directly from the source text of *The SPARC Architecture Manual-Version 9* and from JPS2's predecessor, the *SPARC Joint Programming Specification*. Some theoretical material contained in *The SPARC Architecture Manual-Version 9* has been omitted, but for some implementors, this theoretical information is important. In particular, operating system programmers who write memory management software, compiler writers who write machine-specific optimizers, and anyone who writes code to run on all SPARC V9-compatible machines should obtain and use *The SPARC Architecture Manual-Version 9*.

Software that is intended to be portable across all SPARC V9 processors should adhere to *The SPARC Architecture Manual-Version* 9.

Material in this document identified as relevant to SPARC JPS2 (or just "JPS2") processors may not apply to other SPARC V9 processors. Therefore, in Appendixes D, H, I, J, and K, we duplicated the information contained in the same appendixes of *The SPARC Architecture Manual-Version 9*. Because we have added and deleted a significant number of tables and figures, the table and figure numbers in this guide are not parallel with the numbers in *The SPARC Architecture Manual-Version 9*.

In this book, the word *architecture* refers to the machine details that are visible to an assembly language programmer or to the compiler code generator. It does not include details of the implementation that are not visible or easily observable by software.

In this chapter, we discuss:


- *Navigating the SPARC Joint Programming Specification 2* on page 2
- *Fonts and Notational Conventions* on page 3
- *SPARC V9 Architecture* on page 5

## 1.1 Navigating the *SPARC Joint Programming Specification 2*

If you are new to SPARC, read Chapter 3, *Architectural Overview*, study the definitions in Chapter 2, *Definitions*, then look into the subsequent sections and appendixes for more details in areas of interest to you.

If you are familiar with SPARC V8 but not SPARC V9, you should review the list of changes in Appendix K. For additional details of architectural changes, review the following:

- Chapter 4, *Data Formats*, for a description of the supported data formats
- Chapter 5, *Registers*, for a description of the register set
- Chapter 6, *Instructions*, for a description of the new instructions
- Chapter 7, *Traps*, for a description of the trap model
- Chapter 8, *Memory Models*, for a description of the memory models
- Chapter 9, *Multithreaded Processing (MTP)*, for a description of new MTP features
- *Appendix , Instruction Definitions*, for descriptions of the instructions

Finally, if you are familiar with the SPARC V9 architecture and want to familiarize yourself with the Sun- and Fujitsu-specific implementations, study the following chapters and appendices in the Sun- and Fujitsu-specific Extension documents:

- *Chapter 2, Definitions*
- *Appendix , Instruction Definitions*, for descriptions of specific instruction extensions
- *Appendix C, Implementation Dependencies*, for descriptions of resolutions of all SPARC V9 implementation dependencies
- *Appendix E, Opcode Maps*, to see how opcode extensions fit into the SPARC V9 opcode maps
- *Appendix F, Memory Management Unit*, to see the common features of the SPARC JPS2 Memory Management Unit and the implementation-specific features of that MMU.
- *Appendix G, Assembly Language Syntax*, to see extensions to the SPARC V9 assembly language syntax; in particular, synthetic instructions are documented in this appendix
- Appendix L, *Address Space Identifiers (ASIs)*, for a complete list of supported ASIs
- Appendix M, *Caches and Cache Coherency*, for a description of caches on JPS2 processors
- Appendix N, *Interrupt Handling*, for information on how interrupts are handled

- Appendix O, *Reset, RED_state, and error_state*, for a detailed description of resets, RED_state, and Error_state
- Appendix P, *Error Handling*, for a description of error handling in JPS2 processors

## 1.2 Fonts and Notational Conventions

Fonts are used as follows:

- *Italic* font is used for emphasis, book titles, and the first instance of a word that is defined.
- *Italic* font is also used for assembly language terms.
- *Italic sans serif* font is used for exception and trap names. For example, "The *privileged_action* exception...."
- `Typewriter` font (Courier) is used for register fields (named bits), instruction fields, and read-only register fields. For example: "The `rs1` field contains...."
- Subfields, e.g. `TICK.npt`, are indicated by periods ('.').
- UPPERCASE items are generally acronyms, instruction names, or register names. Some common acronyms appear in the glossary in Chapter 2, *Definitions*. **Note:** Names of some instructions contain both upper- and lowercase letters.
- Underbar characters join words in register, register field, exception, and trap names. **Note:** Such words can be split across lines at the underbar without an intervening hyphen. For example: "This is true whenever the integer_condition_ code field...."

The following notational conventions are used:

- Square brackets, [ ], indicate a numbered register in a register file. For example: "`r`[0] contains...."
- Angle brackets, < >, indicate a bit number or colon-separated range of bit numbers within a field. For example: "Bits `FSR`<29:28> and `FSR`<12> are...."
- Curly braces, { }, indicate textual substitution. For example, the string "ASI_PRIMARY{_LITTLE}" expands to "ASI_PRIMARY" and "ASI_PRIMARY_LITTLE."
- The ∥ symbol designates concatenation of bit vectors. A comma (**,**) on the left side of an assignment separates quantities that are concatenated for the purpose of assignment. For example, if X, Y, and Z are 1-bit vectors and the 2-bit vector T equals $11_2$, then

  $(X, Y, Z) \leftarrow 0 \parallel T$

  results in X = 0, Y = 1, and Z = 1.

## 1.2.1 Implementation Dependencies

The implementors of SPARC V9 processors are allowed to resolve some aspects of the architecture in machine-dependent ways. Each possible implementation dependency is indicated in *The SPARC Architecture Manual-Version 9* by the notation "**IMPL. DEP. #*nn***: Some descriptive text." The number ***nn*** enumerates the dependencies in Appendix C. References to SPARC V9 implementation dependencies are indicated, as in *The SPARC Architecture Manual-Version 9*, by the notation "(impl. dep. #*nn*)." In *SPARC Joint Programming Specification 2*, we have replaced all definitions of and references to SPARC V9 implementation dependencies with implementation-specific descriptions.

## 1.2.2 Notation for Numbers

Numbers throughout this specification are decimal (base-10) unless otherwise indicated. Numbers in other bases are followed by a numeric subscript indicating their base (for example, $1001_2$, FFFF $0000_{16}$). Long binary and hex numbers within the text have spaces inserted every four characters to improve readability. Within C or assembly language examples, numbers may be preceded by "0x" to indicate base-16 (hexadecimal) notation (for example, 0xFFFF0000).

## 1.2.3 Informational Notes

This guide provides several different types of information in notes, as follows:

---

**Programming Note –** Programming notes contain incidental information about implementation-specific programming.

---

---

**V9 Compatibility Note –** Implementation notes contain information that is relevant to SPARC JPS1 and SPARC V9 implementations. Such information may not pertain to other SPARC V9 implementations.

---

---

**JPS Compatibility Note –** Compatibility notes containing information relevant to SPARC JPS1 and SPARC JPS2 processors.

---

## 1.3 SPARC V9 Architecture

This section briefly describes features, attributes, and components of the SPARC V9 architecture and, further, describes correct implementation of the architecture specification and SPARC V9-compliance levels.

### 1.3.1 Features

SPARC V9 includes the following principal features:

- **A linear 64-bit address space** with 64-bit addressing.
- **32-bit-wide instructions** — These are aligned on 32-bit boundaries in memory. Only load and store instructions access memory and perform I/O.
- **Few addressing modes** — A memory address is given as either "register + register" or "register + immediate."
- **Triadic register addresses** — Most computational instructions operate on two register operands or one register and a constant and place the result in a third register.
- **A large windowed register file** — At any one instant, a program sees 8 global integer registers plus a 24-register window of a larger register file. The windowed registers can be used as a cache of procedure arguments, local values, and return addresses.
- **Floating point** — The architecture provides an IEEE 754-compatible floating-point instruction set, operating on a separate register file that provides 32 single-precision (32-bit), 32 double-precision (64-bit), 16 quad-precision (128-bit) registers, or a mixture thereof.
- **Fast trap handlers** — Traps are vectored through a table.
- **Multiprocessor synchronization instructions** — One instruction performs an atomic read-then-set-memory operation; another performs an atomic exchange-register-with-memory operation; another compares the contents of a register with a value in memory and exchanges memory with the contents of another register if the comparison was equal (compare and swap); two others synchronize the order of shared memory operations as observed by virtual processors.
- **Predicted branches** — The branch with prediction instructions allows the compiler or assembly language programmer to give the hardware a hint about whether a branch will be taken.
- **Branch elimination instructions** — Several instructions can be used to eliminate branches altogether (for example, Move on Condition). Eliminating branches increases performance in superscalar and superpipelined implementations.

- **Hardware trap stack** — A hardware trap stack is provided to allow nested traps. It contains all of the machine state necessary to return to the previous trap level. The trap stack makes the handling of faults and error conditions simpler, faster, and safer.
- **Relaxed memory order (RMO) model** — In addition to the TSO and PSO memory models defined for SPARC V8, SPARC JPS2 offers a weak memory model called *Relaxed Memory Order*, or RMO. RMO allows the hardware to schedule memory accesses in any order as long as the program computes the correct result (adheres to processor consistency).

## 1.3.2 Attributes

SPARC V9 is a processor *instruction set architecture* (ISA) derived from SPARC V8; both architectures come from a reduced instruction set computer (RISC) lineage. As architectures, SPARC V9 and SPARC V8 allow for a spectrum of processor and system *implementations* at a variety of price/performance points for a range of applications, including scientific/engineering, programming, real-time, and commercial applications.

### 1.3.2.1 Design Goals

SPARC JPS2 is designed to be a target for optimizing compilers and high-performance hardware implementations. Implementations of SPARC JPS2 provide exceptionally high execution rates and short time-to-market development schedules.

### 1.3.2.2 Register Windows

The JPS2 processor is derived from SPARC®, which was formulated at Sun Microsystems in 1985. SPARC is based on the RISC I and II designs engineered at the University of California at Berkeley from 1980 through 1982. The SPARC “register window” architecture, pioneered in the UC Berkeley designs, allows for straightforward, high-performance compilers and a reduction in memory load/store instructions.

Note that supervisor software, not user programs, manages the register windows. The supervisor can save a minimum number of registers (approximately 24) during a context switch, thereby optimizing context-switch latency.

## 1.3.3 System Components

The SPARC V9 architecture allows for a spectrum of I/O, memory management unit (MMU), and cache system subarchitectures.

#### 1.3.3.1 SPARC JPS2 MMU

The SPARC V9 ISA does not mandate a single MMU design for all system implementations. Rather, designers are free to use the MMU that is most appropriate for their application or no MMU at all, if they wish.

Although SPARC V9 allows its implementations freedom in their MMU designs, SPARC JPS2 defines a common MMU architecture (see *Appendix F, Memory Management Unit*) with some specifics left to implementations (see Appendix F in the JPS2 Extension documents).

#### 1.3.3.2 Privileged Software

SPARC V9 does not assume that all implementations must execute identical privileged software. Thus, certain traits that are visible to privileged software have been tailored to the requirements of the system.

#### 1.3.3.3 Binary Compatibility

The most important SPARC V9 architectural mandate is binary compatibility of nonprivileged programs across implementations. Binaries executed in nonprivileged mode should behave identically on all SPARC V9 systems when those systems are running an operating system known to provide a standard execution environment. One example of such a standard environment is the SPARC V9 Application Binary Interface (ABI).

Although different SPARC V9 systems can execute nonprivileged programs at different rates, they will generate the same results as long as they are run under the same memory model. See Chapter 8, *Memory Models*, for more information.

Additionally, SPARC V9 is binary upward-compatible from SPARC V8 for applications running in nonprivileged mode that conform to the SPARC V8 ABI.

### 1.3.4 Architectural Definition

The SPARC V9 architecture is defined by the chapters and normative appendixes of *The SPARC Architecture Manual-Version 9*. A correct implementation of the architecture interprets a program strictly according to the rules and algorithms specified in the s and normative appendixes.

*SPARC Joint Programming Specification 2* defines a set of conforming implementations of the SPARC V9 architecture.

## 1.3.5 SPARC V9 Compliance

SPARC International is responsible for certifying that implementations comply with the SPARC V9 Architecture. Two levels of compliance are distinguished; an implementation may be certified at either level.

- **Level 1** – The implementation correctly interprets all of the nonprivileged instructions by any method, including direct execution, simulation, or emulation. This level supports user applications and is the architecture component of the SPARC V9 ABI.
- **Level 2** – The implementation correctly interprets both nonprivileged and privileged instructions by any method, including direct execution, simulation, or emulation. A Level 2 implementation includes all hardware, supporting software, and firmware necessary to provide a complete and correct implementation.

Note that a Level-2-compliant implementation is also Level-1 compliant.

**IMPL. DEP. #1:** Whether an instruction is implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.

Compliant implementations shall not add to or deviate from this standard except in aspects described as implementation dependent. See *Appendix C, Implementation Dependencies*.

An implementation may be claimed to be compliant only if it has been

1. Submitted to SPARC International for testing, and
2. Issued a Certificate of Compliance by SPARC International.

A system incorporating a certified implementation may also claim compliance. A claim of compliance must designate the level of compliance.

Prior to testing, a statement must be submitted for each implementation; this statement must:

- Resolve the implementation dependencies listed in *Appendix C, Implementation Dependencies*
- Identify the presence (but not necessarily the function) of any extensions
- Designate any instructions that require emulation

These statements become the property of SPARC International and may be released publicly.

Appendix C of the JPS2 Extension documents describes the manner in which implementation dependencies have been resolved.

# Definitions

This chapter defines concepts and terminology common to all implementations of SPARC JPS2.

**AFAR** Asynchronous Fault Address Register.

**AFSR** Asynchronous Fault Status Register.

**aliased** Said of each of two virtual addresses that refer to the same physical address.

**address space identifier (ASI)** An 8-bit value that identifies an address space. For each instruction or data access, the integer unit appends an ASI to the address. *See also* **implicit ASI**.

**application program** A program executed with the virtual processor in nonprivileged *mode*. **Note:** Statements made in this specification regarding application programs may not be applicable to programs (for example, debuggers) that have access to *privileged* virtual processor state (for example, as stored in a memory-image dump).

**ASI** Address space identifier.

**ASR** Ancillary State Register.

**big-endian** An addressing convention. Within a multiple-byte integer, the byte with the smallest address is the most significant; a byte's significance decreases as its address increases.

**BLD** Block load.

**BST** Block store.

**bypass ASI** An ASI that refers to memory space and for which the MMU does not perform address translation (that is, memory is accessed using a direct physical address).

**byte** Eight consecutive bits of data.

**clean window** A register window in which all of the registers contain 0, a valid address from the current address space, or valid data from the current address space.

**coherence** A set of states guaranteeing that all memory accesses are globally visible to all caches on a shared-memory bus.

**completed** A memory transaction is said to be completed when an idealized memory has executed the transaction with respect to all processors. A load is considered completed when no subsequent memory transaction can affect the value returned by the load. A store is considered completed when no subsequent load can return the value that was overwritten by the store.

**consistency** *See* **coherence**.

**context** A set of translations that supports a particular address space. *See also* **Memory Management Unit (MMU)**.

**copyback** The process of copying back a dirty cache line in response to a cache hit while snooping.

**core** In a SPARC JPS2 processor, the term core may be used to refer to either a virtual processor or a physical core.

**CPI** Cycles per instruction. The number of clock cycles it takes to execute an instruction.

**cross-call** An interprocessor call in a multiprocessor system.

**current window** The block of 24 `r` registers that is currently in use. The Current Window Pointer (`CWP`) register points to the current window.

**DCTI** Delayed control transfer instruction.

**demap** To invalidate a mapping in the MMU.

**denormalized number** A nonzero floating-point number the exponent of which has a value of zero. A more complete definition is provided in IEEE Standard 754-1985.

**deprecated** The term applied to an architectural feature (such as an instruction or register) for which a SPARC V9 implementation provides support only for compatibility with previous versions of the architecture. Use of a deprecated feature must generate correct results but may compromise software performance. Deprecated features should not be used in new SPARC V9 software and may not be supported in future versions of the architecture.

**disable (core)** The process of removing a virtual processor from operation, which will normally complete during the next system or power-on reset.

**disabled (core)** A virtual processor that is out of operation (not executing instructions and not participating in cache coherency).

**dispatch** To send a previously fetched instruction to one or more functional units for execution. Typically, the instruction is dispatched from a reservation station or other buffer of instructions waiting to be executed. (Other conventions for this term exist, but the JPS2 document attempts to use *dispatch* consistently as defined here.) *See also* **issued**.

**doubleword** An 8-byte datum. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

**enable (core)** The process of preparing a virtual processor for operation, which will normally complete during the next system or power-on reset.

**enabled (core)** A virtual processor that is in operation (participating in cache coherency, but not executing instructions unless it is also unparked). *See also* **running** *and* **disabled**.

**exception** A condition that requires special processing, typically handled by the generation of a trap and execution of trap-handler software. Some exceptions (for example, floating-point exceptions; see `FSR.tem`) may be masked (that is, trap generation disabled) so that the decision on whether or not to apply special processing can be deferred and made by software at a later time. *See also* **trap**.

**extended word** An 8-byte datum, nominally containing integer data. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

**`f` register** A floating-point register. SPARC V9 includes single-, double-, and quad-precision `f` registers.

**`fcc`*N*** One of the floating-point condition code fields `fcc0`, `fcc1`, `fcc2`, or `fcc3`.

**floating-point exception** An exception that occurs during the execution of an FPop instruction. The exceptions are *unfinished_FPop*, *unimplemented_FPop*, *sequence_error*, *hardware_error*, *invalid_fp_register*, or *IEEE_754_exception*.

**floating-point operate (FPop) instructions** Instructions that perform floating-point calculations, as defined by the `FPop1` and `FPop2` opcodes. FPop instructions do not include `FBfcc` instructions or loads and stores between memory and the floating-point unit.

**floating-point trap type** The specific type of a floating-point exception, encoded in the `FSR.ftt` field.

**floating-point unit** A processing unit that contains the floating-point registers and performs floating-point operations, as defined by this specification.

**FPop** Floating-point operate instruction.

**`FPRS`** Floating-Point Register State (register).

**`FSR`** Floating-Point Status Register.

**FPU** Floating-point unit.

**halfword** A 2-byte datum. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

**IEEE 754** IEEE Standard 754-1985, the IEEE Standard for Binary Floating-Point Arithmetic.

**IEEE-754 exception** A floating-point exception, as specified by IEEE Std 754-1985. Listed within this specification as *IEEE_754_exception*.

**implementation** Hardware or software that conforms to all of the specifications of an instruction set architecture (ISA).

**implementation dependent** An aspect of the architecture that can legitimately vary among implementations. In many cases, the permitted range of variation is specified in the SPARC V9 standard. When a range is specified, compliant implementations must not deviate from that range.

**implicit ASI** The address space identifier that is supplied by the hardware on all instruction accesses and on data accesses that do not contain an explicit ASI or a reference to the contents of the ASI register.

**informative appendix** An appendix containing information that is useful but not required to create an implementation that conforms to the SPARC V9 specification. *See also* **normative appendix**.

**initiated** *Synonym*: **issued**.

**instruction field** A bit field within an instruction word.

**instruction group** One or more independent instructions that can be dispatched for simultaneous execution.

**instruction set architecture** A set that defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. Does not define clock cycle times, cycles per instruction, data paths, etc. This specification defines the SPARC JPS2 ISA.

**integer unit** A processing unit that performs integer and control-flow operations and contains general-purpose integer registers and virtual processor state registers, as defined by this specification.

**interrupt request** A request for service presented to a virtual processor by an external device.

**ISA** Instruction set architecture.

**issued** (1) A memory transaction (load, store, or atomic load-store) is said to be "issued" when a virtual processor has sent the transaction to the memory subsystem and the completion of the request is out of the virtual processor's control. *Synonym*: **initiated**.
(2) An instruction (or sequence of instructions) is said to be *issued* when released from the virtual processor's instruction fetch unit. Typically, instructions are issued to a reservation station or other buffer of instructions

waiting to be executed. (Other conventions for this term exist, but the JPS2 document attempts to use “issue” consistently as defined here.) *See also* **dispatched**.

**IU** Integer Unit.

**leaf procedure** A procedure that is a leaf in the program’s call graph; that is, one that does not call (by using `CALL` or `JMPL`) any other procedures.

**little-endian** An addressing convention. Within a multiple-byte integer, the byte with the smallest address is the least significant; a byte’s significance increases as its address increases.

**load** An instruction that reads (but does not write) memory or reads (but does not write) location(s) in an alternate address space. Some examples of *Load* include loads into integer or floating-point registers, block loads, Load Quadword Atomic, and alternate address space variants of those instructions. *See also* **load-store** *and* **store**, the definitions of which are mutually exclusive with *load*.

**load-store** An instruction that explicitly both reads and writes memory or explicitly reads and writes location(s) in an alternate address space. *Load-store* includes instructions such as `CASA`, `CASXA`, `LDSTUB`, and the deprecated `SWAP` instruction. *See also* **load** *and* **store**, the definitions of which are mutually exclusive with *load-store*.

**may** A keyword indicating flexibility of choice with no implied preference. **Note:** “May” indicates that an action or operation is allowed; “can” indicates that it is possible.

**Memory Management Unit (MMU)** The address translation hardware in the SPARC JPS2 implementation that translates 64-bit virtual address into physical addresses. The MMU is composed of the TLBs, ASRs, and ASI registers used to manage address translation. *See also* **context**, **physical address**, *and* **virtual address**.

**MTP** MultiThreaded Processor. A processor cluster containing more than one virtual processor core. (May also be used as an acronym for Multithreaded Processing.)

**multiprocessor system** A system containing more than one virtual processor, which share some resources (notably, memory).

**must** *Synonym*: **shall**.

**next program counter (nPC)** A register that contains the address of the instruction to be executed next if a trap does not occur.

**NFO** Nonfault access only.

**nonfaulting load** A load operation that behaves identically to a normal load operation, except when supplied an invalid effective address by software. In that case, a register load triggers an exception whereas a nonfaulting load appears (with the assistance of system software) to ignore the exception and loads its destination register with a value of zero.

In a JPS2 virtual processor, hardware treats a nonfaulting load exactly the same as a normal load. Trap handler software is responsible for writing a value of zero to the destination register of a nonfaulting load.

*Contrast with* **speculative load**.

**nonprivileged** An adjective that describes:

(1) the state of a virtual processor when `PSTATE.priv` = 0, that is, nonprivileged mode;

(2) virtual processor state information that is accessible to software while the virtual processor is in either privileged mode or nonprivileged mode; for example, nonprivileged registers, nonprivileged ASRs, or, in general, nonprivileged state;

(3) an instruction that can be executed when the virtual processor is in either privileged mode or nonprivileged mode.

**nonprivileged mode** The mode in which a virtual processor is operating when `PSTATE.priv` = 0. *See also* **privileged**.

**normal trap** A trap processed in `execute_state` (or equivalently, a non-`RED_state` trap). *Contrast with* **`RED_state` trap**.

**normative appendix** An appendix containing specifications that must be met by an implementation conforming to the this specification. *See also* **informative appendix**.

**nontranslating ASI** An ASI that does not refer to memory (for example, refers to control/status register(s)) and for which the MMU does not perform address translation.

**nPC** Next program counter.

**NPT** Nonprivileged trap.

**NUMA** Non-uniform memory access.

**`NWINDOWS`** The number of register windows present in a particular implementation.

**opcode** A bit pattern that identifies a particular instruction.

**optional** A feature not required for SPARC V9 compliance.

**PA** Physical address.

**park** The process of suspending a virtual processor from operation. There may be a delay until the virtual processor is parked, but no heavyweight operation (such as reset) is required to complete the parking process.

**parked** Describes a virtual processor that is suspended from operation. When parked, a virtual processor is not issuing instructions for execution but (if enabled) still maintains cache coherency. *See also* **disabled** *and* **running**.

**PC** Program counter.

**PCR** Performance Control Register.

**physical address** An address that maps real physical memory or I/O device space. *See also* **virtual address**.

**physical core** The term *physical processor core*, or just *physical core*, is similar to the term fiber but represents a broader collection of hardware. A physical core includes an execution pipeline (fiber) and associated structures, such as caches, that are required for performing the execution of instructions from one or more software threads. A physical core contains one or more strands. The physical core provides the necessary resources for the threads on each strand to make forward progress at a reasonable rate. A multi-stranded physical core can execute multiple software threads either by time multiplexing or partitioning resources (or any combination there of).

**PIC** Performance Instrumentation Counter.

**PIPT** Physically indexed, physically tagged.

**POR** Power-on reset.

**prefetchable** (1) An attribute of a memory location that indicates to an MMU that `PREFETCH` operations to that location may be applied.
(2) A memory location condition for which the system designer has determined that no undesirable effects will occur if a `PREFETCH` operation to that location is allowed to succeed. Typically, normal memory is prefetchable.
Nonprefetchable locations include those that, when read, change state or cause external events to occur. For example, some I/O devices are designed with registers that clear on read; others have registers that initiate operations when read. *See* **side effect**.

**privileged** An adjective that describes:
(1) the state of a virtual processor when `PSTATE.priv` = 1, that is, *privileged mode*;
(2) virtual processor state that is only accessible to software while the virtual processor is in *privileged mode*; for example, privileged registers, privileged ASRs, or, in general, privileged state;
(3) an instruction that can be executed only when the virtual processor is in *privileged mode*.

**privileged mode** The mode in which a virtual processor is operating when `PSTATE.priv` = 1. *See also* **nonprivileged**.

**processor** A *processor* is the unit on which a shared interface is provided to control the configuration and execution of a collection of strands. A processor contains one or more physical cores, each of which contains one or more strands. On a more

physical side, a processor is a physical module that plugs into a system. And a processor is expected to appear logically as a single agent on the system interconnect fabric. *Synonym:* **processor module**.

**processor core** *See* **virtual processor**.

**processor module** *Synonym:* **processor**.

**program counter (PC)** A register that contains the address of the instruction currently being executed by the IU.

**PSO** Partial store order.

**quadword** A 16-byte datum. **Note:** The definition of this term is architecture dependent and may be different from that used in other processor architectures.

**r register** An integer register. Also called a general-purpose register or working register.

**RD** Rounding direction.

**RDPR** Read Privileged Register.

**RED_state** **R**eset, **E**rror, and **D**ebug state. The virtual processor state when PSTATE.red = 1. A restricted execution environment used to process resets and traps that occur when TL = MAXTL – 1.

**RED_state trap** A trap processed in RED_state. *Contrast with* **normal trap**.

**reserved** Describing an instruction field, certain bit combinations within an instruction field, or a register field that is reserved for definition by future versions of the architecture.

*Reserved instruction fields* shall read as 0, unless the implementation supports extended instructions within the field. The behavior of SPARC V9 processors when they encounter nonzero values in reserved instruction fields is as defined in *Reserved Opcodes and Instruction Fields* on page 123.

*Reserved bit combinations within instruction fields* are defined in Appendix A, *Instruction Definitions*. In all cases, SPARC V9 processors shall decode and trap on these reserved combinations.

*Reserved fields within registers* should always be written by software with values of those fields previously read from that register or with zeroes; they should read as zero in hardware. Software intended to run on future versions of SPARC should not assume that these fields will read as 0 or any other particular value. Throughout this specification, figures and tables illustrating registers and instruction encodings indicate reserved fields and combinations with an em dash (—).

**reset trap** A vectored transfer of control to privileged software through a fixed-address reset trap table. Reset traps cause entry into RED_state.

**restricted** Describing an address space identifier (ASI) that may be accessed only while the virtual processor is operating in privileged mode.

**running** A virtual processor that is in operation (maintaining cache coherency and issuing instructions for execution) and not parked.

**rs1, rs2, rd** The integer or floating-point register operands of an instruction. `rs1` and `rs2` are source registers; `rd` is the destination register.

**RMO** Relaxed memory order.

**SFAR** Synchronous Fault Address Register.

**SFSR** Synchronous Fault Status Register.

**shall** A keyword indicating a mandatory requirement. Designers shall implement all such mandatory requirements to ensure interoperability with other SPARC V9-compliant products. *Synonym:* **must**.

**should** A keyword indicating flexibility of choice with a strongly preferred implementation. *Synonym:* **it is recommended**.

**side effect** The result of a memory location having additional actions beyond the reading or writing of data. A side effect can occur when a memory operation on that location is allowed to succeed. Locations with side effects include those that, when accessed, change state or cause external events to occur. For example, some I/O devices contain registers that clear on read; others have registers that initiate operations when read. *See also* **prefetchable**.

**SIMD** Single Instruction/Multiple Data. A class of instructions that perform identical operations on multiple data contained (or "packed") in each source operand.

**SIR** Software-initiated reset.

**snooping** The process of maintaining coherency between caches in a shared-memory bus architecture. All cache controllers monitor (snoop) the bus to determine whether they have a copy of the shared cache block.

**speculative load** A load operation that is issued by a virtual processor speculatively, that is, before it is known whether the load will be executed in the flow of the program. Speculative accesses are used by hardware to speed program execution and are transparent to code. An implementation, through a combination of hardware and system software, must nullify speculative loads on memory locations that have side effects; otherwise, such accesses produce unpredictable results. *Contrast with* **nonfaulting load**.

**store** An instruction that writes (but does not explicitly read) memory or writes (but does not explicitly read) location(s) in an alternate address space. Some examples of *Store* includes stores from either integer or floating-point registers, block stores, Partial Store, and alternate address space variants of those instructions. *See also* **load** and **load-store**, the definitions of which are mutually exclusive with *store*.

**strand** A term that identifies the hardware state used to hold a software thread in order to execute it. Strand is specifically the software visible architected state (PC, next PC, general purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.) of a thread and any microarchitecture state required by hardware for its execution.

**subnormal number** *Synonym:* **denormalized number**.

**superscalar** An implementation that allows several instructions to be issued, executed, and committed in one clock cycle.

**supervisor software** Software that executes when the virtual processor is in privileged mode.

**suspend** *See* **park**.

**TBA** Trap base address.

**thread** A term that refers to a software entity that can be run on hardware. A thread is scheduled, may or may not be actively running on hardware at any given time, and may migrate around the hardware of a system.

**TLB** *See* **Translation Lookaside Buffer (TLB)**.

**TLB hit** The desired translation is present in the TLB.

**TLB miss** The desired translation is not present in the TLB.

**TPC** Trap-saved `PC`.

**Translation Lookaside Buffer (TLB)** A cache within an MMU that contains recent partial translations. TLBs speed up closely following translations by often eliminating the need to reread Translation Table Entries (TTEs) from memory.

**trap** The action taken by a virtual processor when it changes the instruction flow in response to the presence of an exception, reset, a `Tcc` instruction, or an interrupt. The action is a vectored transfer of control to supervisor software through a table, the address of which is specified by the privileged Trap Base Address (`TBA`) register. *See also* **exception**.

**TSB** Translation storage buffer. A table of the address translations that is maintained by software in system memory and that serves as a cache of the address translations.

**TSO** Total store order.

**TTE** Translation Table Entry. Describes the virtual-to-physical translation and page attributes for a specific page in the Page Table. In some cases, the term is explicitly used for the entries in the TSB.

**unassigned** A value (for example, an ASI number), the semantics of which are not architecturally mandated and which may be determined independently by each implementation within any guidelines given.

**undefined** An aspect of the architecture that has deliberately been left unspecified. Software should have no expectation of, nor make any assumptions about, an undefined feature or behavior. Use of such a feature can deliver unexpected results, may or may not cause a trap, can vary among implementations, and can vary with time on a given implementation.

Notwithstanding any of the above, undefined aspects of the architecture shall not cause security holes (such as allowing user software to access privileged state), put the virtual processor into privileged mode, or put a virtual processor into an unrecoverable state.

**unimplemented** An architectural feature that is not directly executed in hardware because it is optional or is emulated in software.

**uniprocessor system** A system containing a single virtual processor.

**unpark** The process of bringing a virtual processor out of suspension. There may be a delay until the virtual processor is unparked, but no heavyweight operation (such as a reset) is required to complete the unparking process.

**unparked** *Synonym:* **running**.

**unpredictable** *Synonym:* **undefined**.

**unrestricted** Describes an address space identifier (ASI) that can be used in both privileged and nonprivileged modes; that is, regardless of the value of `PSTATE.priv`.

**user application program** *Synonym:* **application program**.

**VA** Virtual address.

**virtual address** An address produced by a processor that maps all systemwide, program-visible memory. Virtual addresses usually are translated by a combination of hardware and software to physical addresses, which can be used to access physical memory.

**virtual core** *Synonym:* **virtual processor**.

**virtual processor, virtual processor core** The term *virtual processor core*, or *virtual processor*, is used to identify each strand in a processor. Each virtual processor corresponds to a specific strand on a specific physical core where there may be multiple physical cores, each with multiple strands. Each virtual processor has its own interrupt ID and the operating system can schedules independent threads on each virtual processor.

**VIS™** VIS Instruction Set.

**WDR** Watchdog reset.

**word** A 4-byte datum. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

**WRPR** Write Privileged Register.

**XIR** Externally initiated reset.

# Architectural Overview

SPARC V9 architecture supports 32- and 64-bit integer and 32- 64-, and 128-bit floating-point as its principal data types. The 32- and 64-bit floating-point types conform to IEEE Std 754-1985. The 128-bit floating-point type conforms to IEEE Std 1596.5-1992. The architecture defines general-purpose integer, floating-point, and special state/status register instructions, all encoded in 32-bit-wide instruction formats. The load/store instructions address a linear, $2^{64}$-byte virtual address space.

Text in this chapter is largely excerpted from *The SPARC Architecture Manual, Version 9*, edited by David L. Weaver and Tom Germond. Even though the SPARC JPS2 architecture has grown from the earlier, simpler SPARC V9 model, the following sections still provide useful background for understanding SPARC JPS2:


- *SPARC V9 Processor Architecture* on page 21
- *Instructions* on page 22
- *Traps* on page 27


The following section is new in SPARC JPS2:


- *Multithreaded Processor (MTP)* on page 28


## 3.1 SPARC V9 Processor Architecture

A SPARC V9 processor logically consists of an integer unit (IU) and a floating-point unit (FPU), each with its own registers. This organization allows for implementations with concurrent integer and floating-point instruction execution. Integer registers are 64 bits wide; floating-point registers are 32, 64, or 128 bits wide. Instruction operands are single registers, register pairs, register quadruples, or immediate constants.

A SPARC JPS2 virtual processor can run in either of two modes: *privileged* or *nonprivileged*. In privileged mode, the processor can execute any instruction, including privileged instructions. In nonprivileged mode, an attempt to execute a privileged instruction causes a trap to privileged software.

### 3.1.1 Integer Unit (IU)

The integer unit contains the general-purpose registers and controls the overall operation of the virtual processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU.

In addition, a SPARC JPS2 virtual processor implements two additional sets of alternate global registers: one for MMU handling and another for interrupt handling.

**IMPL. DEP. #2:** An implementation of the SPARC V9 IU may contain from 64 to 528 general-purpose 64-bit `r` registers. This corresponds to a grouping of the registers into 8 global `r` registers, 8 alternate global `r` registers, plus a circular stack of from 3 to 32 sets of 16 registers each, known as register windows. The number of register windows present (`NWINDOWS`) is implementation dependent in SPARC V9.

`NWINDOWS` = 8 in a SPARC JPS2 virtual processor.

### 3.1.2 Floating-Point Unit (FPU)

The FPU has thirty-two 32-bit (single-precision) floating-point registers, thirty-two 64-bit (double-precision) floating-point registers, and sixteen 128-bit (quad-precision) floating-point registers, some of which overlap. Double-precision values occupy an even-odd pair of single-precision register, and quad-precision values occupy a quad-aligned group of four single-precision registers.

If an FPU is not present or is not enabled, then an attempt to execute a floating-point instruction generates an *fp_disabled* trap. In either case, privileged-mode software must do the following:

- Enable the FPU and reexecute the trapping instruction, or
- Emulate the trapping instruction

## 3.2 Instructions

Instructions fall into the following basic categories:

- Memory access
- Integer arithmetic / logical / shift

- Control transfer
- State register access
- Floating-point operate
- Conditional move
- Register window management

These classes are discussed in the following subsections.

## 3.2.1 Memory Access

Load, store, load-store, and PREFETCH instructions are the only instructions that access memory. They use two r registers or an r register and a signed 13-bit immediate value to calculate a 64-bit, byte-aligned memory address. The Integer Unit appends an ASI to this address.

The destination field of the load/store instruction specifies either one or two r registers or one, two, or four f registers that supply the data for a store or that receive the data from a load.

Integer load and store instructions support byte, halfword (16-bit), word (32-bit), and doubleword (64-bit) accesses. Some versions of integer load instructions perform sign extension on 8-, 16-, and 32-bit values as they are loaded into a 64-bit destination register. Floating-point load and store instructions support word, doubleword, and quadword memory accesses.

CASA/CASXA, SWAP, and LDSTUB are special atomic memory access instructions that concurrent processes use for synchronization and memory updates.

The Atomic Quad Load instruction supplies an indivisible 128-bit (16-byte) load that is important in certain system software applications.

### 3.2.1.1 Memory Alignment Restrictions

A memory access on a SPARC JPS2 virtual processor must typically be aligned on an address boundary greater than or equal to the size of the datum being accessed. An improperly aligned address in a load, store, or load-store instruction may trigger an exception and cause a subsequent trap. For details, see *Memory Alignment Restrictions* on page 106.

### 3.2.1.2 Addressing Conventions

SPARC V9 uses big-endian byte order by default: the address of a quadword, doubleword, word, or halfword is the address of its most significant byte. Increasing the address means decreasing the significance of the unit being accessed. All instruction accesses are performed using big-endian byte order. SPARC V9 also can support little-endian byte order for data accesses only: the address of a quadword,

doubleword, word, or halfword is the address of its least significant byte. Increasing the address means increasing the significance of the unit being accessed. See *Processor State (PSTATE) Register* on page 72 for information about changing the implicit byte order to little-endian.

Addressing conventions are illustrated in FIGURE 6-4 on page 107 and FIGURE 6-5 on page 109.

### 3.2.1.3 Load / Store Alternate

Versions of load / store instructions, the *load/store alternate* instructions, can specify an arbitrary 8-bit address space identifier for the load / store data access. Access to alternate spaces $00_{16}$–$7F_{16}$ is restricted, and access to alternate spaces $80_{16}$–$FF_{16}$ is unrestricted. Some of the ASIs are available for implementation-dependent uses. Supervisor software can use the implementation-dependent ASIs to access special protected registers, such as MMU, cache control, and virtual processor state registers, and other processor- or system-dependent values. See *Address Space Identifiers (ASIs)* on page 109 for more information.

Alternate space addressing is also provided for the atomic memory access instructions `LDSTUB`, `SWAP`, and `CASA` / `CASXA`.

### 3.2.1.4 Separate I and D Memories

The interpretation of address can be unified, in which case the same translations and caching are applied to both instructions and data. Alternatively, addresses can be split, in which case instruction references use one translation mechanism and cache and data references use another, although the same main memory is shared.

In such split-memory systems, the coherency mechanism may be split, so that a write into data memory is not immediately reflected in instruction memory. For this reason, programs that modify their own code (self-modifying code) and that wish to be portable across all SPARC V9 processors must issue `FLUSH` instructions, or a system call with a similar effect, to bring the instruction and data caches into a consistent state. SPARC JPS2 virtual processors have coherent instruction and data caches. Therefore, `FLUSH` instructions are required for self-modifying code on those virtual processors to flush pipeline instruction buffers that possibly contain modified instructions but are not required for cache coherency.

### 3.2.1.5 Input / Output (I / O)

SPARC V9 assumes that input / output registers are accessed through load / store alternate instructions, normal load / store instructions, or read / write Ancillary State Register instructions (`RDASR`, `WRASR`).

**IMPL. DEP. #123:** The semantic effect of accessing input/output (I/O) locations is implementation dependent.

**IMPL. DEP. #6:** Whether the I/O registers can be accessed by nonprivileged code is implementation dependent.

**IMPL. DEP. #7:** The addresses and contents of I/O registers are implementation dependent.

#### 3.2.1.6 Memory Synchronization

Two instructions are used for synchronization of memory operations: `FLUSH` and `MEMBAR`. Their operation is explained in *Flush Instruction Memory* on page 268 and *Memory Barrier* on page 293, respectively. **Note**: `STBAR` is also available, but it is deprecated and should not be used in newly developed software.

### 3.2.2 Arithmetic / Logical / Shift Instructions

The arithmetic/logical/shift instructions perform arithmetic, tagged arithmetic, logical, and shift operations. With one exception, these instructions compute a result that is a function of two source operands; the result is either written into a destination register or discarded. The exception, `SETHI`, can be used in combination with another arithmetic or logical instruction to create a 32-bit constant in an `r` register.

Shift instructions shift the contents of an `r` register left or right by a given count. The shift distance is specified by a constant in the instruction or by the contents of an `r` register.

The integer multiply instruction performs a $64 \times 64 \rightarrow 64$-bit operation. The integer division instructions perform $64 \div 64 \rightarrow 64$-bit operations. Division by zero causes a trap. Some versions of the 32-bit multiply and divide instructions set the condition codes.

The tagged arithmetic instructions assume that the least-significant two bits of each operand are a data-type tag. These instructions set the integer condition code (`icc`) and extended integer condition code (`xcc`) overflow bits on 32-bit (`icc`) or 64-bit (`xcc`) arithmetic overflow. In addition, if any of the operands' tag bits are nonzero, `icc` is set. The `xcc` overflow bit is not affected by the tag bits.

### 3.2.3 Control Transfer

Control-transfer instructions (CTIs) include PC-relative branches and calls, register-indirect jumps, and conditional traps. Most of the control-transfer instructions are delayed; that is, the instruction immediately following a control-transfer instruction

in logical sequence is dispatched before the control transfer to the target address is completed. Note that the next instruction in logical sequence may not be the instruction following the control-transfer instruction in memory.

The instruction following a delayed control-transfer instruction is called a *delay* instruction. A bit in a delayed control-transfer instruction (the *annul bit*) can cause the delay instruction to be annulled (that is, to have no effect) if the branch is not taken (or in the "branch always" case if the branch is taken).

---

**Note –** SPARC V8 specified that the delay instruction was always fetched, even if annulled, and that an annulled instruction could not cause any traps. SPARC V9 does not require the delay instruction to be fetched if it is annulled.

---

Branch and `CALL` instructions use PC-relative displacements. The jump and link (`JMPL`) and return (`RETURN`) instructions use a register-indirect target address. They compute their target addresses either as the sum of two `r` registers or as the sum of an `r` register and a 13-bit signed immediate value. The "branch on condition codes without prediction" instruction provides a displacement of ±8 Mbytes; the "branch on condition codes with prediction" instruction provides a displacement of ±1 Mbyte; the "branch on register contents" instruction provides a displacement of ±128 Kbytes; and the `CALL` instruction's 30-bit word displacement allows a control transfer to any address within ±2 gigabytes ($\pm 2^{31}$ bytes).

---

**Note –** The return from privileged trap instructions (`DONE` and `RETRY`) get their target address from the appropriate `TPC` or `TNPC` register.

---

## 3.2.4 State Register Access

The read and write state register instructions read and write the contents of state registers visible to nonprivileged software (`Y`, `CCR`, `ASI`, `PC`, `TICK`, and `FPRS`). The read and write privileged register instructions read and write the contents of state registers visible only to privileged software (`TPC`, `TNPC`, `TSTATE`, `TT`, `TICK`, `TBA`, `PSTATE`, `TL`, `PIL`, `CWP`, `CANSAVE`, `CANRESTORE`, `CLEANWIN`, `OTHERWIN`, `WSTATE`, and `VER`).

**IMPL. DEP. #8:** Software can use read / write ancillary state register instructions to read / write implementation-dependent processor registers (ASRs 16–31).

**IMPL. DEP. #9:** Whether each of the implementation-dependent read / write ancillary state register instructions (for ASRs 16–31) is privileged is implementation dependent.

### 3.2.5 Floating-Point Operate

Floating-point operate (FPop) instructions perform all floating-point calculations; they are register-to-register instructions that operate on the floating-point registers. Like arithmetic/logical/shift instructions, FPops compute a result that is a function of one or two source operands. Specific floating-point operations are selected by a subfield of the FPop1/FPop2 instruction formats.

Although not part of JPS2 commonality, the floating-point multiply-add and multiply-subtract instructions described in A.24 of the **SPARC64 VI** Extension document to JPS2 are expected to be part of the commonality in a future JPS.

### 3.2.6 Conditional Move

Conditional move instructions conditionally copy a value from a source register to a destination register, depending on an integer or floating-point condition code or upon the contents of an integer register. These instructions increase performance by reducing the number of branches.

### 3.2.7 Register Window Management

Register window instructions manage the register windows. SAVE and RESTORE are nonprivileged and cause a register window to be pushed or popped. FLUSHW is nonprivileged and causes all of the windows except the current one to be flushed to memory. SAVED and RESTORED are used by privileged software to end a window spill or fill trap handler.

## 3.3 Traps

A *trap* is a vectored transfer of control to privileged software through a trap table that may contain the first 8 instructions (32 for some frequently used traps) of each trap handler. The base address of the table is established by software in a state register (the Trap Base Address Register, TBA). The displacement within the table is encoded in the type number of each trap and the level of the trap. One-half of the table is reserved for hardware traps; one-quarter is reserved for software traps generated by trap (Tcc) instructions; the final quarter is reserved for future expansion of the architecture.

A trap causes the current PC and nPC to be saved in the TPC and TNPC registers. It also causes the CCR, ASI, PSTATE, and CWP registers to be saved in TSTATE. TPC, TNPC, and TSTATE are entries in a hardware trap stack, where the number of entries

in the trap stack is equal to the number of trap levels supported (which is 5 in a JPS2 virtual processor). A trap also sets bits in the PSTATE register, one of which can enable an alternate set of global registers for use by the trap handler. Normally, the CWP is not changed by a trap; on a window spill or fill trap; however, the CWP is changed to point to the register window to be saved or restored.

A trap can be caused by a Tcc instruction, an asynchronous exception, an instruction-induced exception, or an interrupt request not directly related to a particular instruction. Before executing each instruction, a virtual processor determines if there are any pending exceptions or interrupt requests. If any are pending, the virtual processor selects the highest-priority exception or interrupt request and causes a trap.

See Chapter 7, *Traps*, for a complete description of traps.

## 3.4 Multithreaded Processor (MTP)

A SPARC JPS2 implementation may include multiple virtual processor cores on the same processor module to provide a dense, high throughput system. This may be achieved by having a combination of multiple physical processor cores and/or multiple strands (threads) per physical processor core. Chapter 9, *Multithreaded Processing (MTP)* specifies a common interface between hardware and software for such products, referred to here as Multithreaded Processors (MTPs). It addresses issues common to MTPs, regardless of the microarchitecture of the individual physical processor cores.

The MTP Programming Model describes a set of privileged registers that are used for identification and configuration of MTPs. Equally important, the MTP Programming Model describes certain behavior that must be common for MTPs. The set of registers and the common behavior are covered in the following sections, grouped into a number of topics.

---

**JPS Compatibility Note –** Support for MTP is new in JPS2.

---

# Data Formats

The SPARC JPS2 architecture recognizes these fundamental data types:

- Signed integer: 8, 16, 32, and 64 bits
- Unsigned integer: 8, 16, 32, and 64 bits
- SIMD data formats: pixel (32-bits), fixed16 (64-bits), and fixed32 (64 bits)
- Floating point: 32, 64, and 128 bits

The widths of the data types are as follows:

- Byte: 8 bits
- Halfword: 16 bits
- Word: 32 bits
- Tagged word: 32 bits (30-bit value plus 2-bit tag)
- Doubleword: 64 bits
- Extended word: 64 bits
- Quadword: 128 bits

The signed integer values are stored as two's-complement numbers with a width commensurate with their range. Unsigned integer values, bit vectors, Boolean values, character strings, and other values representable in binary form are stored as unsigned integers with a width commensurate with their range. The floating-point formats conform to the IEEE Standard for Binary Floating-point Arithmetic, IEEE Std 754-1985. In tagged words, the least significant two bits are treated as a tag; the remaining 30 bits are treated as a signed integer.

Data formats are described in these sections:

- *Integer Data Formats* on page 30
- *Floating-Point Data Formats* on page 33
- *Graphics Data Formats* on page 37

Names are assigned to individual subwords of the multiword data formats as described in these sections:

- *Signed Integer Double* on page 31
- *Unsigned Integer Double* on page 33
- *Floating Point, Double Precision* on page 34
- *Floating Point, Quad Precision* on page 35

# 4.1 Integer Data Formats

TABLE 4-1 describes the width and ranges of the signed, unsigned, and tagged integer data formats.

**TABLE 4-1** Signed Integer, Unsigned Integer, and Tagged Format Ranges

| Data Type | Width (bits) | Range |
|---|---|---|
| Signed integer byte | 8 | $-2^7$ to $2^7 - 1$ |
| Signed integer halfword | 16 | $-2^{15}$ to $2^{15} - 1$ |
| Signed integer word | 32 | $-2^{31}$ to $2^{31} - 1$ |
| Signed integer tagged word | 32 | $-2^{29}$ to $2^{29} - 1$ |
| Signed integer double | 64 | $-2^{63}$ to $2^{63} - 1$ |
| Signed extended integer | 64 | $-2^{63}$ to $2^{63} - 1$ |
| Unsigned integer byte | 8 | 0 to $2^8 - 1$ |
| Unsigned integer halfword | 16 | 0 to $2^{16} - 1$ |
| Unsigned integer word | 32 | 0 to $2^{32} - 1$ |
| Unsigned integer tagged word | 32 | 0 to $2^{30} - 1$ |
| Unsigned integer double | 64 | 0 to $2^{64} - 1$ |
| Unsigned extended integer | 64 | 0 to $2^{64} - 1$ |

TABLE 4-2 describes the memory and register alignment for integer data. All registers in the integer register file are 64 bits wide, but can be used to contain smaller (narrower) data sizes.

**TABLE 4-2** Integer Doubleword Alignment

| | | Memory Address | | Register Number | |
|---|---|---|---|---|---|
| Subformat Name | Subformat Field | Required Alignment | Address (big-endian)* | Required Alignment | Register Number |
| SD-0 | `signed_dbl_integer<63:32>` | 0 **mod** 8 | *n* | 0 **mod** 2 | *r* |
| SD-1 | `signed_dbl_integer<31:0>` | 4 **mod** 8 | *n* + 4 | 1 **mod** 2 | *r* + 1 |
| SX | `signed_ext_integer<63:0>` | 0 **mod** 8 | *n* | — | *r* |
| UD-0 | `unsigned_dbl_integer<63:32>` | 0 **mod** 8 | *n* | 0 **mod** 2 | *r* |
| UD-1 | `unsigned_dbl_integer<31:0>` | 4 **mod** 8 | *n* + 4 | 1 **mod** 2 | *r* + 1 |
| UX | `unsigned_ext_integer<63:0>` | 0 **mod** 8 | *n* | — | *r* |

* The Memory Address in this table applies to big-endian memory accesses. Word and byte order are reversed when little-endian accesses are used.

The data types are illustrated in the following subsections.

## 4.1.1 Signed Integer Data Types

Figures in this section illustrate the following signed data types:

- Signed integer byte
- Signed integer halfword
- Signed integer word
- Signed integer doubleword
- Signed extended integer

### 4.1.1.1 Signed Integer Byte, Halfword, and Word

FIGURE 4-1 illustrates the signed integer byte, halfword, and word data formats.

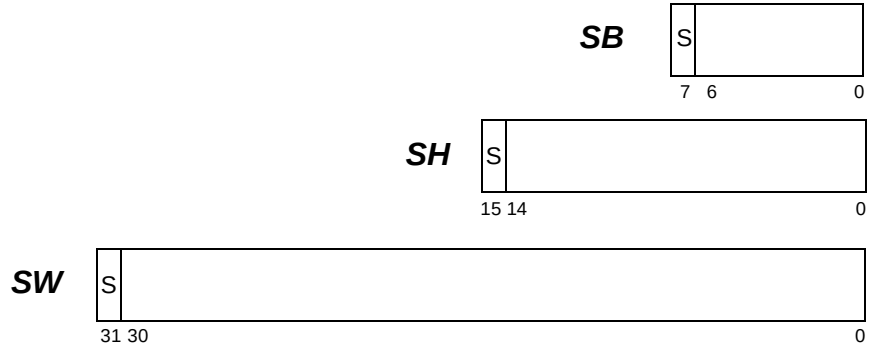



**FIGURE 4-1** Signed Integer Byte, Halfword, and Word Data Formats

### 4.1.1.2 Signed Integer Double

FIGURE 4-2 illustrates both components (SD-0 and SD-1) of the signed integer double data format.

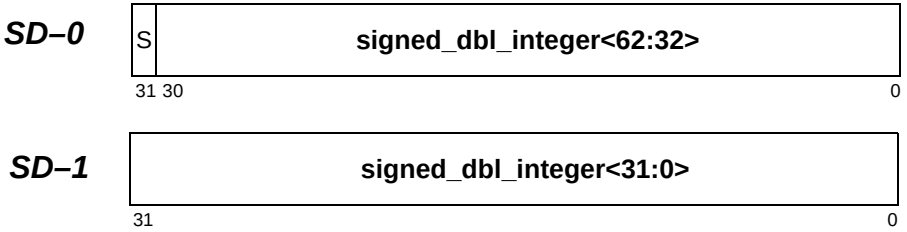



**FIGURE 4-2** Signed Integer Double Data Format

#### 4.1.1.3 Signed Extended Integer

FIGURE 4-3 illustrates the signed extended integer (SX) data format.

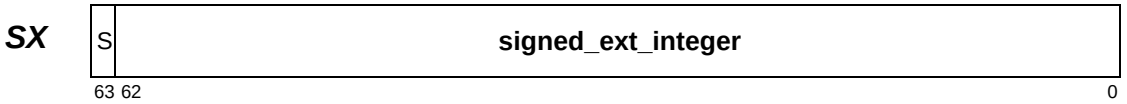



**FIGURE 4-3** Signed Extended Integer Data Format

### 4.1.2 Unsigned Integer Data Types

Figures in this section illustrate the following unsigned data types:

- Unsigned integer byte
- Unsigned integer halfword
- Unsigned integer word
- Unsigned integer doubleword
- Unsigned extended integer

#### 4.1.2.1 Unsigned Integer Byte, Halfword, and Word

FIGURE 4-4 illustrates the unsigned integer byte data format.

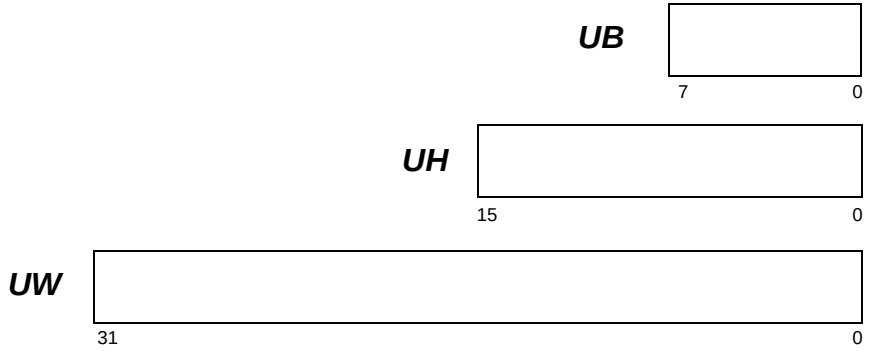



**FIGURE 4-4** Unsigned Integer Byte, Halfword, and Word Data Formats

#### 4.1.2.2 Unsigned Integer Double

FIGURE 4-5 illustrates both components (UD-0 and UD-1) of the unsigned integer double data format.

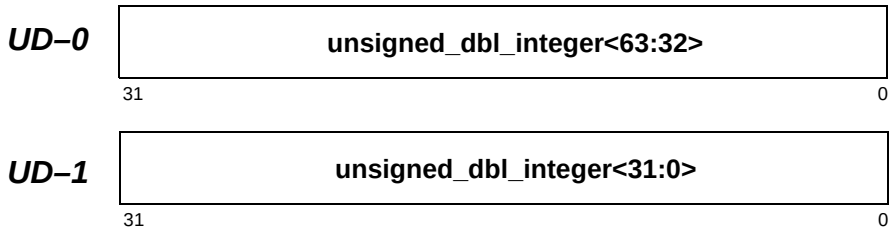



**FIGURE 4-5** Unsigned Integer Double Data Format

#### 4.1.2.3 Unsigned Extended Integer

FIGURE 4-6 illustrates the unsigned extended integer (UX) data format.

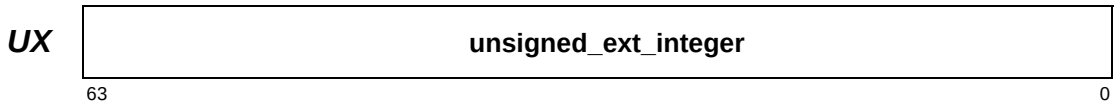



**FIGURE 4-6** Unsigned Extended Integer Data Format

### 4.1.3 Tagged Word

FIGURE 4-7 illustrates the tagged word data format.

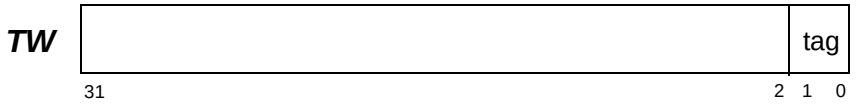



**FIGURE 4-7** Tagged Word Data Format

## 4.2 Floating-Point Data Formats

Single-precision, double-precision, and quad-precision floating-point data types are described below.

## 4.2.1 Floating Point, Single Precision

FIGURE 4-8 illustrates the floating-point single-precision data format, and TABLE 4-3 describes the formats.

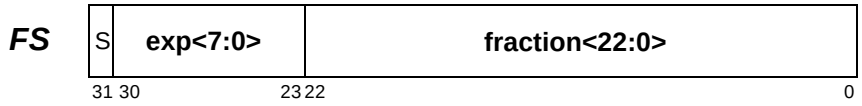



**FIGURE 4-8** Floating-Point Single-Precision Data Format

**TABLE 4-3** Floating-Point Single-Precision Format Definition

| | |
|---|---|
| s = sign (1 bit) | |
| e = biased exponent (8 bits) | |
| f = fraction (23 bits) | |
| *u* = undefined | |
| Normalized value (0 < e < 255): | $(-1)^s \times 2^{e-127} \times 1.f$ |
| Subnormal value (e = 0): | $(-1)^s \times 2^{-126} \times 0.f$ |
| Zero (e = 0) | $(-1)^s \times 0$ |
| Signalling NaN | s = *u*; e = 255 (max); f = .0*uu--uu*<br>(At least one bit of the fraction must be nonzero) |
| Quiet NaN | s = *u*; e = 255 (max); f = .1*uu--uu* |
| − ∞ (negative infinity) | s = 1; e = 255 (max); f = .000--00 |
| + ∞ (positive infinity) | s = 0; e = 255 (max); f = .000--00 |

## 4.2.2 Floating Point, Double Precision

FIGURE 4-9 illustrates both components (FD-0 and FD-1) of the floating-point double-precision data format, and TABLE 4-4 describes the formats.

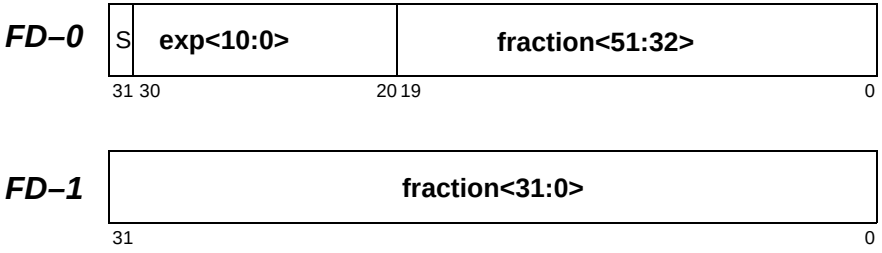



**FIGURE 4-9** Floating-Point Double-Precision Data Format

**TABLE 4-4** Floating-Point Double-Precision Format Definition

| s = sign (1 bit)<br>e = biased exponent (11 bits)<br>f = fraction (52 bits)<br>*u* = undefined | |
|---|---|
| Normalized value (0 < e < 2047): | $(-1)^s \times 2^{e-1023} \times 1.f$ |
| Subnormal value (e = 0): | $(-1)^s \times 2^{-1022} \times 0.f$ |
| Zero (e = 0) | $(-1)^s \times 0$ |
| Signalling NaN | s = *u*; e = 2047 (max); f = .0*uu*--*uu*<br>(At least one bit of the fraction must be nonzero) |
| Quiet NaN | s = *u*; e = 2047 (max); f = .1*uu*--*uu* |
| − ∞ (negative infinity) | s = 1; e = 2047 (max); f = .000--00 |
| + ∞ (positive infinity) | s = 0; e = 2047 (max); f = .000--00 |

## 4.2.3 Floating Point, Quad Precision

FIGURE 4-10 illustrates all four components (FQ-0 through FQ-3) of the floating-point quad-precision data format, and TABLE 4-5 describes the formats.

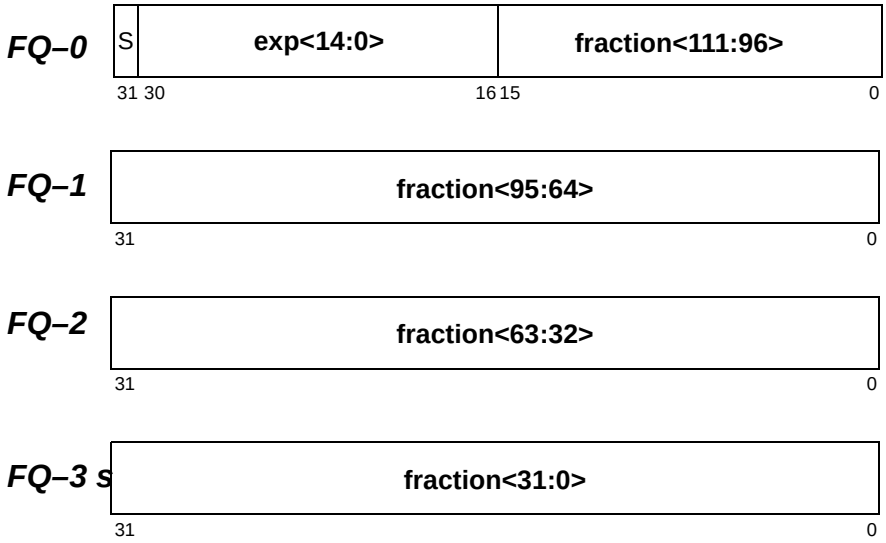



**FIGURE 4-10** Floating-Point Quad-Precision Data Format

**TABLE 4-5** Floating-Point Quad-Precision Format Definition

| |
|---|
| s = sign (1 bit) |
| e = biased exponent (15 bits) |
| f = fraction (112 bits) |
| *u* = undefined |

| | |
|---|---|
| Normalized value (0 < e < 32767): | $(-1)^s \times 2^{e-16383} \times 1.f$ |
| Subnormal value (e = 0): | $(-1)^s \times 2^{-16382} \times 0.f$ |
| Zero (e = 0) | $(-1)^s \times 0$ |
| Signalling NaN | s = *u*; e = 32767 (max); f = .0*uu*--*uu*<br>(At least one bit of the fraction must be nonzero) |
| Quiet NaN | s = *u*; e = 32767 (max); f = .1*uu*--*uu* |
| − ∞ (negative infinity) | s = 1; e = 32767 (max); f = .000--00 |
| + ∞ (positive infinity) | s = 0; e = 32767 (max); f = .000--00 |

## 4.2.4 Floating-Point Data Alignment in Memory and Registers

TABLE 4-6 describes the address and memory alignment for floating-point data.

**TABLE 4-6** Floating-Point Doubleword and Quadword Alignment

| | | Memory Address | | Register Number | |
|---|---|---|---|---|---|
| **Subformat Name** | **Subformat Field** | **Required Alignment** | **Address (big-endian)*** | **Required Alignment** | **Register Number** |
| FD-0 | `s:exp<10:0>:fraction<51:32>` | 0 **mod** 4 † | *n* | 0 **mod** 2 | *f* |
| FD-1 | `fraction<31:0>` | 0 **mod** 4 † | *n* + 4 | 1 **mod** 2 | *f* + 1 |
| FQ-0 | `s:exp<14:0>:fraction<111:96>` | 0 **mod** 4 ‡ | *n* | 0 **mod** 4 | *f* |
| FQ-1 | `fraction<95:64>` | 0 **mod** 4 ‡ | *n* + 4 | 1 **mod** 4 | *f* + 1 |
| FQ-2 | `fraction<63:32>` | 0 **mod** 4 ‡ | *n* + 8 | 2 **mod** 4 | *f* + 2 |
| FQ-3 | `fraction<31:0>` | 0 **mod** 4 ‡ | *n* + 12 | 3 **mod** 4 | *f* + 3 |

* The Memory Address in this table applies to big-endian memory accesses. Word and byte order are reversed when little-endian accesses are used.

† Although a floating-point doubleword is required only to be word-aligned in memory, it is recommended that it be doubleword-aligned (that is, the address of its FD-0 word should be 0 **mod** 8 so that it can be accessed with doubleword loads/stores instead of multiple singleword loads/stores).

‡ Although a floating-point quadword is required only to be word-aligned in memory, it is recommended that it be quadword-aligned (that is, the address of its FQ-0 word should be 0 **mod** 16).

# 4.3 Graphics Data Formats

SIMD (single instruction / multiple data) instructions perform identical operations on multiple data contained ("packed") in each source operand. This section describes the data formats used by SIMD instructions, which are commonly used in media-oriented (audio / video / graphics) applications.

## 4.3.1 Pixel Data Format

A pixel consists of four unsigned 8-bit integers contained in a 32-bit word (see FIGURE 4-11).

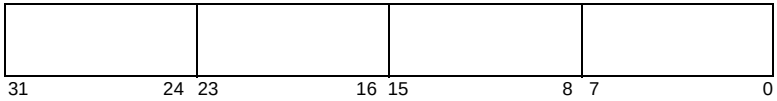



**FIGURE 4-11** Pixel Data Format

---

**Programming Note –** Typically, pixels represent intensity values for an image; for example, $\alpha$, G, B, R, where $\alpha$ = transparency, G = green, B = blue, and R = red. A SPARC JPS2 virtual processor supports:

- *Band interleaved* images, with the various color components of a point in the image stored together
- *Band sequential* images, with all of the values for one color component stored together

Conventional use is to store $\alpha$, R, G, and B values in order of most-significant to least-significant byte within the 32-bit word.

---

## 4.3.2 Fixed-Point Data Formats

A fixed-point datum is represented by a signed integer value, with an assumed binary point that is managed by software and not visible to the processor. Fixed-point data values provide an intermediate format with enough precision and dynamic range for filtering and simple image computations on pixel values. Complex calculations needing more dynamic range or precision than that provided by the fixed-point data types can be performed by means of floating-point data.

Conversion from pixel data to fixed-point data occurs through pixel multiplication. Conversion from fixed-point data to pixel data is done with the `FPACK` instructions, which clip and truncate to an 8-bit unsigned integer value. Conversion from 32-bit fixed-point to 16-bit fixed-point is also supported with the `FPACKFIX` instruction.

### 4.3.2.1 Fixed16 Data Format

The fixed-point 16-bit data format consists of four 16-bit, signed, fixed-point values contained in a 64-bit word. FIGURE 4-12 illustrates the Fixed16 Data format.

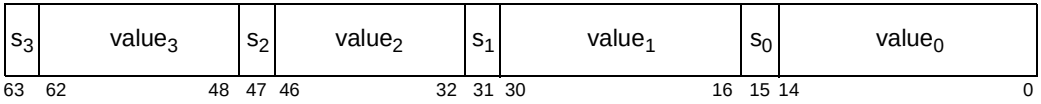



**FIGURE 4-12** Fixed16 Data Format

### 4.3.2.2 Fixed32 Data Format

The fixed-point 32-bit format consists of two 32-bit, signed, fixed-point values contained in a 64-bit word. FIGURE 4-13 illustrates the Fixed32 Data format.

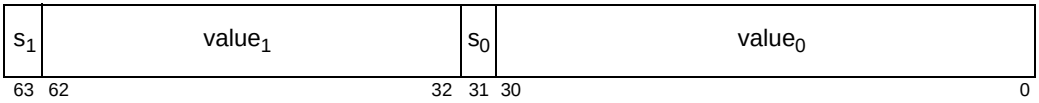



**FIGURE 4-13** Fixed32 Data Format

# Registers

Registers are described in these two main sections:

- *Registers Accessible in Nonprivileged Mode* on page 40
- *Registers Accessible in Privileged Mode Only* on page 71

Within the category of registers accessible in nonprivileged mode are the following subcategories:

- *General-Purpose r Registers* on page 41
- *Floating-Point f Registers* on page 46
- *Floating-Point State Register (FSR)* on page 53
- *State Registers* on page 62, including the following registers:
  - *Program Counters (PC, nPC)* on page 62
  - *32-bit Multiply/Divide Register (Y)* on page 63
  - *Integer Condition Codes Register (CCR)* on page 64
  - *Floating-Point Registers State (FPRS) Register* on page 65
  - *Address Space Identifier (ASI) Register* on page 66
  - *Tick (TICK) Register* on page 66
  - *Performance Control Register (PCR) (ASR 16)* on page 67
  - *Performance Instrumentation Counter (PIC) Register (ASR 17)* on page 68
  - *Graphics Status Register (GSR) (ASR 19)* on page 69
  - *System Tick (STICK) Register (ASR 24)* on page 70

Within the category of registers accessible in privileged mode are the following subcategories:

- *Non-Register-Window PR State Registers* on page 71, including the following registers:
  - *Processor State (PSTATE) Register* on page 72
  - *Trap Level Register (TL)* on page 77
  - *Processor Interrupt Level (PIL) Register* on page 78
  - *Trap Program Counter (TPC) Register* on page 78
  - *Trap Next Program Counter (TNPC) Register* on page 79
  - *Trap State (TSTATE) Register* on page 80
  - *Trap Type (TT) Registers* on page 81
  - *Trap Base Address (TBA) Register* on page 82

    - *Version (VER) Register* on page 82
- *Register Window PR State Registers* on page 83, including the following registers:
    - *Current Window Pointer (CWP) Register* on page 84
    - *Savable Windows (CANSAVE) Register* on page 85
    - *Restorable Windows (CANRESTORE) Register* on page 85
    - *Other Windows (OTHERWIN) Register* on page 85
    - *Window State (WSTATE) Register* on page 85
    - *Clean Windows (CLEANWIN) Register* on page 86
- *State Registers* on page 86, including the following registers:
    - *Dispatch Control Register (DCR) (ASR 18)* on page 86
    - *SET_SOFTINT (Set Bit(s) in SOFTINT Register) (ASR 20)* on page 88
    - *CLEAR_SOFTINT (Clear Bit(s) in SOFTINT Register) (ASR 21)* on page 88
    - *SOFTINT Register (ASR 22)* on page 89
    - *Tick Compare (TICK_COMPARE) Register (ASR 23)* on page 90
    - *System Tick Compare (STICK_COMPARE) Register (ASR 25)* on page 90
- *ASI-Accessible Registers* on page 91, including the following registers:
    - *Data Cache Unit Control Register (DCUCR)* on page 91
    - *Data Watchpoint Registers* on page 94
    - *Instruction Trap Register* on page 96
    - *Interrupt ASI Registers* on page 97
    - *MTP Registers Accessed through ASIs* on page 98
    - *Scratchpad Registers (ASI_SCRATCHPAD_n_REG)* on page 98


---

**JPS1 Compatibility Note –** MTP registers (p. 98) and Scratchpad registers (p. 98) are new in JPS2.

---

For convenience, some registers in this chapter are illustrated as fewer than 64 bits wide. Any bits not shown are reserved for future extensions to the architecture. Such reserved bits are read as zeroes and, when written by software, should be written with the values of those bits previously read from that register or with zeroes.

---

**V9 Compatibility Note –** Figures and tables in this chapter are reproduced from *The SPARC Architecture Manual-Version 9*.

---

# 5.1 Registers Accessible in Nonprivileged Mode

The registers described in this subsection are visible to nonprivileged (application or “user mode”) software.

## 5.1.1 General-Purpose r Registers

A SPARC JPS2 virtual processor contains 160 general-purpose 64-bit r registers. They are partitioned into eight *global* registers, three sets of eight *alternate global* registers, plus eight 16-register sets.

SPARC instructions use 5-bit fields to reference r registers. That is, 32 r registers are visible to software at any moment. Which 32 out of the 160 r registers are selected is described below. The visible 32 r registers are named r[0] through r[31], as illustrated in FIGURE 5-1.

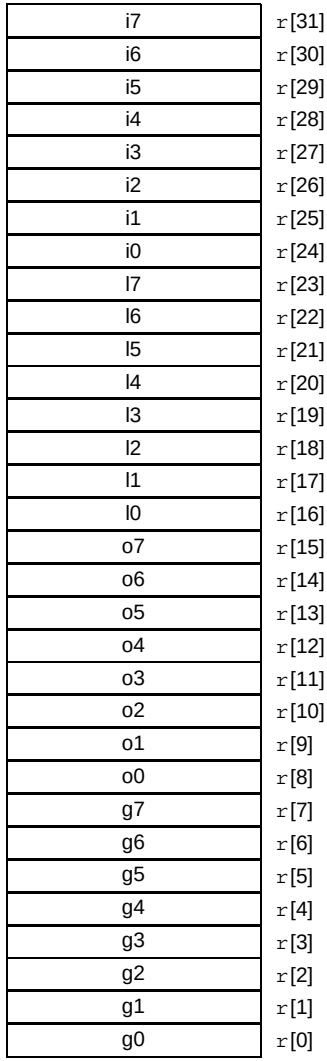



**FIGURE 5-1** General-Purpose Registers (Visible at Any Given Time)

### 5.1.1.1 Global r Registers

Registers r[0]–r[7] refer to a set of eight registers called the global registers (g0–g7). At any time, one of four sets of eight registers is enabled and can be accessed as a global register. The currently enabled set of global registers is selected by the Alternate Global (ag), Interrupt Global (ig), and MMU Global (mg) fields in the PSTATE register. See *Processor State (PSTATE) Register* on page 72 for a description of the ag, ig, and mg fields.

Global register zero (g0) always reads as zero; writes to it have no software-visible effect.

### 5.1.1.2 Windowed r Registers

One 16-register set consists of eight *in* registers and eight *local* registers. Registers r[24]–r[31] refer to the *in* registers of one of the 16-register sets, and are also called i0–i7. r[16]–r[23] refer to the *local* registers of the same 16-register set and are also called l0–l7. r[8]–r[15] refer to the *in* registers of an adjacent 16-register set, and are also called o0–o7. FIGURE 5-2 illustrates this.

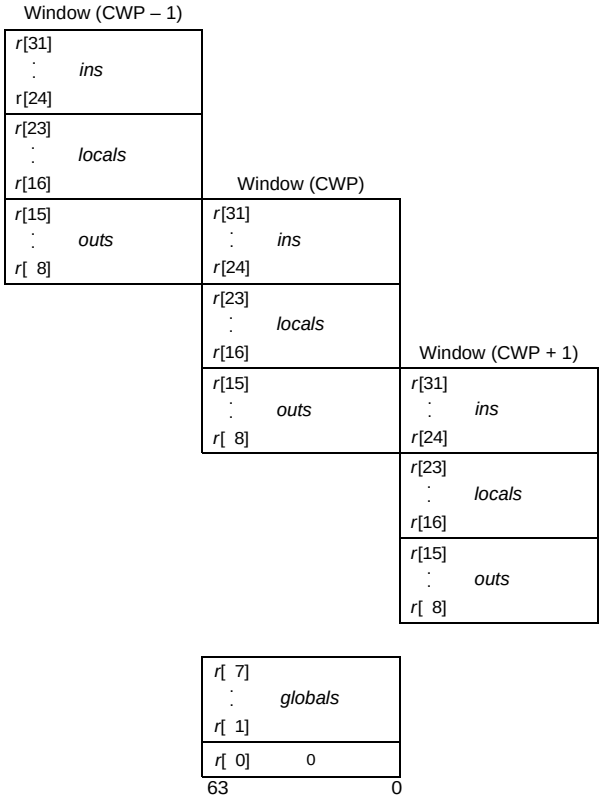
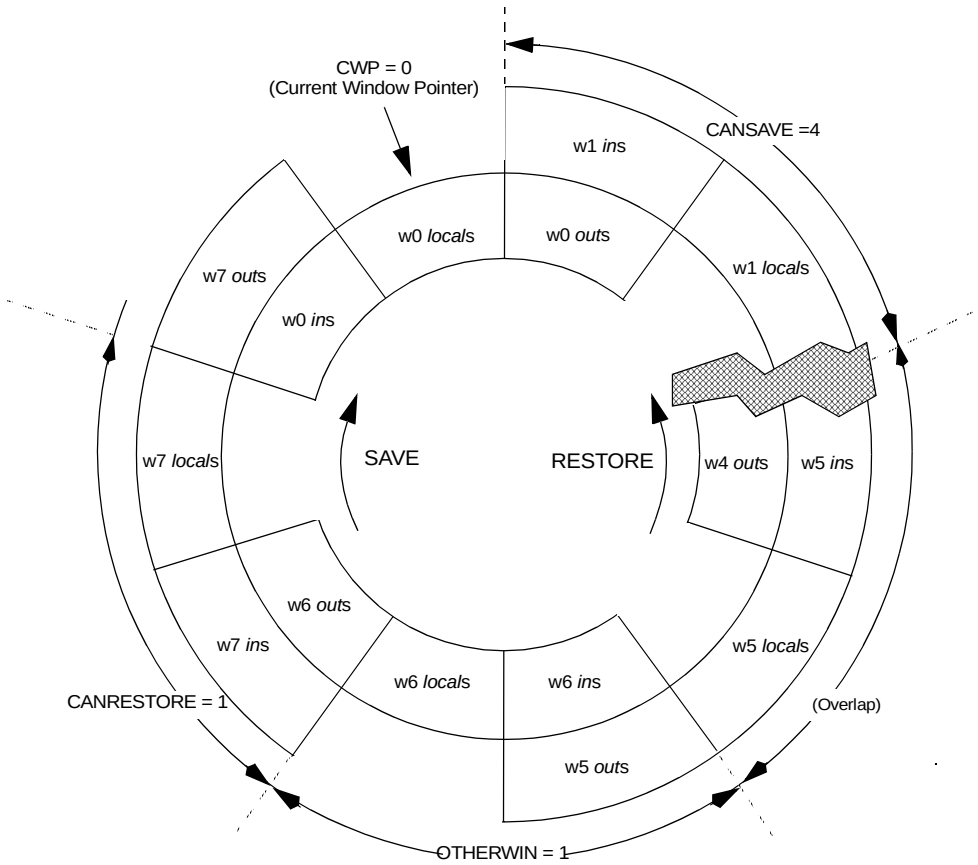



**FIGURE 5-2** Three Overlapping Windows and the Eight Global Registers

A set of 24 r registers that become visible at the same time as r[8]–r[31] is called a register window. The registers that become r[8]–r[15] in a register window are called the *out* registers of the window. Note that the *in* registers of a register window become the *out* registers of an adjacent register window. See TABLE 5-1 and FIGURE 5-3.

**TABLE 5-1** Window Addressing

| Windowed Register Address | r Register Address |
|---|---|
| *in*[0] – *in*[7] | r[24] – r[31] |
| *local*[0] – *local*[7] | r[16] – r[23] |
| *out*[0] – *out*[7] | r[ 8] – r[15] |
| *global*[0] – *global*[7] | r[ 0] – r[ 7] |




CANSAVE + CANRESTORE + OTHERWIN = NWINDOWS – 2

The current window (window 0) and the overlap window (window 5) account for the two windows in the right side of the equation. The "overlap window" is the window that must remain unused because its *ins* and *outs* overlap two other valid windows.

**FIGURE 5-3** Windowed r Registers for NWINDOWS = 8

The names *in*, *local*, and *out* originate from the fact that the *out* registers are typically used to pass parameters from (out of) a calling routine and that the called routine receives those parameters as its *in* registers.

Since the PSTATE register is writable only by privileged software, existing nonprivileged SPARC V8 software operates correctly on a SPARC JPS2 virtual processor if supervisor software ensures that nonprivileged software sees a consistent set of global registers.

---

**V9 Compatibility Note –** The number of windows or register sets, NWINDOWS, ranges from 3 to 32 in SPARC V9.

---

The total number of r registers in a given implementation is 8 (for the global registers), plus 24 (8 alternate global registers, 8 interrupt global registers, and 8 MMU global registers), plus the number of window sets times 16 registers/set. In a SPARC JPS2 virtual processor, NWINDOWS is fixed at 8. Therefore, a JPS2 virtual processor has 160 r registers.

The current window in the windowed portion of r registers is given by the current window pointer (CWP) register. The CWP is decremented by the RESTORE instruction and incremented by the SAVE instruction.

### *Overlapping Windows*

Each window shares its *in*s with one adjacent window and its *out*s with another. The outs of the CWP – 1 (modulo NWINDOWS) window are addressable as the ins of the current window, and the outs in the current window are the ins of the CWP + 1 (modulo NWINDOWS) window. The *local*s are unique to each window.

An outs register with address *o*, where $8 \le o \le 15$, refers to exactly the same register as ($o+16$) does after the CWP is incremented by 1 (modulo NWINDOWS). Likewise, an *in* register with address *i*, where $24 \le i \le 31$, refers to exactly the same register as address ($i-16$) does after the CWP is decremented by 1 (modulo NWINDOWS). See FIGURE 5-1 on page 41 and FIGURE 5-2 on page 42.

Since CWP arithmetic is performed modulo NWINDOWS, the highest-numbered implemented window (window 7 in SPARC JPS2) overlaps with window 0. The outs of window NWINDOWS − 1 are the ins of window 0. Implemented windows are numbered contiguously from 0 through NWINDOWS −1.

**Programming Note –** Since the procedure call instructions (CALL and JMPL) do not change the CWP, a procedure can be called without changing the window. See *Leaf-Procedure Optimization* on page 521.

Because the windows overlap, the number of windows available to software is one less than the number of implemented windows; that is, NWINDOWS – 1 or 7 in SPARC JPS2. When the register file is full, the outs of the newest window are the ins of the oldest window, which still contains valid data.

Window overflow is detected by the CANSAVE register, and window underflow is detected by the CANRESTORE register, both of which are controlled by privileged software. A window overflow (underflow) condition causes a window spill (fill) trap.

The *local* and *out* registers of a register window are guaranteed to contain either zeroes or old values that belong to the current context upon reentering the window through a SAVE instruction. If software executes a RESTORE followed by a SAVE, then (due to the possible intervention of a trap between the RESTORE and the SAVE) the resulting window's local and out registers may have changed since the SAVE was executed. Those registers may even contain "dirty" data, that is, data created by software running in a different context. However, if the clean_window protocol is being used, system software must guarantee that registers in the current window after a SAVE always contains only zeroes or valid data from that context. See *Clean Windows (CLEANWIN) Register* on page 86, *Savable Windows (CANSAVE) Register* on page 85, and *Restorable Windows (CANRESTORE) Register* on page 85.

**Implementation Note –** A JPS2 virtual processor supports the guarantees made in the preceding paragraph by using system software. No special hardware support is required (other than the CLEANWIN register).

*Register Window Management* on page 125 describes how the windowed integer registers are managed.

### 5.1.1.3 Special r Registers

The use of two of the r registers is fixed, in whole or in part, by the architecture:

- The value of r[0] is always zero; writes to it have no program-visible effect.
- The CALL instruction writes its own address into register r[15] (out register 7).

### *Register-Pair Operands*

LDD, LDDA, STD, and STDA instructions access a pair of words in adjacent r registers and require even-odd register alignment. The least significant bit of an r register number in these instructions is unused and must always be supplied as 0 by software (or an *illegal_instruction* exception will occur).

When the r[0]–r[1] register pair is used as a destination in LDD or LDDA, only r[1] is modified. When the r[0]–r[1] register pair is used as a source operand in STD or STDA, 0 is read from r[0] so 0 is written to the 32-bit word at the lowest destination address, and the least significant 32 bits of r[1] are written to the 32-bit word at the highest address.

An attempt to execute an LDD, LDDA, STD, or STDA instruction that refers to a misaligned (odd) destination register number causes an *illegal_instruction* trap.

### *Register Usage*

See *General-Purpose r Registers* on page 41 for information about the conventional usage of the r registers.

In FIGURE 5-3, NWINDOWS = 8. The eight *global register*s are not illustrated. CWP = 0, CANSAVE = 4, OTHERWIN = 1, and CANRESTORE = 1. If the procedure using window w0 executes a RESTORE, then window w7 becomes the current window. If the procedure using window w0 executes a SAVE, then window w1 becomes the current window.

## 5.1.2 Floating-Point f Registers

The floating-point register set consists of thirty-two 64-bit registers, which may be accessed as follows:

- Thirty-two 64-bit double-precision registers, referenced as f[0], f[2], ..., f[62]
- Sixteen 128-bit quad-precision registers, referenced as f[0], f[4], ..., f[60]
- Thirty-two 32-bit single-precision registers, referenced as f[0], f[1], ..., f[31] (only the lower half of the floating-point register file can be accessed as single-precision registers)

The floating-point registers are arranged so that some of them overlap, that is, are aliased. The layout and numbering of the floating-point registers are shown in TABLE 5-2, TABLE 5-3, and TABLE 5-4. Unlike the windowed r registers, all of the floating-point registers are accessible at any time. The floating-point registers can be

read and written by floating-point operate (FPop1/FPop2 format) instructions, by load/store single/double/quad floating-point instructions, by VIS™ instructions, and by block load and block store instructions.

**TABLE 5-2** Single-Precision Floating-Point Registers, with Aliasing

| Operand Register ID | From Register |
|---|---|
| f31 | f31<31:0> |
| f30 | f30<31:0> |
| f29 | f29<31:0> |
| f28 | f28<31:0> |
| f27 | f27<31:0> |
| f26 | f26<31:0> |
| f25 | f25<31:0> |
| f24 | f24<31:0> |
| f23 | f23<31:0> |
| f22 | f22<31:0> |
| f21 | f21<31:0> |
| f20 | f20<31:0> |
| f19 | f19<31:0> |
| f18 | f18<31:0> |
| f17 | f17<31:0> |
| f16 | f16<31:0> |
| f15 | f15<31:0> |
| f14 | f14<31:0> |
| f13 | f13<31:0> |
| f12 | f12<31:0> |
| f11 | f11<31:0> |
| f10 | f10<31:0> |
| f9 | f9<31:0> |
| f8 | f8<31:0> |
| f7 | f7<31:0> |
| f6 | f6<31:0> |
| f5 | f5<31:0> |
| f4 | f4<31:0> |
| f3 | f3<31:0> |
| f2 | f2<31:0> |

**TABLE 5-2** Single-Precision Floating-Point Registers, with Aliasing *(Continued)*

| Operand Register ID | From Register |
|---|---|
| f1 | f1<31:0> |
| f0 | f0<31:0> |

**TABLE 5-3** Double-Precision Floating-Point Registers, with Aliasing

| Operand Register ID | Operand Field | From Register |
|---|---|---|
| f62 | <63:0> | f62<63:0> |
| f60 | <63:0> | f60<63:0> |
| f58 | <63:0> | f58<63:0> |
| f56 | <63:0> | f56<63:0> |
| f54 | <63:0> | f54<63:0> |
| f52 | <63:0> | f52<63:0> |
| f50 | <63:0> | f50<63:0> |
| f48 | <63:0> | f48<63:0> |
| f46 | <63:0> | f46<63:0> |
| f44 | <63:0> | f44<63:0> |
| f42 | <63:0> | f42<63:0> |
| f40 | <63:0> | f40<63:0> |
| f38 | <63:0> | f38<63:0> |
| f36 | <63:0> | f36<63:0> |
| f34 | <63:0> | f34<63:0> |
| f32 | <63:0> | f32<63:0> |
| f30 | <31:0> | f31<31:0> |
| | <63:32> | f30<31:0> |
| f28 | <31:0> | f29<31:0> |
| | <63:32> | f28<31:0> |
| f26 | <31:0> | f27<31:0> |
| | <63:32> | f26<31:0> |
| f24 | <31:0> | f25<31:0> |
| | <63:32> | f24<31:0> |
| f22 | <31:0> | f23<31:0> |
| | <63:32> | f22<31:0> |

**TABLE 5-3** Double-Precision Floating-Point Registers, with Aliasing *(Continued)*

| Operand Register ID | Operand Field | From Register |
|---|---|---|
| f20 | <31:0> | f21<31:0> |
| | <63:32> | f20<31:0> |
| f18 | <31:0> | f19<31:0> |
| | <63:32> | f18<31:0> |
| f16 | <31:0> | f17<31:0> |
| | <63:32> | f16<31:0> |
| f14 | <31:0> | f15<31:0> |
| | <63:32> | f14<31:0> |
| f12 | <31:0> | f13<31:0> |
| | <63:32> | f12<31:0> |
| f10 | <31:0> | f11<31:0> |
| | <63:32> | f10<31:0> |
| f8 | <31:0> | f9<31:0> |
| | <63:32> | f8<31:0> |
| f6 | <31:0> | f7<31:0> |
| | <63:32> | f6<31:0> |
| f4 | <31:0> | f5<31:0> |
| | <63:32> | f4<31:0> |
| f2 | <31:0> | f3<31:0> |
| | <63:32> | f2<31:0> |
| f0 | <31:0> | f1<31:0> |
| | <63:32> | f0<31:0> |

**TABLE 5-4** Quad-Precision Floating-Point Registers, with Aliasing *(1 of 3)*

| Operand Register ID | Operand Field | From Register |
|---|---|---|
| f60 | <63:0> | f62<63:0> |
| | <127:64> | f60<63:0> |
| f56 | <63:0> | f58<63:0> |
| | <127:64> | f56<63:0> |
| f52 | <63:0> | f54<63:0> |
| | <127:64> | f52<63:0> |

**TABLE 5-4** Quad-Precision Floating-Point Registers, with Aliasing *(2 of 3)*

| Operand Register ID | Operand Field | From Register |
|---|---|---|
| f48 | <63:0> | f50<63:0> |
| | <127:64> | f48<63:0> |
| f44 | <63:0> | f46<63:0> |
| | <127:64> | f44<63:0> |
| f40 | <63:0> | f42<63:0> |
| | <127:64> | f40<63:0> |
| f36 | <63:0> | f38<63:0> |
| | <127:64> | f36<63:0> |
| f32 | <63:0> | f34<63:0> |
| | <127:64> | f32<63:0> |
| f28 | <31:0> | f31<31:0> |
| | <63:32> | f30<31:0> |
| | <95:64> | f29<31:0> |
| | <127:96> | f28<31:0> |
| f24 | <31:0> | f27<31:0> |
| | <63:32> | f26<31:0> |
| | <95:64> | f25<31:0> |
| | <127:96> | f24<31:0> |
| f20 | <31:0> | f23<31:0> |
| | <63:32> | f22<31:0> |
| | <95:64> | f21<31:0> |
| | <127:96> | f20<31:0> |
| f16 | <31:0> | f19<31:0> |
| | <63:32> | f18<31:0> |
| | <95:64> | f17<31:0> |
| | <127:96> | f16<31:0> |
| f12 | <31:0> | f15<31:0> |
| | <63:32> | f14<31:0> |
| | <95:64> | f13<31:0> |
| | <127:96> | f12<31:0> |

**TABLE 5-4** Quad-Precision Floating-Point Registers, with Aliasing *(3 of 3)*

| Operand Register ID | Operand Field | From Register |
|---|---|---|
| f8 | <31:0> | f11<31:0> |
| | <63:32> | f10<31:0> |
| | <95:64> | f9<31:0> |
| | <127:96> | f8<31:0> |
| f4 | <31:0> | f7<31:0> |
| | <63:32> | f6<31:0> |
| | <95:64> | f5<31:0> |
| | <127:96> | f4<31:0> |
| f0 | <31:0> | f3<31:0> |
| | <63:32> | f2<31:0> |
| | <95:64> | f1<31:0> |
| | <127:96> | f0<31:0> |

### 5.1.2.1 Floating-Point Register Number Encoding

Register numbers for single, double, and quad registers are encoded differently in the 5-bit register number field of a floating-point instruction. If the bits in a register number field are labeled b<4>–b<0> (where b<4> is the most significant bit of the register number), the encoding of floating-point register numbers into 5-bit instruction fields is as given in TABLE 5-5.

**TABLE 5-5** Floating-Point Register Number Encoding

| Register Operand Type | 6-bit Register Number | | | | | | Encoding in a 5-bit Register Field in an Instruction | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Single | 0 | b<4> | b<3> | b<2> | b<1> | b<0> | b<4> | b<3> | b<2> | b<1> | b<0> |
| Double | b<5> | b<4> | b<3> | b<2> | b<1> | 0 | b<4> | b<3> | b<2> | b<1> | b<5> |
| Quad | b<5> | b<4> | b<3> | b<2> | 0 | 0 | b<4> | b<3> | b<2> | 0 | b<5> |

**Note –** In SPARC V8, bit 0 of double and quad register numbers encoded in instruction fields was required to be zero. Therefore, all SPARC V8 floating-point instructions can run unchanged on a SPARC JPS2 virtual processor, using the encoding in TABLE 5-5.

### 5.1.2.2 Double and Quad Floating-Point Operands

A single `f` register can hold one single-precision operand; a double-precision operand requires an aligned pair of `f` registers, and a quad-precision operand requires an aligned quadruple of `f` registers. At a given time, the floating-point registers can hold a maximum of 32 single-precision, 16 double-precision, or 8 quad-precision values in the lower half of the floating-point register file, plus an additional 16 double-precision or 8 quad-precision values in the upper half, or mixtures of the three sizes.

---

**Programming Note –** JPS2 virtual processors trap on `LDDF(A)` and `LDQF(A)` accesses to poorly aligned double- and quad-precision data in memory (impl. dep. #109). Therefore, data to be loaded into a floating-point double or quad register that is not doubleword aligned in memory should be loaded into the lower 16 double registers (8 quad registers) by means of single-precision `LDF` instructions. If desired, the data can then be copied into the upper 16 double registers (8 quad registers).

---

An attempt to execute an instruction that refers to a misaligned floating-point register operand (that is, a quad-precision operand in a register whose 6-bit register number is not 0 **mod** 4) shall cause an *fp_exception_other* trap, with `FSR.ftt` = 6 (*invalid_fp_register*).

---

**Programming Note –** Given the encoding in TABLE 5-5, it is impossible to specify a double-precision register with a misaligned register number.

---

SPARC JPS2 does not implement quad-precision operations in hardware (impl. dep. #1). All SPARC V9 FP quad (including load and store) operations trap to the OS kernel and are emulated. Whether quad-precision multiply-add and multiply-subtract instructions are emulated in software is implementation-dependent (impl. dep. #1). Since JPS2 processors do not implement quad floating-point arithmetic operations in hardware, the *fp_exception_other* trap with `FSR.ftt` = 6 (*invalid_fp_register*) does not occur in a JPS2 processor.

## 5.1.3 Floating-Point State Register (FSR)

The Floating-Point State Register (FSR) fields, illustrated in FIGURE 5-4, contain FPU mode and status information. The lower 32 bits of the FSR are read and written by the STFSR and LDFSR instructions; all 64 bits of the FSR are read and written by the STXFSR and LDXFSR instructions, respectively. The ver, ftt, and reserved ("—") fields are not modified by LDFSR or LDXFSR.

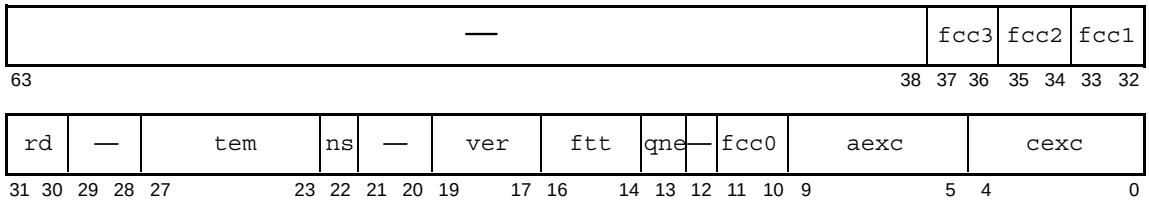



**FIGURE 5-4** FSR Fields

Bits 63–38, 29–28, 21–20, and 12 are reserved. When read by an STXFSR instruction, these bits shall read as zero. Software should issue LDXFSR instructions only with zero values in these bits, unless the values of these bits are exactly those derived from a previous STXFSR.

The subsections on pages 53 through 60 describe the remaining fields in the FSR.

### 5.1.3.1 FSR_fp_condition_codes (fcc0, fcc1, fcc2, fcc3)

The four sets of floating-point condition code fields are labeled fcc0, fcc1, fcc2, and fcc3 (fcc*N* refers to any floating-point condition code field).

---

**V9 Compatibility Note –** SPARC V9's fcc0 is the same as SPARC V8's fcc.

---

The fcc0 field consists of bits 11 and 10 of the FSR, fcc1 consists of bits 33 and 32, fcc2 consists of bits 35 and 34, and fcc3 consists of bits 37 and 36. Execution of a floating-point compare instruction (FCMP or FCMPE) updates one of the fcc*n* fields in the FSR, as selected by the instruction. The fcc*n* fields are read and written by STXFSR and LDXFSR instructions, respectively. The fcc0 field can also be read and written by STFSR and LDFSR, respectively. FBfcc and FBPfcc instructions base their control transfers on these fields. The MOVcc and FMOVcc instructions can conditionally copy a register, based on the state of these fields.

In TABLE 5-6, $f_{rs1}$ and $f_{rs2}$ correspond to the single, double, or quad values in the floating-point registers specified by a floating-point compare instruction's `rs1` and `rs2` fields. The question mark (?) indicates an unordered relation, which is true if either $f_{rs1}$ or $f_{rs2}$ is a signalling NaN or a quiet NaN. If `FCMP` or `FCMPE` generates an *fp_exception_ieee_754* exception, then `fcc`*n* is unchanged.

**TABLE 5-6** Floating-Point Condition Codes (`fcc`*n*) Fields of `FSR`

| Content of *fccn* | Indicated Relation |
|---|---|
| 0 | $f_{rs1} = f_{rs2}$ |
| 1 | $f_{rs1} < f_{rs2}$ |
| 2 | $f_{rs1} > f_{rs2}$ |
| 3 | $f_{rs1}$ ? $f_{rs2}$ (*unordered*) |

### 5.1.3.2 FSR_rounding_direction (`rd`)

Bits 31 and 30 select the rounding direction for floating-point results according to IEEE Std 754-1985. TABLE 5-7 shows the encodings.

**TABLE 5-7** Rounding Direction (`rd`) Field of `FSR`

| rd | Round Toward |
|---|---|
| 0 | Nearest (even, if tie) |
| 1 | 0 |
| 2 | + ∞ |
| 3 | – ∞ |

If the interval mode bit of Graphics Status register is equal to 1 (`GSR.im` = 1), then the value of `FSR.rd` is ignored and floating-point results are instead rounded according to `GSR.irnd`. See *Graphics Status Register (GSR) (ASR 19)* on page 69 for further details.

### 5.1.3.3 FSR_trap_enable_mask (`tem`)

Bits 27 through 23 are enable bits for each of the five IEEE-754 floating-point exceptions that can be indicated in the current_exception field (`cexc`). SeeFIGURE 5-7 on page 61. If a floating-point operate instruction generates one or more exceptions and the `tem` bit corresponding to any of the exceptions is 1, then this condition causes an *fp_exception_ieee_754* trap. A `tem` bit value of 0 prevents the corresponding IEEE 754 exception type from generating a trap.

### 5.1.3.4 FSR_nonstandard_fp (ns) (Impl. Dep. #18)

When FSR.ns = 1, bit 22 causes a SPARC JPS2 virtual processor to produce implementation-defined results that may not correspond to IEEE Std 754-1985.

---

**V9 Compatibility Note –** For instance, to obtain higher performance, implementations may convert a subnormal floating-point operand or result to zero when FSR.ns is set. For implementations in which no nonstandard floating-point mode exists, the ns bit of the FSR should always read as 0, and writes to it should be ignored.

---

A SPARC JPS2 virtual processor implements FSR.ns (impl. dep. #18). The effects of FSR.ns = 1 are as follows:

- When a floating-point source operand is subnormal, in some cases it is replaced by a floating-point zero value of the same sign (instead of causing an exception).

  **IMPL. DEP. #18a:** The cases in which this replacement is performed are implementation dependent.

  If the operand is a divisor, it will cause a "division-by-zero".

- When a floating-point operation generates a subnormal value, the value is replaced with a floating-point zero value of the same sign.

  **IMPL. DEP. #18b:** The means by which that replacement occurs are implementation dependent.

  A JPS2 implementation may implement replacement of a subnormal value by any of the following methods:

  - Always perform the replacement in hardware, never causing an exception.
  - Always perform the replacement in hardware, but also cause an *fp_exception_ieee_754* "inexact," "underflow," or "division-by-zero" exception (which may be masked with FSR.tem).
  - Sometimes perform the replacement in hardware, and sometimes cause an *fp_exception_other* exception with FSR.ftt = 2 (unfinished_FPop) so that trap handler software can perform the replacement.

If GSR.im = 1, then the value of fsr.ns is ignored and the virtual processor operates as if FSR.ns = 0. See *Graphics Status Register (GSR) (ASR 19)* on page 69 for further details.

### 5.1.3.5 FSR_version (ver)

**IMPL. DEP. #19:** Bits 19 through 17 identify one or more particular implementations of the FPU architecture.

For each SPARC V9 IU implementation (as identified by its VER.impl field), there may be one or more FPU implementations, or none. This field identifies the particular FPU implementation present. The value in FSR.ver for each implementation is strictly implementation dependent. Consult the appropriate document for each implementation for its setting of FSR.ver.

Version number 7 is reserved to indicate that no hardware floating-point controller is present.

The ver field is read-only; it cannot be modified by the LDFSR and LDXFSR instructions.

### 5.1.3.6 FSR_floating-point_trap_type (ftt)

Several conditions can cause a floating-point exception trap. When a floating-point exception trap occurs, ftt (bits 16 through 14 of the FSR) identifies the cause of the exception, the "floating-point trap type." After a floating-point exception occurs, the ftt field encodes the type of the floating-point exception until it is cleared (set to zero) by execution of an STFSR, STXFSR, or an FPop that does not cause a trap due to a floating-point exception.

The ftt field can be read by the STFSR and STXFSR instructions. The LDFSR and LDXFSR instructions do not affect ftt.

Privileged software that handles floating-point traps must execute an STFSR (or STXFSR) to determine the floating-point trap type. STFSR and STXFSR shall zero ftt after the store completes without generating an exception. If the store generates an exception and does not complete, ftt remains unchanged.

---

**Programming Note –** Neither LDFSR nor LDXFSR can be used for the purpose of clearing the ftt field, since both leave ftt unchanged. However, executing a nontrapping floating-point operate (FPop) instruction such as "`fmovs %f0,%f0`" prior to returning to nonprivileged mode will zero ftt. The ftt remains valid until the next FPop instruction completes execution.

---

The ftt field encodes the floating-point trap type according to TABLE 5-8. **Note**: The value "7" is reserved for future expansion.

**TABLE 5-8** Floating-Point Trap Type (ftt) Field of FSR)

| ftt | Trap Type | Trap Vector |
|---|---|---|
| 0 | None | No trap taken |
| 1 | IEEE_754_exception | *fp_exception_ieee_754* |
| 2 | unfinished_FPop | *fp_exception_other* |
| 3 | unimplemented_FPop | *fp_exception_other* |

**TABLE 5-8** Floating-Point Trap Type (ftt) Field of FSR) *(Continued)*

| | | |
|---|---|---|
| 4 | sequence_error | Does not occur in SPARC JPS2 |
| 5 | hardware_error | Does not occur in SPARC JPS2 |
| 6 | invalid_fp_register | Does not occur in SPARC JPS2 |
| 7 | *Reserved* | |

IEEE_754_exception, unfinished_FPop, and unimplemented_FPop will likely arise occasionally in the normal course of computation and must be recoverable by system software.

When a floating-point trap occurs, the following results are observed by software:

1. The value of aexc is unchanged.
2. When an *fp_exception_ieee_754* trap occurs, a bit corresponding to the trapping exception is set in cexc. On other traps, the value of cexc is unchanged.
3. The source and destination registers are unchanged.
4. The value of fcc*n* is unchanged.

The foregoing describes the result seen by a user trap handler if an IEEE exception is signalled, either immediately from an IEEE_754 exception or after recovery from an unfinished_FPop or unimplemented_FPop. In either case, cexc as seen by the trap handler reflects the exception causing the trap.

In the cases of an *fp_exception_other* exception with a floating-point trap type of unfinished_FPop or unimplemented_FPop that does not subsequently generate an IEEE trap, the recovery software should set cexc, aexc, and the destination register or fcc*n*, as appropriate.

***ftt = IEEE_754 exception.*** The IEEE_754 exception floating-point trap type indicates the occurrence of a floating-point exception conforming to IEEE Std 754-1985. The IEEE_754 exception type (overflow, inexact, etc.) is set in the cexc field.

The aexc and fcc*n* fields and the destination f register are not affected by an *fp_exception_ieee_754* trap.

***ftt = unfinished_FPop.*** The unfinished_FPop floating-point trap type indicates that the virtual processor was unable to generate correct results or that exceptions as defined by IEEE Std 754-1985 have occurred. Where exceptions have occurred, the cexc field is unchanged.

**IMPL. DEP. #248:** The conditions under which an *fp_exception_other* exception with floating-point trap type of unfinished_FPop can occur are implementation dependent.

***ftt = unimplemented_FPop.*** The unimplemented_FPop floating-point trap type indicates that the virtual processor decoded an FPop that it does not implement. In this case, the `cexc` field is unchanged.

All quad-precision FPop variations in a SPARC JPS2 virtual processor cause an *fp_exception_other* exception, setting `ftt` = unimplemented_FPop.

---

**V9 Compatibility Note –** `ftt` = sequence_error. Reserved for sequence errors, but never generated in a JPS2 processor.

---

---

**V9 Compatibility Note –** `ftt` = hardware_error. Reserved for hardware errors, but never generated in a JPS2 processor.

---

---

**V9 Compatibility Note –** `ftt` = invalid_fp_register. Reserved for invalid floating-point operations, but never generated by a JPS2 processor. This trap indicates that one or more operands of an FPop are misaligned; that is, a quad-precision register number is not 0 **mod** 4. An implementation shall generate an *fp_exception_other* trap with `FSR.ftt` = invalid_fp_register in this case.

---

---

**JPS Compatibility Note –** SPARC JPS2 processors do not implement quad FPops in hardware, so a quad FPop generates an unimplemented_FPop trap regardless of the specified `f` registers. `ftt` = invalid_fp_register never occurs in a SPARC JPS2 processor.

---

### 5.1.3.7 FSR_FQ_not_empty (`qne`)

This bit is reserved in SPARC JPS2 virtual processors.

### 5.1.3.8 FSR_accrued_exception (`aexc`)

Bits 9 through 5 accumulate IEEE_754 floating-point exceptions as long as floating-point exception traps are disabled through the `tem` field. See FIGURE 5-5 on page 60. After an FPop completes, the `tem` and `cexc` fields are logically ANDed together. If the result is nonzero, `aexc` is left unchanged and an *fp_exception_ieee_754* trap is generated; otherwise, the new `cexc` field is ORed into the `aexc` field and no trap is generated. Thus, while (and only while) traps are masked, exceptions are accumulated in the `aexc` field.

This field is also written with the appropriate value when an `LDFSR` or `LDXFSR` instruction is executed.

### 5.1.3.9 FSR_current_exception (cexc)

Bits 4 through 0, when set, indicate that one or more IEEE_754 floating-point exceptions were generated by the most recently executed FPop instruction. The absence of an exception causes the corresponding bit to be cleared. See FIGURE 5-10 on page 64.

---

**Programming Note –** If the FPop traps and software emulate or finish the instruction, the system software in the trap handler is responsible for creating a correct FSR.cexc value before returning to a nonprivileged program.

---

The cexc bits are set as described in *Floating-Point Exception Fields* on page 60, by the execution of an FPop that either does not cause a trap or causes an *fp_exception_ieee_754* exception with FSR.ftt = IEEE_754_exception. An IEEE 754 exception that traps shall cause exactly one bit in FSR.cexc to be set, corresponding to the detected IEEE Std 754-1985 exception.

Floating-point operations which cause an overflow or underflow condition may also cause an "inexact" condition. For overflow and underflow conditions, FSR.cexc bits are set and trapping occurs as follows:

- If an IEEE 754 overflow condition occurs:
  - if tem.ofm=0 and tem.nxm=0, the cexc.ofc and cexc.nxc bits are both set to 1, the other three bits of cexc are set to 0, and an *fp_exception_ieee_754* trap does *not* occur.
  - if tem.ofm=0 and tem.nxm=1,the cexc.nxc bit is set to 1,the other four bits of cexc are set to 0, and an *fp_exception_ieee_754* trap *does* occur.
  - if tem.ofm=1, the cexc.ofc bit is set to 1, the other four bits of cexc are set to 0, and an *fp_exception_ieee_754* trap *does* occur,
- If an IEEE 754 underflow condition occurs:
  - if tem.ufm=0 and tem.nxm=0, the cexc.ufc and cexc.nxc bits are both set to 1, the other three bits of cexc are set to 0, and an *fp_exception_ieee_754* trap does *not* occur.
  - if tem.ufm=0 and tem.nxm=1, the cexc.nxc bit is set to 1, the other four bits of cexc are set to 0, and an *fp_exception_ieee_754* trap *does* occur.
  - if tem.ufm=1, the cexc.ufc bit is set to 1, the other four bits of cexc are set to 0, and an *fp_exception_ieee_754* trap *does* occur.

The above behavior is summarized in TABLE 5-9 (where “x” indicates “don’t care”):

**TABLE 5-9** Setting of FSR.cexc bits

| Conditions | | | | | | Results | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Exception(s) Detected in f.p. operation | | | Trap Enable Mask bits (in FSR.tem) | | | *fp_exception_ieee_754* | Current Exception bits (in FSR.cexc) | | | |
| of | uf | nx | OFM | UFM | NXM | Trap Occurs? | ofc | ufc | nxc | Notes |
| - | - | - | x | x | x | no | 0 | 0 | 0 | |
| - | - | ✔ | x | x | 0 | no | 0 | 0 | 1 | |
| - | ✔ | ✔ | x | 0 | 0 | no | 0 | 1 | 1 | (1) |
| ✔ | - | ✔ | 0 | x | 0 | no | 1 | 0 | 1 | (2) |
| - | - | ✔ | x | x | 1 | yes | 0 | 0 | 1 | |
| - | ✔ | ✔ | x | 0 | 1 | yes | 0 | 0 | 1 | (1) |
| - | ✔ | ✔ | x | 1 | x | yes | 0 | 1 | 0 | (1) |
| ✔ | - | ✔ | 1 | x | x | yes | 1 | 0 | 0 | (2) |
| ✔ | - | ✔ | 0 | x | 1 | yes | 0 | 0 | 1 | (2) |

Notes:

(1) Underflow is always accompanied by inexact.

(2) Overflow is always accompanied by inexact.

If the execution of an FPop causes a trap other than *fp_exception_ieee_754*, `FSR.cexc` is left unchanged.

## 5.1.3.10 Floating-Point Exception Fields

The current and accrued exception fields and the trap enable mask assume the following definitions of the floating-point exception conditions (per IEEE Std 754-1985):

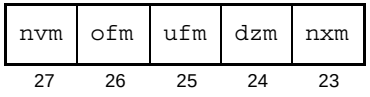



**FIGURE 5-5** Trap Enable Mask (`tem`) Fields of `FSR`

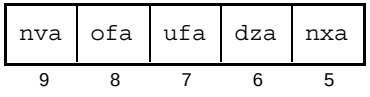



**FIGURE 5-6** Accrued Exception Bits (`aexc`) Fields of `FSR`

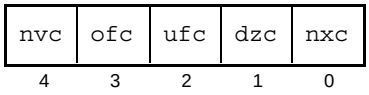



**FIGURE 5-7** Current Exception Bits (cexc) Fields of FSR

### *FSR_invalid (nvc, nva)*

An operand is improper for the operation to be performed. For example, 0.0 ÷ 0.0 and ∞ – ∞ are invalid; 1 = invalid operand(s), 0 = valid operand(s).

### *FSR_overflow (ofc, ofa)*

The result, rounded as if the exponent range were unbounded, would be larger in magnitude than the destination format's largest finite number; 1 = overflow, 0 = no overflow.

### *FSR_underflow (ufc, ufa)*

The rounded result is inexact and would be smaller in magnitude than the smallest normalized number in the indicated format; 1 = underflow, 0 = no underflow.

Underflow is never indicated when the correct unrounded result is 0. Otherwise:

If ufm = 0: Underflow occurs if a nonzero result is tiny and a loss of accuracy occurs.

If ufm = 1: Underflow occurs if a nonzero result is tiny.

SPARC V9 allows tininess to be detected either before or after rounding. In all cases and regardless of the setting of ufm, a SPARC JPS2 processor detects tininess before rounding (impl. dep. #55).

### *FSR_division-by-zero (dzc, dza)*

X ÷ 0.0, where X is subnormal or normalized; 1 = division by zero, 0 = no division by zero.

**Note**: 0.0 ÷ 0.0 *does not* set the dzc or dza bits.

### *FSR_inexact (nxc, nxa)*

The rounded result of an operation differs from the infinitely precise unrounded result; 1 = inexact result, 0 = exact result.

#### 5.1.3.11 FSR Conformance

A JPS2 implementation implements the `tem`, `cexc`, and `aexc` fields in hardware (impl. dep. #22).

---

**Programming Note –** Privileged software (or a combination of privileged and nonprivileged software) must be capable of simulating the operation of the FPU in order to handle the unimplemented_FPop, unfinished_FPop, and IEEE_754_exception floating-point trap types properly. Thus, a user application program always sees an `FSR` that is fully compliant with IEEE Std 754-1985.

---

**Note –** A SPARC JPS2 virtual processor is IEEE 754 compliant with the addition of appropriate trap handler software.

---

### 5.1.4 State Registers

Registers described in this section are the nonprivileged registers accessible with Read State Register and Write State Register instructions.

The SPARC V9 architecture provides for up to 25 ancillary state registers (ASRs), numbered from 7 through 31. ASRs numbered 7–15 are reserved for future use by the architecture and should not be referenced by software.

A state register is read and written with the Read State Register and Write State Register instructions, respectively. These instructions are privileged if the accessed register is privileged.

The SPARC V9 architecture leaves ASRs numbered 16–31 available for implementation-dependent uses. SPARC JPS2 virtual processors implement the nonprivileged state registers defined in the following subsections.

#### 5.1.4.1 Program Counters (`PC`, `nPC`)

The `PC` contains the address of the instruction currently being executed. The `nPC` holds the address of the next instruction to be executed if a trap does not occur. The low-order two bits of `PC` and `nPC` always contain 0.

For a delayed control transfer, the instruction that immediately follows the transfer instruction is known as the delay instruction. This delay instruction is executed (unless the control transfer instruction annuls it) before control is transferred to the target. Right before the execution of the delay instruction, the `nPC` points to the target of the control transfer instruction, and the `PC` points to the delay instruction. See Chapter 6, *Instructions*.

The PC is used implicitly as a destination register by all control transfer instructions. It can be read directly by an RDPC instruction.

---

**Note –** nPC is set by Delayed Control Transfer instructions (DCTIs) but is not accessible as an ASR register.

---

### 5.1.4.2 32-bit Multiply / Divide Register (Y)

---

**V9 Compatibility Note –** The Y register is deprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC V9 software. It is recommended that all instructions that reference the Y register (that is, SMUL, SMULcc, UMUL, UMULcc, MULScc, SDIV, SDIVcc, UDIV, UDIVcc, RDY, and WRY be avoided. For suitable substitute instructions, see the following pages: for the multiply instructions, see page 398; for the multiply step instruction, see page 400; for division instructions, see page 390; for the read instruction, see page 402; and for the write instruction, see page 418.

---

The low-order 32 bits of the Y register, illustrated in FIGURE 5-8, contain the more significant word of the 64-bit product of an integer multiplication, as a result of either a 32-bit integer multiply (SMUL, SMULcc, UMUL, UMULcc) instruction or an integer multiply step (MULScc) instruction. The Y register also holds the more significant word of the 64-bit dividend for a 32-bit integer divide (SDIV, SDIVcc, UDIV, UDIVcc) instruction.

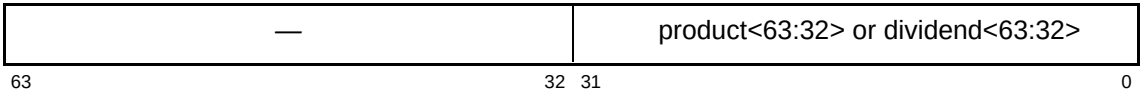



**FIGURE 5-8** Y Register

Although Y is a 64-bit register, its high-order 32 bits are reserved and always read as 0.

The Y register may be explicitly read and written by the RDY and WRY instructions, respectively.

### 5.1.4.3 Integer Condition Codes Register (CCR)

The Condition Codes Register (CCR), shown in FIGURE 5-9, holds the integer condition codes.

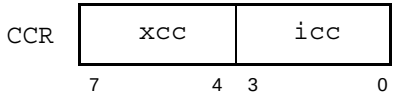



**FIGURE 5-9** Condition Codes Register

#### *CCR Condition Code Fields (xcc and icc)*

All instructions that set integer condition codes set both the xcc and icc fields. The xcc condition codes indicate the result of an operation when viewed as a 64-bit operation. The icc condition codes indicate the result of an operation when viewed as a 32-bit operation. For example, if an operation results in the 64-bit value 0000 0000 FFFF $FFFF_{16}$, the 32-bit result is negative (icc.*N* is set to 1) but the 64-bit result is nonnegative (xcc.*N* is set to 0).

Each of the 4-bit condition-code fields is composed of four 1-bit subfields, as shown in FIGURE 5-10.

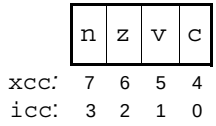



**FIGURE 5-10** Integer Condition Codes (CCR_icc and CCR_xcc)

The n bits indicate whether the 2's-complement ALU result was negative for the last instruction that modified the integer condition codes; 1 = negative, 0 = not negative.

The z bits indicate whether the ALU result was zero for the last instruction that modified the integer condition codes; 1 = zero, 0 = nonzero.

The v bits signify whether the ALU result was within the range of (was representable in) 64-bit (xcc) or 32-bit (icc) 2's complement notation for the last instruction that modified the integer condition codes; 1 = overflow, 0 = no overflow.

The c bits indicate whether a 2's complement carry (or borrow) occurred during the last instruction that modified the integer condition codes. Carry is set on addition if there is a carry out of bit 63 (xcc) or bit 31 (icc). Carry is set on subtraction if there is a borrow into bit 63 (xcc) or bit 31 (icc); 1 = carry, 0 = no carry.

**CCR_extended_integer_cond_codes (*xcc*).** Bits 7 through 4 are the IU condition codes, which indicate the results of an integer operation, with both of the operands and the result considered to be 64 bits wide. These bits are modified by the arithmetic and logical instructions, the names of which end with the letters "cc" (for

example, ANDcc) and by the WRCCR instruction. They can be modified by a DONE or RETRY instruction, which replaces these bits with the ccr field of the TSTATE register. The BPcc and Tcc instructions may cause a transfer of control based on the values of these bits. The MOVcc instruction can conditionally move the contents of an integer register based on the state of these bits. The FMOVcc instruction can conditionally move the contents of a floating-point register according to the state of these bits.

**CCR_integer_cond_codes (*icc*).** Bits 3 through 0 are the IU condition codes, which indicate the results of an integer operation, with both of the operands and the result considered to be 32 bits wide. These bits are modified by the arithmetic and logical instructions, the names of which end with the letters "cc" (for example, ANDcc) and by the WRCCR instruction. They can be modified by a DONE or RETRY instruction, which replaces these bits with the ccr field of the TSTATE register. The BPcc, Bicc, and Tcc instructions may cause a transfer of control based on the values of these bits. The MOVcc instruction can conditionally move the contents of an integer register based on the state of these bits. The FMOVcc instruction can conditionally move the contents of a floating-point register based on the state of these bits.

### 5.1.4.4 Floating-Point Registers State (FPRS) Register

The Floating-Point Registers State (FPRS) Register, shown in FIGURE 5-11, holds control information for the floating-point register file; this information is readable and writable by nonprivileged software.

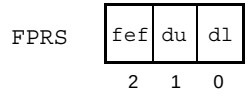



**FIGURE 5-11** Floating-Point Registers State Register

#### *FPRS_enable_fp (fef)*

Bit 2, fef, determines whether the FPU is enabled. If it is disabled, executing a floating-point instruction causes an *fp_disabled* trap. If this bit is set but the PSTATE.pef bit is not set, then executing a floating-point instruction causes an *fp_disabled* trap; that is, both FPRS.fef and PSTATE.pef must be set to enable floating-point operations.

### *FPRS_dirty_upper (du)*

Bit 1 is the “dirty” bit for the upper half of the floating-point registers; that is, f32–f62. It is set whenever any of the upper floating-point registers is modified. The virtual processor may set it pessimistically; it may be set whenever a floating-point instruction is issued, even though that instruction never completes and no output register is modified. The du bit is cleared only by software.

### *FPRS_dirty_lower (dl)*

Bit 0 is the “dirty” bit for the lower 32 floating-point registers; that is, f0–f31. It is set whenever any of the lower floating-point registers is modified. The virtual processor may set it pessimistically; it may be set whenever a floating-point instruction is issued, even though that instruction never completes and no output register is modified. The dl bit is cleared only by software.

## 5.1.4.5 Address Space Identifier (ASI) Register

The Address Space Identifier register (FIGURE 5-12) specifies the address space identifier to be used for load and store alternate instructions that use the “rs1 + simm13” addressing form. Nonprivileged (user-mode) software may write any value into the ASI register; however, values with bit 7 = 0 indicate restricted ASIs. When a nonprivileged instruction makes an access that uses an ASI with bit 7 = 0, a *privileged_action* exception is generated. See *Address Space Identifiers (ASIs)* on page 109 for details.

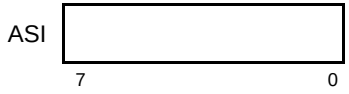



**FIGURE 5-12** Address Space Identifier Register

## 5.1.4.6 Tick (TICK) Register

FIGURE 5-13 illustrates the TICK register.

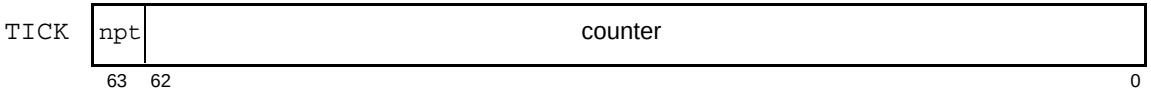



**FIGURE 5-13** TICK Register

The counter field of the TICK register is a 63-bit counter that counts processor clock cycles. Bit 63 of the TICK register is the nonprivileged trap (npt) bit, which controls access to the TICK register by nonprivileged software. Privileged software

can always read the TICK register with either the RDPR or RDTICK instruction. Privileged software can always write the TICK register with the WRPR instruction; there is no WRTICK instruction.

Nonprivileged software can read the TICK register by using the RDTICK instruction when TICK.npt = 0. When TICK.npt = 1, an attempt by nonprivileged software to read the TICK register causes a *privileged_action* exception. Nonprivileged software cannot write the TICK register.

TICK.npt is set to 1 by a power-on reset trap. The value of TICK.counter is reset after a power-on reset trap.

After the TICK register is written, reading the TICK register returns a value incremented (by 1 or more) from the last value written, rather than from some previous value of the counter. The number of counts between a write and a subsequent read does not accurately reflect the number of processor cycles between the write and the read. Software may rely only on read-to-read counts of the TICK register for accurate timing, not on write-to-read counts.

**IMPL. DEP. #105:** The difference between the values read from the TICK register on two reads should reflect the number of processor cycles executed between the reads. If an accurate count cannot always be returned, any inaccuracy should be small, bounded, and documented. An implementation may implement fewer than 63 bits in TICK.counter; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any upper bits not implemented must read as zero.

---

**Programming Note –** TICK.npt may be used by a secure operating system to control access by user software to high-accuracy timing information. The operation of the timer might be emulated by the trap handler, which could read TICK.counter and “fuzz” the value to lower accuracy.

---

### 5.1.4.7 Performance Control Register (PCR) (ASR 16)

The PCR is a read/write register used to control performance monitoring events collected in counter pairs, via the Performance Instrumentation Counter (PIC) register (ASR 17) (see page 68). Unused PCR bits read as zero; they should be written only with zeroes or with values previously read from them.

When the virtual processor is operating in privileged mode (PSTATE.priv = 1), PCR may be freely read and written by software.

**IMPL. DEP. #250:** When the virtual processor is operating in nonprivileged mode (PSTATE.priv = 0), the accessibility of PCR as a unit and of individual fields of PCR is implementation dependent. Also, which exception is raised upon detection of an access privilege violation is implementation dependent.

See Appendix Q, *Performance Instrumentation*, in the JPS2 Extension documents for a detailed discussion of the PCR and PIC register usage and event count definitions.

The Performance Control Register is illustrated in FIGURE 5-14 and described in TABLE 5-10.

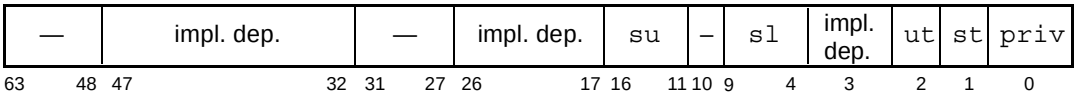



**FIGURE 5-14** Performance Control Register (PCR) (ASR 16)

**IMPL. DEP. #207:** The values and semantics of bits 47:32, 26:17, and bit 3 of the PCR are implementation dependent.

**TABLE 5-10** PCR Bit Description

| Bit | Field | Description |
|---|---|---|
| 47:32 | — | These bits are implementation dependent (impl. dep #207). |
| 26:17 | — | These bits are implementation dependent (impl. dep. #207). |
| 16:11 | su | Six-bit field selecting 1 of 64 event counts in the upper half (bits <63:32>) of the PIC. |
| 9:4 | sl | Six-bit field selecting 1 of 64 event counts in the lower half (bits <31:0>) of the PIC. |
| 3 | — | This bit is implementation dependent (impl. dep. #207). |
| 2 | ut | User Trace Enable. If set to 1, events in nonprivileged (user) mode are counted. |
| 1 | st | System Trace Enable. If set to 1, events in privileged (system) mode are counted.<br>Notes:<br>If both PCR.ut and PCR.st are set to 1, all selected events are counted.<br>If both PCR.ut and PCR.st are zero, counting is disabled.<br>PCR.ut and PCR.st are global fields which apply to all PIC pairs. |
| 0 | priv | Privileged. Controls access to the PIC register (via RDPIC or WRPIC instructions). If PCR.priv = 0, an attempt to access PIC will succeed regardless of the privilege state (PSTATE.priv). If PCR.priv = 1, access to PIC is restricted to privileged software; that is, an attempt to access PIC while PSTATE.priv = 1 will succeed but an attempt to access PIC while PSTATE.priv = 0 will result in a *privileged_action* exception.<br>PCR.priv may also have implementation-dependent effects on the accessibility (via RDPCR and WRPCR instructions) of fields in PCR itself (impl. dep. #250). |

### 5.1.4.8 Performance Instrumentation Counter (PIC) Register (ASR 17)

PIC contains two 32-bit counters that count performance-related events (such as instruction counts, cache misses, TLB misses, and pipeline stalls). Which events are actively counted at any given time is selected by the PCR register.

The difference between the values read from the PIC register at two different times reflects the number of events that occurred between register reads. Software can only rely on read-to-read PIC accesses to get an accurate count and not a write-to-read of the PIC counters.

The PIC is normally a nonprivileged-access, read / write register. However, if the priv bit of the PCR (ASR 16) is set, attempted access by nonprivileged (user) code causes a *privileged_action* exception.

Multiple PICs may be implemented. Each is accessed by way of ASR 17, using an implementation-dependent PIC pair selection field in PCR (ASR 16) (impl. dep. #207). Read / write access to the PIC will access the picu / picl counter pair selected by PCR.

The PIC is described below and illustrated in FIGURE 5-15.

| Bit | Field | Description |
|---|---|---|
| 63:32 | picu | 32-bit counter representing the count of an event selected by the SU field of the Performance Control Register (PCR) (ASR 16). See Appendix Q, *Performance Instrumentation*, in JPS2 Extension documents for a detailed definition of these counters. |
| 31:0 | picl | 32-bit counter representing the count of an event selected by the SL field of the Performance Control Register (PCR) (ASR 16). See Appendix Q, *Performance Instrumentation*, in JPS2 Extension documents for a detailed definition of these counters. |

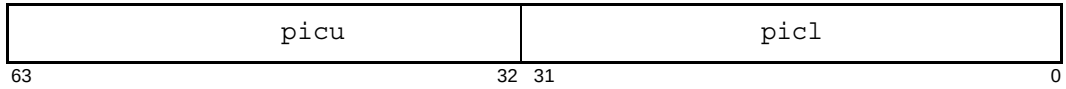



**FIGURE 5-15** Performance Instrumentation Counter (PIC) (ASR 17)

### *Counter Overflow*

On overflow, the effective counter wraps to 0, SOFTINT register bit 15 is set to 1, and an interrupt level 15 trap is generated if not masked by PSTATE.ie and PIL. The counter overflow trap is triggered on the transition from value FFFF $FFFF_{16}$ to value 0.

## 5.1.4.9 Graphics Status Register (GSR) (ASR 19)

The Graphics Status Register is a nonprivileged read / write register that is implicitly referenced by many VIS instructions. The GSR can be read through RDGSR (see A.51 on page 343) and written through WRGSR (see A.70 on page 380).

The GSR is illustrated in FIGURE 5-16 and described in TABLE 5-11.

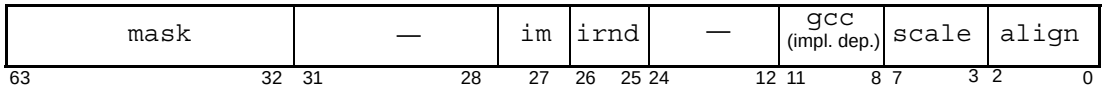



**FIGURE 5-16** Graphic Status Register (GSR) (ASR 19)

**TABLE 5-11** GSR Bit Description

| Bit | Field | Description |
|---|---|---|
| 63:32 | mask<31:0> | This field specifies the mask used by the BSHUFFLE instruction. The field contents are set by the BMASK instruction. |
| 31:28 | *Reserved* | |
| 27 | im | Interval Mode: When im = 1, the values in FSR.rd and FSR.ns are ignored; the virtual processor operates as if FSR.ns = 0 and rounds floating-point results according to GSR.irnd. |
| 26:25 | irnd<1:0> | IEEE Std 754-1985 rounding direction to use in Interval Mode (GSR.IM = 1), as follows:<br>**irnd** — **Round toward ...**<br>0 — Nearest (even, if tie)<br>1 — 0<br>2 — + ∞<br>3 — – ∞<br>When GSR.im = 1, the value in GSR.irnd overrides the value in FSR.rd. |
| 24:12 | *Reserved* | |
| 11:8 | gcc | Graphic condition codes.<br>**IMPL. DEP. #303:** Whether the GSR.gcc bits are implemented is implementation dependent. If the gcc bits are not implemented, GSR bits 11:8 are reserved. |
| 7:3 | scale<4:0> | Shift count in the range 0–31, used by the FPACK instructions for formatting. |
| 2:0 | align<2:0> | Least three significant bits of the address computed by the last executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction. |

Accesses to the Graphics Status Register cause an *fp_disabled* trap if PSTATE.pef or FPRS.fef is 0.

### 5.1.4.10 System Tick (STICK) Register (ASR 24)

The System Tick (STICK) register provides a synchronized systemwide clock that can be used for timestamping. The counter field of the STICK register is a 63-bit counter that increments at a rate determined by a clock signal external to the processor. Bit 63 of the STICK register is the nonprivileged trap (npt) bit, which

controls access to the STICK register by nonprivileged software. Privileged software can always read the STICK register with RDSTICK instruction. Privileged software can always write the STICK register with the WRSTICK instruction.

The STICK register is illustrated in FIGURE 5-17 and described below.


**FIGURE 5-17** STICK Register

Nonprivileged software can read the STICK register by using the RDSTICK instruction when STICK.npt = 0. When STICK.npt = 1, an attempt by nonprivileged software to read the STICK register causes a *privileged_action* exception. Nonprivileged software cannot write the STICK register. If PSTATE.priv = 0 when WRSTICK instruction is executed, a *privileged_opcode* exception is signalled.

STICK.npt is set to 1 by a power-on reset trap. The value of STICK.counter is cleared after a power-on reset trap.

After the STICK register is written, reading the STICK register returns a value incremented (by 1 or more) from the last value written, rather than from some previous value of the counter.

---

**Note –** The STICK register is unaffected by any reset other than a power-on reset.

---

# 5.2 Registers Accessible in Privileged Mode Only

The registers described in this subsection are visible only to software running in privileged mode; that is, when PSTATE.priv = 1.

## 5.2.1 Non-Register-Window PR State Registers

Registers described in this section are visible only to software running privileged mode; that is, when PSTATE.priv = 1, and are accessible with RDPR and WRPR instructions.

### 5.2.1.1 Processor State (PSTATE) Register

The Processor State register (PSTATE) shown in FIGURE 5-18 holds the current state of the virtual processor. There is only one instance of the PSTATE register. See Chapter 7, *Traps*, for more details.

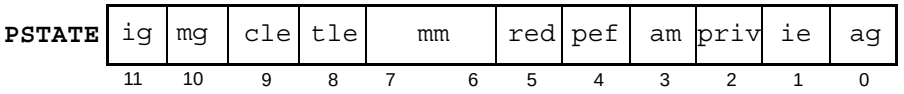



**FIGURE 5-18** PSTATE Fields

Writing PSTATE is nondelayed; that is, new machine state written to PSTATE is visible to the next instruction executed. The privileged RDPR and WRPR instructions are used to read and write PSTATE, respectively.

Subsections on pages 67 through 71 describe the fields contained in the PSTATE register.

#### *Global Register Sets*

The SPARC JPS2 virtual processor provides Interrupt and MMU Global Register sets in addition to the two global register sets specified by SPARC V9. The currently active set of global registers is specified by the ag, ig, and mg bits according to TABLE 5-12.

---

**V9 Compatibility Note –** The ig and mg fields are saved on the trap stack along with the 10 bits of the PSTATE register that are defined in SPARC V9.

---

**TABLE 5-12** PSTATE Global Register Selection Encoding

| ag | ig | mg | Globals selected for use | Automatically Set by ‡ |
|---|---|---|---|---|
| 0 | 0 | 0 | Normal Global registers | |
| 0 | 0 | 1 | MMU Global registers (mg) | traps caused by$^{\nabla}$: *fast_instruction_access_MMU_miss, fast_data_access_MMU_miss, fast_data_access_protection, data_access_exception, instruction_access_exception* |
| 0 | 1 | 0 | Interrupt Global registers (ig) | *interrupt_vector* trap$^{\nabla}$ |
| 0 | 1 | 1 | *Reserved*$^{\dagger}$ | |
| 1 | 0 | 0 | Alternate Global registers (ag) | Any trap other than those listed above$^{\nabla}$ |
| 1 | 0 | 1 | *Reserved*$^{\dagger}$ | |

**TABLE 5-12** PSTATE Global Register Selection Encoding *(Continued)*

| ag | ig | mg | Globals selected for use | Automatically Set by ‡ |
|---|---|---|---|---|
| 1 | 1 | 0 | *Reserved*† | |
| 1 | 1 | 1 | *Reserved*† | |

† An attempt to write this reserved combination of ag, ig, and mg bit values using a WRPR instruction causes an *illegal_instruction* trap. An attempt to write it by other means causes implementation-dependent results. (impl. dep #308)

‡ Since PSTATE is preserved in the TSTATE register when a trap occurs, the previous value of these bits are normally restored upon return from a trap (via DONE or RETRY instruction)

▽ This global register selection may also be set in certain implementation-dependent circumstances. See impl. dep. #307.

When an *interrupt_vector* trap (trap type = $60_{16}$) is taken, a SPARC JPS2 virtual processor selects the Interrupt Global Registers by setting ig and clearing ag and mg. When a *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, *fast_data_access_protection*, *data_access_exception*, or *instruction_access_exception* trap is taken, the virtual processor selects the MMU Global Registers by setting mg and clearing ag and ig. When any other type of trap occurs, the virtual processor selects the Alternate Global Registers by setting ag and clearing ig and mg.

Executing a DONE or RETRY instruction restores the previous {ag, ig, mg} state before the trap is taken. Programmers can also set or clear these three bits by writing to the PSTATE register with a WRPR instruction.

An attempt to write a reserved combination of ag, ig, and mg bit values to the PSTATE register using a WRPR instruction causes an *illegal_instruction* exception.

However, the virtual processor does not check for a reserved encoding in TSTATE. Hence, executing a DONE or RETRY may result in undefined behavior in this case.

**IMPL. DEP. #308:** The result of an attempt to write a reserved combination of ag, ig, and mg bit values to the PSTATE register by means other than a WRPR instruction (for example, copying them from TSTATE during a DONE or RETRY instruction) is implementation dependent.

### *PSTATE_interrupt_globals (ig)*

When PSTATE.ig = 1, the virtual processor interprets integer register numbers in the range 0–7 as referring to the interrupt global register set. See above. When an *interrupt_vector* trap (trap type = $60_{16}$) is taken, a SPARC JPS2 virtual processor sets ig to 1 and sets ag and mg to 0.

---

**V9 Compatibility Note –** PSTATE.ig is an extension to the SPARC V9 PSTATE register.

---

### *PSTATE_MMU_globals (mg)*

When PSTATE.mg = 1, the virtual processor interprets integer register numbers in the range 0–7 as referring to the MMU global register set. This bit must not be set to 1 if either ag or ig is also set to 0. See the first footnote in TABLE 5-12 above.

A SPARC JPS2 virtual processor sets mg to 1 and sets ig and ag to 0 when any of the following traps are taken:

- *fast_instruction_access_MMU_miss* trap (trap type = $64_{16}$)
- *fast_data_access_MMU_miss* trap (trap type = $68_{16}$)
- *fast_data_access_protection* trap (trap type = $6C_{16}$)
- *data_access_exception* trap (trap type = $30_{16}$)
- *instruction_access_exception* trap (trap type = $08_{16}$)

---

**V9 Compatibility Note –** PSTATE.mg is an extension to the SPARC V9 PSTATE register.

---

### *PSTATE_current_little_endian (cle)*

When PSTATE.cle = 1, all data accesses using an implicit ASI are performed in little-endian byte order. When PSTATE.cle = 0, all data accesses using an implicit ASI are performed in big-endian byte order. Instruction accesses are always performed in big-endian byte order. Specific ASIs are shown in TABLE 6-2 on page 110.

### *PSTATE_trap_little_endian (tle)*

When a trap is taken, the current PSTATE register is pushed onto the trap stack and the PSTATE.tle bit is copied into PSTATE.cle in the new PSTATE register. This behavior allows system software to have a different implicit byte ordering than the current process. Thus, if PSTATE.tle is set to 1, data accesses using an implicit ASI in the trap handler are little-endian. The original state of PSTATE.cle is restored when the original PSTATE register is restored from the trap stack.

### *PSTATE_mem_model (mm)*

This 2-bit field determines the memory model in use by the virtual processor. The defined values in the SPARC JPS2 virtual processor are shown in TABLE 5-13.

**TABLE 5-13** MM Encodings

| MM Value | SPARC V9 |
|---|---|
| 00 | Total Store Order (TSO) |
| 01 | Partial Store Order (PSO) |
| 10 | Relaxed Memory Order (RMO) |
| 11 | Reserved |

The current memory model is determined by the value of PSTATE.mm. Software should always refrain from using the combination 11 because it is reserved for future SPARC V9 extensions.

- **Total Store Order (TSO)** — Loads are ordered with respect to earlier loads. Stores are ordered with respect to earlier loads and stores. Thus, loads can bypass earlier stores but cannot bypass earlier loads; stores cannot bypass earlier loads and stores. Programs that execute correctly in either PSO or RMO will execute correctly in the TSO model.
- **Partial Store Order (PSO)** — Loads and stores ordered with respect to earlier loads; atomic load-stores are ordered with respect to loads. Explicit MEMBAR instructions are required to order store and atomic load-store instructions with respect to each other.
- **Relaxed Memory Order (RMO)** — Hardware can schedule memory accesses in any order, as long as the program computes the correct result. In other words, RMO places no ordering constraints on memory references beyond those required for virtual processor self-consistency. When ordering is required, it must be provided explicitly in the programs by MEMBAR instructions.

**IMPL. DEP. #113:** Whether the PSO or RMO models are supported by SPARC JPS2 systems is implementation dependent.

**IMPL. DEP. #119:** The effect of writing an unimplemented memory mode designation into PSTATE.mm is implementation dependent.

### *PSTATE_RED_state (red)*

When PSTATE.red is set to 1, the virtual processor is operating in RED (Reset, Error, and Debug) state. See *RED_state* on page 131. The virtual processor sets PSTATE.red when any hardware reset occurs. It also sets PSTATE.red when a trap is taken while TL = (MAXTL – 1). Software can exit RED_state by one of two methods:

1. Execute a DONE or RETRY instruction, which restores the stacked copy of PSTATE and clears PSTATE.red if it was 0 in the stacked copy.
2. Write a 0 to PSTATE.red with a WRPR instruction.

**Programming Note –** Changing PSTATE.red may cause a change in address mapping on some systems. It is recommended that writes of PSTATE.red be placed in the delay slot of a JMPL; the target of this JMPL should be in the new address mapping. The JMPL sets the nPC, which becomes the PC for the instruction that follows the 'WRPR' in its delay slot. The effect of the WRPR instruction becomes immediately visible to software, and the instruction fetch for the JMPL target is controlled by the *new* PSTATE.

### *PSTATE_enable_floating-point (pef)*

When set to 1, the pef bit enables the floating-point unit, which allows privileged software to manage the FPU. For the FPU to be usable, both PSTATE.pef and FPRS.fef must be set. Otherwise, any floating-point instruction (including the future JPS-specific multiply-add and multiply-subtract instructions) that tries to reference the FPU causes an *fp_disabled* trap.

### *PSTATE_address_mask (am)*

When PSTATE.am = 1, the high-order 32 bits of all instruction and data addresses are set to 0 in the following cases:

- Before data addresses are sent out of the virtual processor
- For instruction accesses to caches (both internal and external)
- Before being stored to a general-purpose register for CALL, JMPL, and RDPC instructions (impl. dep. #125; see below)
- Before being stored to TPC[*n*] and TNPC[*n*] when a trap occurs (impl. dep. #125; see specific SPARC JPS2 Extension documents).

When a bypass ASI (ASI_PHYS_*) is used in a load or store instruction, the setting of PSTATE.am is ignored and the full 64-bit address is used. (See ASI $14_{16}$, ASI_PHYS_USE_EC, for an example.)

**IMPL. DEP. #125:** When PSTATE.am = 1, the value of the high-order 32-bits of the PC transmitted to the specified destination register(s) by CALL, JMPL, RDPC, and saved during a trap is implementation dependent.

**IMPL. DEP. #241:** When PSTATE.am = 1 and an exception occurs, the value written to the more-significant 32 bits of the Data Synchronous Fault Address Register (DSFAR) is implementation dependent.

**Programming Note –** The PSTATE.am bit must be set when 32-bit software is executed.

#### *PSTATE_privileged_mode (priv)*

When PSTATE.priv = 1, the virtual processor is in privileged mode.

#### *PSTATE_interrupt_enable (ie)*

When PSTATE.ie = 1, the virtual processor can accept interrupts.

#### *PSTATE_alternate_globals (ag)*

When PSTATE.ag = 1, the virtual processor interprets integer register numbers in the range 0–7 as referring to the alternate global register set. See Note in TABLE 5-12 on page 72. When ig, mg, and ag are all 0, the virtual processor interprets integer register numbers in the range 0–7 as referring to the normal global register set.

PSTATE.ag is set automatically when any trap other than the following occurs:

- *fast_instruction_access_MMU_miss* (tt = $64_{16}$)
- *fast_data_access_MMU_miss* (tt = $68_{16}$)
- *fast_data_access_protection* (tt = $6C_{16}$)
- *data_access_exception* (tt = $30_{16}$)
- *instruction_access_exception* (tt = $08_{16}$)
- *interrupt_vector* (tt = $60_{16}$)

Setting this bit to 1 is mutually exclusive with setting the PSTATE.mg or PSTATE.ig bit; at most, one of them may be set at any given time. A SPARC JPS2 virtual processor sets ig and mg to 0 any time it automatically sets ag to 1.

### 5.2.1.2 Trap Level Register (TL)

The Trap Level register (TL; FIGURE 5-19) specifies the current trap level. TL = 0 is the normal (nontrap) level of operation. TL > 0 implies that one or more traps are being processed. The maximum valid value that the TL register may contain is MAXTL, which is always equal to the number of supported trap levels beyond level 0; MAXTL must be ≥ 4. See Chapter 7, *Traps*, for more details about the TL register.

SPARC JPS2 supports five trap levels beyond level 0; that is, MAXTL = 5 in a SPARC JPS2 virtual processor (impl. dep. #101).

After a power-on rest (POR), TL is set to MAXTL (5 in SPARC JPS2).

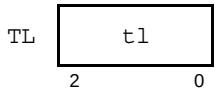



**FIGURE 5-19** Trap Level Register

Writing the TL register with a value greater than MAXTL (five for SPARC JPS2) causes the value MAXTL to be written to TL.

Writing the TL register with a WRPR instruction does not alter any other machine state; that is, it is not equivalent to taking or returning from a trap.

### 5.2.1.3 Processor Interrupt Level (PIL) Register

The Processor Interrupt Level register (PIL; see FIGURE 5-20) specifies the interrupt level above which the virtual processor will accept software interrupts which are controlled by the SOFTINT register. See *SOFTINT Register (ASR 22)* on page 89 for the description of software interrupts. Traps other than software interrupts are not affected by PIL.

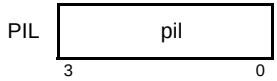



**FIGURE 5-20** Processor Interrupt Level Register

---

**V9 Compatibility Note –** On SPARC V8 processors, the level 15 interrupt is considered to be nonmaskable, so it has different semantics from other interrupt levels. SPARC V9 processors do not treat a level 15 interrupt differently from other interrupt levels. See *Software-Initiated Reset (SIR) Traps* on page 158.

---

### 5.2.1.4 Trap Program Counter (TPC) Register

The Trap Program Counter register (TPC; FIGURE 5-21) contains the program counter (PC) from the previous trap level. There are MAXTL instances of the TPC (five in SPARC JPS2), but only one is accessible at any time. The current value in the TL register determines which instance of the TPC[TL] register is accessible. An attempt to read or write the TPC register when TL = 0 causes an *illegal_instruction* exception.

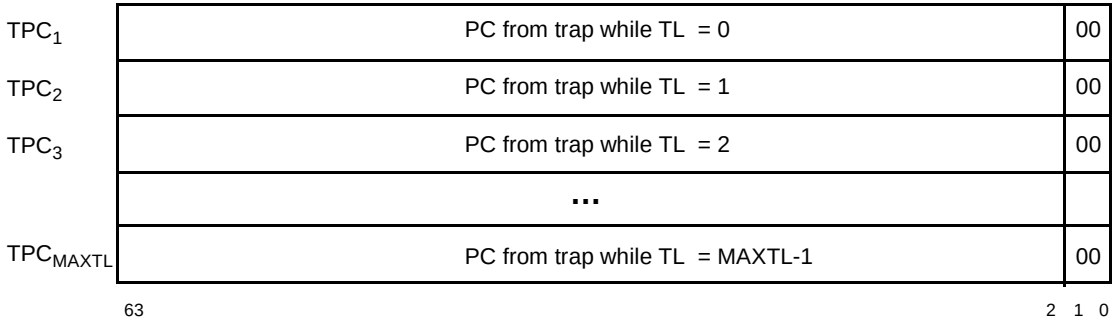



**FIGURE 5-21** Trap Program Counter Register

After a power-on reset the contents of TPC[1] through TPC[MAXTL] are undefined. During normal operation, the value of TPC[*n*], when *n* is greater than the current trap level (*n* > TL), is undefined.

When executing with TL = *n*, the following events affect TPC:

| Event | Effect |
|---|---|
| Trap | TPC[*n* + 1] ← PC |
| RETRY instruction | PC ← TPC[*n*] |
| RDPR (TPC) | r[rd] ← TPC[*n*] |
| WRPR (TPC) | TPC[*n*] ← *value* |
| Power-on reset (*POR*) | All TPC values are left undefined |

### 5.2.1.5 Trap Next Program Counter (TNPC) Register

The Trap Next Program Counter register (TNPC; FIGURE 5-22) is the next program counter (nPC) from the previous trap level. There are MAXTL instances (five in SPARC JPS2) of the TNPC, but only one is accessible at any time. The current value in the TL register determines which instance of the TNPC register is accessible. An attempt to read or write the TNPC register when TL = 0 causes an *illegal_instruction* exception.

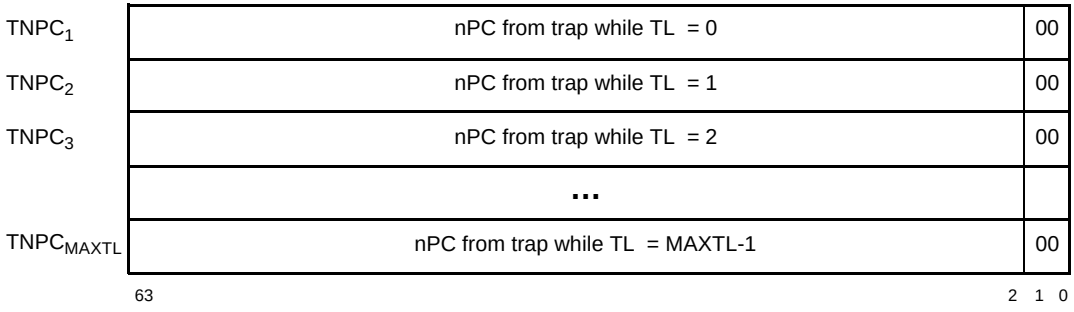



**FIGURE 5-22** Trap Next Program Counter Register

After a power-on reset, the contents of TNPC[1] through TNPC[MAXTL] are undefined. During normal operation, the value of TNPC[*n*], when *n* is greater than the current trap level (*n* > TL), is undefined.

When executing with TL = *n*, the following events affect TNPC:

| Event | Effect |
|---|---|
| Trap | TNPC[*n* + 1] ← NPC |
| RETRY instruction | NPC ← TNPC[*n*] |
| RDPR (TNPC) | r[rd] ← TNPC[*n*] |
| WRPR (TNPC) | TNPC[*n*] ← *value* |
| DONE instruction | PC ← TNPC[*n*]; NPC ← TNPC[*n*] + 4 |
| Power-on reset (*POR*) | All TNPC values are left undefined |

### 5.2.1.6 Trap State (TSTATE) Register

The Trap State register (TSTATE; FIGURE 5-23) contains the state from the previous trap level, comprising the contents of the CCR, ASI, CWP, and PSTATE registers from the previous trap level. There are MAXTL instances (five in SPARC JPS2) of the TSTATE register, but only one is accessible at a time. The current value in the TL register determines which instance of TSTATE is accessible. An attempt to read or write the TSTATE register when TL = 0 causes an *illegal_instruction* exception.

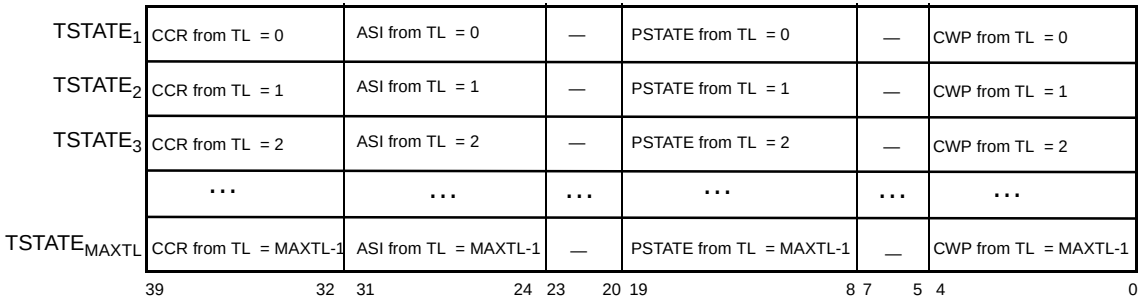



**FIGURE 5-23** Trap State Register

After a power-on reset the contents of TSTATE[1] through TSTATE[MAXTL] are undefined. During normal operation the value of TSTATE[*n*], when *n* is greater than the current trap level (*n* > TL), is undefined.

Because of the addition of the ig and mg bits in the PSTATE register in SPARC JPS2, a 12-bit PSTATE value is stored in TSTATE instead of the 10-bit value specified in SPARC V9.

When executing with TL = *n*, the following events affect TSTATE:

| Event | Effect |
|---|---|
| Trap | TSTATE[*n* + 1] ← (registers) |
| RETRY instruction | (registers) ← TSTATE[*n*] |
| RDPR (TSTATE) | r[rd] ← TSTATE[*n*] |
| WRPR (TSTATE) | TSTATE[*n*] ← *value* |
| DONE instruction | (registers) ← TSTATE[*n*] |
| Power-on reset (*POR*) | All TSTATE values are left undefined |

### 5.2.1.7 Trap Type (TT) Registers

The Trap Type register (TT; FIGURE 5-24) normally contains the trap type of the trap that caused entry to the current trap level. On a reset trap, the TT register contains the trap type of the reset (see TABLE 7-1 on page 132). There are MAXTL (5 in SPARC JPS2) instances of the TT register, but only one is accessible at a time. The current value in the TL register determines which instance of the TT register is accessible. An attempt to read or write the TT register when TL = 0 causes an *illegal_instruction* exception.

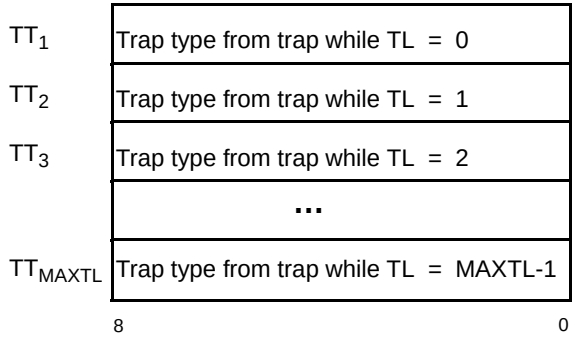



**FIGURE 5-24** Trap Type Register

After a power-on reset the contents of TT[1] through TT[MAXTL-1] are undefined and TT [MAXTL] = $001_{16}$. During normal operation the value of TT[*n*], when *n* is greater than the current trap level (*n* > TL) is undefined.

When executing with TL = *n*, the following events affect TT:

| Event | Effect |
|---|---|
| Trap | TT[*n* + 1] ← (trap type) |
| RDPR (TT) | r[rd] ← TT[*n*] |
| WRPR (TT) | TT[*n*] ← *value* |
| Power-on reset (*POR*) | All TT values are left undefined |

### 5.2.1.8 Trap Base Address (TBA) Register

The Trap Base Address register (TBA) shown in FIGURE 5-25 provides the upper 49 bits of the address used to select the trap vector for a trap. The lower 15 bits of the TBA always read as zero, and writes to them are ignored.

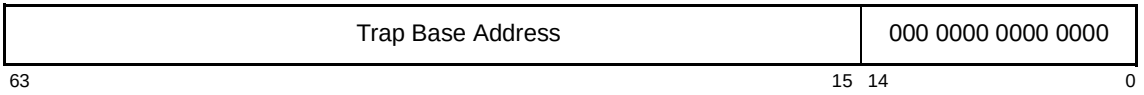



**FIGURE 5-25** Trap Base Address Register

The full address for a trap vector is specified by the TBA, TL, TT[tl], and five zeroes, as shown in FIGURE 5-26.

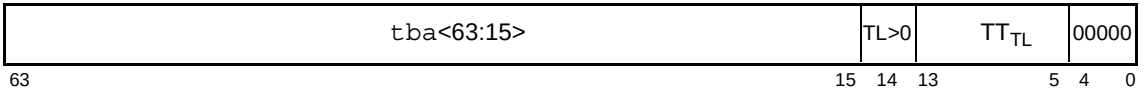



**FIGURE 5-26** Trap Vector Address Format

The "TL>0" bit is 0 if TL = 0 when the trap was taken, and 1 if TL > 0 when the trap was taken. This implies that there are two trap tables: one for traps from TL = 0 and one for traps from TL > 0. See Chapter 7, *Traps*, for more details on trap vectors.

### 5.2.1.9 Version (VER) Register

The Version register, shown in FIGURE 5-27, specifies the fixed parameters pertaining to a particular processor implementation and mask set. The VER register is read-only, readable by the RDPR instruction.

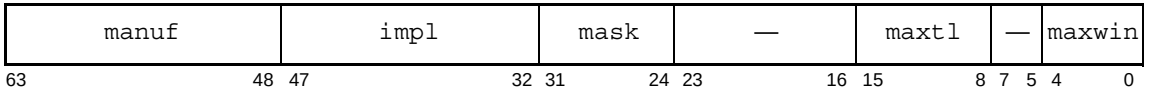



**FIGURE 5-27** Version Register

**IMPL. DEP. #104:** VER.manuf contains a 16-bit manufacturer code. This field is optional and, if not present, shall read as 0. VER.manuf may indicate the original supplier of a second-sourced processor. It is intended that the contents of VER.manuf track the JEDEC semiconductor manufacturer code as closely as possible. If the manufacturer does not have a JEDEC semiconductor manufacturer code, SPARC International will assign a value for VER.manuf.

**IMPL. DEP. #13:** VER.impl uniquely identifies an implementation or class of software-compatible implementations of the architecture. Values $FFF0_{16}$–$FFFF_{16}$ are reserved and are not available for assignment.

The value of VER.impl is assigned as described in the SPARC JPS2 supplements.

VER.mask specifies the current mask set revision and is chosen by the implementor. It generally increases numerically with successive releases of the processor but does not necessarily increase by one for consecutive releases.

VER.maxtl contains the maximum number of trap levels supported, that is, MAXTL, the maximum value of the contents of the TL register.

VER.maxwin contains the maximum index number available for use as a valid CWP value in an implementation; that is, VER.maxwin contains the value NWINDOWS – 1.

For a SPARC JPS2 processor, MAXTL = 5 and MAXWIN = NWINDOWS – 1 = 7; therefore, VER.maxtl = 5 and VER.maxwin = 7.

## 5.2.2 Register Window PR State Registers

The state of the register windows is determined by a set of privileged registers. They can be read/written by privileged software using the RDPR/WRPR instructions. In addition, these registers are modified by instructions related to register windows and are used to generate traps that allow supervisor software to spill, fill, and clean register windows.

---

**V9 Compatibility Note –** Privileged registers CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN contain values in the range 0 to NWINDOWS – 1. The effect of writing a value greater than NWINDOWS – 1 to any of these registers is undefined. Although the width of each of these five registers is nominally 5 bits, the width is implementation dependent and shall be between $\lceil \log_2(\text{NWINDOWS}) \rceil$ and 5 bits, inclusive. If fewer than 5 bits are implemented, the unimplemented upper bits shall read as 0 and writes to them shall have no effect. All five registers should have the same width.

---

Because NWINDOWS = 8 in SPARC JPS2, only the lower 3 bits are implemented in the CWP, CANSAVE, CANRESTORE, CLEANWIN, and OTHERWIN registers (impl. dep. #126). When any of these registers are moved into a 64-bit integer register with an RDPR instruction, the most significant 61 bits are set to 0. When any are written with a WRPR instruction, the most significant 61 bits are ignored.

For details of how the window-management registers are used by hardware, see *Register Window Management Instructions* on page 118.

---

**Programming Note –** CANSAVE, CANRESTORE, OTHERWIN and CLEANWIN must never be set to 7 on a JPS2 virtual processor. Setting any of these to 7 violates the register window state definition in Section 6.4.1. Notice that hardware does not enforce this restriction; it is up to system software to keep the window state consistent.

---

**V9 Compatibility Note –** Since NWINDOWS = 8 on a SPARC JPS2 virtual processor, bits 4:3 of the CWP, CANSAVE, CANRESTORE, OTHERWIN, CLEANWIN registers are unused and always read as 0.

---

### 5.2.2.1 Current Window Pointer (CWP) Register

The CWP register, shown in FIGURE 5-28, is a counter that identifies the current window into the set of integer registers. See *Register Window Management Instructions* on page 118 and Chapter 7, *Traps*, for information on how hardware manipulates the CWP register.

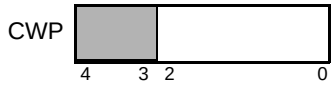



**FIGURE 5-28** Current Window Pointer Register

---

**V9 Compatibility Note –** The following differences between SPARC V8 and SPARC V9 are visible only to privileged software; they are invisible to nonprivileged software.

**1.** In SPARC V9, SAVE increments CWP and RESTORE decrements CWP. In SPARC V8, the opposite is true: SAVE decrements PSR.CWP and RESTORE increments PSR.CWP.

**2.** PSR.CWP in SPARC V8 is changed on each trap. In SPARC V9, CWP is not affected by a trap other than a window fill, spill, or *clean_window* trap.

---

### 5.2.2.2 Savable Windows (CANSAVE) Register

The CANSAVE register, shown in FIGURE 5-29, contains the number of register windows following CWP that are not in use and are, hence, available to be allocated by a SAVE instruction without generating a window spill exception.

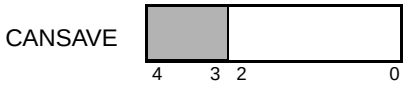



**FIGURE 5-29** CANSAVE Register

### 5.2.2.3 Restorable Windows (CANRESTORE) Register

The CANRESTORE register, shown in FIGURE 5-30, contains the number of register windows preceding CWP that are in use by the current program and can be restored (by the RESTORE instruction) without generating a window fill exception.

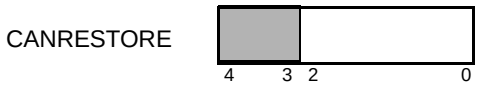



**FIGURE 5-30** CANRESTORE Register

### 5.2.2.4 Other Windows (OTHERWIN) Register

The OTHERWIN register, shown in FIGURE 5-31, contains the count of register windows that will be spilled/filled by a separate set of trap vectors based on the contents of WSTATE.other. If OTHERWIN is zero, register windows are spilled/filled by use of trap vectors based on the contents of WSTATE.normal.

The OTHERWIN register can be used to split the register windows among different address spaces and handle spill/fill traps efficiently by use of separate spill/fill vectors.

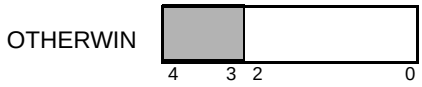



**FIGURE 5-31** OTHERWIN Register

### 5.2.2.5 Window State (WSTATE) Register

The WSTATE register, shown in FIGURE 5-32, specifies bits that are inserted into $TT_{TL}$<4:2> on traps caused by window spill and fill exceptions. These bits are used to select one of eight different window spill and fill handlers. If OTHERWIN = 0 at the time a trap is taken because of a window spill or window fill exception, then the

WSTATE.normal bits are inserted into TT[tl]. Otherwise, the WSTATE.other bits are inserted into TT[tl]. See *Register Window Management Instructions* on page 118, for details of the semantics of OTHERWIN.

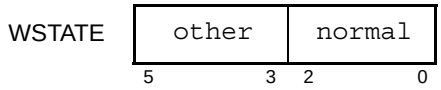



**FIGURE 5-32** WSTATE Register

#### 5.2.2.6 Clean Windows (CLEANWIN) Register

The CLEANWIN register, shown in FIGURE 5-33, contains the number of windows that can be used by the SAVE instruction without causing a *clean_window* exception.

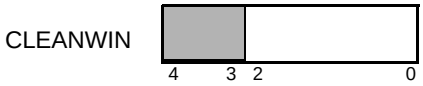



**FIGURE 5-33** CLEANWIN Register

The CLEANWIN register counts the number of register windows that are "clean" with respect to the current program; that is, register windows that contain only zeroes, valid addresses, or valid data from that program. Registers in these windows need not be cleaned before they can be used. The count includes the register windows that can be restored (the value in the CANRESTORE register) and the register windows following CWP that can be used without cleaning. When a clean window is requested (by a SAVE instruction) and none is available, a *clean_window* exception occurs to cause the next window to be cleaned.

### 5.2.3 State Registers

Registers described in this section are the privileged registers accessible with Read State Register and Write State Register instructions.

#### 5.2.3.1 Dispatch Control Register (DCR) (ASR 18)

The DCR provides control over the dispatch unit and branch prediction logic. The DCR is a read / write register. Unused bits read as 0; unused bits should be written only with zero or values previously read from them. DCR is a privileged register; attempted access by nonprivileged (user) code causes a *privileged_opcode* trap.

The Dispatch Control Register is illustrated in FIGURE 5-34 and described in TABLE 5-14.

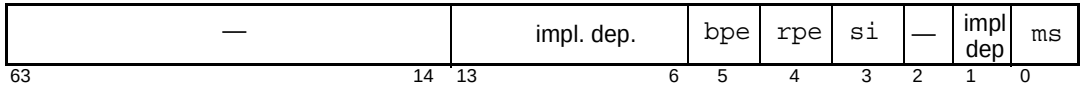



**FIGURE 5-34** Dispatch Control Register (ASR 18)

**IMPL. DEP. #204:** The existence, values, and semantics of DCR bits 5:3 and 0 are implementation dependent. If each is implemented, standard (recommended) semantics are as described below. If not implemented, each bit reads as 0 and writes to it are ignored.

**TABLE 5-14** DCR Bit Description

| Bit | Field | Description |
|---|---|---|
| 63:32 | — | Reserved. |
| 31:6 | — | **IMPL. DEP. #203:** The values and semantics of bits 13:6 and 1 of DCR are implementation dependent. |
| Branch and Return Control | | |
| 5 | bpe | Branch Prediction Enable. When bpe = 1, conditional branches are predicted through internal hardware. When bpe = 0, all branches are predicted not taken. After power-on reset initialization, this bit is set to 0. This bit is also automatically set to 0 on any trap causing RED_state entry (but not zeroed when privileged code enters RED_state by setting the red bit in PSTATE). |
| 4 | rpe | Return Address Prediction Enable. When rpe = 0, the return address prediction stack is disabled. Even when encountering a JMPL instruction, instruction fetch will continue on a sequential path until the return address is generated and a mispredict is signalled. When rpe = 1, the virtual processor may attempt to predict the target address of JMPL instructions and prefetch subsequent instructions accordingly.<br>After power-on reset initialization, this bit is set to 0. This bit is also automatically set to 0 on any trap causing a RED_state entry (but left unchanged when privileged code enters RED_state by setting PSTATE.red). |
| Instruction Dispatch Control | | |
| 3 | si | Single Issue Debug Mode Disable. When si = 0, only one instruction will be outstanding at a time. Superscalar instruction dispatch is disabled, and only one instruction is executed at a time. When si = 1, normal pipelining is enabled. The virtual processor can issue new instructions prior to the completion of previously issued instructions.<br>After power-on reset initialization, this bit is set to 0. This bit is also automatically zeroed on any trap causing RED_state entry (but left unchanged when privileged code enters RED_state by setting PSTATE.red). |

**TABLE 5-14** DCR Bit Description *(Continued)*

| Bit | Field | Description |
|---|---|---|
| 2 | — | Reserved. |
| 1 | — | **IMPL. DEP. #203:** The values and semantics of bits 13:6 and 1 of DCR are implementation dependent. |
| 0 | ms | Multiscalar Dispatch Debug Mode Enable. When ms = 0, the virtual processor operates in scalar mode, issuing and executing one instruction at a time. Pipelined operation is still controlled by the si bit. ms = 1 enables superscalar (normal) instruction issue.<br>After power-on reset initialization, this bit is set to 0. The bit is also zeroed automatically on any trap causing RED_state entry (but left unchanged when privileged code enters RED_state by setting PSTATE.red). |

### 5.2.3.2 SET_SOFTINT (Set Bit(s) in SOFTINT Register) (ASR 20)

A Write State Register instruction (WRSOFTINT_SET) to ASR 20 sets selected bits in the SOFTINT Register (ASR 22) (see page 89). That is, bits 16:0 of the write data are ORed into SOFTINT; any '1' bit in the write data causes the corresponding bit of SOFTINT to be set to 1. Bits 63:17 of the write data are ignored.

Access to ASR 20 is privileged, write-only.

**Programming Note –** There is no actual "register" (machine state) corresponding to ASR 20; it is just a programming interface to conveniently set selected bits to '1' in the SOFTINT register, ASR 22.

FIGURE 5-35 illustrates the SET_SOFTINT Register.

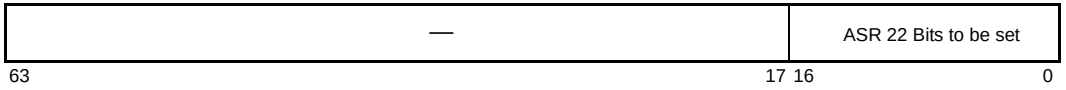



**FIGURE 5-35** SET_SOFTINT Pseudo-Register (ASR 20)

### 5.2.3.3 CLEAR_SOFTINT (Clear Bit(s) in SOFTINT Register) (ASR 21)

A Write State Register instruction (WRSOFTINT_CLR) to ASR 21 clears selected bits in the SOFTINT register (ASR 22) (see page 89). That is, bits 16:0 of the write data are inverted and ANDed into SOFTINT; any '1' bit in the write data causes the corresponding bit of SOFTINT to be set to 0. Bits 63:17 of the write data are ignored.

Access to ASR 21 is privileged, write-only.

**Programming Note –** There is no actual “register” (machine state) corresponding to ASR 21; it is just a programming interface to conveniently set selected bits to ‘0’ in the SOFTINT register, ASR 22.

FIGURE 5-36 illustrates the CLEAR_SOFTINT Register.

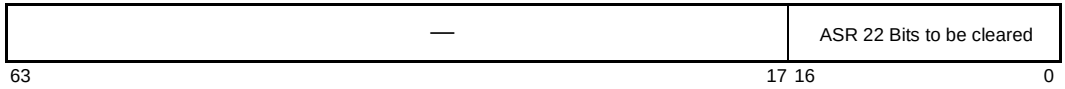



**FIGURE 5-36** CLEAR_SOFTINT Pseudo-Register (ASR 21)

### 5.2.3.4 SOFTINT Register (ASR 22)

Privileged software uses this privileged, read / write register to schedule interrupts. SOFTINT can be read with a RDSOFTINT instruction (Read State Register 22) and written with a WRSOFTINT instruction (Write State Register 22).

The SOFTINT Register is illustrated in FIGURE 5-37 and described in TABLE 5-15.

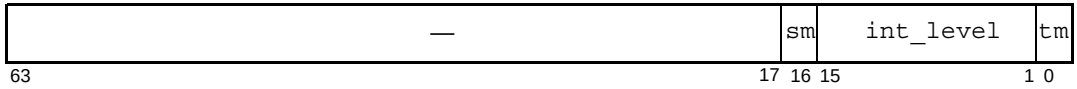



**FIGURE 5-37** SOFTINT Register (ASR 22)

**TABLE 5-15** SOFTINT Bit Description

| Bit | Field | Description |
|---|---|---|
| 16 | sm (stick_int) | When the STICK_COMPARE (ASR 25) register’s int_dis (interrupt disable) field is 0 (that is, system tick compare is *enabled*) and its stick_cmpr field matches the value in the STICK register, then the stick_int field in ASR 22 is set to 1 and a level 14 interrupt is generated. See *System Tick (STICK) Register (ASR 24)* on page 70 for details. |
| 15:1 | int_level | When a bit is set within this field (bits 15:1), an interrupt is caused at the corresponding interrupt level.<br>Note: Level-14 interrupt is triggered by either int_level<14>, stick_int or tick_int.<br>Note: int_level<15> (which triggers a level-15 interrupt (*interrupt_level_15*)) can be set either by a write to the SOFTINT register or by a PIC overflow. |

**TABLE 5-15** SOFTINT Bit Description *(Continued)*

| Bit | Field | Description |
|---|---|---|
| 0 | tm (tick_int) | When the TICK_COMPARE (ASR 23) register's int_dis (interrupt disable) field is 0 (that is, tick compare is *enabled*) and its tick_cmpr field matches the value in the TICK register, then the tick_int field in ASR 22 is set to 1 and a level-14 interrupt is generated. See *Tick Compare (TICK_COMPARE) Register (ASR 23))* for details. |

See Section N.6 for additional information regarding the SOFTINT register.

### 5.2.3.5 Tick Compare (TICK_COMPARE) Register (ASR 23)

The TICK_COMPARE register allows system software to cause a trap when the TICK register reaches a specified value. Nonprivileged accesses to this register cause a *privileged_opcode* trap (see *Exception and Interrupt Descriptions* on page 160). After a power-on reset trap, the int_dis bit is set to 1 (disabling tick compare interrupts) and the tick_cmpr value is set to 0.

The TICK_COMPARE Register is described below and illustrated in FIGURE 5-38.

| Bit | Field | Description |
|---|---|---|
| 63 | int_dis | Interrupt Disable. If set, tick compare interrupts are disabled. |
| 62:0 | tick_cmpr | Tick Compare Field. When this field exactly matches TICK.counter and TICK_COMPARE.int_dis = 0, SOFTINT.tick_int is set to 1. This has the effect of posting a level-14 interrupt to the virtual processor when the virtual processor has (PIL < 14) and (PSTATE.ie = 1). The level-14 interrupt handler must check SOFTINT<14>, TICK_INT, and STICK_INT to determine which was the source of the level-14 interrupt. |

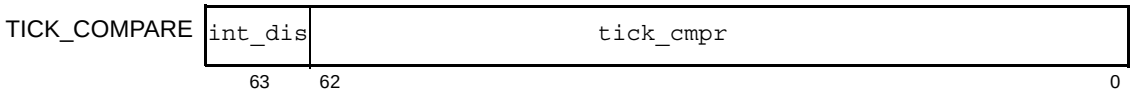



**FIGURE 5-38** TICK_COMPARE Register

### 5.2.3.6 System Tick Compare (STICK_COMPARE) Register (ASR 25)

The STICK_COMPARE register allows system software to cause a trap when the STICK register reaches a specified value. Nonprivileged accesses to this register cause a *privileged_opcode* trap (see *Exception and Interrupt Descriptions* on page 160). After a power-on reset trap, the int_dis bit is set to 1 (disabling system tick compare interrupts), and the STICK_CMPR value is set to 0.

The System Tick Compare Register is defined below and illustrated in FIGURE 5-39.

| Bit | Field | Description |
|---|---|---|
| 63 | int_dis | Interrupt Disable. If set, system tick compare interrupts are disabled. |
| 62:0 | stick_cmpr | System Tick Compare Field. When this field exactly matches STICK.counter and STICK_COMPARE.int_dis = 0, SOFTINT.stick_int is set to 1. This has the effect of posting a level-14 interrupt to the virtual processor when the virtual processor has (PIL < 14) and (PSTATE.ie = 1). The level-14 interrupt handler must check SOFTINT<14>, tick_int, and stick_int to determine which was the source of the level-14 interrupt. |

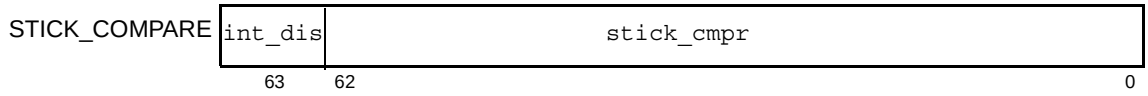



**FIGURE 5-39** STICK_COMPARE Register

## 5.2.4 ASI-Accessible Registers

In this section the Data Cache Unit Control Register, Data Watchpoint registers, Instruction Trap Register, Interrupt ASI registers, scratchpad registers, and MTP registers are described. A complete list of JPS2 ASIs is shown in TABLE L-1 on page 569.

### 5.2.4.1 Data Cache Unit Control Register (DCUCR)

ASI $45_{16}$ (ASI_DCU_CONTROL_REGISTER), VA = $0_{16}$

The Data Cache Unit Control Register contains fields that control several memory-related hardware functions. The functions include instruction, prefetch, write and data caches, MMUs, and watchpoint setting.

After a power-on reset (POR), all fields of DCUCR are set to 0. After a WDR, XIR, or SIR, all fields of DCUCR defined in this section are set to 0. The effect of reset on implementation-dependent fields of DCUCR is implementation dependent (impl. dep. #240).

The Data Cache Unit Control Register is illustrated in FIGURE 5-40 and described in TABLE 5-16. In the table, bits are grouped by function rather than by strict bit sequence.

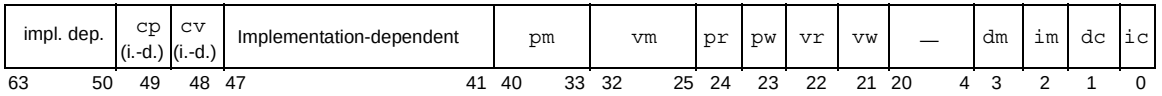



**FIGURE 5-40** DCU Control Register Access Data Format (ASI $45_{16}$)

**TABLE 5-16** DCUCR Description *(1 of 3)*

| Bits | Field | Type | Use — Description |
|---|---|---|---|
| 63:50 | impl. dep. | | **IMPL. DEP. #309:** The presence and semantics of bits 63:50 of DCUCR are implementation dependent. If any of these bits is not implemented, it reads as 0 and writes to it are ignored. |
| 49:48 | cp, cv | RW | **IMPL. DEP. #232:** Whether cp and cv bits are implemented in the DCU Control Register is implementation dependent in JPS2.<br>If cp is implemented, it determines the physical cacheability of memory accesses when the IMMU or DMMU is disabled (im = 0 or DM = 0). 1 = cacheable, 0 = noncacheable.<br>If cv is implemented, it determines the virtual cacheability of memory accesses when the DMMU is disabled (dm = 0); 1 = cacheable, 0 = noncacheable.<br>If DCUCR.cp is implemented and the MMUs are disabled, then each memory access is performed as if it was to a location with a TTE entry containing E = **not** DCUCR.cp. That is, an access when DCUCR.cp = 0 would be performed as if there were side effects and an access when DCUCR.cp = 1 would be performed without regard to side effects.<br>**Note:** The cp and cv bits of DCUCR must be changed with care. It is recommended that a MEMBAR #Sync be executed before and after cp or cv is changed. Also, software must manage cache states to be consistent before and after cp or cv is changed. |
| 47:41 | impl. dep. | | **IMPL. DEP. #240:** The presence and semantics of bits 47:41 of DCUCR are implementation dependent. If any of these bits is not implemented, it reads as 0 and writes to it are ignored. |
| | | | ***Watchpoint Control*** |
| 40:33 | pm**<7:0>** | | DCU Physical Address Data Watchpoint Mask. The Physical Address Data Watchpoint Register contains the physical address of a 64-bit word to be watched. The 8-bit Physical Address Data Watch Point Mask controls which byte(s) within the 64-bit word should be watched. If all 8 bits are cleared, the physical watchpoint is disabled. If the watchpoint is enabled and a data reference overlaps any of the watched bytes in the watchpoint mask, then a physical watchpoint trap is generated. Watchpoint behavior for a Partial Store instruction may differ (see impl. dep. #249).<br>Please see the table in the vm field description. |

**TABLE 5-16** DCUCR Description *(2 of 3)*

| Bits | Field | Type | Use — Description |
|---|---|---|---|
| 32:25 | vm**<7:0>** | | DCU Virtual Address Data Watchpoint Mask. The Virtual Address Data Watchpoint Register contains the virtual address of a 64-bit word to be watched. This 8-bit mask controls which byte(s) within the 64-bit word should be watched. If all 8 bits are cleared, then the virtual watchpoint is disabled. If watchpoint is enabled and a data reference overlaps any of the watched bytes in the watchpoint mask, then a virtual watchpoint trap is generated. Watchpoint behavior for a Partial Store instruction may differ (see impl. dep. #249).<br>VA / PA data watchpoint byte mask examples are shown below. |

| Watchpoint Mask (pm **or** vm) | Least Significant 3 Bits of Address of Bytes Watched 7654 3210 |
|---|---|
| $00_{16}$ | Watchpoint disabled |
| $01_{16}$ | 0000 0001 |
| $32_{16}$ | 0011 0010 |
| $FF_{16}$ | 1111 1111 |

| | | | |
|---|---|---|---|
| 24, 23 | pr**,** pw | | DCU Physical Address Data Watchpoint Enable. If pr (pw) is 1, then a data read (write) that matches the range of addresses in the Physical Watchpoint Register causes a watchpoint trap. If both pr and pw are set, a watchpoint trap will occur on either a read or write access. |
| 22, 21 | vr**,** vw | | DCU Virtual Address Data Watchpoint Enable. If vr (vw) is 1, then a data read (write) that matches the range of addresses in the Virtual Watchpoint Register causes a watchpoint trap. If both vr and vw are set, a watchpoint trap will occur on either a read or write access. |
| 20:4 | — | | *Reserved.* |
| | | | ***MMU Control*** |
| 3 | dm | | DMMU Enable. If dm = 0, the DMMU is disabled (pass-through mode).<br>**Note:** When the MMU / TLB is disabled, a virtual address is passed through as a physical address. |
| 2 | im | | IMMU Enable. If im = 0, the IMMU is disabled (pass-through mode). |

**TABLE 5-16** DCUCR Description *(3 of 3)*

| Bits | Field | Type | Use — Description |
|---|---|---|---|
| | | | ***Cache Control*** |
| 1 | dc | | **IMPL. DEP. #252:** The presence of DCUCR bit 1 (DCUCR.dc, Data Cache Enable) is implementation dependent. If dc is not implemented, it reads as zero, writes to it are ignored, and software should only write zero or a value previously read from dc to dc. The remainder of this description assumes that DC is implemented. The function of dc is to enable / disable operation of the data cache closest to the virtual processor (D-cache); dc = 1 enables the D-cache and dc = 0 disables it. When dc = 0, memory accesses (loads, stores, atomic load-stores) are satisfied by caches lower in the cache hierarchy. It is implementation dependent whether or not memory accesses update the D-cache while the D-cache is disabled (dc = 0). If memory accesses do not update the D-cache, then when the D-cache is reenabled (dc is set to 1) any D-cache lines still marked as "valid" may be inconsistent with the state of memory or other caches. In that case, software must handle any inconsistencies by flushing the inconsistent lines from the D-cache. |
| **0** | ic | | **IMPL. DEP. #253:** The presence of DCUCR bit 0 (DCUCR.ic, Instruction Cache Enable) is implementation dependent. If ic is not implemented, it reads as zero, writes to it are ignored, and software should only write zero or a value previously read from ic to ic. The remainder of this description assumes that IC is implemented. The function of ic is to enable / disable operation of the instruction cache closest to the virtual processor (I-cache); ic = 1 enables the I-cache and ic = 0 disables it. When ic = 0, instruction fetches are satisfied by caches lower in the cache hierarchy. It is implementation dependent whether or not instruction fetches update the I-cache while the I-cache is disabled (ic = 0). If instruction fetches do not update the I-cache, then when the I-cache is reenabled (ic is set to 1) any I-cache lines still marked as "valid" may be inconsistent with the state of memory or other caches. In that case, software must handle any inconsistencies by invalidating the inconsistent lines in the I-cache. |

### 5.2.4.2 Data Watchpoint Registers

SPARC JPS2 processors implement "break before" watchpoint traps. When the address of a data access matches a preset physical or virtual watchpoint address, instruction execution is stopped immediately before the watched memory location is accessed. TABLE 5-17 lists ASIs that are affected by the two watchpoint traps.

**TABLE 5-17** ASIs Affected by Watchpoint Traps

| ASI Type | ASI Range (Unimplemented ASIs Excluded) | Data MMU | Watchpoint If Matching VA | Watchpoint If Matching PA |
|---|---|---|---|---|
| Translating ASIs | $04_{16}$–$11_{16}$, $18_{16}$–$19_{16}$, $24_{16}$–$2C_{16}$, $70_{16}$–$71_{16}$, $78_{16}$–$79_{16}$, $80_{16}$–$EE_{16}$, $F0_{16}$–$FF_{16}$ | on | Y | Y |
| | | off | N | Y |
| Bypass ASIs | $14_{16}$, $15_{16}$, $1C_{16}$, $1D_{16}$, $34_{16}$, $3C_{16}$ | — | N | Y |
| Nontranslating ASIs | $2D_{16}$–$33_{16}$, $35_{16}$–$3B_{16}$, $3D_{16}$–$6F_{16}$, $72_{16}$–$77_{16}$, $7A_{16}$–$7F_{16}$, $EF_{16}$ | — | N | N |

For 128-bit (quad) atomic load, LDQF, and 64-byte block load and store instructions, a watchpoint trap is generated only if the watchpoint overlaps the lowest-address eight bytes of the access.

---

**Programming Note –** To avoid trapping infinitely, software should emulate the instruction that caused the trap and return from the trap by using a DONE instruction or turn off the watchpoint before returning from a watchpoint trap handler.

---

**IMPL. DEP. #244:** Implementation-dependent feature(s) may be present that degrade the reliability of data watchpoints. If such features are present, it must be possible to disable them such that data watchpoints function as described in this section. Furthermore, those features should be disabled by default.

Two 64-bit data watchpoint registers provide the means to monitor data accesses during program execution. When Virtual/Physical Data Watchpoint is enabled, the virtual/physical addresses of all data references are compared against the content of the corresponding watchpoint register. If a match occurs, a *VA_watchpoint* or *PA_watchpoint* trap is taken before the data reference instruction is completed. The virtual address watchpoint trap has higher priority than the physical address watchpoint trap.

Separate 8-bit byte masks allow watchpoints to be set for a range of addresses. Each zero bit in the byte mask causes the comparison to ignore the corresponding byte in the address. These watchpoint byte masks and the watchpoint enable bits reside in the Data Cache Unit Control Register.

### *Virtual Address Data Watchpoint Register*

ASI $58_{16}$, VA = $38_{16}$

Name: VA Data Watchpoint Register

FIGURE 5-41 illustrates the Virtual Address Watchpoint Register,

**db_va** is the most significant 61 bits of the 64-bit virtual data watchpoint address.

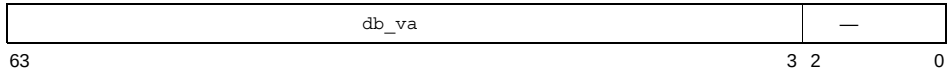



**FIGURE 5-41** VA Data Watchpoint Register Format

### *Physical Address Data Watchpoint Register*

ASI $58_{16}$, VA=$40_{16}$

Name: PA Data Watchpoint Register

FIGURE 5-42 illustrates the PA Data Watchpoint Register.

**db_pa** is the most significant 61 bits of the physical data watchpoint address. The minimum width of a SPARC JPS2 physical address is 43 bits (impl. dep. #224).

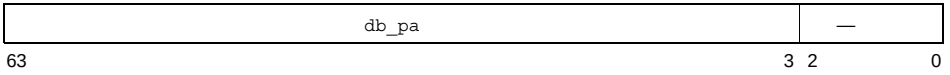



**FIGURE 5-42** PA Data Watchpoint Register Format

---

**Note –** Implementations may provide fewer than 64 bits of physical address space (impl. dep. #224). Therefore, software is responsible for zero-extending any physical address narrower than 64 bits out to a full 64 bits before writing that address into the PA Data Watchpoint Register.

---

### 5.2.4.3 Instruction Trap Register

ASI $60_{16}$ (ASI_IIU_INST_TRAP), VA=$0_{16}$

The Instruction Trap Register can be used to generate a trap whenever an instruction belonging to a specified class of instruction is dispatched.

**IMPL. DEP. #205:** The presence of the Instruction Trap Register in a SPARC JPS2 virtual processor is implementation dependent. If implemented, the standard (recommended) implementation is as described in this section.

When an instruction is dispatched and its opcode bits match the pattern specified in the Instruction Trap Register, then an *illegal_instruction* exception occurs. A range of opcodes can be specified through the use of the mask and match fields of the Instruction Trap Register.

If an instruction breakpoint triggers an *illegal_instruction* trap, the *illegal_instruction* trap will have a higher priority than that of a *privileged_opcode* trap (exceptional, see FIGURE 7-3 on page 145).

The Instruction Trap Register is described below and illustrated in FIGURE 5-43.

| Bits | Field | Type | Description |
|---|---|---|---|
| **63:32** | mask | RW | A “1” entry enables comparison of the corresponding Match bit against the issued instructions. Bit 63 corresponds to match bit 31, bit 32 to match bit 0. This field is initialized to all zeroes on power-on reset. If mask is all zeroes, then the Instruction Trap Register never generates a trap. |
| **31:0** | match | RW | Contains a bit pattern to match against the issued instruction stream. If a match is found, an *illegal_instruction* exception is generated. Specifically:<br>*illegal_instruction* generated when ((*instruction* & mask) = (match & mask)) && (mask!=0) |

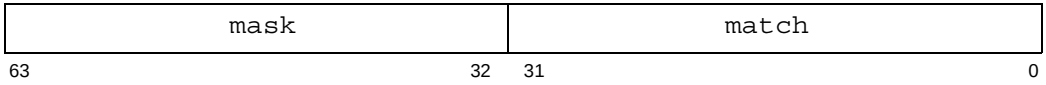



**FIGURE 5-43** Instruction Trap Register

**IMPL. DEP. #245:** On SPARC JPS2 virtual processors, the encoding of the least significant 11 bits of the displacement field of `CALL` and branch (`BPcc`, `FBPfcc`, `Bicc`, `BPr`) instructions in an instruction cache is implementation-dependent. Specifically, those bits' encoding in an instruction cache is not necessarily the same as their architectural encoding (which appears in main memory).

---

**Programming Note –** The 32-bit instruction value matched against the Instruction Trap Register is the instruction word fetched from the instruction cache. However, the encoding of the least significant 11 bits of `CALL` and branch instructions may be different in the instruction cache from the architecturally specified encoding (impl. dep. #245, above). Therefore, software intended to be portable across SPARC JPS2 implementations that write the Instruction Trap Register to cause a trap on `CALL` or branch instructions must set bits 10:0 of the `mask` field to 0 to mask out the implementation-dependent bits from the comparison.

---

A store to the Instruction Trap Register requires `MEMBAR #Sync` plus either `FLUSH`, `DONE`, or `RETRY` before the point that its effect must be visible to instruction accesses. That is, `MEMBAR #Sync` alone is not sufficient. In either case, one of these instructions must be executed before the next store or load of any type using a translating or bypass ASI, to avoid data corruption.

---

**Note –** (1) The Instruction Trap Register generates an exception based on instruction opcodes, not on their addresses (as do traditional breakpoints).

(2) As a historical note: This mechanism was designed to provide a way around hardware errors that may be found in silicon during bringup. For example, if an instruction is failing on a particular mask set, it can be trapped and emulated in software with the Instruction Trap Register mechanism.

---

### 5.2.4.4 Interrupt ASI Registers

See *Interrupt ASI Registers* on page 590 for detailed descriptions of ASI register used in handling interrupts.

### 5.2.4.5 MTP Registers Accessed through ASIs

All Multithreaded Processing (MTP) registers are accessed through ASIs. See *Accessing MTP Registers* on page 189, for descriptions of ASI registers used to control MTP functions.

### 5.2.4.6 Scratchpad Registers (ASI_SCRATCHPAD_*n*_REG)

**IMPL. DEP. #302:** Whether Scratchpad registers are present in a SPARC JPS2 virtual processor is implementation dependent.

Implementation of scratchpad registers is recommended for all future SPARC processors, including MTP implementations.

When implemented, each virtual processor includes eight Scratchpad registers (64 bits each, read / write accessible). The addresses of the scratchpad registers are defined in TABLE 5-18. The use of the Scratchpad registers is completely defined by software.

For conventional usage of Scratchpad registers, see *Scratchpad Register Usage* on page 538.

---

**Implementation Note –** It is important that Scratchpad registers be implemented as fast-access ASI registers. They are intended to be used by performance-critical trap handler code. It is not practical to give a precise definition of “fast.” It is assumed that processor teams will use good judgement and work closely with the corresponding software teams to determine the performance requirements and implications for these registers.

---

**TABLE 5-18** Scratchpad Registers

| Register Asi Name | ASI # | VA | Access |
|---|---|---|---|
| ASI_SCRATCHPAD_0_REG | $4F_{16}$ | $00_{16}$ | RD / WR |
| ASI_SCRATCHPAD_1_REG | $4F_{16}$ | $08_{16}$ | RD / WR |
| ASI_SCRATCHPAD_2_REG | $4F_{16}$ | $10_{16}$ | RD / WR |
| ASI_SCRATCHPAD_3_REG | $4F_{16}$ | $18_{16}$ | RD / WR |
| ASI_SCRATCHPAD_4_REG | $4F_{16}$ | $20_{16}$ | RD / WR |
| ASI_SCRATCHPAD_5_REG | $4F_{16}$ | $28_{16}$ | RD / WR |
| ASI_SCRATCHPAD_6_REG | $4F_{16}$ | $30_{16}$ | RD / WR |
| ASI_SCRATCHPAD_7_REG | $4F_{16}$ | $38_{16}$ | RD / WR |

# Instructions

Instructions are fetched by the virtual processor from memory and are executed, annulled, or trapped. Instructions are encoded in 4 major formats and partitioned into 11 general categories. We describe instructions in these sections:


- *Instruction Execution* on page 99
- *Instruction Formats and Fields* on page 100
- *Instruction Categories* on page 104
- *Register Window Management* on page 125


## 6.1 Instruction Execution

The instruction at the memory location specified by the program counter is fetched and then executed. Instruction execution may change program-visible virtual processor and/or memory state. As a side effect of its execution, new values are assigned to the program counter (PC) and the next program counter (nPC).

An instruction may generate an exception if it encounters some condition that makes it impossible to complete normal execution. Such an exception may in turn generate a precise trap. Other events may also cause traps: an exception caused by a previous instruction (a deferred trap), an interrupt or asynchronous error (a disrupting trap), or a reset request (a reset trap). If a trap occurs, control is vectored into a trap table. See Chapter 7, *Traps*, for a detailed description of exception and trap processing.

If a trap does not occur and the instruction is not a control transfer, the next program counter is copied into the PC, and the nPC is incremented by 4 (ignoring overflow, if any). There are two types of control-transfer instructions (CTIs): delayed and immediate. For a delayed CTI, at the end of the execution of an instruction, nPC is copied to into the PC and the target address is copied into nPC. For an immediate CTI, at the end of execution, the target is copied to PC and target+4 is copied to nPC.

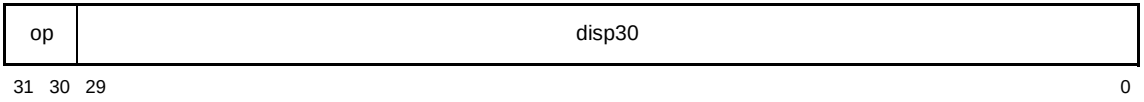
In the SPARC instruction set, many CTIs do not transfer control until after a delay of one instruction, hence the term “delayed CTI” (DCTI). Thus, the two program counters provide for a delayed-branch execution model.

For each instruction access and each normal data access, the IU appends an 8-bit address space identifier, or ASI, to the 64-bit memory address. Load / store alternate instructions (see *Address Space Identifiers (ASIs)* on page 109) can provide an arbitrary ASI with their data addresses or can use the ASI value currently contained in the ASI register.

# 6.2 Instruction Formats and Fields

Instructions are encoded in four major 32-bit formats and several minor formats, as shown in FIGURE 6-1, FIGURE 6-2 on page 101, and FIGURE 6-3 on page 102.

*Format 1 (op = 1):* `CALL`




*Format 2 (op = 0):* `SETHI` *and Branches* (`Bicc`, `BPcc`, `BPr`, `FBfcc`, `FBPfcc`)

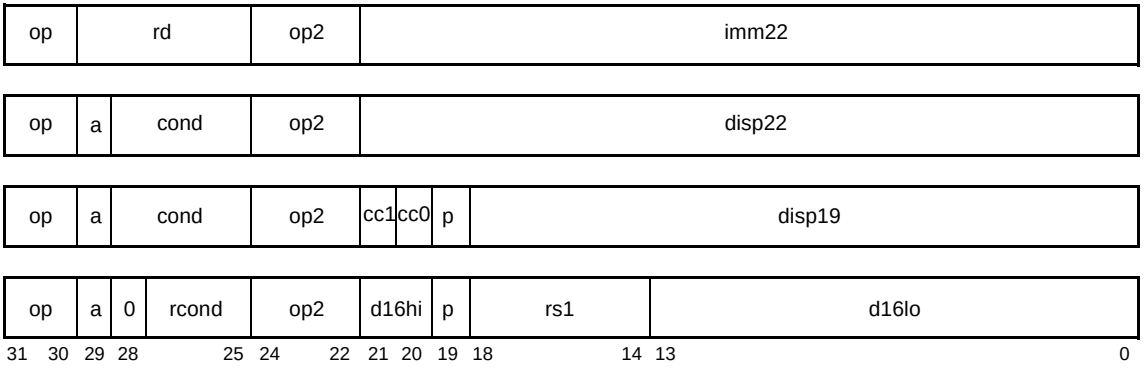



**FIGURE 6-1** Summary of Instruction Formats: Formats 1 and 2

*Format 3 (op = 2 or 3):* Arithmetic, Logical, MOVr, MEMBAR, Prefetch, Load, *and* Store

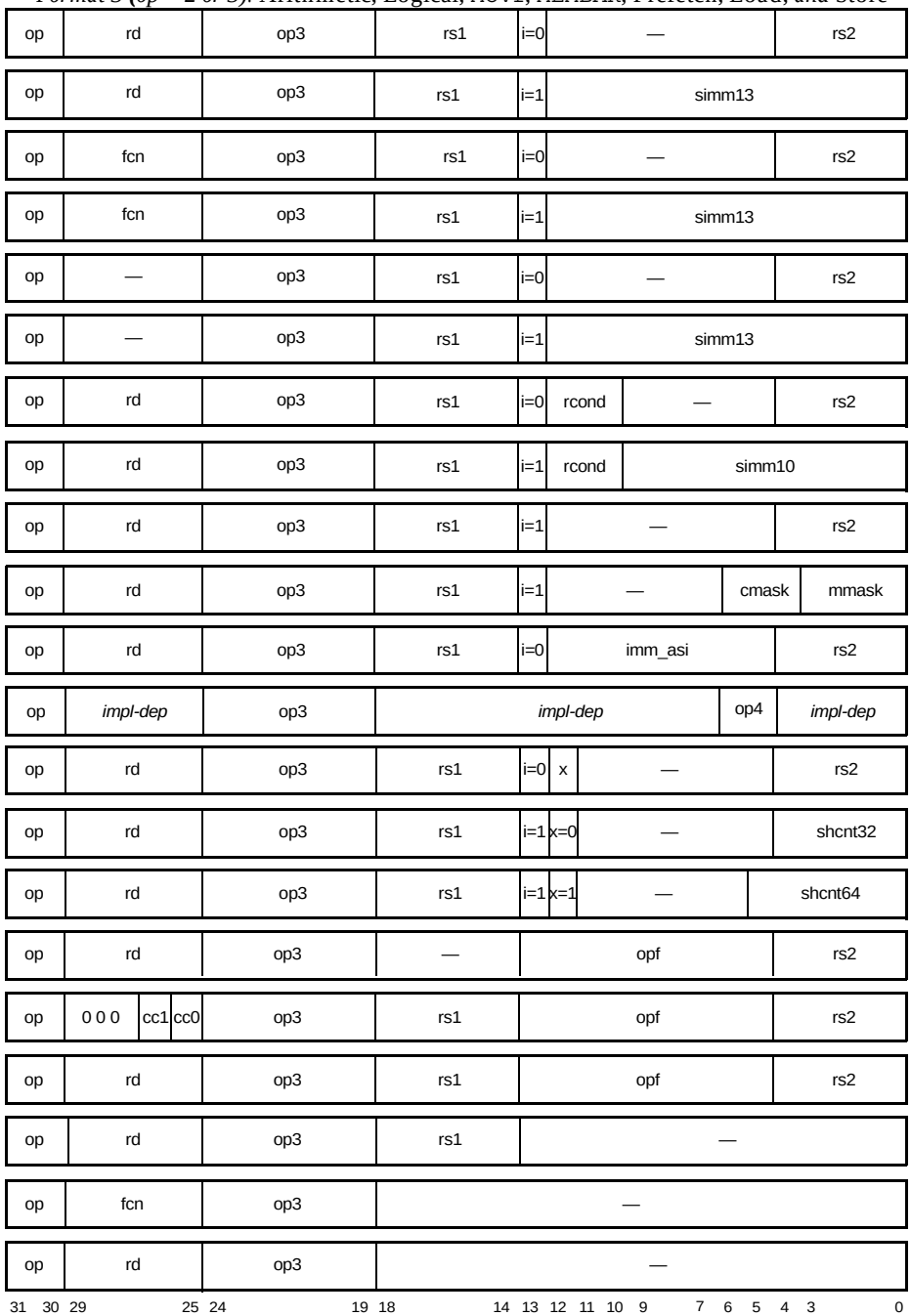



**FIGURE 6-2** Summary of Instruction Formats: Format 3

*Format 4 (op = 2):* MOVcc, FMOVr, FMOVcc***, and*** Tcc

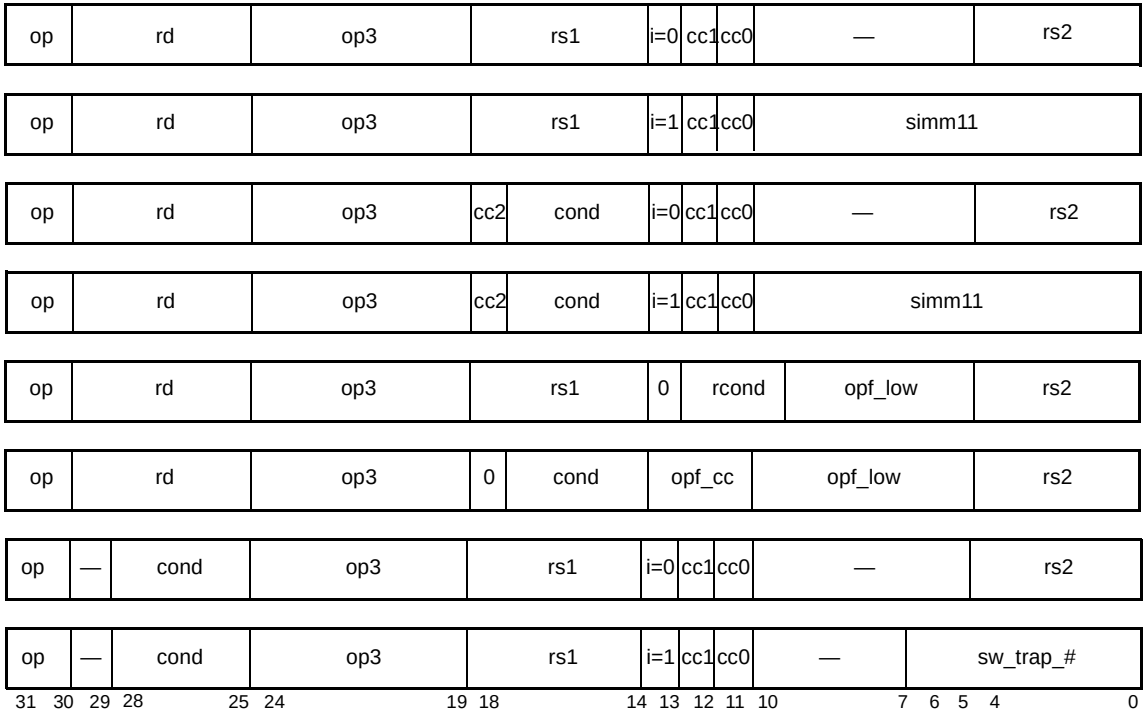



**FIGURE 6-3** Summary of Instruction Formats: Format 4

The instruction fields are interpreted as described in TABLE 6-1.

**TABLE 6-1** Instruction Field Interpretation *(1 of 3)*

| Field | Description |
|---|---|
| a | The a bit annuls the execution of the following instruction if the branch is conditional and not taken, or if it is unconditional. |
| cc2, cc1, cc0 | cc2, cc1, and cc0 specify the condition codes (icc, xcc, fcc0, fcc1, fcc2, fcc3) to be used in the following instructions:<br>• Branch on Floating-Point Condition Codes with Prediction Instructions (FBPfcc)<br>• Branch on Integer Condition Codes with Prediction (BPcc)<br>• Floating-Point Compare Instructions (FCMP and FCMPE)<br>• Move Integer Register If Condition Is Satisfied (MOVcc)<br>• Move Floating-Point Register If Condition Is Satisfied (FMOVcc)<br>• Trap on Integer Condition Codes (Tcc).<br>In instructions such as Tcc that do not contain the cc2 bit, the missing cc2 bit takes on a default value. See TABLE E-10 on page 464 for a description of these fields' values. |

**TABLE 6-1** Instruction Field Interpretation *(2 of 3)*

| Field | Description |
|---|---|
| cmask | This 3-bit field specifies sequencing constraints on the order of memory references and the processing of instructions before and after a MEMBAR instruction. |
| cond | This 4-bit field selects the condition tested by a branch instruction. See *Appendix E, Opcode Maps*, for descriptions of its values. |
| d16hi, d16lo | These 2-bit and 14-bit fields together comprise a word-aligned, sign-extended, PC-relative displacement for a branch-on-register-contents with prediction (BPr) instruction. |
| disp19 | This 19-bit field is a word-aligned, sign-extended, PC-relative displacement for an integer branch-with-prediction (BPcc) instruction or a floating-point branch-with-prediction (FBPfcc) instruction. |
| disp22, disp30 | These 22-bit and 30-bit fields are word-aligned, sign-extended, PC-relative displacements for a branch or call, respectively. |
| fcn | This 5-bit field provides additional opcode bits to encode the DONE, RETRY, and PREFETCH(A) instructions. |
| i | The i bit selects the second operand for integer arithmetic and load/store instructions. If i = 0, then the operand is r[rs2]. If i = 1, then the operand is simm10, simm11, or simm13, depending on the instruction, sign-extended to 64 bits. |
| imm22 | This 22-bit field is a constant that SETHI places in bits 31:10 of a destination register. |
| imm_asi | This 8-bit field is the address space identifier in instructions that access alternate space. |
| impl-dep | The meaning of these fields is completely implementation dependent for IMPDEP2A and IMPDEP2B instructions. |
| mmask | This 4-bit field imposes order constraints on memory references appearing before and after a MEMBAR instruction. |
| op, op2 | These 2- and 3-bit fields encode the three major formats and the Format 2 instructions. See *Appendix E, Opcode Maps*, for descriptions of their values. |
| op3 | This 6-bit field (together with one bit from op) encodes the Format 3 instructions. See *Appendix E, Opcode Maps*, for descriptions of its values. |
| op4 | This 2-bit field encodes part of the opcode for Format 3 implementation-dependent instructions. See *Implementation-Dependent Instructions* on page 272 for descriptions of its values. |
| opf | This 9-bit field encodes the operation for a floating-point operate (FPop) instruction. See *Appendix E, Opcode Maps*, for possible values and their meanings. |
| opf_cc | Selects the condition code set (icc, xcc, or fcc*n*) to be used in FMOVcc instructions. |
| opf_low | This 6-bit field encodes the specific operation for a Move Floating-Point Register if condition is satisfied (FMOVcc) or Move Floating-Point Register if contents of integer register match condition (FMOVr) instruction. |

**TABLE 6-1** Instruction Field Interpretation *(3 of 3)*

| Field | Description |
|---|---|
| p | This 1-bit field encodes static prediction for `BPcc` and `FBPfcc` instructions; branch prediction bit (`p`) encodings are shown below.<br>**p** — **Branch Prediction**<br>0 — Predict that branch will not be taken<br>1 — Predict that branch will be taken |
| rcond | This 3-bit field selects the register-contents condition to test for a move, based on register contents (`MOVr` or `FMOVr`) instruction or a Branch on Register Contents with Prediction (`BPr`) instruction. See *Appendix E, Opcode Maps*, for descriptions of its values. |
| rd | This 5-bit field is the address of the destination register (if any) for all instructions, or the source register for store and swap instructions. |
| rs1 | This 5-bit field is the address of the first `r` or `f` register(s) source operand. |
| rs2 | This 5-bit field is the address of the (usually) second `r` or `f` register(s) source operand; often applicable if `i` = 0. |
| shcnt32 | This 5-bit field provides the shift count for 32-bit shift instructions. |
| shcnt64 | This 6-bit field provides the shift count for 64-bit shift instructions. |
| simm10 | This 10-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for a `MOVr` instruction when `i` = 1. |
| simm11 | This 11-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for a `MOVcc` instruction when `i` = 1. |
| simm13 | This 13-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for many instructions (often used in an integer arithmetic instruction or the displacement for a load/store instruction, only when `i` = 1). |
| sw_trap# | This 7-bit field is an immediate value that is used as the second ALU operand for a Trap on Condition Code instruction. |
| x | The `x` bit selects whether a 32- or 64-bit shift will be performed. |

## 6.3 Instruction Categories

SPARC JPS2 instructions can be grouped into the following categories:

- Memory access
- Memory synchronization
- Integer arithmetic
- Control transfer (CTI)
- Conditional moves
- Register window management

- State register access
- Privileged register access
- Floating-point operate
- Implementation dependent
- Reserved

Each of these categories is described in the following subsections.

## 6.3.1 Memory Access Instructions

Load, store, load-store, and PREFETCH instructions are the only instructions that access memory. All of the instructions except CASA, CASXA, and Partial Store use either two r registers or an r register and simm13 to calculate a 64-bit byte memory address. For example, Compare and Swap uses a single r register to specify a 64-bit byte memory address. To this 64-bit address, the IU appends an ASI that encodes address space information.

The destination field of a memory reference instruction specifies the r or f register(s) that supply the data for a store or that receive the data from a load or LDSTUB. For SWAP, the destination register identifies the r register to be exchanged atomically with the calculated memory location. For Compare and Swap, an r register is specified, the value of which is compared with the value in memory at the computed address. If the values are equal, then the destination field specifies the r register that is to be exchanged atomically with the addressed memory location. If the values are unequal, then the destination field specifies the r register that is to receive the value at the addressed memory location; in this case, the addressed memory location remains unchanged. The LDFSR/LDXFSR and the STFSR/STXFSR are special load and store instructions that load or store the floating-point status instead of acting on a r or f register.

The destination field of a PREFETCH instruction (fcn) is used to encode the type of the prefetch.

Integer load and store instructions support byte, halfword (2 bytes), word (4 bytes), and doubleword (8 bytes) accesses. Floating-point load and store instructions support word, doubleword, and quadword[1] memory accesses. LDSTUB accesses bytes, SWAP accesses words, CASA accesses words, and CASXA accesses doublewords. The Atomic Quad Load instruction accesses a quadword (16 bytes). Block loads and stores access 64-byte aligned data. PREFETCH accesses at least 64 bytes.

1. No JPS2 processor implements the LDQF, LDQFA, STQF, and STQFA instructions in hardware; they generate an exception and are emulated in supervisor software.

**Programming Note –** For some instructions, by using `simm13`, any location in the lowest or highest 4096 bytes of an address space can be accessed without using a register to hold part of the address.

### 6.3.1.1 Memory Alignment Restrictions

A halfword access must be *aligned* on a 2-byte boundary, a word access (including an instruction fetch) must be aligned on a 4-byte boundary, an extended-word (`LDX`, `LDXA`, `STX`, `STXA`) or integer doubleword (`LDD`, `LDDA`, `STD`, `STDA`) access must be aligned on an 8-byte boundary, an integer quadword (`LDQ_Atomic`) access must be aligned on a 16-byte boundary, and a Block Load (`BLD`) or Store (`BST`) access must be aligned on a 64-byte boundary.

A floating-point doubleword access (`LDDF`, `LDDFA`, `STDF`, `STDFA`) should be aligned on an 8-byte boundary, but is only required to be aligned on a word (4-byte) boundary. A floating-point doubleword access to an address which is 4-byte aligned but not 8-byte aligned may result in less efficient and nonatomic access (and may even cause a trap and be emulated in software (impl. dep. #109)), so 8-byte alignment is recommended.

A floating-point quadword access (`LDQF`, `LDQFA`, `STQF`, `STQFA`) should be aligned on a 16-byte boundary, but is only required to be aligned on a word (4-byte) boundary. A floating-point quadword access to an address which is 4-byte or 8-byte aligned but not 16-byte aligned may result in less efficient and nonatomic access (and may even cause a trap and be emulated in software (impl. dep. #111)), so 16-byte alignment is recommended.

An improperly aligned address in a load, store, or load-store instruction causes a *mem_address_not_aligned* exception to occur, with these exceptions:

- An `LDDF` or `LDDFA` instruction accessing an address that is word aligned but not doubleword aligned causes an *LDDF_mem_address_not_aligned* exception (impl. dep. #109).
- An `STDF` or `STDFA` instruction accessing an address that is word aligned but not doubleword aligned causes an *STDF_mem_address_not_aligned* exception (impl. dep. #110).

### 6.3.1.2 Addressing Conventions

A JPS2 virtual processor uses big-endian byte order for all instruction accesses and, by default, for data accesses. It is possible to access data in little-endian format by using selected ASIs. It is also possible to change the default byte order for implicit data accesses. See *Processor State (PSTATE) Register* on page 72 for more information.[1]

1. Interested readers on more background information on big- vs. little-endian can also refer to Cohen, D., "On Holy Wars and a Plea for Peace," *Computer* 14:10 (October 1981), pp. 48-54.

***Big-endian Addressing Convention.*** Within a multiple-byte integer, the byte with the smallest address is the most significant; a byte's significance decreases as its address increases. The big-endian addressing conventions are illustrated in FIGURE 6-4 and described below the figure.

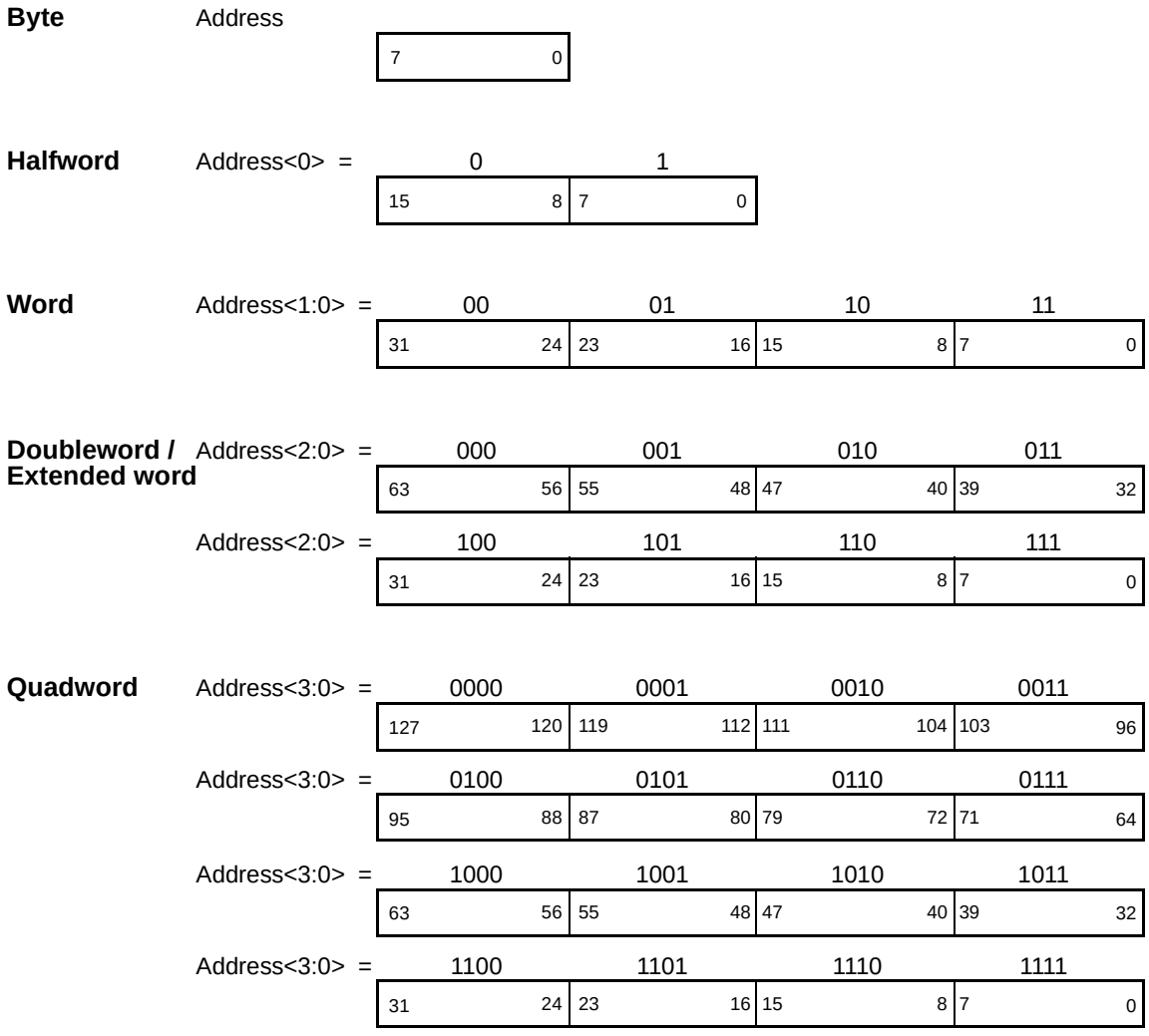



**FIGURE 6-4** Big-endian Addressing Conventions

**byte** A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.

**halfword** For a load/store halfword instruction, two bytes are accessed. The most significant byte (bits 15–8) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 1.

**word** For a load/store word instruction, four bytes are accessed. The most significant byte (bits 31–24) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 3.

**doubleword or extended word** For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The most significant byte (bits 63–56) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 7.

For the deprecated integer load/store double instructions (LDD/STD), two big-endian words are accessed. The word at the address specified in the instruction corresponds to the even register specified in the instruction; the word at address + 4 corresponds to the following odd-numbered register.

**quadword** For a load/store quadword instruction, 16 bytes are accessed. The most significant byte (bits 127–120) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 15.

***Little-endian Addressing Convention.*** Within a multiple-byte integer, the byte with the smallest address is the least significant; a byte's significance increases as its address increases. The little-endian addressing conventions are illustrated in FIGURE 6-5 and defined below the figure.

**byte** A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.

**halfword** For a load/store halfword instruction, two bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 15–8) is accessed at the address + 1.

**word** For a load/store word instruction, four bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 31–24) is accessed at the address + 3.

**doubleword or extended word** For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 63–56) is accessed at the address + 7.

For the deprecated integer load/store double instructions (LDD/STD), two little-endian words are accessed. The word at the address specified in the instruction corresponds to the even register in the instruction; the word at the address specified in the instruction +4 corresponds to the following odd-numbered register. With respect to little endian memory, an LDD (STD} instruction behaves as if it is composed of two 32-bit loads (stores), each of which is byte-swapped independently before being written into each destination register (memory word).

**quadword** For a load/store quadword instruction, 16 bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 127–120) is accessed at the address + 15.

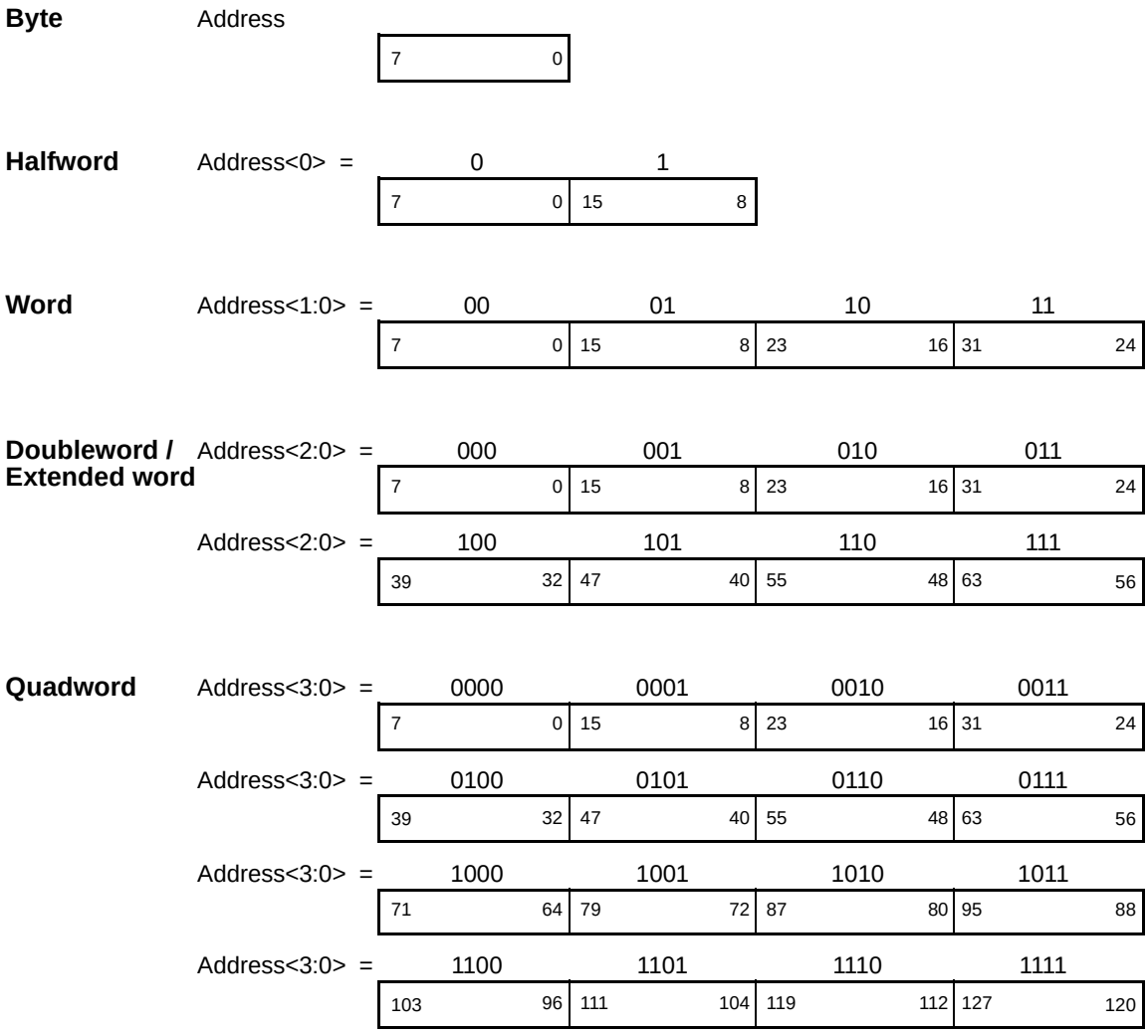



**FIGURE 6-5** Little-endian Addressing Conventions

### 6.3.1.3 Address Space Identifiers (ASIs)

Alternate-space load, store, and load-store instructions specify an explicit ASI to use for their data access; when `i` = 0, the explicit ASI is provided in the instruction's imm_asi field, and when `i` = 1, it is provided in the `ASI` register.

Non-alternate-space load, store, and load-store instructions use an implicit ASI value that depends on the current trap level (TL) and the value of PSTATE.cle. Instruction fetches use an implicit ASI that depends only on the current trap level. The cases are enumerated in TABLE 6-2.

**TABLE 6-2** ASIs Used for Data Accesses and Instruction Fetches

| Access Type | TL | PSTATE.cle | ASI Used |
|---|---|---|---|
| Instruction Fetch | = 0 | any | ASI_PRIMARY |
| | > 0 | any | ASI_NUCLEUS* |
| Non-alternate-space Load, Store, or Load-Store | = 0 | 0 | ASI_PRIMARY |
| | | 1 | ASI_PRIMARY_LITTLE |
| | > 0 | 0 | ASI_NUCLEUS* |
| | | 1 | ASI_NUCLEUS_LITTLE** |
| Alternate-space Load, Store, or Load-Store | any | any | ASI explicitly specified in the instruction (subject to privilege-level restrictions) |

*On some early SPARC V9 implementations, ASI_PRIMARY may have been used for this case.

**On some early SPARC V9 implementations, ASI_PRIMARY_LITTLE may have been used for this case.

See also *Addressing and Alternate Address Spaces* on page 173.

ASIs $00_{16}$ through $7F_{16}$ are restricted; only privileged software is allowed to access them. An attempt to access a restricted ASI by nonprivileged software results in a *privileged_action* exception (impl. dep #103(6)). ASIs $80_{16}$ through $FF_{16}$ are unrestricted; software is allowed to access them whether the virtual processor is operating in privileged or nonprivileged mode, as summarized in TABLE 6-3.

**TABLE 6-3** Allowed Accesses to ASIs

| Value | Access Type | Processor State (PSTATE.priv) | Result of ASI Access |
|---|---|---|---|
| $00_{16}$–$7F_{16}$ | Restricted | Nonprivileged (0) | *privileged_action* exception |
| | | Privileged (1) | Valid access |
| $80_{16}$–$FF_{16}$ | Unrestricted | Nonprivileged (0) | Valid access |
| | | Privileged (1) | Valid access |

**IMPL. DEP. #29:** Some ASIs are implementation dependent in SPARC JPS2. See TABLE L-1 on page 569 for details.

**V9 Compatibility Note –** Some ASIs that were implementation dependent in SPARC V9 are now mandatory in SPARC JPS2.

**IMPL. DEP. #30:** In SPARC JPS2 implementations, all 8 bits of each ASI specifier must be decoded. Refer to Appendix L, *Address Space Identifiers (ASIs)*, of this specification for details.

---

**V9 Compatibility Note –** In SPARC V9, an implementation could choose to decode only a subset of the 8-bit ASI specifier.

---

#### 6.3.1.4 Separate Instruction Memory

A SPARC V9 implementation may choose to access instruction and data through the same address space and use hardware to keep data and instruction memory consistent at all times. It may also choose to overload independent address spaces for data and instructions and allow them to become inconsistent when data writes are made to addresses shared with the instruction space.

A SPARC V9 program containing self-modifying code should use `FLUSH` instruction(s) after executing stores to modify instruction memory and before executing the modified instruction(s), to ensure the consistency of program execution.

#### 6.3.1.5 Memory Synchronization Instructions

Two forms of memory barrier (`MEMBAR`) instructions allow programs to manage the order and completion of memory references. Ordering `MEMBAR`s induce a partial ordering between sets of loads and stores and future loads and stores. Sequencing `MEMBAR`s exert explicit control over completion of loads and stores (or other instructions). Both barrier forms are encoded in a single instruction, with subfunctions bit-encoded in `cmask` and `mmask` fields.

### 6.3.2 Integer Arithmetic and Logical Instructions

The integer arithmetic and logical instructions are generally triadic-register-address instructions that compute a result that is a function of two source operands. They either write the result into the destination register `r[rd]` or discard it. One of the source operands is always `r[rs1]`. The other source operand depends on the `i` bit in the instruction; if `i` = 0, then the operand is `r[rs2]`; if `i` = 1, then the operand is the constant `simm10`, `simm11`, or `simm13`, sign-extended to 64 bits.

**Note**: The value of `r`[0] always reads as zero, and writes to it are ignored.

#### 6.3.2.1 Setting Condition Codes

Most integer arithmetic instructions have two versions: one sets the integer condition codes (`icc` and `xcc`) as a side effect; the other does not affect the condition codes. A special comparison instruction for integer values is not needed since it is easily synthesized with the "subtract and set condition codes" (`SUBcc`) instruction. See *Synthetic Instructions* on page 514 for details.

#### 6.3.2.2 Shift Instructions

Shift instructions shift an `r` register left or right by a constant or variable amount. None of the shift instructions change the condition codes.

#### 6.3.2.3 Set High 22 Bits of Low Word

The "set high 22 bits of low word of an `r` register" instruction (`SETHI`) writes a 22-bit constant from the instruction into bits 31 through 10 of the destination register. It clears the low-order 10 bits and high-order 32 bits, and it does not affect the condition codes. Its primary use is to construct constants in registers.

#### 6.3.2.4 Integer Multiply / Divide

The integer multiply instruction performs a $64 \times 64 \rightarrow 64$-bit operation; the integer divide instructions perform $64 \div 64 \rightarrow 64$-bit operations. For compatibility with SPARC V8, $32 \times 32 \rightarrow 64$-bit multiply instructions, $64 \div 32 \rightarrow 32$-bit divide instructions, and the multiply step instruction are provided. Division by zero causes a *division_by_zero* exception.

#### 6.3.2.5 Tagged Add / Subtract

The tagged add / subtract instructions assume tagged-format data, in which the tag is the two low-order bits of each operand. If either of the two operands has a nonzero tag or if 32-bit arithmetic overflow occurs, tag overflow is detected. If tag overflow occurs, then `TADDcc` and `TSUBcc` set the `CCR.icc.v` bit; if 64-bit arithmetic overflow occurs, then they set the `CCR.xcc.v` bit.

The trapping versions (`TADDccTV`, `TSUBccTV`) are deprecated. See A.71.16 and A.71.17 for details.

### 6.3.3 Control-Transfer Instructions (CTIs)

These are the basic control-transfer instruction types:

- Conditional branch (`Bicc`, `BPcc`, `BPr`, `FBfcc`, `FBPfcc`)
- Unconditional branch

- Call and link (CALL)
- Jump and link (JMPL, RETURN)
- Return from trap (DONE, RETRY)
- Trap (Tcc)

A control-transfer instruction functions by changing the value of the next program counter (nPC) or by changing the value of both the program counter (PC) and the next program counter (nPC). When only the next program counter, nPC, is changed, the effect of the transfer of control is delayed by one instruction. Most control transfers in SPARC V9 are of the delayed variety. The instruction following a delayed control transfer instruction is said to be in the *delay slot* of the control transfer instruction. Some control transfer instructions (branches) can optionally annul, that is, not execute, the instruction in the delay slot, depending upon whether the transfer is taken or not taken. Annulled instructions have no effect upon the program-visible state, nor can they cause a trap.

---

**Programming Note –** The annul bit increases the likelihood that a compiler can find a useful instruction to fill the delay slot after a branch, thereby reducing the number of instructions executed by a program. For example, the annul bit can be used to move an instruction from within a loop to fill the delay slot of the branch that closes the loop.

Likewise, the annul bit can be used to move an instruction from either the “else” or “then” branch of an “if-then-else” program block to the delay slot of the branch that selects between them. Since a full set of conditions is provided, a compiler can arrange the code (possibly reversing the sense of the condition) so that an instruction from either the “else” branch or the “then” branch can be moved to the delay slot.

Use of annulled branches provided some benefit in older, single-issue SPARC implementations. JPS2 processors are superscalar SPARC implementations, on which the only benefit of annulled branches might be a slight reduction in code size. Therefore, the use of annulled branch instructions is no longer encouraged.

---

TABLE 6-4 defines the value of the program counter and the value of the next program counter after execution of each instruction. Conditional branches have two forms: branches that test a condition (including branch-on-register), represented in the table by Bcc, and branches that are unconditional, that is, always or never taken, represented in the table by BA and BN respectively. The effect of an annulled branch is shown in the table through explicit transfers of control, rather than by fetching and annulling the instruction.

**TABLE 6-4** Control Transfer Characteristics

| Instruction Group | Address Form | Delayed | Taken | Annul Bit | New PC | New nPC |
|---|---|---|---|---|---|---|
| Non-CTIs | — | — | — | — | nPC | nPC + 4 |
| Bcc | PC-relative | Yes | Yes | 0 | nPC | EA |
| Bcc | PC-relative | Yes | No | 0 | nPC | nPC + 4 |
| Bcc | PC-relative | Yes | Yes | 1 | nPC | EA |
| Bcc | PC-relative | Yes | No | 1 | nPC + 4 | nPC + 8 |
| BA | PC-relative | Yes | Yes | 0 | nPC | EA |
| BN | PC-relative | Yes | No | 0 | nPC | nPC + 4 |
| BA | PC-relative | Yes | Yes | 1 | EA | EA + 4 |
| BN | PC-relative | Yes | No | 1 | nPC + 4 | nPC + 8 |
| CALL | PC-relative | Yes | — | — | nPC | EA |
| JMPL, RETURN | Register-indirect | Yes | — | — | nPC | EA |
| DONE | Trap state | No | — | — | TNPC[TL] | TNPC[TL] + 4 |
| RETRY | Trap state | No | — | — | TPC[TL] | TNPC[TL] |
| Tcc | Trap vector | No | Yes | — | EA | EA + 4 |
| Tcc | Trap vector | No | No | — | nPC | nPC + 4 |

The effective address, ea in TABLE 6-4, specifies the target of the control transfer instruction. The effective address is computed in different ways, depending on the particular instruction.

- **PC-relative effective address —** A PC-relative effective address is computed by sign extending the instruction's immediate field to 64-bits, left-shifting the word displacement by two bits to create a byte displacement, and adding the result to the contents of the PC.
- **Register-indirect effective address —** A register-indirect effective address computes its target address as either r[rs1] + r[rs2] if i = 0, or r[rs1] + sign_ext(simm13) if i = 1.
- **Trap vector effective address —** A trap vector effective address first computes the software trap number as the least significant 7 bits of r[rs1] + r[rs2] if i = 0, or as the least significant 7 bits of r[rs1] + *sw_trap#* if i = 1. The trap level, TL, is incremented. The hardware trap type is computed as 256 + *sw_trap#* and stored in TT[TL]. The effective address is generated by concatenation of the contents of the TBA register, the "TL > 0" bit, and the contents of TT[TL]. See *When executing with TL = n, the following events affect TT*: on page 82 for details.
- **Trap state effective address —** A trap state effective address is not computed but is taken directly from either TPC[TL] or TNPC[TL].

**V9 Compatibility Note –** SPARC V8 specified that the delay instruction was always fetched, even if annulled, and that an annulled instruction could not cause any traps. SPARC V9 does not require the delay instruction to be fetched if it is annulled.

SPARC V8 left as undefined the result of executing a delayed conditional branch that had a delayed control transfer in its delay slot. For this reason, programmers should avoid such constructs when backward compatibility is an issue.

#### 6.3.3.1 Conditional Branches

A conditional branch transfers control if the specified condition is true. If the annul bit is 0, the instruction in the delay slot is always executed. If the annul bit is 1, the instruction in the delay slot is executed only when the conditional branch is taken. **Note**: The annul behavior of a taken conditional branch is different from that of an unconditional branch.

#### 6.3.3.2 Unconditional Branches

An unconditional branch transfers control unconditionally if its specified condition is "always"; it never transfers control if its specified condition is "never." If the annul bit is 0, then the instruction in the delay slot is always executed. If the annul bit is 1, then the instruction in the delay slot is *never* executed. **Note**: The annul behavior of an unconditional branch is different from that of a taken conditional branch.

#### 6.3.3.3 CALL and JMPL Instructions

The CALL instruction writes the contents of the PC, which points to the CALL instruction itself, into r[15] (out register 7) and then causes a delayed transfer of control to a PC-relative effective address. The value written into r[15] is visible to the instruction in the delay slot.

The JMPL instruction writes the contents of the PC, which points to the JMPL instruction itself, into r[rd] and then causes a register-indirect delayed transfer of control to the address given by "r[rs1] + r[rs2]" or "r[rs1] + a signed immediate value." The value written into r[rd] is visible to the instruction in the delay slot.

When PSTATE.am = 1, the value of the high-order 32 bits transmitted to r[15] by the CALL instruction or to R[rd] by the JMPL instruction is zero.

### 6.3.3.4 RETURN Instruction

The RETURN instruction is used to return from a trap handler executing in nonprivileged mode. RETURN combines the control-transfer characteristics of a JMPL instruction with r[0] specified as the destination register and the register-window semantics of a RESTORE instruction.

### 6.3.3.5 DONE and RETRY Instructions

The DONE and RETRY instructions are used by privileged software to return from a trap. These instructions restore the machine state to values saved in the TSTATE register.

RETRY returns to the instruction that caused the trap in order to reexecute it. DONE returns to the instruction pointed to by the value of nPC associated with the instruction that caused the trap, that is, the next logical instruction in the program. DONE presumes that the trap handler did whatever was requested by the program and that execution should continue.

### 6.3.3.6 Trap Instruction (Tcc)

The Tcc instruction initiates a trap if the condition specified by its cond field matches the current state of the condition code register specified by its cc field; otherwise, it executes as a NOP. If the trap is taken, it increments the TL register, computes a trap type that is stored in TT[tl], and transfers to a computed address in the trap table pointed to by TBA. See *When executing with TL = n, the following events affect TT*: on page 82.

A Tcc instruction can specify one of 128 software trap types. When a Tcc is taken, 256 plus the 7 least significant bits of the sum of the Tcc's source operands is written to TT[tl]. The only visible difference between a software trap generated by a Tcc instruction and a hardware trap is the trap number in the TT register. See Chapter 7, *Traps*, for more information.

---

**Programming Note –** Tcc can be used to implement breakpointing, tracing, and calls to supervisor software. Tcc can also be used for runtime checks, such as out-of-range array index checks or integer overflow checks.

---

### 6.3.3.7 Conditional Move Instructions

This subsection describes two groups of instructions that copy or move the contents of any integer or floating-point register.

***MOVcc and FMOVcc Instructions.*** The MOVcc and FMOVcc instructions copy the contents of any integer or floating-point register to a destination integer or floating-point register if a condition is satisfied. The condition to test is specified in the instruction and may be any of the conditions allowed in conditional delayed control-transfer instructions. This condition is tested against one of the 6 sets of condition codes (icc, xcc, fcc0, fcc1, fcc2, and fcc3), as specified by the instruction. For example:

```
fmovdg  %fcc2, %f20, %f22
```

moves the contents of the double-precision floating-point register %f20 to register %f22 if floating-point condition code number 2 (fcc2) indicates a greater-than relation (FSR.fcc2 = 2). If fcc2 does not indicate a greater-than relation (FSR.fcc2 ≠ 2), then the move is not performed.

The MOVcc and FMOVcc instructions can be used to eliminate some branches in programs. In most implementations, branches will be more expensive than the MOVcc or FMOVcc instructions. For example, the following C statement:

```
if (A > B) X  = 1; else X  = 0;
```

can be coded as

```
cmp     %i0, %i2          ! (A > B)
or      %g0, 0, %i3       ! set X  = 0
movg    %xcc,1, %i3       ! overwrite X with 1 if A > B
```

which eliminates the need for a branch.

***MOVr and FMOVr Instructions.*** The MOVr and FMOVr instructions allow the contents of any integer or floating-point register to be moved to a destination integer or floating-point register if the contents of a register satisfy a specified condition. The conditions to test are enumerated in TABLE 6-5.

**TABLE 6-5** MOVr and FMOVr Test Conditions

| Condition | Description |
|---|---|
| NZ | Nonzero |
| Z | Zero |
| GEZ | Greater than or equal to zero |
| LZ | Less than zero |
| LEZ | Less than or equal to zero |
| GZ | Greater than zero |

Any of the integer registers may be tested for one of the conditions, and the result used to control the move. For example,

```
movrnz  %i2, %l4, %l6
```

moves integer register %l4 to integer register %l6 if integer register %i2 contains a nonzero value.

MOVr and FMOVr can be used to eliminate some branches in programs or can emulate multiple unsigned condition codes by using an integer register to hold the result of a comparison.

## 6.3.4 Register Window Management Instructions

This subsection describes the instructions that manage register windows in SPARC JPS2. The privileged registers affected by these instructions are described in *Register Window PR State Registers* on page 83.

### 6.3.4.1 SAVE Instruction

The SAVE instruction allocates a new register window and saves the caller's register window by incrementing the CWP register.

If CANSAVE = 0, then execution of a SAVE instruction causes a window spill exception, that is, one of the *spill_n_{normal, other}* exceptions.

If CANSAVE ≠ 0 but the number of clean windows is zero, that is,

$$(\text{CLEANWIN} - \text{CANRESTORE}) = 0$$

then SAVE causes a *clean_window* exception.

If SAVE does not cause an exception, it performs an ADD operation, decrements CANSAVE, and increments CANRESTORE. The source registers for the ADD are from the old window (the one to which CWP pointed before the SAVE), while the result is written into a register in the new window (the one to which the incremented CWP points).

### 6.3.4.2 RESTORE Instruction

The RESTORE instruction restores the previous register window by decrementing the CWP register.

If CANRESTORE = 0, execution of a RESTORE instruction causes a window fill exception, that is, one of the *fill_n_{normal, other}* exceptions.

If RESTORE does not cause an exception, it performs an ADD operation, decrements CANRESTORE, and increments CANSAVE. The source registers for the ADD are from the old window (the one to which CWP pointed before the RESTORE), and the result is written into a register in the new window (the one to which the decremented CWP points).

---

**Programming Note –** This note describes a common convention for use of register windows, SAVE, RESTORE, CALL, and JMPL instructions.

A procedure is invoked by executing a CALL (or a JMPL) instruction. If the procedure requires a register window, it executes a SAVE instruction. A routine that does not allocate a register window of its own (possibly a leaf procedure) should not modify any windowed registers except out registers 0 through 6. See *Leaf-Procedure Optimization* on page 521.

A procedure that uses a register window returns by executing both a RESTORE and a JMPL instruction. A procedure that has not allocated a register window returns by executing a JMPL only. The target address for the JMPL instruction is normally 8 plus the address saved by the calling instruction, that is, the instruction after the instruction in the delay slot of the calling instruction.

The SAVE and RESTORE instructions can be used to atomically establish a new memory stack pointer in an r register and switch to a new or previous register window. See *Register Allocation Within a Window* on page 523.

---

### 6.3.4.3 SAVED Instruction

The SAVED instruction should be used by a spill trap handler to indicate that a window spill has completed successfully. It increments CANSAVE:

CANSAVE ← (CANSAVE + 1)

If the saved window belongs to a different address space (OTHERWIN ≠ 0), it decrements OTHERWIN:

OTHERWIN ← (OTHERWIN – 1)

Otherwise, the saved window belongs to the current address space (OTHERWIN = 0), so SAVED decrements CANRESTORE:

CANRESTORE ← (CANRESTORE – 1)

If CANSAVE ≥ (NWINDOWS – 2) or CANRESTORE = 0 just prior to execution of a SAVED instruction, the subsequent behavior of the processor is undefined. In neither of these cases can SAVED generate a register window state that is both valid (see *Register Window State Definition* on page 125) and consistent with the state prior to the SAVED.

### 6.3.4.4 RESTORED Instruction

The RESTORED instruction should be used by a fill trap handler to indicate that a window has been filled successfully. It increments CANRESTORE:

CANRESTORE ← (CANRESTORE + 1)

If the restored window replaces a window that belongs to a different address space (OTHERWIN ≠ 0), it decrements OTHERWIN:

OTHERWIN ← (OTHERWIN – 1)

Otherwise, the restored window belongs to the current address space (OTHERWIN = 0), so RESTORED decrements CANSAVE:

CANSAVE ← (CANSAVE – 1)

If CLEANWIN is less than NWINDOWS – 1, the RESTORED instruction increments CLEANWIN:

**if** (CLEANWIN < (NWINDOWS – 1)) **then** CLEANWIN ← (CLEANWIN + 1)

If CANSAVE = 0 or CANRESTORE ≥ (NWINDOWS – 2) just prior to execution of a RESTORED instruction, the subsequent behavior of the processor is undefined. In neither of these cases can RESTORED generate a register window state that is both valid (see *Register Window State Definition* on page 125) and consistent with the state prior to the RESTORED.

### 6.3.4.5 Flush Windows Instruction

The FLUSHW instruction flushes all of the register windows, except the current window, by performing repetitive spill traps. The FLUSHW instruction causes a spill trap if any register window (other than the current window) has valid contents. The number of windows with valid contents is computed as:

NWINDOWS – 2 – CANSAVE

If this number is nonzero, the FLUSHW instruction causes a spill trap. Otherwise, FLUSHW has no effect. If the spill trap handler exits with a RETRY instruction, the FLUSHW instruction continues causing spill traps until all the register windows except the current window have been flushed.

## 6.3.5 State Register Access

The read / write state register instructions access program-visible state and status registers. These instructions read / write the state registers into / from r registers. A read / write State Register instruction is privileged only if the accessed register is privileged.

The supported RDASR and WRASR instructions are described in TABLE 6-6; for more information see *State Registers* on page 86.

**TABLE 6-6** Supported RDASR and WRASR Instructions

| ASR # | ASR Name | Description | R, W? | Priv? |
|---|---|---|---|---|
| 0 | Y[D] | Y register (deprecated) | RW | No |
| 2 | CCR | Condition Codes Register | RW | No |
| 3 | ASI | ASI | RW | No |
| 4 | TICK | Tick (timer) | R | Yes / No[1] |
| 5 | PC | Program Counter | R | No |
| 6 | FPRS | Floating-Point Register Status | RW | No |
| 16 | PCR | Performance Control Register | RW | Yes / No[2] |
| 17 | PIC | Performance Instrumentation Counters | RW | Yes / No[3] |
| 18 | DCR | Dispatch Control Register | RW | Yes |
| 19 | GSR | Graphics Status Register | RW | No |
| 20 | SET_SOFTINT (pseudo) | Set bits in SOFTINT | W | Yes |
| 21 | CLEAR_SOFTINT (pseudo) | Clear bits in SOFTINT | W | Yes |
| 22 | SOFTINT | Software interrupt register | RW | Yes |
| 23 | TICK_COMPARE | TICK compare | RW | Yes |
| 24 | STICK | System TICK (timer) | RW | Yes / No[4] |
| 25 | STICK_COMPARE | STICK compare | RW | Yes |
| 26-31 | Implementation dependent | — | — | — |

1. Writes are always privileged; reads are privileged if TICK.npt = 1; otherwise, reads are nonprivileged.
2. When PSTATE.priv = 0, the accessibility of PCR is implementation dependent. (impl. dep. #250) See *Performance Control Register (PCR) (ASR 16)* on page 67.
3. All accesses are privileged if PCR.priv = 1; otherwise, all accesses are nonprivileged.
4. Writes are always privileged; reads are privileged if STICK.npt = 1; otherwise, reads are nonprivileged.

## 6.3.6 Privileged Register Access

The read / write privileged register instructions access state and status registers that are visible only to privileged software. These instructions read / write privileged registers into / from r registers. The read / write privileged register instructions are privileged.

## 6.3.7 Floating-Point Operate (FPop) Instructions

Floating-point operate instructions (FPops) are generally triadic-register-address instructions. They compute a result that is a function of one or two source operands and place the result in one or more destination f registers, with two exceptions:

- Floating-point convert operations, which use one source and one destination operand
- Floating-point compare operations, which do not write to an f register but update one of the fcc*n* fields of the FSR instead

The term "FPop" refers to those instructions encoded by the FPop1 and FPop2 opcodes and does *not* include branches based on the floating-point condition codes (FBfcc and FBPfcc) or the load / store floating-point instructions.

The FMOVcc instructions function for the floating-point registers as the MOVcc instructions do for the integer registers. See *MOVcc and FMOVcc Instructions* on page 117.

The FMOVr instructions function for the floating-point registers as the MOVr instructions do for the integer registers. See *MOVr and FMOVr Instructions* on page 117.

If no floating-point unit is present or if PSTATE.pef = 0 or FPRS.fef = 0, then any instruction, including an FPop instruction, that attempts to access an FPU register generates an *fp_disabled* exception.

All FPop instructions clear the ftt field and set the cexc field unless they generate an exception. Floating-point compare instructions also write one of the fcc*n* fields. All FPop instructions that can generate IEEE exceptions set the cexc and aexc fields unless they generate an exception. FABS(s,d,q), FMOV(s,d,q), FMOVcc(s,d,q), FMOVr(s,d,q), and FNEG(s,d,q) cannot generate IEEE exceptions, so they clear cexc and leave aexc unchanged.

**IMPL. DEP. #3:** An implementation may indicate that a floating-point instruction did not produce a correct IEEE Std 754-1985 result by generating an *fp_exception_other* exception with FSR.ftt = unfinished_FPop or unimplemented FPop. In this case, privileged software must emulate any functionality not present in the hardware.

SPARC JPS2 processors do not implement any quad-precision floating-point operations in hardware. Instead, these operations cause an *fp_exception_other* trap with FSR.ftt = *unimplemented_FPop*, and system software emulates quad operations (impl. dep. #1).

See *ftt = unfinished_FPop* on page 57 to see which instructions can produce an unfinished_FPop exception. See *ftt = unimplemented_FPop* on page 58 to see which instructions can produce an unimplemented_FPop exception.

## 6.3.8 Implementation-Dependent Instructions

SPARC V9 provides two instructions that are entirely implementation dependent: IMPDEP1 and IMPDEP2.

In SPARC JPS2, the IMPDEP1 opcode space is used by VIS™ instructions.

In SPARC JPS2, IMPDEP2 is subdivided into IMPDEP2A and IMPDEP2B. IMPDEP2A remains implementation dependent. However, some implementations use the IMPDEP2B opcode space for floating-point multiply-add/multiply-subtract instructions, which are expected to be incorporated into a future JPS. Therefore, for future compatibility, it is recommended that SPARC JPS2 implementations not use IMPDEP2B instructions, unless they are used compatibly with the Fujitsu SPARC64 VI implementation.

## 6.3.9 Reserved Opcodes and Instruction Fields

If a conforming SPARC V9 implementation attempts to execute an instruction that is not specifically defined in this specification, it behaves as follows:

- If the instruction encodes an implementation-specific extension to the instruction set, that extension is executed.
- If the instruction does not encode an extension to the instruction set, but would decode as a valid instruction if nonzero bits in reserved instruction field(s) were ignored (read as 0):
  - The recommended behavior is to generate an *illegal_instruction* exception (or, in the FPop opcode space, an *fp_exception_other* exception with FSR.ftt = 3 (unimplemented_FPop)).
  - Alternatively, the implementation can ignore the nonzero reserved field bits and execute the instruction as if those bits had been zero.
- If the instruction does not encode an extension to the instruction set and would still not decode as a valid instruction if nonzero bits in reserved instruction field(s) were ignored, then the instruction is invalid and causes an exception. Specifically, attempting to execute an invalid instruction in the FPop opcode space

causes an *fp_exception_other* trap (with FSR.ftt = unimplemented_FPop); attempting to execute any other invalid instruction causes an *illegal_instruction* trap.

See *Appendix E, Opcode Maps*, for an enumeration of the reserved opcodes.

---

**Implementation Note –** As described above, implementations are strongly encouraged, but not strictly required, to trap on nonzero values in reserved instruction fields.

---

**Programming Note –** For software portability, software (such as assemblers, static compilers, and dynamic compilers) that generates SPARC instructions must always generate zeroes in instruction fields marked "reserved" ("—").

---

## 6.3.10 Summary of Unimplemented Instructions

Certain SPARC V9 instructions are not implemented in hardware in a SPARC JPS2 processor (impl. dep. #1). Executing any of these instructions results in the implementation-dependent behavior described in TABLE 6-7.

**TABLE 6-7** SPARC JPS2 Actions on Unimplemented Instructions

| Instructions | Trap Taken | Notes |
|---|---|---|
| Quad FPops (including FdMULq) | *fp_exception_other* | FSR.ftt = *unimplemented_FPop* |
| POPC | *illegal_instruction* | (none) |
| RDPR FQ | *illegal_instruction* | There is no FQ |
| LDQF | *illegal_instruction* | (none) |
| STQF | *illegal_instruction* | (none) |

---

**Note –** The operating system emulates all of these instructions except RDPR FQ.

---

## 6.4 Register Window Management

The state of the register windows is determined by the contents of the set of privileged registers described in *Register Window PR State Registers* on page 83. Those registers are affected by the instructions described in *Register Window Management Instructions* on page 118. Privileged software can read/write these state registers directly by using RDPR/WRPR instructions.

### 6.4.1 Register Window State Definition

For the state of the register windows to be consistent, the following must always be true:

$$\text{CANSAVE} + \text{CANRESTORE} + \text{OTHERWIN} = \text{NWINDOWS} - 2$$

TABLE 5-2 on page 47 shows how the register windows are partitioned to obtain the above equation. The partitions are as follows:

- The current window plus the window that must not be used because it overlaps two other valid windows. In FIGURE 5-2, these are windows 0 and 5, respectively. They are always present and account for the 2 subtracted from NWINDOWS in the right side of the above equation.
- Windows that do not have valid contents and that can be used (through a SAVE instruction) without causing a spill trap. These windows (windows 1–4 in FIGURE 5-2) are counted in CANSAVE.
- Windows that have valid contents for the current address space and that can be used (through the RESTORE instruction) without causing a fill trap. These windows (window 7 in FIGURE 5-2) are counted in CANRESTORE.
- Windows that have valid contents for an address space other than the current address space. An attempt to use these windows through a SAVE (RESTORE) instruction results in a spill (fill) trap to a separate set of trap vectors, as discussed in the following subsection. These windows (window 6 in FIGURE 5-2) are counted in OTHERWIN.

In addition,

$$\text{CLEANWIN} \geq \text{CANRESTORE}$$

since CLEANWIN is the sum of CANRESTORE and the number of clean windows following CWP.

For the window-management features of the architecture described in this section to be used, the state of the register windows must be kept consistent at all times, except within the trap handlers for window spilling, filling, and cleaning. While window

traps are being handled, the state may be inconsistent. Window spill/fill trap handlers should be written so that a nested trap can be taken without destroying state.

---

**Programming Note –** System software is responsible for keeping the state of the register windows consistent at all times. Failure to do so will cause undefined behavior. For example, CANSAVE, CANRESTORE, and OTHERWIN must never be greater than or equal to 7 (NWINDOWS – 1).

---

## 6.4.2 Register Window Traps

Window traps are used to manage overflow and underflow conditions in the register windows, support clean windows, and implement the FLUSHW instruction.

### 6.4.2.1 Window Spill and Fill Traps

A window overflow occurs when a SAVE instruction is executed and the next register window is occupied (CANSAVE = 0). An overflow causes a spill trap that allows privileged software to save the occupied register window in memory, thereby making it available for use.

A window underflow occurs when a RESTORE instruction is executed and the previous register window is not valid (CANRESTORE = 0). An underflow causes a fill trap that allows privileged software to load the registers from memory.

### 6.4.2.2 *Clean_Window* Trap

The virtual processor provides the *clean_window* trap so that software can create a secure environment in which it is guaranteed that register windows contain only data from the same address space.

A clean register window is one in which all of the registers, including uninitialized registers, contain either 0 or data assigned by software executing in the address space to which the window belongs. A clean window cannot contain register values from another process, that is, software operating in a different address space.

Supervisor software specifies the number of windows that are clean with respect to the current address space in the CLEANWIN register. This number includes register windows that can be restored (the value in the CANRESTORE register) and the register windows following CWP that can be used without cleaning. Therefore, the number of clean windows that are available to be used by the SAVE instruction is

```
CLEANWIN – CANRESTORE
```

The SAVE instruction causes a *clean_window* exception if this value is 0. This behavior allows supervisor software to clean a register window before it is accessed by a user.

### 6.4.2.3 Vectoring of Fill / Spill Traps

To make handling of fill and spill traps efficient, SPARC V9 provides multiple trap vectors for the fill and spill traps. These trap vectors are determined as follows:

- Supervisor software can mark a set of contiguous register windows as belonging to an address space different from the current one. The count of these register windows is kept in the OTHERWIN register. A separate set of trap vectors (*fill_n_other* and *spill_n_other*) is provided for spill and fill traps for these register windows (as opposed to register windows that belong to the current address space).
- Supervisor software can specify the trap vectors for fill and spill traps by presetting the fields in the WSTATE register. This register contains two subfields, each three bits wide. The WSTATE.normal field determines one of eight spill (fill) vectors to be used when the register window to be spilled (filled) belongs to the current address space (OTHERWIN = 0). If the OTHERWIN register is nonzero, the WSTATE.other field selects one of eight *fill_n_other* (*spill_n_other*) trap vectors.

See Chapter 7, *Traps*, for more details on how the trap address is determined.

### 6.4.2.4 CWP on Window Traps

On a window trap, the CWP is set to point to the window that must be accessed by the trap handler, as follows. (**Note**: All arithmetic on CWP is done **modulo** NWINDOWS.)

- If the spill trap occurs because of a SAVE instruction (when CANSAVE = 0), there is an overlap window between the CWP and the next register window to be spilled:

  CWP ← (CWP + 2) **mod** NWINDOWS

  If the spill trap occurs because of a FLUSHW instruction, there can be unused windows (CANSAVE) in addition to the overlap window between the CWP and the window to be spilled:

  CWP ← (CWP + CANSAVE + 2) **mod** NWINDOWS

---

**Programming Note –** All spill traps can set CWP using the calculation:
CWP ← (CWP + CANSAVE + 2) **mod** NWINDOWS
since CANSAVE is 0 whenever a trap occurs because of a SAVE instruction.

---

- On a fill trap, the window preceding CWP must be filled:

  CWP ← (CWP – 1) **mod** NWINDOWS

- On a *clean_window* trap, the window following CWP must be cleaned. Then

  CWP ← (CWP + 1) **mod** NWINDOWS

### 6.4.2.5 Window Trap Handlers

The trap handlers for fill, spill, and *clean_window* traps must handle the trap appropriately and return, by using the RETRY instruction, to reexecute the trapped instruction. The state of the register windows must be updated by the trap handler, and the relationships among CLEANWIN, CANSAVE, CANRESTORE, and OTHERWIN must remain consistent. Follow these recommendations:

- A spill trap handler should execute the SAVED instruction for each window that it spills.
- A fill trap handler should execute the RESTORED instruction for each window that it fills.
- A *clean_window* trap handler should increment CLEANWIN for each window that it cleans:

  CLEANWIN ← (CLEANWIN + 1)

Window trap handlers in SPARC JPS2 can be very efficient. See *Example Code for Spill Handler* on page 534 for details and sample code.

# Traps

A trap is a vectored transfer of control to supervisor software through a trap table that contains the first eight (32 for *clean_window*, spill, fill, *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection* traps) instructions of each trap handler. The base address of the table is established by supervisor software, by writing the Trap Base Address (TBA) register. The displacement within the table is determined by the trap type and the current trap level (TL). One-half of the table is reserved for hardware traps; one-quarter is reserved for software traps generated by Tcc instructions; the remaining quarter is reserved for future use.

A trap behaves like an unexpected procedure call. It causes the hardware to do the following:

1. Save certain virtual processor state (program counters, CWP, ASI, CCR, PSTATE, and the trap type) on a hardware register stack.
2. Enter privileged execution mode with a predefined PSTATE.
3. Begin executing trap handler code in the trap vector.

When the trap handler has finished, it uses either a DONE or RETRY instruction to return.

A trap may be caused by a Tcc instruction, an instruction-induced exception, a reset, an asynchronous error, or an interrupt request not directly related to a particular instruction. The virtual processor must appear to behave as though, before executing each instruction, it determines if there are any pending exceptions or interrupt requests. If there are pending exceptions or interrupt requests, the virtual processor selects the highest-priority exception or interrupt request and causes a trap.

Thus, an *exception* is a condition that makes it impossible for the virtual processor to continue executing the current instruction stream without software intervention. A *trap* is the action taken by the virtual processor when it changes the instruction flow in response to the presence of an exception, interrupt, reset, or Tcc instruction.

A catastrophic error exception is due to the detection of a hardware malfunction from which, due to the nature of the error, the state of the machine at the time of the exception cannot be restored. Since the machine state cannot be restored, execution after such an exception may not be resumable. An example of such an error is an uncorrectable bus parity error.

**IMPL. DEP. #31**: The causes and effects of catastrophic errors (for example, *internal_processor_error*) are implementation-dependent. They may cause precise, deferred, or disrupting traps.

Traps are described in these sections:


- *Virtual Processor States, Normal and RED_state Traps* on page 130
- *Trap Categories* on page 134
- *Trap Control* on page 138
- *Trap-Table Entry Addresses* on page 139
- *Trap Processing* on page 146
- *Exception and Interrupt Descriptions* on page 160


# 7.1 Virtual Processor States, Normal and `RED_state` Traps

A JPS2 virtual processor is always in one of three discrete states:

- `execute_state`, which is the normal execution state of the virtual processor
- `RED_state` (**R**eset, **E**rror, and **D**ebug state), which is a restricted execution state reserved for processing traps that occur when `TL` = `MAXTL` – 1, and for processing hardware- and software-initiated resets
- `error_state`, which is a halted state that is entered as a result of a trap when `TL = MAXTL`

`TL` and `PSTATE.red` also affect the vector address. `TL` also determines where (e.g. which entry of `TSTATE`) the states are saved. See *RED_state Characteristics* on page 597 for more details.

Traps processed in `execute_state` are called *normal traps*. Traps processed in `RED_state` are called `RED_state` *traps*.

---

**V9 Compatibility Note –** `Red_state` traps were called “special traps” in SPARC V9. The name was changed to clarify the terminology.

---

## 7.1.1 RED_state

RED_state is an acronym for **R**eset, **E**rror, and **D**ebug state. The virtual processor enters RED_state under any one of the following conditions:

- A trap is taken when TL = MAXTL –1.
- A POR, WDR, or XIR reset occurs.
- An SIR occurs when TL < MAXTL.
- System software sets PSTATE.red = 1.

RED_state serves two mutually exclusive purposes:

- During trap processing, it indicates that no more trap levels are available; that is, if another nested trap is taken, the virtual processor will enter error_state and halt. RED_state provides system software with a restricted execution environment.
- It provides the execution environment for all reset processing.

RED_state is indicated by PSTATE.red. When this bit is set to 1, the virtual processor is in RED_state; when this bit is zero, the virtual processor is not in RED_state, independent of the value of TL. Executing a DONE or RETRY instruction in RED_state restores the stacked copy of the PSTATE register, which zeroes the PSTATE.red flag if it was zero in the stacked copy. System software can also set or clear the PSTATE.red flag with a WRPR instruction, which also forces the virtual processor to enter or exit RED_state, respectively. In this case, the WRPR instruction should be placed in the delay slot of a jump so that the PC can be changed in concert with the state change.

---

**Note –** Setting TL = MAXTL with a WRPR instruction does not also set PSTATE.red = 1, nor does it alter any other machine state. The values of PSTATE.red and TL are independent.

Setting PSTATE.red with a WRPR instruction causes the virtual processor to execute in RED_state. This results in the execution environment, as defined in *RED_state Execution Environment* on page 132. However, it is different from a RED_state trap in the sense that there are no trap-related changes in the machine state (for example., TL does not change).

---

### 7.1.1.1 RED_state Trap Table

Traps occurring in RED_state or traps that cause the virtual processor to enter RED_state use an abbreviated trap vector. The RED_state trap vector is constructed so that it can overlay the normal trap vector if necessary. TABLE 7-1 illustrates the RED_state trap vector layout.

**TABLE 7-1** RED_state Trap Vector Layout

| Offset | TT | Reason |
|---|---|---|
| $00_{16}$ | 0 | *Reserved* (SPARC V8 reset) |
| $20_{16}$ | 1 | Power-on reset (POR) |
| $40_{16}$ | 2† | Watchdog reset (WDR) |
| $60_{16}$ | 3‡ | Externally initiated reset (XIR) |
| $80_{16}$ | 4 | Software-initiated reset (SIR) |
| $A0_{16}$ | * | All other exceptions in RED_state |

‡TT = 3 if an *externally_initiated_reset* (XIR) occurs while the virtual processor is not in error_state; TT = trap type of the exception that caused entry into error_state if the externally initiated reset occurs in error_state.

†TT = 2 if a watchdog reset occurs while the virtual processor is not in error_state; TT = trap type of the exception that caused entry into error_state if a watchdog reset (WDR) occurs in error_state.

*TT = trap type of the exception. See TABLE 7-3 on page 141.

**IMPL. DEP. #114:** The RED_state trap vector is located at an implementation-dependent address referred to as RSTVaddr. The value of RSTVaddr is a constant within each implementation.

### 7.1.1.2 RED_state Execution Environment

In RED_state, the virtual processor is forced to execute in a restricted environment by overriding the values of some virtual processor control and state registers.

The values are overridden, not set, allowing them to be switched atomically.

**IMPL. DEP. #115:** SPARC JPS2 defines much of RED_state behavior (see Section 7.5, Appendix F, and Appendix O); any RED_state behavior that is not defined in JPS2 remains implementation dependent.

When RED_state is entered because of component failures, the handler should attempt to recover from potentially catastrophic error conditions or to disable the failing components. When RED_state is entered after a reset, the software should create the environment necessary to restore the system to a running state.

### 7.1.1.3 RED_state Entry Traps

The following traps are processed in RED_state in all cases:

- **Power-on reset (POR)** — Implemented in hardware in SPARC JPS2 processors. POR causes the virtual processor to start execution at this trap table entry.

- **Watchdog reset (WDR)** — Implemented in hardware in SPARC JPS2; this trap is also used as a recovery mechanism from `error_state` in SPARC JPS2. Upon an entry to `error_state`, the virtual processor automatically invokes a watchdog reset to enter `RED_state`.
- **Externally initiated reset (XIR)** — Implemented in hardware in SPARC JPS2; typically used as a nonmaskable interrupt method for debug.

In addition, the following trap is processed in `RED_state` if `TL < MAXTL` when the trap is taken. Otherwise, it causes the virtual processor to enter `error_state`:

- **Software-initiated reset (SIR)**

Traps that occur when `TL` = `MAXTL` – 1 also set `PSTATE.red` = 1; that is, any trap handler entered with `TL` = `MAXTL` runs in `RED_state`.

Any non-reset trap that sets `PSTATE.red` = 1 or that occurs when `PSTATE.red` = 1 branches to a special entry in the `RED_state` trap vector at `RSTVaddr` + $A0_{16}$. Reset traps are described in *Reset Traps* on page 136.

### 7.1.1.4 `RED_state` Software Considerations

In effect, `RED_state` reserves one level of the trap stack for recovery and reset processing. Software should be designed to require only `MAXTL` – 1 trap levels for normal processing. That is, any trap that causes `TL` = `MAXTL` is an exceptional condition that should cause entry to `RED_state`.

The architected value for `MAXTL` in SPARC JPS2 is 5; typical usage of the trap levels is shown in TABLE 7-2.

**TABLE 7-2** Typical Usage for Trap Levels

| TL | Usage |
|---|---|
| 0 | Normal execution |
| 1 | System calls; interrupt handlers; instruction emulation |
| 2 | Window spill/fill handler |
| 3 | Page-fault handler |
| 4 | Reserved for error handling |
| 5 | `RED_state` handler |

**Note –** To log the state of the virtual processor, `RED_state` handler software needs either a spare register or a preloaded pointer to a save area. To support recovery, the operating system might reserve one of the alternate global registers (for example, `%a7`) for use in `RED_state`.

### 7.1.2 error_state

The virtual processor enters error_state when a trap occurs while the virtual processor is already at its maximum supported trap level, that is, when TL = MAXTL.

**IMPL. DEP. #39:** The virtual processor may enter error_state when an implementation dependent error condition occurs.

**IMPL. DEP. #40:** Effects when error_state is entered are implementation-dependent, but it is recommended that as much processor state as possible be preserved upon entry to error_state. In addition, a SPARC JPS2 virtual processor may have other error_state entry traps that are implementation dependent.

**IMPL. DEP. #254:** The means of exiting error_state are implementation dependent. A suggested method is for the virtual processor, upon entering error_state, to automatically generate a *watchdog_reset* (WDR).

## 7.2 Trap Categories

An exception or interrupt request can cause any of the following trap types:

- Precise trap
- Deferred trap
- Disrupting trap
- Reset trap

### 7.2.1 Precise Traps

A *precise trap* is induced by a particular instruction and occurs before any program-visible state has been changed by the trap-inducing instructions. When a precise trap occurs, several conditions must be true:

- The PC saved in TPC[tl] points to the instruction that induced the trap and the nPC saved in TNPC[tl] points to the instruction that was to be executed next.
- All instructions issued before the one that induced the trap have completed execution.
- Any instructions issued after the one that induced the trap remain unexecuted.

Among the actions the trap handler software might take after a precise trap are these:

- Return to the instruction that caused the trap and reexecute it by executing a RETRY instruction (PC ← old PC, nPC ← old nPC).

- Emulate the instruction that caused the trap and return to the succeeding instruction by executing a DONE instruction (PC ← old nPC, nPC← old nPC + 4).
- Terminate the program or process associated with the trap.

## 7.2.2 Deferred Traps

A *deferred trap* is also induced by a particular instruction, but unlike a precise trap, a deferred trap may occur after program-visible state has been changed. Such state may have been changed by the execution of either the trap-inducing instruction itself or by one or more other instructions.

Associated with a particular deferred-trap implementation, the following must exist:

- An instruction that causes a potentially outstanding deferred-trap exception to be taken as a trap
- Privileged instructions that access the state information needed by the supervisor software to emulate the deferred-trap-inducing instruction and to resume execution of the trapped instruction stream.

---

**Programming Note –** Resuming execution may require the emulation of instructions that had not completed execution at the time of the deferred trap, that is, those instructions in the deferred-trap queue.

---

**IMPL. DEP. #32**: Whether any deferred traps (and, possibly, associated deferred-trap queues) are present is implementation dependent.

Among the actions software can take after a deferred trap are these:

- Emulate the instruction that caused the exception, emulate or cause to execute any other execution-deferred instructions that were in an associated deferred-trap state queue, and use RETRY to return control to the instruction at which the deferred trap was invoked.
- Terminate the program or process associated with the trap.

## 7.2.3 Disrupting Traps

A *disrupting trap* is neither a precise trap nor a deferred trap. A disrupting trap is caused by a *condition* (for example, an interrupt) rather than directly by a particular instruction; that cause distinguishes it from precise and deferred traps. When a disrupting trap has been serviced, trap handler software normally arranges for program execution to resume where it left off. That differentiates disrupting traps from reset traps, which trap to a unique reset address from which execution of the program that was running when the reset occurred is never expected to resume.

Disrupting traps are controlled by a combination of the Processor Interrupt Level (PIL) register and the Interrupt Enable (ie) field of PSTATE. A disrupting trap condition is ignored when interrupts are disabled (PSTATE.ie = 0) or when the condition's interrupt level is less than or equal to that specified in PIL.

A disrupting trap may be due either to an interrupt request not directly related to a previously executed instruction or to an exception related to a previously executed instruction. Interrupt requests may be either internal or external. An interrupt request can be induced by the assertion of a signal not directly related to any particular virtual processor or memory state, for example, the assertion of an "I/O done" signal.

A disrupting trap related to an earlier instruction causing an exception is similar to a deferred trap, in that it occurs after instructions following the trap-inducing instruction have modified the processor or memory state. The difference is that the condition that caused the instruction to induce the disrupting trap may lead to unrecoverable errors, since the implementation may not preserve the necessary state. An example is an ECC data-access error reported after the corresponding load instruction has completed.

Disrupting trap conditions should persist until the corresponding trap is taken.

Among the actions that trap-handler software might take after a disrupting trap are these:

- Use RETRY to return to the instruction at which the trap was invoked (PC ← old PC, nPC ← old nPC).
- Terminate the program or process associated with the trap.

### 7.2.4 Reset Traps

A *reset trap* occurs when supervisor software or the implementation's hardware determines that the machine must be reset to a known state. Reset traps differ from disrupting traps in that trap handler software for resets is never expected to resume execution of the program that was running when the reset trap occurred.

**IMPL. DEP. #37:** Some of a virtual processor's behavior during a reset trap is implementation dependent. See *RED_state Trap Processing* on page 153 for details.

The following reset traps are defined for SPARC V9:

- **Software-initiated reset (SIR)** — Initiated by software by executing the SIR instruction.
- **Power-on reset (POR)** — Initiated when power is applied (or reapplied) to the virtual processor.
- **Watchdog reset (WDR**) — Initiated in response to watchdog timer overflow or entry into error_state (impl. dep. #254).

- **Externally initiated reset (XIR)** — Initiated in response to an external signal. This reset trap is normally used for critical system events, such as power failure.

## 7.2.5 Uses of the Trap Categories

The SPARC V9 *trap model* makes the following stipulations:

1. Reset traps, except *software_initiated_reset* traps, occur asynchronously to program execution.

2. When recovery from an exception can affect the interpretation of subsequent instructions, such exceptions shall be precise. These exceptions are:
   - *software_initiated_reset*
   - *instruction_access_exception*
   - *privileged_action*
   - *privileged_opcode*
   - *trap_instruction*
   - *instruction_access_error*
   - *clean_window*
   - *fp_disabled*
   - *LDDF_mem_address_not_aligned*
   - *STDF_mem_address_not_aligned*
   - *LDQF_mem_address_not_aligned* (not used in SPARC JPS2)
   - *STQF_mem_address_not_aligned* (not used in SPARC JPS2)
   - *tag_overflow*
   - *spill_n_normal*
   - *spill_n_other*
   - *fill_n_normal*
   - *fill_n_other*

   See 7.6, *Exception and Interrupt Descriptions* on page 160 for other precise traps.

3. Exceptions that occur as the result of program execution are precise in a JPS2 virtual processor (impl. dep. #33).

4. An exception caused after the initial access of a multiple-access load or store instruction (for example, load/store doubleword or quadword, block load, block store, `LDSTUB`, `CASA`, `CASXA`, or `SWAP`) that causes a catastrophic exception may be precise, deferred, or disrupting. Thus, a trap due to the second memory access can occur after the processor or memory state has been modified by the first access.

5. Implementation-dependent catastrophic exceptions may cause precise, deferred, or disrupting traps (impl. dep. #31).

6. Exceptions caused by external events unrelated to the instruction stream, such as interrupts, are disrupting.

A deferred trap may occur one or more instructions after the trap-inducing instruction is dispatched.

## 7.3 Trap Control

Several registers control how any given trap is processed, for example:

- The interrupt enable (`ie`) field in `PSTATE` and the Processor Interrupt Level (`PIL`) register control interrupt processing.
- The enable floating-point unit (`fef`) field in `FPRS`, the floating-point unit enable (`pef`) field in `PSTATE`, and the trap enable mask (`tem`) in the FSR control floating-point traps.
- The `TL` register, which contains the current level of trap nesting, controls whether a trap causes entry to `execute_state`, `RED_state`, or `error_state`.
- `PSTATE.tle` determines whether implicit data accesses in the trap routine will be performed with the big- or little-endian byte order.

### 7.3.1 `PIL` Control

Between the execution of instructions, the IU prioritizes the outstanding exceptions and interrupt requests. At any given time, only the highest priority exception or interrupt request is taken as a trap. When there are multiple outstanding exceptions or interrupt requests, SPARC V9 assumes that lower-priority interrupt requests will persist and lower-priority exceptions will recur if an exception-causing instruction is reexecuted.

For interrupt requests, the IU compares the interrupt request level against the Processor Interrupt Level (`PIL`) register. If the interrupt request level is greater than `PIL`, then the virtual processor takes the interrupt request trap, assuming there are no higher-priority exceptions outstanding.

**IMPL. DEP. #34:** How quickly a virtual processor responds to an interrupt request is implementation dependent.

### 7.3.2 FSR.tem Control

The occurrence of floating-point traps of type *IEEE_754_exception* can be controlled with the user-accessible trap enable mask (tem) field of the FSR. If a particular bit of tem is 1, the associated *IEEE_754_exception* can cause an *fp_exception_ieee_754* trap.

If a particular bit of tem is 0, the associated *IEEE_754_exception* does not cause an *fp_exception_ieee_754* trap. Instead, the occurrence of the exception is recorded in the FSR's accrued exception field (aexc).

If an *IEEE_754_exception* results in an *fp_exception_ieee_754* trap, then the destination f register, fcc*n*, and aexc fields remain unchanged. However, if an *IEEE_754_exception* does not result in a trap, then the f register, fcc*n*, and aexc fields are updated to their new values.

## 7.4 Trap-Table Entry Addresses

Privileged software initializes the trap base address (TBA) register to the upper 49 bits of the trap-table base address. Bit 14 of the vector address (the tl > 0 field) is set based on the value of tl just before the trap is taken; that is, to 0 if tl = 0 and to 1 if tl > 0. Bits 13–5 of the trap vector address are the contents of the TT register. The lowest five bits of the trap address, bits 4–0, are always 0 (hence, each trap-table entry is at least $2^5$ or 32 bytes long). Each entry in the trap table may contain the first eight instructions of the corresponding trap handler. FIGURE 7-1 illustrates the trap vector address.

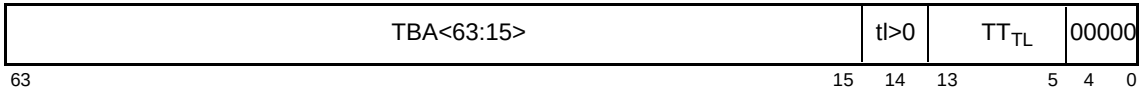



**FIGURE 7-1** Trap Vector Address

## 7.4.1 Trap Table Organization

The trap table layout is as illustrated in FIGURE 7-2.

| Value of tl Before the Trap | Trap Table Contents | Trap Type |
|---|---|---|
| tl = 0 | Hardware traps | $000_{16}$–$07F_{16}$ |
| | *Spill/fill* traps | $080_{16}$–$0FF_{16}$ |
| | Software traps | $100_{16}$–$7F_{16}$ |
| | *Reserved* | $180_{16}$–$1FF_{16}$ |
| tl > 0 | Hardware traps | $200_{16}$–$27F_{16}$ |
| | *Spill/fill* traps | $280_{16}$–$2FF_{16}$ |
| | Software traps | $300_{16}$–$37F_{16}$ |
| | *Reserved* | $380_{16}$–$3FF_{16}$ |

**FIGURE 7-2** Trap Table Layout

The trap table for `tl` = 0 comprises 512 thirty-two-byte entries; the trap table for `tl` > 0 comprises 512 more 32-byte entries. Therefore, the total size of a full trap table is 512 × 32 × 2, or 32 Kbytes. However, if privileged software does not use software traps (`Tcc` instructions) at `tl` > 0, the table can be made 24 Kbytes long.

## 7.4.2 Trap Type (`TT`)

When a normal trap occurs, a value that uniquely identifies the trap is written into the current 9-bit `TT` register (`TT[tl]`) by hardware. Control is then transferred into the trap table to an address formed by the `TBA` register (`tl>0`) and `TT[tl]` (see *Trap Base Address (TBA) Register* on page 82).

---

**Note –** The trap type for the *clean_window* exception is $024_{16}$. Three subsequent trap vectors ($025_{16}$–$027_{16}$) are reserved to allow for an inline (branchless) trap handler. Three subsequent trap vectors are reserved for each *spill*/*fill* vector, to allow for an inline (branchless) trap handler.

The spill/fill, *clean_window*, and MMU-related traps (*fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection*) trap types are spaced such that their trap-table entries are 128 bytes (32 instructions) long in SPARC JPS2. This length allows the complete code for one spill/fill routine, a *clean_window* routine, or a normal MMU miss handling routine to reside in one trap-table entry.

---

When a RED_state trap occurs, the TT register is set as described in *RED_state* on page 131. Control is then transferred into the RED_state trap table to an address formed by the RSTVaddr and an offset depending on the condition.

TT values $000_{16}$–$0FF_{16}$ are reserved for hardware traps. TT values $100_{16}$–$17F_{16}$ are reserved for software traps (traps caused by execution of a Tcc instruction). TT values $180_{16}$–$1FF_{16}$ are reserved for future uses.

**IMPL. DEP. #35:** TT values $070_{16}$ to $07F_{16}$ are reserved for implementation-dependent exceptions. The existence of *implementation_dependent_n* traps and whether any that do exist are precise, deferred, or disrupting is implementation dependent. TT values $060_{16}$ through $06F_{16}$ are defined for JPS2 processors and $070_{16}$ through $07F_{16}$ remain implementation-dependent; see TABLE 7-3 and Appendix C, *Implementation Dependencies*.

The assignment of TT values to traps is shown in TABLE 7-3; TABLE 7-4 lists the traps in priority order. Traps marked with an open bullet (❍) are optional and possibly implementation dependent. Traps marked with a closed bullet (●) are mandatory; that is, hardware must detect and trap these exceptions and interrupts and must set the defined TT values. In the table, ag = alternate globals, mg = MMU globals, and ig = interrupt globals. "-*NA*-" means "not applicable".

**TABLE 7-3** Exception and Interrupt Requests, by TT Value *(1 of 3)*

| SPARC V9 M/O | JPS2 M/O | Exception or Interrupt Request | TT | Global Register Set | Priority |
|---|---|---|---|---|---|
| ● | ● | Reserved | $000_{16}$ | -NA- | -NA- |
| ● | ● | *power_on_reset* | $001_{16}$ | AG | 0 |
| ❍ | ● | *watchdog_reset* | $002_{16}$ | AG | 1 |
| ❍ | ● | *externally_initiated_reset* | $003_{16}$ | AG | 1 |
| ● | ● | *software_initiated_reset* | $004_{16}$ | AG | 1 |
| ● | ● | *RED_state_exception* | † | AG | 1 |
| ● | ● | Reserved | $006_{16}$–$007_{16}$ | -NA- | -NA- |
| ● | ● | *instruction_access_exception* | $008_{16}$ | MG | 5 |
| ❍ | — | *instruction_access_MMU_miss*‡ | $009_{16}$ | — | 2 |
| ❍ | ● | *instruction_access_error* | $00A_{16}$ | AG | 3 |
| ● | ● | Reserved | $00B_{16}$–$00F_{16}$ | -NA- | -NA- |
| ● | ● | *illegal_instruction* | $010_{16}$ | AG | 7 |
| ● | ● | *privileged_opcode* | $011_{16}$ | AG | 6 |
| ❍ | ❍ | *unimplemented_LDD* | $012_{16}$ | AG | 6 |
| ❍ | ❍ | *unimplemented_STD* | $013_{16}$ | AG | 6 |
| ● | ● | Reserved | $014_{16}$–$01F_{16}$ | -NA- | -NA- |

**TABLE 7-3** Exception and Interrupt Requests, by TT Value *(2 of 3)*

| SPARC V9 M/O | JPS2 M/O | Exception or Interrupt Request | TT | Global Register Set | Priority |
|---|---|---|---|---|---|
| ● | ● | *fp_disabled* | $020_{16}$ | AG | 8 |
| ❍ | ● | *fp_exception_ieee_754* | $021_{16}$ | AG | 11 |
| ❍ | ● | *fp_exception_other* | $022_{16}$ | AG | 11 |
| ● | ● | *tag_overflow* | $023_{16}$ | AG | 14 |
| ❍ | ● | *clean_window* | $024_{16}^{\diamond}$ | AG | 10 |
| ● | ● | *division_by_zero* | $028_{16}$ | AG | 15 |
| ❍ | ❍ | *internal_processor_error* | $029_{16}$ | impl. dep. | impl. dep |
| ● | ● | Reserved | $02A_{16}$–$02F_{16}$ | -NA- | -NA- |
| ● | ● | *data_access_exception* | $030_{16}$ | impl. dep. | 12 |
| ❍ | — | *data_access_MMU_miss*‡ | $031_{16}$ | — | 12 |
| ❍ | ● | *data_access_error* | $032_{16}$ | AG | 12 |
| ❍ | — | *data_access_protection*‡ | $033_{16}$ | — | 12 |
| ● | ● | *mem_address_not_aligned* | $034_{16}$ | AG | 10 |
| ❍ | ● | *LDDF_mem_address_not_aligned* (impl. dep. #109) | $035_{16}$ | AG | 10 |
| ❍ | ● | *STDF_mem_address_not_aligned* (impl. dep. #110) | $036_{16}$ | AG | 10 |
| ● | ● | *privileged_action* | $037_{16}$ | AG | 11 |
| ❍ | ❍ | *LDQF_mem_address_not_aligned* (impl. dep. #111) | $038_{16}$ | AG | 10 |
| ❍ | ❍ | *STQF_mem_address_not_aligned* (impl. dep. #112) | $039_{16}$ | AG | 10 |
| ● | ● | Reserved | $03A_{16}$–$03F_{16}$ | -NA- | -NA- |
| ❍ | ❍ | *async_data_error* | $040_{16}$ | impl. dep. | 2 |
| ● | ● | *interrupt_level_n* (*n* = 1–15) | $041_{16}$–$04F_{16}$ | AG | 32-*n* |
| ● | ● | Reserved | $050_{16}$–$05F_{16}$ | -NA- | -NA- |
| ❍ | ● | *interrupt_vector* | $060_{16}$ | IG | 16 |
| ❍ | ● | *PA_watchpoint* | $061_{16}$ | AG | 12 |
| ❍ | ● | *VA_watchpoint* | $062_{16}$ | AG | 11 |
| ❍ | ● | *ECC_error* | $063_{16}$ | AG | 33 |
| ❍ | ● | *fast_instruction_access_MMU_miss* | $064_{16}^{\diamond}$ | MG | 2 |
| ❍ | ● | *fast_data_access_MMU_miss* | $068_{16}^{\diamond}$ | MG | 12 |
| ❍ | ● | *fast_data_access_protection* | $06C_{16}^{\diamond}$ | MG | 12 |
| ❍ | ❍ | *implementation_dependent_exception_n* (impl. dep. #35) | $070_{16}$–$07F_{16}$ | impl. dep. | impl. dep. |

**TABLE 7-3** Exception and Interrupt Requests, by TT Value *(3 of 3)*

| SPARC V9 M/O | JPS2 M/O | Exception or Interrupt Request | TT | Global Register Set | Priority |
|---|---|---|---|---|---|
| ● | ● | *spill_n_normal* (*n* = 0–7) | $080_{16}$–$09F_{16}$ | AG | 9 |
| ● | ● | *spill_n_other* (*n* = 0–7) | $0A0_{16}$–$0BF_{16}$ | AG | 9 |
| ● | ● | *fill_n_normal* (*n* = 0–7) | $0C0_{16}$–$0DF_{16}$ | AG | 9 |
| ● | ● | *fill_n_other* (*n* = 0–7) | $0E0_{16}$–$0FF_{16}$ | AG | 9 |
| ● | ● | *trap_instruction* | $100_{16}$–$17F_{16}$ | AG | 16 |
| ● | ● | Reserved | $180_{16}$–$1FF_{16}$ | -NA- | -NA- |

† *RED_state_exception* uses the trap vector entry reserved for trap type $005_{16}$, but the trap type recorded in TT is the trap type of the original exception that triggered *RED_state_exception*.

‡ This exception type is not used in JPS2.

◊ The trap vector entry (32 bytes) for this trap type plus the next three trap types (total of 128 bytes) are permanently reserved for this exception.

**TABLE 7-4** Exception and Interrupt Requests, by Priority (0 = Highest) *(1 of 3)*

| SPARC V9 M/O | JPS2 M/O | Exception or Interrupt Request | TT | Global Register Set | Priority▽ |
|---|---|---|---|---|---|
| ● | ● | *power_on_reset* (POR) | $001_{16}$ | AG | 0 |
| ❍ | ● | *externally_initiated_reset* (XIR) | $003_{16}$ | AG | 1 |
| ❍ | ● | *watchdog_reset* (WDR) | $002_{16}$ | AG | 1 |
| ● | ● | *software_initiated_reset* (SIR) | $004_{16}$ | AG | 1 |
| ● | ● | *RED_state_exception* | † | AG | 1 |
| ❍ | ❍ | *instruction_access_MMU_miss*‡ | $009_{16}$ | — | 2 |
| ❍ | ❍ | *async_data_error* | $040_{16}$ | impl. dep. | 2 |
| ❍ | ● | *fast_instruction_access_MMU_miss* | $064_{16}$◊ | MG | 2 |
| ❍ | ● | *instruction_access_error* | $00A_{16}$ | AG | 3 |
| ● | ● | *instruction_access_exception* | $008_{16}$ | MG | 5 |
| ● | ● | *privileged_opcode* | $011_{16}$ | AG | 6 |
| ❍ | ❍ | *unimplemented_LDD* | $012_{16}$ | AG | 6 |
| ❍ | ❍ | *unimplemented_STD* | $013_{16}$ | AG | 6 |
| ● | ● | *illegal_instruction* | $010_{16}$ | AG | 7 |
| ● | ● | *fp_disabled* | $020_{16}$ | AG | 8 |

**TABLE 7-4** Exception and Interrupt Requests, by Priority (0 = Highest) *(2 of 3)*

| SPARC V9 M/O | JPS2 M/O | Exception or Interrupt Request | TT | Global Register Set | Priority[▽] |
|---|---|---|---|---|---|
| ● | ● | *spill_n_normal* (*n* = 0–7) | $080_{16}$–$09F_{16}$ | AG | 9 |
| ● | ● | *spill_n_other* (*n* = 0–7) | $0A0_{16}$–$0BF_{16}$ | AG | 9 |
| ● | ● | *fill_n_normal* (*n* = 0–7) | $0C0_{16}$–$0DF_{16}$ | AG | 9 |
| ● | ● | *fill_n_other* (*n* = 0–7) | $0E0_{16}$–$0FF_{16}$ | AG | 9 |
| ❍ | ● | *clean_window* | $024_{16}$[◊] | AG | 10 |
| ❍ | ● | *LDDF_mem_address_not_aligned* (impl. dep. #109) | $035_{16}$ | AG | 10 |
| ❍ | ● | *STDF_mem_address_not_aligned* (impl. dep. #110) | $036_{16}$ | AG | 10 |
| ❍ | ❍ | *LDQF_mem_address_not_aligned* (impl. dep. #111) | $038_{16}$ | AG | 10 |
| ❍ | ❍ | *STQF_mem_address_not_aligned* (impl. dep. #112) | $039_{16}$ | AG | 10 |
| ● | ● | *mem_address_not_aligned* | $034_{16}$ | AG | 10 |
| ❍ | ● | *fp_exception_ieee_754* | $021_{16}$ | AG | 11 |
| ❍ | ● | *fp_exception_other* | $022_{16}$ | AG | 11 |
| ● | ● | *privileged_action* | $037_{16}$ | AG | 11 |
| ❍ | ● | *VA_watchpoint* | $062_{16}$ | AG | 11 |
| ● | ● | *data_access_exception* | $030_{16}$ | MG | 12 |
| ❍ | ● | *fast_data_access_MMU_miss* | $068_{16}$[◊] | MG | 12 |
| ❍ | ❍ | *data_access_MMU_miss*[‡] | $031_{16}$ | — | 12 |
| ❍ | ● | *data_access_error* | $032_{16}$ | AG | 12 |
| ❍ | ● | *PA_watchpoint* | $061_{16}$ | AG | 12 |
| ❍ | ● | *fast_data_access_protection* | $06C_{16}$[◊] | MG | 12 |
| ❍ | ❍ | *data_access_protection*[‡] | $033_{16}$ | — | 12 |
| ● | ● | *tag_overflow* | $023_{16}$ | AG | 14 |
| ● | ● | *division_by_zero* | $028_{16}$ | AG | 15 |
| ● | ● | *trap_instruction* | $100_{16}$–$17F_{16}$ | AG | 16 |
| ❍ | ● | *interrupt_vector* | $060_{16}$ | IG | 16 |
| ● | ● | *interrupt_level_n* (*n* = 1–15) | $041_{16}$–$04F_{16}$ | AG | 32-*n* |

**TABLE 7-4** Exception and Interrupt Requests, by Priority (0 = Highest) *(3 of 3)*

| SPARC V9 M/O | JPS2 M/O | Exception or Interrupt Request | TT | Global Register Set | Priority[▽] |
|---|---|---|---|---|---|
| ❍ | ● | *ECC_error* | $063_{16}$ | AG | 33 |
| ❍ | ❍ | *implementation_dependent_exception_n* (impl. dep. #35) | $070_{16}$–$07F_{16}$ | impl. dep. | *impl. dep.* |
| ❍ | ❍ | *internal_processor_error* | $029_{16}$ | impl. dep. | impl. dep |

[†] *RED_state_exception* uses the trap vector entry reserved for trap type $005_{16}$, but the trap type recorded in TT is the trap type of the original exception that triggered *RED_state_exception*.

[‡] This exception type is not used in JPS2

[◊] The trap vector entry (32 bytes) for this trap type plus the next three trap types (total of 128 bytes) are permanently reserved for this exception.

[▽] Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36 on page 146), including relative priorities within a given priority level.

### 7.4.2.1 Trap Type for Spill/Fill Traps

The trap type for window *spill*/*fill* traps is determined on the basis of the contents of the OTHERWIN and WSTATE registers as described below and shown in FIGURE 7-3.

| Bit | Field | Description |
|---|---|---|
| 8:6 | spill_or_fill | $010_2$ for spill traps; $011_2$ for fill traps |
| 5 | other | (OTHERWIN ≠ 0) |
| 4:2 | wtype | If (OTHER) then WSTATE.OTHER; else WSTATE.NORMAL |

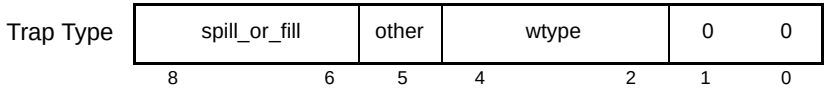



**FIGURE 7-3** Trap Type Encoding for Spill/Fill Traps

## 7.4.3 Trap Priorities

TABLE 7-3 on page 141 and TABLE 7-4 on page 143 show the assignment of traps to TT values and the relative priority of traps and interrupt requests. A trap priority is an ordinal number, with 0 indicating the highest priority and greater priority numbers indicating decreasing priority; that is, if $X < Y$, a pending exception or interrupt request with priority $X$ is taken instead of a pending exception or interrupt request with priority $Y$.

**IMPL. DEP. #36:** The relative priorities of traps defined in SPARC JPS2 are fixed. However, the absolute priorities of those traps are implementation dependent (because a future version of the architecture may define new traps). The priorities (both absolute and relative) of any additional traps are implementation dependent.

However, the TT values for the exceptions and interrupt requests shown in TABLE 7-3 and TABLE 7-4 must remain the same for every implementation.

The trap priorities given above always need to be considered in light of how the virtual processor actually issues and executes instructions. For example, if an *instruction_access_error* occurs (priority 3), it will be taken even if the instruction is an SIR (priority 1). This situation occurs because the virtual processor gets the *instruction_access_error* during instruction fetch and never actually issues or executes the instruction, so the SIR instruction is never seen by the execution units of the virtual processor. This is an obvious case, but there are other more subtle cases.

## 7.5 Trap Processing

The virtual processor's action during trap processing depends on the trap type, the current level of trap nesting (given in the TL register), and PSTATE. When a trap occurs, the mapping of the visible set of the global registers changes to one of three sets of trap global register—MMU globals, interrupt globals, or alternate globals—based on the type of trap.

The following traps use RED_state trap processing:

- Reset requests
- Catastrophic errors
- Traps taken when TL = MAXTL – 1
- Traps taken when the virtual processor is in RED_state

All other traps use normal trap processing.

During normal operation, the virtual processor is in execute_state. It processes traps in execute_state and continues.

When a non-reset trap or software-initiated reset (SIR) occurs with TL = MAXTL, there are no more levels on the trap stack, so the virtual processor enters error_state and halts. To avoid this catastrophic failure, SPARC V9 provides the RED_state virtual processor state. Traps processed in RED_state use a special trap vector and a special trap-vectoring algorithm. RED_state vectoring and the setting of the TT value for RED_state traps are described in *RED_state Trap Table* on page 131.

Traps that occur with TL = MAXTL – 1 are processed in RED_state. In addition, reset traps are also processed in RED_state. Reset trap processing is described in *RED_state Trap Processing* on page 153. Finally, supervisor software can force the virtual processor into RED_state by setting the PSTATE.red bit to 1.

Once the virtual processor has entered RED_state, no matter how it got there, all subsequent traps are processed in RED_state until software returns the virtual processor to execute_state or a normal or SIR trap is taken when TL = MAXTL, which puts the virtual processor in error_state. TABLE 7-5, TABLE 7-6, and TABLE 7-7 describe the virtual processor mode and trap-level transitions involved in handling traps.

**TABLE 7-5** Trap Received While in execute_state

| | New State, After Receiving Trap Type | | | |
|---|---|---|---|---|
| **Original State** | **Non-reset Trap or Interrupt** | **POR** | **XIR, WDR (Impl. Dep.)** | **SIR** |
| execute_state<br>TL < MAXTL – 1 | execute_state<br>TL ← TL + 1 | RED_state<br>TL = MAXTL | RED_state<br>TL ← TL + 1 | RED_state<br>TL ← TL + 1 |
| execute_state<br>TL = MAXTL – 1 | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL |
| execute_state†<br>TL = MAXTL | error_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | error_state<br>TL = MAXTL |

† This state occurs when software changes TL to MAXTL and does not set PSTATE.RED, or if it clears PSTATE.RED while at MAXTL.

**TABLE 7-6** Trap Received While in RED_state

| | New State, After Receiving Trap Type | | | |
|---|---|---|---|---|
| **Original State** | **Non-reset Trap or Interrupt** | **POR** | **XIR, WDR (Impl. Dep.)** | **SIR** |
| RED_state<br>TL < MAXTL – 1 | RED_state<br>TL ← TL + 1 | RED_state<br>TL = MAXTL | RED_state<br>TL ← TL + 1 | RED_state<br>TL ← TL + 1 |
| RED_state<br>TL = MAXTL – 1 | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL |
| RED_state<br>TL = MAXTL | error_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | error_state<br>TL = MAXTL |

**TABLE 7-7** Reset Received While in error_state

| | New State, After Receiving Trap Type | | | |
|---|---|---|---|---|
| **Original State** | **Non-reset Trap or Interrupt** | **POR** | **XIR, WDR (Impl. Dep.)** | **SIR** |
| error_state<br>TL < MAXTL – 1 | — | RED_state<br>TL = MAXTL | RED_state<br>TL ← TL + 1 | — |
| error_state<br>TL = MAXTL – 1 | — | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | — |
| error_state<br>TL = MAXTL | — | RED_state<br>TL = MAXTL | RED_state<br>TL = MAXTL | — |

The virtual processor does not recognize interrupts while it is in error_state.

A trap other than a fast MMU trap (see Section 7.5.1.2 on page 150) or an interrupt vector trap (see Section 7.5.1.3 on page 151) causes the following state changes to occur:

- If the virtual processor is already in RED_state, the new trap is processed in RED_state unless TL = MAXTL. See *Non-reset Traps When the Virtual Processor Is in RED_state* on page 159.
- If the virtual processor is in execute_state and the trap level is one less than its maximum value, that is, TL = MAXTL–1, then the virtual processor enters RED_state. See *RED_state* on page 131 and *Non-reset Traps with TL = MAXTL – 1* on page 153.
- If the virtual processor is in either execute_state or RED_state and the trap level is already at its maximum value, that is, TL = MAXTL, then the virtual processor enters error_state. See *error_state* on page 134.

Otherwise, the trap uses normal trap processing, described in the following section.

## 7.5.1 Normal Trap Processing

Normal traps consist of all non-RED_state traps. The processing of two special classes of normal traps, fast MMU traps and interrupt vector traps is described in sections 7.5.1.2 and 7.5.1.3, respectively. Processing of the remainder of normal traps is described in the following section, 7.5.1.1.

### 7.5.1.1 Processing of Most Normal Traps

During processing of normal traps other than fast MMU or interrupt vector traps, the following state changes occur:

- The trap level is set. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack.

  TL ← TL + 1

- Existing state is preserved.

```
TSTATE[TL].CCR    ←CCR
TSTATE[TL].ASI    ←ASI
TSTATE[TL].PSTATE←PSTATE
TSTATE[TL].CWP    ←CWP
TPC[TL]           ←PC
TNPC[TL]          ←nPC
```

- The trap type is preserved.

  TT[TL] ← the trap type

- The PSTATE register is updated to a predefined state.

```
PSTATE.mm   is unchanged
PSTATE.red  ←0
PSTATE.pef  ←1 (FPU is present)
PSTATE.am   ←0 (address masking is turned off)
PSTATE.priv←1 (the virtual processor enters privileged mode)
PSTATE.ie   ←0 (interrupts are disabled)
PSTATE.ag   ←1 (global regs are replaced with alternate globals)
PSTATE.mg   ←0 (MMU globals are disabled)
PSTATE.ig   ←0 (interrupt globals are disabled)
PSTATE.cle  ←PSTATE.tle (set endian mode for traps)
PSTATE.tle  is unchanged
```

- For a register-window trap (*clean_window*, window spill, or window fill) only, CWP is set to point to the register window that must be accessed by the trap-handler software, that is:
  - If TT[TL] = $024_{16}$ (a *clean_window* trap), then CWP ← CWP + 1.
  - If ($080_{16}$ ≤ TT[TL] ≤ $0BF_{16}$) (window spill trap), then CWP ← CWP + CANSAVE + 2.
  - If ($0C0_{16}$ ≤ TT[TL] ≤ $0FF_{16}$) (window fill trap), then CWP ← CWP – 1.

  For non-register-window traps, CWP is not changed.

- Control is transferred into the trap table.

  PC ← TBA<63:15> ▯ (TL>0) ▯ TT[TL] ▯ 0 0000

  nPC← TBA<63:15> ▯ (TL>0) ▯ TT[TL] ▯ 0 0100

  where "(TL>0)" is 0 if TL = 0, and 1 if TL > 0.

  Interrupts are ignored as long as PSTATE.ie = 0.

**Note –** State in TPC[*n*], TNPC[*n*], TSTATE[*n*], and TT[*n*] is only changed autonomously by the virtual processor when a trap is taken while TL = *n* –1; however, software can change any of these values with a WRPR instruction when TL = *n*.

### 7.5.1.2 Fast MMU Trap Processing

Fast MMU traps (*fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection*) cause the following state changes to occur:

- If the virtual processor is already in RED_state, the new trap is processed in RED_state unless TL = MAXTL. See *Non-reset Traps When the Virtual Processor Is in RED_state* on page 159.
- If the virtual processor is in execute_state and the trap level is one less than its maximum value, that is, TL = MAXTL–1, then the virtual processor enters RED_state. See *RED_state* on page 131 and *Non-reset Traps with TL = MAXTL – 1* on page 153.
- If the virtual processor is in either execute_state or RED_state and the trap level is already at its maximum value, that is, TL = MAXTL, then the virtual processor enters error_state. See *error_state* on page 134.

Otherwise, the trap uses normal trap processing, and the following state changes occur:

- The trap level is set. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack.

  TL ← TL + 1

- Existing state is preserved.

```
TSTATE[TL].CCR    ←CCR
TSTATE[TL].ASI    ←ASI
TSTATE[TL].PSTATE←PSTATE
TSTATE[TL].CWP    ←CWP
TPC[TL]           ←PC
TNPC[TL]          ←nPC
```

- The trap type is preserved.

  TT[TL] ← the trap type

- The PSTATE register is updated to a predefined state.

```
PSTATE.mm   is unchanged
PSTATE.red  ←0
PSTATE.pef  ←1 (FPU is present)
PSTATE.am   ←0 (address masking is turned off)
PSTATE.priv←1 (the virtual processor enters privileged mode)
PSTATE.ie   ←0 (interrupts are disabled)
PSTATE.ag   ←0 (alternate globals are disabled)
PSTATE.mg   ←1 (global regs are replaced with MMU globals)
PSTATE.ig   ←0 (interrupt globals are disabled)
PSTATE.cle  ←PSTATE.tle (set endian mode for traps)
PSTATE.tle  is unchanged
```

- For non-register-window traps, CWP is not changed.
- Control is transferred into the trap table.

PC ← TBA<63:15> || (TL>0) || TT[TL] || 0 0000

nPC← TBA<63:15> || (TL>0) || TT[TL] || 0 0100

where "(TL>0)" is 0 if TL = 0, and 1 if TL > 0.

Interrupts are ignored as long as PSTATE.ie = 0.

---

**Note –** State in TPC[*n*], TNPC[*n*], TSTATE[*n*], and TT[*n*] is only changed autonomously by the virtual processor when a trap is taken while TL = *n* – 1; however, software can change any of these values with a WRPR instruction when TL = *n*.

---

### 7.5.1.3 Interrupt Vector Trap Processing

An *interrupt_vector* trap causes the following state changes to occur:

- If the virtual processor is already in RED_state, the new trap is processed in RED_state unless tl = MAXTL. See *Non-reset Traps When the Virtual Processor Is in RED_state* on page 159.
- If the virtual processor is in execute_state and the trap level is one less than its maximum value, that is, TL = MAXTL – 1, the virtual processor enters RED_state. See *RED_state* on page 131 and *Non-reset Traps with TL = MAXTL – 1* on page 153.
- If the virtual processor is in either execute_state or RED_state and the trap level is already at its maximum value, that is, TL = MAXTL, then the virtual processor enters error_state. See *error_state* on page 134.

Otherwise, the trap uses normal trap processing, and the following state changes occur:

- The trap level is set. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack.

  TL ← TL + 1

- Existing state is preserved.

```
TSTATE[TL].CCR    ←CCR
TSTATE[TL].ASI    ←ASI
TSTATE[TL].PSTATE←PSTATE
TSTATE[TL].CWP    ←CWP
TPC[TL]           ←PC
TNPC[TL]          ←nPC
```

- The trap type is preserved.

  TT[TL] ← the trap type

- The PSTATE register is updated to a predefined state.

```
PSTATE.mm   is unchanged
PSTATE.red  ←0
PSTATE.pef  ←1 (FPU is present)
PSTATE.am   ←0 (address masking is turned off)
PSTATE.priv←1 (the virtual processor enters privileged mode)
PSTATE.ie   ←0 (interrupts are disabled)
PSTATE.ag   ←0 (alternate globals are disabled)
PSTATE.mg   ←0 (MMU globals are disabled)
PSTATE.ig   ←1 (global regs are replaced with interrupt globals)
PSTATE.cle  ←PSTATE.tle (set endian mode for traps)
PSTATE.tle  is unchanged
```

- For non-register-window traps, CWP is not changed.
- Control is transferred into the trap table.

  PC ← TBA<63:15> ▯ (TL>0) ▯ TT[TL] ▯ 0 0000

  nPC← TBA<63:15> ▯ (TL>0) ▯ TT[TL] ▯ 0 0100

  where "(TL>0)" is 0 if TL = 0, and 1 if TL > 0.

  Interrupts are ignored as long as PSTATE.ie = 0.

---

**Note –** State in TPC[*n*], TNPC[*n*], TSTATE[*n*], and TT[*n*] is only changed autonomously by the virtual processor when a trap is taken while TL = *n* – 1; however, software can change any of these values with a WRPR instruction when TL = *n*.

---

## 7.5.2 RED_state Trap Processing

The following conditions invoke RED_state trap processing:

- Traps taken with TL = MAXTL – 1
- Power-on reset traps
- Watchdog reset traps
- Externally initiated reset traps
- Software-initiated reset traps
- Traps taken when the virtual processor is already in RED_state

**IMPL. DEP. #38:** Implementation-dependent registers may or may not be affected by the various reset traps.

### 7.5.2.1 Non-reset Traps with TL = MAXTL – 1

Non-reset traps that occur when TL = MAXTL – 1 are processed in RED_state.

**IMPL. DEP. #307a:** When a trap due to *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, *fast_data_access_protection*, *data_access_exception*, or *instruction_access_exception* occurs and the trap is processed in RED_state (that is, either TL = MAXTL – 1 or PSTATE.red = 1), it is implementation-dependent whether PSTATE.ag or PSTATE.mg is set to 1 during the trap.

**IMPL. DEP. #307b:** When an *interrupt_vector* trap occurs and the trap is processed in RED_state (that is, either TL = MAXTL – 1 or PSTATE.red = 1), an implementation may either set PSTATE.ag to 1 or set PSTATE.ig to 1.

The following state changes occur:

- The trap level is advanced.

  TL ← MAXTL

- Existing state is preserved.

```
TSTATE[TL].CCR    ←CCR
TSTATE[TL].ASI    ←ASI
TSTATE[TL].PSTATE←PSTATE
TSTATE[TL].CWP    ←CWP
TPC[TL]           ←PC
TNPC[TL]          ←nPC
```

- The trap type is preserved.

  TT[TL] ← the trap type

- The PSTATE register is set as follows:

PSTATE.mm ← $00_2$ (TSO)
PSTATE.red ← 1 (enter RED_state)
PSTATE.pef ← 1 (FPU is present)
PSTATE.am ← 0 (address masking is turned off)
PSTATE.priv ← 1 (the virtual processor enters privileged mode)
PSTATE.ie ← 0 (interrupts are disabled)
PSTATE.ag ← implementation dependent (impl. dep. #307)
PSTATE.mg ← implementation dependent (impl. dep. #307)
PSTATE.ig ← implementation dependent (impl. dep. #307)
PSTATE.cle ← PSTATE.tle (set endian mode for traps)
PSTATE.tle ← unchanged[1]

- For a register-window trap only, CWP is set to point to the register window that must be accessed by the trap-handler software, that is:

  If TT[TL] = $024_{16}$ (a *clean_window* trap), then CWP ← CWP + 1.

  If ($080_{16}$ ≤ TT[TL] ≤ $0BF_{16}$) (window *spill* trap), then CWP ← CWP + CANSAVE + 2.

  If ($0C0_{16}$ ≤ TT[TL] ≤ $0FF_{16}$) (window *fill* trap), then CWP ← CWP – 1.

  For non-register-window traps, CWP is not changed.

- Implementation-specific state changes, including:
  - State changes common to all JPS2 implementations (see Appendix O)
  - Implementation-specific state changes (see Appendix O of the JPS2 Extension documents)
- Control is transferred into the RED_state trap table. See *RED_state Trap Vector* on page 599 for further details of RSTVaddr.

  PC ← RSTVaddr<63:8> ▯ $1010\ 0000_2$

  nPC ← RSTVaddr<63:8> ▯ $1010\ 0100_2$

### 7.5.2.2 Power-On Reset (POR) Traps

POR traps occur when power is applied to the virtual processor. If the virtual processor is in error_state, a POR brings the virtual processor out of error_state and places it in RED_state. See Appendix O, *Reset, RED_state, and error_state* for further details. Virtual processor state is undefined after POR, except for the following:

- The trap level is set.

  TL ← MAXTL

- The trap type is set.

  TT[TL] ← $001_{16}$

1. Note that this was left unspecified in SPARC V9.

- The PSTATE register is set as follows:

  PSTATE.mm ←$00_2$ (TSO)
  PSTATE.red ←1 (enter RED_state)
  PSTATE.pef ←1 (FPU is present)
  PSTATE.am ←0 (address masking is turned off)
  PSTATE.priv ←1 (the virtual processor enters privileged mode)
  PSTATE.ie ←0 (interrupts are disabled)
  PSTATE.ag ←1 (global regs are replaced with alternate globals)
  PSTATE.mg ←0 (MMU globals are disabled)
  PSTATE.ig ←0 (interrupt globals are disabled)
  PSTATE.cle ←0 (big-endian mode for nontraps)
  PSTATE.tle ←0 (big-endian mode for traps)

- The TICK register is protected.

  TICK.npt ←1 (TICK unreadable by nonprivileged software)

  STICK.npt ←1 (STICK unreadable by nonprivileged software)

- Implementation-specific state changes, including:
  - State changes common to all JPS2 implementations (see Appendix O)
  - Implementation-specific state changes (see Appendix O of the JPS2 Extension documents)
- Control is transferred into the RED_state trap table.

  PC ← RSTVaddr<63:8> ▯ $0010\ 0000_2$

  nPC← RSTVaddr<63:8> ▯ $0010\ 0100_2$

For any reset when TL = MAXTL, for all $n$ < MAXTL, the values in TPC[$n$], TNPC[$n$], and TSTATE[$n$] are undefined.

See JPS2 Extension documents for more details.

### 7.5.2.3 Watchdog Reset (WDR) Traps

The WDR reset in SPARC JPS2 occurs when a watchdog timer overflows or to provide automatic recovery from error_state (impl. dep. #254). There are several causes of error_state entry (impl. dep. #39), including but not limited to SIR with TL = MAXTL and implementation-dependent watchdog timeout.

Virtual processor state is undefined after WDR, except for the following:

- The trap level is set.

  TL ← min (TL + 1, MAXTL)

- Existing state is preserved.

```
TSTATE[TL].CCR    ←CCR
TSTATE[TL].ASI    ←ASI
TSTATE[TL].PSTATE←PSTATE
TSTATE[TL].CWP    ←CWP
TPC[TL]           ←PC
TNPC[TL]          ←nPC
```

- The trap type is set.

  TT[TL] ← $002_{16}$

- The PSTATE register is set as follows:

```
PSTATE.mm   ←00₂ (TSO)
PSTATE.red  ←1 (enter RED_state)
PSTATE.pef  ←1 (FPU is present)
PSTATE.am   ←0 (address masking is turned off)
PSTATE.priv←1 (the virtual processor enters privileged mode)
PSTATE.ie   ←0 (interrupts are disabled)
PSTATE.ag   ←1 (global regs are replaced with alternate globals)
PSTATE.mg   ←0 (MMU globals are disabled)
PSTATE.ig   ←0 (interrupt globals are disabled)
PSTATE.cle  ←PSTATE.tle (set endian mode for traps)
PSTATE.tle  ←unchanged[1]
```

- Implementation-specific state changes, including:
  - State changes common to all JPS2 implementations (see Appendix O)
  - Implementation-specific state changes (see Appendix O of the JPS2 Extension documents)
- Control is transferred into the RED_state trap table.

  PC ← RSTVaddr<63:8> ▯ $0100\ 0000_2$

  nPC← RSTVaddr<63:8> ▯ $0100\ 0100_2$

For any reset when TL = MAXTL, for all $n$<MAXTL, the values in TPC[$n$], TNPC[$n$], and TSTATE[$n$] are undefined.

### 7.5.2.4 Externally Initiated Reset (XIR) Traps

XIR traps are initiated by an external signal. They behave like an interrupt that cannot be masked by IE = 0 or PIL. Typically, XIR is used for critical system events such as power failure, reset button pressed, failure of external components that does not require a WDR (which aborts operations), or systemwide reset in a multiprocessor. See Appendix O, *Reset, RED_state, and error_state* for further details.

The following state changes occur:

[1] Note that this was left unspecified in SPARC V9.

- The trap level is set.

  TL ← min (TL + 1, MAXTL)

- Existing state is preserved.

```
TSTATE[TL].CCR    ←CCR
TSTATE[TL].ASI    ←ASI
TSTATE[TL].PSTATE←PSTATE
TSTATE[TL].CWP    ←CWP
TPC[TL]           ←PC
TNPC[TL]          ←nPC
```

- The trap type is set.

  TT[TL] ← $003_{16}$

- The PSTATE register is set as follows:

```
PSTATE.mm  ←00₂ (TSO)
PSTATE.red ←1 (enter RED_state)
PSTATE.pef ←1 (FPU is present)
PSTATE.am  ←0 (address masking is turned off)
PSTATE.priv←1 (the virtual processor enters privileged mode)
PSTATE.ie  ←0 (interrupts are disabled)
PSTATE.ag  ←1 (global regs are replaced with alternate globals)
PSTATE.mg  ←0 (MMU globals are disabled)
PSTATE.ig  ←0 (interrupt globals are disabled)
PSTATE.cle ←PSTATE.tle (set endian mode for traps)
PSTATE.tle ←unchanged[1]
```

- Implementation-specific state changes, including:
  - State changes common to all JPS2 implementations (see Appendix O)
  - Implementation-specific state changes (see Appendix O of the JPS2 Extension documents)
- Control is transferred into the RED_state trap table.

  PC ← RSTVaddr<63:8> ▯ $0110\ 0000_2$

  nPC← RSTVaddr<63:8> ▯ $0110\ 0100_2$

For any reset when TL = MAXTL, for all $n$<MAXTL, the values in TPC[$n$], TNPC[$n$], and TSTATE[$n$] are undefined.

See *Externally Initiated Reset (XIR)* on page 598 and Appendix O of the JPS2 Extension documents for more information.

1. Note that this was left unspecified in SPARC V9.

### 7.5.2.5 Software-Initiated Reset (SIR) Traps

SIR traps are initiated by execution of an SIR instruction in privileged mode. Supervisor software uses the SIR trap as a panic operation or a metasupervisor trap. See Appendix O, *Reset, RED_state, and error_state* for further details.

The following state changes occur:

- If TL = MAXTL, then enter error_state. Otherwise, do the following:
- The trap level is set.

  TL ← TL + 1

- Existing state is preserved.

```
TSTATE[TL].CCR     ←CCR
TSTATE[TL].ASI     ←ASI
TSTATE[TL].PSTATE  ←PSTATE
TSTATE[TL].CWP     ←CWP
TPC[TL]            ←PC
TNPC[TL]           ←nPC
```

- The trap type is set.

  $TT[TL] \leftarrow 04_{16}$

- The PSTATE register is set as follows:

```
PSTATE.mm   ←00₂ (TSO)
PSTATE.red  ←1 (enter RED_state)
PSTATE.pef  ←1 (FPU is present)
PSTATE.am   ←0 (address masking is turned off)
PSTATE.priv ←1 (the virtual processor enters privileged mode)
PSTATE.ie   ←0 (interrupts are disabled)
PSTATE.ag   ←1 (global regs are replaced with alternate globals)
PSTATE.mg   ←0 (MMU globals are disabled)
PSTATE.ig   ←0 (interrupt globals are disabled)
PSTATE.cle  ←PSTATE.tle (set endian mode for traps)
PSTATE.tle  ←unchanged[1]
```

- Implementation-specific state changes, including:
  - State changes common to all JPS2 implementations (see Appendix O)
  - Implementation-specific state changes (see Appendix O of the JPS2 Extension documents)
- Control is transferred into the RED_state trap table.

  PC ← RSTVaddr<63:8> ▯ $1000\ 0000_2$

  nPC← RSTVaddr<63:8> ▯ $1000\ 0100_2$

1. Note that this was left unspecified in SPARC V9.

For any reset when TL = MAXTL, for all $n$ < MAXTL, the values in TPC[$n$], TNPC[$n$], and TSTATE[$n$] are undefined.

See *Externally Initiated Reset (XIR)* on page 598 and Appendix O of the JPS2 Extension documents for more information.

### 7.5.2.6 Non-reset Traps When the Virtual Processor Is in RED_state

Non-reset traps taken when the virtual processor is already in RED_state are also processed in RED_state, unless TL = MAXTL, in which case the virtual processor enters error_state.

Assuming that tl < MAXTL, the virtual processor state shall be set as follows:

- The trap level is set.

  TL ← TL + 1

- Existing state is preserved.

```
TSTATE[TL].CCR   ←CCR
TSTATE[TL].ASI   ←ASI
TSTATE[TL].PSTATE←PSTATE
TSTATE[TL].CWP   ←CWP
TPC[TL]          ←PC
TNPC[TL]         ←nPC
```

- The trap type is preserved.

  TT[TL] ← trap type

- The PSTATE register is set as follows:

```
PSTATE.mm  ←00₂ (TSO)
PSTATE.red ←1 (enter RED_state)
PSTATE.pef ←1 (FPU is present)
PSTATE.am  ←0 (address masking is turned off)
PSTATE.priv←1 (the virtual processor enters privileged mode)
PSTATE.ie  ←0 (interrupts are disabled)
PSTATE.ag  ←1 (global regs are replaced with alternate globals)
PSTATE.mg  ←0 (MMU globals are disabled)
PSTATE.ig  ←0 (interrupt globals are disabled)
PSTATE.cle ←PSTATE.tle (set endian mode for traps)
PSTATE.tle ←unchanged[1]
```

- For a register-window trap only, CWP is set to point to the register window that must be accessed by the trap-handler software, that is:

  If TT[TL] = $024_{16}$ (a *clean_window* trap), then CWP ← CWP + 1.

1. Note that this was left unspecified in SPARC V9

If ($080_{16} \leq$ TT[TL] $\leq 0BF_{16}$) (window spill trap), then
CWP ← CWP + CANSAVE + 2.

If ($0C0_{16} \leq$ TT[TL] $\leq 0FF_{16}$) (window fill trap), then CWP ← CWP – 1.

- For non-register-window traps, CWP is not changed.
- Implementation-specific state changes, including:
  - State changes common to all JPS2 implementations (see Appendix O)
  - Implementation-specific state changes (see Appendix O of the JPS2 Extension documents)
- Control is transferred into the RED_state trap table.

PC ← RSTVaddr<63:8> || $1010\ 0000_2$

nPC← RSTVaddr<63:8> || $1010\ 0100_2$

## 7.6 Exception and Interrupt Descriptions

The following sections describe the various exceptions and interrupt requests and the conditions that cause them. Each exception and interrupt request describes the corresponding trap type as defined by the trap model. SPARC JPS2 defines five categories of traps:

- Traps defined by SPARC V9 as mandatory
- Traps that are defined by SPARC V9 as optional but that are mandatory in SPARC JPS2
- Traps that are defined by SPARC V9 as optional and that remain optional in SPARC JPS2
- Traps that are defined by SPARC V9 as implementation dependent and optional but that are mandatory in SPARC JPS2
- Traps that are defined by SPARC V9 as implementation dependent and that remain implementation dependent in SPARC JPS2

All other trap types are reserved.

**Note:** This encoding differs from that shown in *The SPARC Architecture Manual-Version 9*. Each trap is marked as precise, deferred, disrupting, or reset. Example exception conditions are included for each exception type. *Appendix , Instruction Definitions*, enumerates which traps can be generated by each instruction.

## 7.6.1 Traps Defined by SPARC V9 As Mandatory

SPARC V9 defines the following traps as mandatory.

- ***data_access_exception*** [TT = $030_{16}$] (Precise) — An exception occurred on an attempted data access. Detailed information regarding the error is logged into the `ftype` field of Data Synchronous Fault Status Register (ASI $58_{16}$, VA = $18_{16}$). Below is the list of conditions that cause a *data_access_exception* exception.
  - **Invalid ASI** — An attempt to do load or store with undefined or reserved ASI or a disallowed instruction / ASI combination (see *Block Load and Store ASIs* on page 582 and *Partial Store ASIs* on page 582).
  - **Illegal Access to Strongly Ordered Page** — An attempt to access a strongly ordered page by any type of load instruction with nonfaulting ASI.

    An attempt to access a strongly ordered page by `FLUSH` instruction.
  - **Illegal Access to Non-Faulting-Only Page** — An attempt to access a non-faulting-only page by any type of load or store instruction or `FLUSH` instruction with ASI other than nonfaulting ASI.
  - **Illegal Access to Noncacheable Page** —An attempt to access a noncacheable page by atomic instructions (`CASA`, `CASXA`, `SWAP`, `SWAPA`, `LDSTUB`, `LDSTUBA`), or an attempt to access a noncacheable page by atomic quad load instructions (`LDDA` with ASI = $24_{16}$, $2C_{16}$).
- ***division_by_zero*** [TT = $028_{16}$] (Precise) — An integer divide instruction attempted to divide by zero**.**
- ***fill_n_normal*** [TT = $0C0_{16}$–$0DF_{16}$] (Precise)
- ***fill_n_other*** [TT = $0E0_{16}$–$0FF_{16}$] (Precise)

---

**Note –** The SPARC V9 *fill_n_** exceptions supersede the SPARC V8 *window_underflow* exception.

---

- ***fp_disabled*** [TT = $020_{16}$] (Precise) — An attempt was made to execute an FPop, a floating-point branch, or a floating-point load / store instruction while an FPU was not present, `PSTATE.pef` = 0, or `FPRS.fef` = 0.
- ***illegal_instruction*** [TT = $010_{16}$] (Precise) — An attempt was made to execute an instruction with an unimplemented opcode, an `ILLTRAP` instruction, an instruction with invalid field usage, instruction breakpoints, or an instruction that would result in illegal processor state. **Note**: An unimplemented FPop instruction generated an *fp_exception_other* exception with `ftt` = 3, instead of an *illegal_instruction* exception.

  Examples of cases in which *illegal_instruction* is generated include:

  - An instruction encoding does not match any of the opcode map definitions (see *Appendix E, Opcode Maps*).

- An instruction is not implemented in hardware (but if the op and op3 fields of the instruction decode as an FPop, then an *fp_exception_other* exception, with ftt = 3, will be generated instead of *illegal_instruction*).
- An illegal value is present in an instruction i field.
- An illegal value is present in a field that is explicitly defined for an instruction, such as cc2, cc1, cc0, fcn, impl, op2 (IMPDEP2A, IMPDEP2B), rcond, or opf_cc.
- Illegal register alignment (such as odd rd value in a doubleword load instruction).
- RDASR instruction with rs1 = 1, 7–14, 20–21, or 26–31.
- RDASR with rs1 = 15 and nonzero rd.
- RDPR with illegal rs1 value ($16 \leq$ rs1 $\leq 30$).
- RDPR with rs1 $\leq 3$ (TPC, TNPC, TSTATE, or TT) when TL = 0.
- WRPR with illegal rd value ($15 \leq$ rd $\leq 31$).
- WRPR with rd $\leq 3$ (TPC, TNPC, TSTATE, or TT) when TL = 0.
- WRPR to PSTATE register that attempts to set more than one of bits ig, mg, and ag.
- Illegal rd value for LDXFSR, STXFSR, or the deprecated instructions LDFSR or STFSR.
- WRASR instruction with rd = 1, 4, 5, 7–14, 26-31.
- WRASR with rd = 15 and nonzero rs1.
- WRASR with rd = 15 and i = 0.
- DONE or RETRY when tl = 0.
- ILLTRAP instruction.
- Instruction breakpoint occurred (impl. dep. #205).
- A reserved instruction field in Tcc instruction is nonzero.

If a reserved instruction field in an instruction other than Tcc is nonzero, an *illegal_instruction* exception should be generated.[1]

- ***instruction_access_exception*** [TT = $008_{16}$] (Precise) — A protection exception occurred on an instruction access, typically as a result of an attempt to access a privileged page while the virtual processor was executing in nonprivileged mode.
- ***interrupt_level_n*** [TT = $041_{16}$–$04F_{16}$] (Disrupting) — An interrupt request level of *n* was presented to the IU, while PSTATE.ie = 1 and (interrupt request level > PIL).

1. Since it is not strictly required that a nonzero value in a reserved field of an instruction other than Tcc causes an *illegal_instruction* exception, a JPS2 implementation may ignore the contents of reserved instruction fields (for instructions other than Tcc).

- ***mem_address_not_aligned*** [TT = $034_{16}$] (Precise) — A load / store instruction generated a memory address that was not properly aligned according to the instruction, or a JMPL or RETURN instruction generated a non-word-aligned address. (See also *Special Memory Access ASIs* on page 580.)
- ***power_on_reset*** (POR) [TT = $001_{16}$] (Reset) — An external signal was asserted. This trap is issued to bring a system reliably from the power-off to the power-on state.
- ***privileged_action*** [TT = $037_{16}$] (Precise) — An action defined to be privileged has been attempted while PSTATE.priv = 0. Examples: a data access by nonprivileged software using an ASI value with its most significant bit = 0 (a restricted ASI), an attempt to read the TICK register by nonprivileged software when TICK.npt = 1, or an attempt to access the PIC register (using RDPIC or WRPIC) while PSTATE.priv = 0 and PCR.priv = 1.
- ***privileged_opcode*** [TT = $011_{16}$] (Precise) — An attempt was made to execute a privileged instruction while PSTATE.priv = 0.

---

**V9 Compatibility Note –** *privileged_opcode*'s trap type is identical to that of the SPARC V8 *privileged_instruction* trap. The name was changed to distinguish it from the new *privileged_action* trap type.

---

- ***RED_state_exception*** [TT = (see text)] — Caused when TL = MAXTL – 1 and a trap occurs, an event that bring the virtual processor into RED_state. Uses the trap vector entry reserved for trap type $005_{16}$, but the trap type recorded in TT is the trap type of the original exception that triggered *RED_state_exception*.
- ***software_initiated_reset*** (SIR) [TT = $004_{16}$] (Precise / Reset) — Caused by the execution of the SIR instruction. It allows system software to reset the virtual processor.
- ***spill_n_normal*** [TT = $080_{16}$–$09F_{16}$] (Precise)
- ***spill_n_other*** [TT = $0A0_{16}$–$0BF_{16}$] (Precise)

  A SAVE or FLUSHW instruction has determined that the contents of a register window must be saved to memory.

---

**V9 Compatibility Note –** The SPARC V9 *spill_n_** exceptions supersede the SPARC V8 *window_overflow* exception.

---

- ***tag_overflow*** [TT = $023_{16}$] (Precise) — A TADDccTV or TSUBccTV instruction was executed, and either 32-bit arithmetic overflow occurred or at least one of the tag bits of the operands was nonzero.
- ***trap_instruction*** [TT = $100_{16}$–$17F_{16}$] (Precise) — A Tcc instruction was executed and the trap condition evaluated to TRUE.

## 7.6.2 SPARC V9 Optional Traps That Are Mandatory in SPARC JPS2

SPARC V9 defines the following traps as optional. However, the traps are mandatory in SPARC JPS2.

- ***clean_window*** [TT = $024_{16}$–$027_{16}$] (Precise) — A SAVE instruction discovered that the window about to be used contains data from another address space; the window must be cleaned before it can be used.

  **IMPL. DEP. #102:** An implementation may choose either to implement automatic cleaning of register windows in hardware or to generate a *clean_window* trap, when needed, so that window(s) can be cleaned by software. If an implementation chooses the latter option, then support for this trap type is mandatory.

- ***data_access_error*** [TT = $032_{16}$] (Precise or Deferred) — An error occurred on a data access.
- ***externally_initiated_reset*** (XIR) [TT = $003_{16}$] (Reset) — An external signal was asserted. This trap is used for catastrophic events such as power failure, reset button pressed, and systemwide reset in multiprocessor systems.
- ***fp_exception_ieee_754*** [TT = $021_{16}$] (Precise) — An FPop instruction generated an *IEEE_754_exception* and its corresponding trap enable mask (tem) bit was 1. The floating-point exception type, *IEEE_754_exception*, is encoded in the FSR.ftt, and specific *IEEE_754_exception* information is encoded in FSR.cexc.
- ***fp_exception_other*** [TT = $022_{16}$] (Precise) — An FPop instruction generated an exception other than an *IEEE_754_exception*. Examples: the FPop is unimplemented, or there was a sequence or hardware error in the FPU. The floating-point exception type is encoded in the FSR's ftt field.
- ***instruction_access_error*** [TT = $00A_{16}$] (Precise) — An error occurred on an instruction access.
- ***LDDF_mem_address_not_aligned*** [TT = $035_{16}$] (Precise) — An attempt was made to execute an LDDF instruction and the effective address was not doubleword aligned.
- ***STDF_mem_address_not_aligned*** [TT = $036_{16}$] (Precise) — An attempt was made to execute an STDF instruction and the effective address was not doubleword aligned.
- ***watchdog reset*** (WDR) [TT = $002_{16}$] (Reset) — This trap occurs when the watchdog timer overflows or as a transition from error_state to RED_state (impl. dep. #254).

## 7.6.3 SPARC V9 Optional Traps That Are Optional in SPARC JPS2

SPARC V9 defines the following trap as optional, and it remains optional in SPARC JPS2.

- ***async_data_error*** [TT = $040_{16}$] (Precise, Deferred, or Disrupting) — An implementation-dependent exception (impl. dep. #31, #218) that indicates that one or more unrecoverable or uncorrectable but recoverable errors have been detected in the virtual processor. This may include errors detected in the architectural registers (general-purpose registers, floating-point registers, ASRs, or ASI registers) and other key processor hardware. A single *async_data_error* exception may indicate multiple errors and may occur asynchronously to instruction execution. An *async_data_error* exception may cause a precise, deferred, or disrupting trap. When *async_data_error* causes a disrupting trap, the `tpc` and `tnpc` stacked by the trap do not necessarily indicate the instruction or data access that caused the error.

  **IMPL. DEP. #218:** Whether *async_data_error* exception is implemented is implementation dependent. If it does exist, it indicates that an error is detected in a processor core and its trap type is $40_{16}$.

---

**V9 Compatibility Note –** The SPARC V9 *async_data_error* supersedes the less general SPARC V8 *data_store_error* exception.

---

- ***internal_processor_error*** [TT = $029_{16}$] (Precise, Deferred, or Disrupting) — A serious internal error occurred in the virtual processor (impl. dep. #31).
- ***unimplemented_LDD*** [TT = $012_{16}$] (Precise) — An attempt was made to execute an `LDD` instruction, which is not implemented in hardware on this implementation (impl. dep. #107).
- ***unimplemented_STD*** [TT = $013_{16}$] (Precise) — An attempt was made to execute an `STD` instruction, which is not implemented in hardware on this implementation (impl. dep. #108).

## 7.6.4 SPARC V9 Optional Traps Not Used in SPARC JPS2

The following traps are optional in SPARC V9 and are not used in SPARC JPS2:

- ***data_access_MMU_miss*** [TT = $031_{16}$] (Precise or Deferred) — This exception is superseded by *fast_data_access_MMU_miss* (see section 7.6.5).
- ***data_access_protection*** [TT = $033_{16}$] (Precise or Deferred) — This exception is superseded by *fast_data_access_protection* (see section 7.6.5).

- ***LDQF_mem_address_not_aligned*** [TT = $038_{16}$] (Precise or Deferred) — An attempt was made to execute an LDQF instruction and the effective address was word aligned but not quadword aligned. Use of this exception is implementation dependent (impl. dep. #111). A separate trap entry for this exception supports fast software emulation of the LDQF instruction when the effective address is word aligned but not quadword aligned. See A.26, *Load Floating-Point* on page 274.
- ***STQF_mem_address_not_aligned*** [TT = $039_{16}$] (Precise) — An attempt was made to execute an STQF instruction and the effective address was word aligned but not quadword aligned. Use of this exception is implementation dependent (impl. dep. #112). A separate trap entry for the exception supports fast software emulation of the STQF instruction when the effective address is word aligned but not quadword aligned. See A.61, *Store Floating-Point* on page 360.
- ***instruction_access_MMU_miss*** [TT = $009_{16}$] (Precise, Deferred, or Disrupting) — This exception is superseded by *fast_instruction_access_MMU_miss* (see Section 7.6.5).

## 7.6.5 SPARC V9 Implementation-Dependent, Optional Traps That Are Mandatory in SPARC JPS2

SPARC V9 defines the following traps as implementation dependent and optional. The traps are mandatory in SPARC JPS2.

- ***ECC_error*** [TT = $063_{16}$] (Disrupting) — The trap to signal the detection of hardware errors asynchronous to the instruction execution, or to request to save the information logged for the error that was detected and corrected by the virtual processor.

**Note**: some implementations may refer to this trap by the name "*corrected_ECC_error.*"

- ***fast_data_access_MMU_miss*** [TT = $068_{16}$] (Precise) — During an attempted data access, the MMU detected that a translation lookaside buffer did not contain a translation for the virtual address (that is, a TLB miss occurred). Four trap vectors are allocated for this trap, allowing a TLB miss handler of up to 32 instructions to fit within the trap vector area.
- ***fast_data_access_protection*** [TT = $06C_{16}$] (Precise) — During an attempted data write access (by a store or load-store instruction), the instruction had appropriate access privilege but the MMU signalled that the location was write-protected (write to a read-only location).
  Note that on a JPS2 virtual processor, an attempt to read or write to a privileged location while in nonprivileged mode causes the higher-priority *data_access_exception* instead of this exception. Four trap vectors are allocated for this trap, allowing a trap handler of up to 32 instructions to fit within the trap vector area.

- ***fast_instruction_access_MMU_miss*** [TT = $064_{16}$] (Precise) — During an attempted instruction access, the MMU detected a TLB miss. Four trap vectors are allocated for this trap, allowing a trap handler of up to 32 instructions to fit within the trap vector area.
- ***interrupt_vector_trap*** [TT = $060_{16}$] (Disrupting) — The virtual processor has received an interrupt request.
- ***PA_watchpoint*** [TT = $061_{16}$] (Precise) — The virtual processor has detected a physical-address watchpoint.
- ***VA_watchpoint*** [TT = $062_{16}$] (Precise) — The virtual processor has detected a virtual-address watchpoint.

## 7.6.6 SPARC JPS2 Implementation-Dependent Traps

The following traps are implementation dependent in SPARC JPS2.

- ***fast_ECC_error*** [TT = $070_{16}$] (Precise) — A single-bit or multiple-bit ECC error is detected.

  **IMPL. DEP. #202:** Whether or not a *fast_ECC_error* trap exists is implementation dependent. If it does exist, it indicates that an ECC error was detected in an external cache and its trap type is $070_{16}$.
- See *Traps* on page 207 for details on MTP traps.

# Memory Models

The SPARC V9 *memory models* define the semantics of memory operations. The instruction set semantics require that loads and stores *seem* to be performed in the order in which they appear in the dynamic control flow of the program. The *actual* order in which they are processed by the memory may be different. The purpose of the memory models is to specify what constraints, if any, are placed on the order of memory operations.

The memory models apply both to uniprocessor and to shared memory multiprocessors. Formal memory models are necessary for precise definitions of the interactions between multiple virtual processors and input/output devices in a shared memory configuration. Programming shared memory multiprocessors requires a detailed understanding of the operative memory model and the ability to specify memory operations at a low level in order to build programs that can safely and reliably coordinate their activities. See Appendix J, *Programming with the Memory Models*, for additional information on the use of the models in programming real systems.

Although this chapter contains a great deal of theoretical information, we have included that information so the discussion of the implementation-specific memory models has sufficient background.

We describe memory models in these sections:


- *Overview* on page 170
- *Memory, Real Memory, and I/O Locations* on page 171
- *Addressing and Alternate Address Spaces* on page 173
- *SPARC V9 Memory Model* on page 175

# 8.1 Overview

The SPARC V9 architecture is a *model* that specifies the behavior observable by software on SPARC V9 systems. Therefore, access to memory can be implemented in any manner, as long as the behavior observed by software conforms to that of the models described here and defined in Appendix D, *Formal Specification of the Memory Models*.

The SPARC V9 architecture defines three different memory models: *Total Store Order (TSO)*, *Partial Store Order (PSO)*, and *Relaxed Memory Order (RMO)*. All SPARC V9 processors must provide Total Store Order (or a more strongly ordered model, for example, Sequential Consistency) to ensure SPARC V8 compatibility.

**IMPL. DEP. #113:** Whether the PSO or RMO models are supported by SPARC V9 systems is implementation dependent.

FIGURE 8-1 shows the relationship of the various SPARC V9 memory models, from the least restrictive to the most restrictive. Programs written assuming one model will function correctly on any included model.

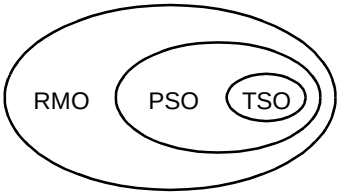



**FIGURE 8-1** Memory Models: Least Restrictive (RMO) to Most Restrictive (TSO)

SPARC V9 provides multiple memory models so that:

- Implementations can schedule memory operations for high performance.
- Programmers can create synchronization primitives using shared memory.

These models are described informally in this subsection and formally in Appendix D, *Formal Specification of the Memory Models*. If there is a conflict in interpretation between the informal description provided here and the formal models, the formal models supersede the informal description.

There is no preferred memory model for SPARC V9. Programs written for Relaxed Memory Order will work in both Partial Store Order and Total Store Order. Programs written for Partial Store Order will work in Total Store Order. Programs written for a weak model, such as RMO, may execute more quickly, since the model

exposes more scheduling opportunities, but may also require extra instructions to ensure synchronization. Multiprocessor programs written for a stronger model will behave unpredictably if run in a weaker model.

Machines that implement *sequential consistency* (also called strong ordering or strong consistency) automatically support programs written for TSO, PSO, and RMO. Sequential consistency is not a SPARC V9 memory model. In sequential consistency, the loads, stores, and atomic load-stores of all virtual processors are performed by memory in a serial order that conforms to the order in which these instructions are issued by individual virtual processors. A machine that implements sequential consistency may deliver lower performance than an equivalent machine that implements a weaker model. Although particular SPARC V9 implementations may support sequential consistency, portable software must not rely on having this model available.

***Notes About the Implementation of the Memory Models.*** From the programmer's point of view, a SPARC V9 implementation completely supports the memory models specified in SPARC V9.

SPARC V9 does not specify exactly how the hardware must support a particular SPARC V9 memory model, except that the hardware support for the V9 memory model must guarantee that a correct program written for that memory model will run correctly on the hardware. For example, a slightly stronger (more restrictive) hardware memory model might be used than that required by the SPARC V9 memory model.

## 8.2 Memory, Real Memory, and I/O Locations

Memory is the collection of locations accessed by the load and store instructions (described in Appendix A, *Instruction Definitions*). Each location is identified by an address consisting of two elements: an *address space identifier* (ASI), which identifies an address space, and a 64-bit *address***,** which is a byte offset into that address space. Memory addresses may be interpreted by the memory subsystem to be either physical addresses or virtual addresses; addresses may be remapped and values cached, provided that memory properties are preserved transparently and coherency is maintained.

When two or more data addresses refer to the same datum, the address is said to be *aliased*. In this case, the virtual processor and memory system must cooperate to maintain consistency; that is, a store to an aliased address must change all values aliased to that address.

Memory addresses identify either real memory or I/O locations.

*Real memory* stores information without side effects. A load operation returns the value most recently stored. Operations are side-effect-free in the sense that a load, store, or atomic load-store to a location in real memory has no program-observable effect, except upon that location.

*I/O locations* may not behave like memory and may have side effects. Load, store, and atomic load-store operations performed on I/O locations may have observable side effects, and loads may not return the value most recently stored. The value semantics of operations on I/O locations are *not* defined by the memory models, but the constraints on the order in which operations are performed is the same as it would be if the I/O locations were real memory. The storage properties, contents, semantics, ASI assignments, and addresses of I/O registers are implementation dependent.

**IMPL. DEP. #118:** The manner in which I/O locations are identified is implementation dependent.

**IMPL. DEP. #120:** The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent.

---

**V9 Compatibility Note –** Operations to I/O locations are *not* guaranteed to be sequentially consistent among themselves, as they are in SPARC V8.

---

SPARC V9 does not distinguish real memory from I/O locations in terms of ordering. All references, both to I/O locations and real memory, conform to the memory model's order constraints. References to I/O locations may need to be interspersed with `MEMBAR` instructions to guarantee the desired ordering.

Systems supporting SPARC V8 applications that use memory mapped I/O locations must ensure that SPARC V8 sequential consistency of I/O locations can be maintained when those locations are referenced by a SPARC V8 application. The MMU either must enforce such consistency or cooperate with system software or the virtual processor to provide it.

**IMPL. DEP. #121:** An implementation may choose to identify certain addresses and use an implementation-dependent memory model for references to them.

# 8.3 Addressing and Alternate Address Spaces

An address in SPARC V9 is a tuple consisting of an 8-bit address space identifier (ASI) and a 64-bit byte-address offset in the specified address space. Memory is byte-addressed, with halfword accesses aligned on 2-byte boundaries, word accesses (which include instruction fetches) aligned on 4-byte boundaries, extended-word and doubleword accesses aligned on 8-byte boundaries, and quadword quantities aligned on 16-byte boundaries. With the possible exception of the cases described in *Memory Alignment Restrictions* on page 106, an improperly aligned address in a load, store, or load-store instruction always causes a trap to occur. The largest datum that is guaranteed to be atomically read or written is an aligned doubleword.[1] Also, memory references to different bytes, halfwords, and words in a given doubleword are treated for ordering purposes as references to the same location. Thus, the unit of ordering for memory is a doubleword.

---

**Note –** The doubleword is the coherency unit for update, but programmers should not assume that doubleword floating-point values are updated as a unit unless they are doubleword-aligned and always updated with double-precision loads and stores. Some programs use pairs of single-precision operations to load and store double-precision floating-point values when the compiler cannot determine that they are doubleword aligned.

Also, although quad-precision operations are defined in the SPARC V9 architecture, the granularity of loads and stores for quad-precision floating-point values may be word or doubleword.

---

The virtual processor provides an address space identifier with every address. This ASI may serve several purposes:

- To identify which of several distinguished address spaces the 64-bit address offset is addressing
- To provide additional access control and attribute information, for example, to identify the processing that is to be performed if an access fault occurs, or to specify the endianness of the reference
- To specify the address of an internal control register in the virtual processor, cache, or memory management hardware

1. Two exceptions to this are the special `ASI_NUCLEUS_QUAD_LDD[_L]` and `ASI_QUAD_LDD_PHYS[_L]` which provide hardware support for an atomic quad load to be used for `TTE` loads from TSBs.

The memory management hardware can associate an independent $2^{64}$-byte memory address space with each ASI. If this is done, it becomes possible to allow system software easy access to the address space of the faulting program when processing exceptions or to implement access to a client program's memory space by a server program.

The architecturally specified ASIs are listed in Appendix L, *Address Space Identifiers (ASIs)*.

When TL = 0, normal accesses by the virtual processor to memory when fetching instructions and performing loads and stores implicitly specify ASI_PRIMARY or ASI_PRIMARY_LITTLE, depending on the setting of the PSTATE.cle bit.

When TL > 0, the implicit ASI for instruction fetches is ASI_NUCLEUS; loads and stores will use ASI_NUCLEUS if PSTATE.cle = 0 or ASI_NUCLEUS_LITTLE if PSTATE.cle = 1 (impl. dep. #124).

SPARC V9 supports the PRIMARY{_LITTLE}, SECONDARY{_LITTLE}, and NUCLEUS{_LITTLE} address spaces.

Accesses to other address spaces use the load/store alternate instructions. For these accesses, the ASI is either contained in the instruction (for the register-register addressing mode) or taken from the ASI register (for register-immediate addressing).

ASIs are either nonrestricted or restricted. A nonrestricted ASI is one that may be used independently of the privilege level (PSTATE.priv) at which the virtual processor is running. Restricted ASIs require that the virtual processor be in privileged mode for a legal access to occur. Restricted ASIs have their high-order bit equal to 0. The relationship between virtual processor state and ASI restriction is shown in TABLE 6-3 on page 110.

Several restricted ASIs are provided as mandated by SPARC V9: ASI_AS_IF_USER_PRIMARY{_LITTLE} and ASI_AS_IF_USER_SECONDARY{_LITTLE}. The intent of these ASIs is to give system software efficient access to the memory space of a program.

The normal address space is *primary address space*, which is accessed by the unrestricted ASI_PRIMARY{_LITTLE}. The *secondary address space*, which is accessed by the unrestricted ASI_SECONDARY{_LITTLE}, is provided to allow a server program to access a client program's address space.

ASI_PRIMARY_NOFAULT{_LITTLE} and ASI_SECONDARY_NOFAULT{_LITTLE} support *nonfaulting loads***.** These ASIs are aliased to ASI_PRIMARY{_LITTLE} and ASI_SECONDARY{_LITTLE}, respectively, and have exactly the same action. They may be used to color (that is, distinguish into classes) loads in the instruction stream so that, in combination with a judicious mapping of low memory and a specialized trap handler, an optimizing compiler can move loads outside of conditional control structures.

**Note –** Nonfaulting loads allow optimizations that move loads ahead of conditional control structures that guard their use; thus, they can minimize the effects of load latency by improving instruction scheduling. The semantics of nonfaulting loads are the same as for any other load, except when nonrecoverable catastrophic faults occur (for example, a reference to a nonexisting or invalid page). When such a fault occurs, it is ignored and the hardware and system software cooperate to make the load appear to complete normally, returning a zero result. The compiler's optimizer generates load-alternate instructions with the ASI field or register set to `ASI_PRIMARY_NOFAULT{_LITTLE}` or `ASI_SECONDARY_NOFAULT{_LITTLE}` for those loads it determines should be nonfaulting.

To minimize unnecessary processing if a fault does occur, map low addresses (especially address zero) to a page of all zeroes, so that references through a `NULL` pointer do not cause unnecessary traps.

## 8.4 SPARC V9 Memory Model

The SPARC V9 processor architecture specifies the organization and structure of a central processing unit but does not specify a memory system architecture. Appendix F, *Memory Management Unit*, summarizes the MMU support required by a SPARC JPS2 virtual processor. Appendix F of the JPS2 Extension documents describe implementations.

The memory models specify the possible order relationships between memory-reference instructions issued by a virtual processor and the order and visibility of those instructions as seen by other virtual processors. The memory model is intimately intertwined with the program execution model for instructions.

### 8.4.1 SPARC V9 Program Execution Model

The SPARC V9 processor model consists of three units: an Issue Unit, a Reorder Unit, and an Execute Unit, as shown in FIGURE 8-2.

The Issue Unit reads instructions over the instruction path from memory and issues them in *program order* to the Reorder Unit. Program order is precisely the order determined by the control flow of the program and the instruction semantics, under the assumption that each instruction is performed independently and sequentially.

Issued instructions are collected and potentially reordered in the Reorder Unit, and then dispatched to the Execute Unit. Instruction reordering allows an implementation to perform some operations in parallel and to better allocate

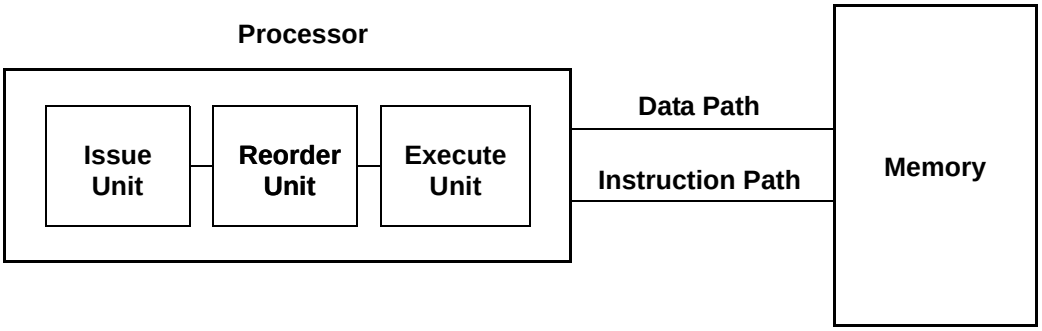



**FIGURE 8-2** Processor Model: Uniprocessor System

resources. The reordering of instructions is constrained to ensure that the results of program execution are the same as they would be if the instructions were performed in program order. This property is called *processor self-consistency*.

Processor self-consistency requires that the result of execution, in the absence of any shared memory interaction with another virtual processor, be identical to the result that would be observed if the instructions were performed in program order. In the model in FIGURE 8-2, instructions are issued in program order and placed in the reorder buffer. The virtual processor is allowed to reorder instructions, provided it does not violate any of the data-flow constraints for registers or for memory.

The data-flow order constraints for register reference instructions are these:

1. An instruction cannot be performed until all earlier instructions that set a register it uses have been performed (read-after-write hazard; write-after-write hazard).
2. An instruction cannot be performed until all earlier instructions that use a register it sets have been performed (write-after-read hazard).

---

**V9 Compatibility Note –** An implementation can avoid blocking instruction execution in case 2 and the write-after-write hazard in case 1 by using a renaming mechanism that provides the old value of the register to earlier instructions and the new value to later uses.

---

The data-flow order constraints for memory-reference instructions are those for register reference instructions, plus the following additional constraints:

1. A memory-reference instruction that sets (stores to) a location cannot be performed until all previous instructions that use (load from) the location have been performed (write-after-read hazard).
2. A memory-reference instruction that uses (loads) the value at a location cannot be performed until all earlier memory-reference instructions that set (store to) the location have been performed (read-after-write hazard).

Memory-barrier instructions (MEMBAR and STBAR) and the active memory model specified by PSTATE.mm also constrain the issue of memory-reference instructions. See *MEMBAR Instruction* on page 178 and *Memory Models* on page 180 for a detailed description.

The constraints on instruction execution assert a partial ordering on the instructions in the reorder buffer. Every one of the several possible orderings is a legal execution ordering for the program. See Appendix D, *Formal Specification of the Memory Models*, for more information.

## 8.4.2 Virtual Processor / Memory Interface Model

Each virtual processor in a multiprocessor system is modeled as shown in FIGURE 8-3; that is, having two independent paths to memory: one for instructions and one for data.

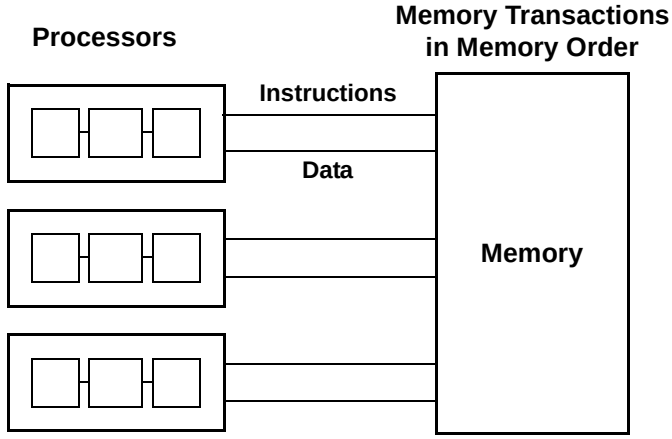



**FIGURE 8-3** Data Memory Paths: Multiprocessor System

Data caches are maintained by hardware to be consistent (coherent). Instruction caches need not be kept consistent with data caches and therefore require explicit program action to ensure consistency when a program modifies an executing instruction stream. Memory is shared in terms of address space, but it may be nonhomogeneous and distributed in an implementation. Mapping and caches are ignored in the model, since their functions are transparent to the memory model.[2]

In real systems, addresses may have attributes that the virtual processor must respect. The virtual processor executes loads, stores, and atomic load-stores in whatever order it chooses, as constrained by program order and the current memory model. The ASI-address couples it generates are translated by a memory

2. The model described here is only a model; implementations of SPARC V9 systems are unconstrained as long as their observable behaviors match those of the model.

management unit (MMU), which associates attributes with the address and may, in some instances, abort the memory transaction and signal an exception to the virtual processor.

For example, a region of memory may be marked as nonprefetchable, noncacheable, read-only, or restricted. It is the MMU's responsibility, working in conjunction with system software, to ensure that memory attribute constraints are not violated. See Appendix F of the JPS2 Extension documents for more information.

Instructions are performed in an order constrained by local dependencies. Using this dependency ordering, an execution unit submits one or more pending memory transactions to the memory. The memory performs transactions in *memory order*. The memory unit may perform transactions submitted to it out of order; hence, the execution unit must not concurrently submit two or more transactions that are required to be ordered, unless the memory unit can still guarantee in-order semantics.

The memory accepts transactions, performs them, and then acknowledges their completion. Multiple memory operations may be in progress at any time and may be initiated in a nondeterministic fashion in any order, provided that all transactions to a location preserve the per-virtual processor partial orders. Memory transactions may complete in any order. Once initiated, all memory operations are performed atomically: loads from one location all see the same value, and the result of stores is visible to all potential requestors at the same instant.

The order of memory operations observed at a single location is a *total order* that preserves the partial orderings of each virtual processor's transactions to this address. There may be many legal total orders for a given program's execution.

### 8.4.3 `MEMBAR` Instruction

`MEMBAR` serves two distinct functions in SPARC V9. One variant of the `MEMBAR`, the ordering `MEMBAR`, provides a way for the programmer to control the order of loads and stores issued by a virtual processor. The other variant of `MEMBAR`, the sequencing `MEMBAR`, enables the programmer to explicitly control order and completion for memory operations. Sequencing `MEMBAR`s are needed only when a program requires that the effect of an operation becomes globally visible rather than simply being scheduled.[3] Because both forms are bit-encoded into the instruction, a single `MEMBAR` can function both as an ordering `MEMBAR` and as a sequencing `MEMBAR`.

3. Sequencing `MEMBAR`s are needed for some input / output operations, forcing stores into specialized stable storage, context switching, and occasional other system functions. Using a sequencing `MEMBAR` when one is not needed may cause a degradation of performance. See Appendix J, *Programming with the Memory Models*, for examples of the use of sequencing `MEMBAR`s.

### 8.4.3.1 Ordering MEMBAR Instructions

Ordering MEMBAR instructions induce an ordering in the instruction stream of a single virtual processor. Sets of loads and stores that appear before the MEMBAR in program order are ordered with respect to sets of loads and stores that follow the MEMBAR in program order. Atomic operations (LDSTUB(A), SWAP(A), CASA, and CASXA) are ordered by MEMBAR as if they were both a load and a store, since they share the semantics of both. An STBAR instruction, with semantics that are a subset of MEMBAR, is provided for SPARC V8 compatibility. MEMBAR and STBAR operate on all pending memory operations in the reorder buffer, independently of their address or ASI, ordering them with respect to all future memory operations. This ordering applies only to memory-reference instructions issued by the virtual processor issuing the MEMBAR. Memory-reference instructions issued by other virtual processors are unaffected.

The ordering relationships are bit-encoded as shown in TABLE 8-1. For example, MEMBAR $01_{16}$, written as "membar #LoadLoad" in assembly language, requires that all load operations appearing before the MEMBAR in program order complete before any of the load operations following the MEMBAR in program order complete. Store operations are unconstrained in this case. MEMBAR $08_{16}$ (#StoreStore) is equivalent to the STBAR instruction; it requires that the values stored by store instructions appearing in program order prior to the STBAR instruction be visible to other virtual processors before issuing any store operations that appear in program order following the STBAR.

In TABLE 8-1 these ordering relationships are specified by the "<*m*" symbol, which signifies memory order. See Appendix D, *Formal Specification of the Memory Models*, for a formal description of the <*m* relationship.

**TABLE 8-1** Ordering Relationships Selected by Mask

| Ordering Relation, Earlier < Later | Suggested Assembler Tag | Mask Value | nmask Bit # |
|---|---|---|---|
| Load <*m* Load | #LoadLoad | $01_{16}$ | 0 |
| Store <*m* Load | #StoreLoad | $02_{16}$ | 1 |
| Load <*m* Store | #LoadStore | $04_{16}$ | 2 |
| Store <*m* Store | #StoreStore | $08_{16}$ | 3 |

Selections may be combined to form more powerful barriers. For example, a MEMBAR instruction with a mask of $09_{16}$ (#LoadLoad | #StoreStore) orders loads with respect to loads and stores with respect to stores, but it does not order loads with respect to stores, or vice versa.

### 8.4.3.2 Sequencing MEMBAR Instructions

A sequencing MEMBAR exerts explicit control over the completion of operations. The three sequencing MEMBAR options each have a different degree of control and a different application.

- **Lookaside Barrier —** Ensures that loads following this MEMBAR are from memory and not from a lookaside into a write buffer. Lookaside Barrier requires that pending stores issued prior to the MEMBAR be completed before any load from that address following the MEMBAR may be issued. A Lookaside Barrier MEMBAR may be needed to provide lock fairness and to support some plausible I/O location semantics. See the example in *Control and Status Registers* on page 558.
- **Memory Issue Barrier —** Ensures that all memory operations appearing in program order before the sequencing MEMBAR complete before any new memory operation may be initiated. See the example in *I/O Registers with Side Effects* on page 558.
- **Synchronization Barrier —** Ensures that all instructions (memory reference and others) preceding the MEMBAR complete and that the effects of any fault or error have become visible before any instruction following the MEMBAR in program order is initiated. A Synchronization Barrier MEMBAR fully synchronizes the virtual processor that issues it.

TABLE 8-2 shows the encoding of these functions in the MEMBAR instruction.

**TABLE 8-2** Sequencing Barrier Selected by Mask

| Sequencing Function | Assembler Tag | Mask Value | cmask Bit # |
|---|---|---|---|
| Lookaside Barrier | #Lookaside | $10_{16}$ | 0 |
| Memory Issue Barrier | #MemIssue | $20_{16}$ | 1 |
| Synchronization Barrier | #Sync | $40_{16}$ | 2 |

## 8.4.4 Memory Models

The SPARC V9 memory models are defined below in terms of order constraints placed upon memory-reference instruction execution, in addition to the minimal set required for self-consistency. These order constraints take the form of MEMBAR operations implicitly performed following some memory-reference instructions.

### 8.4.4.1 Relaxed Memory Order (RMO)

*Relaxed Memory Order* places no ordering constraints on memory references beyond those required for processor self-consistency. When ordering is required, it must be provided explicitly in the programs by MEMBAR instructions.

#### 8.4.4.2 Partial Store Order (PSO)

*Partial Store Order* may be provided for compatibility with existing SPARC V8 programs. Programs that execute correctly in the RMO memory model will execute correctly in the PSO model.

The rules for PSO are these:

- Loads are blocking and ordered with respect to earlier loads.
- Atomic load-stores are ordered with respect to loads.

Thus, PSO ensures the following behavior:

- Each load and atomic load-store instruction behaves as if it were followed by a `MEMBAR` with a mask value of $05_{16}$.
- Explicit `MEMBAR` instructions are required to order store and atomic load-store instructions with respect to each other.

#### 8.4.4.3 Total Store Order (TSO)

*Total Store Order* must be provided for compatibility with existing SPARC V8 programs. Programs that execute correctly in either RMO or PSO will execute correctly in the TSO model.

Following are the rules for TSO:

- Loads are blocking and ordered with respect to earlier loads.
- Stores are ordered with respect to stores.
- Atomic load-stores are ordered with respect to loads and stores.

Thus, TSO ensures the following behavior:

- Each load instruction behaves as if it were followed by a `MEMBAR` with a mask value of $05_{16}$.
- Each store instruction behaves as if it were followed by a `MEMBAR` with a mask of $08_{16}$.
- Each atomic load-store behaves as if it were followed by a `MEMBAR` with a mask of $0D_{16}$.

### 8.4.5 Mode Control

The memory model is specified by the 2-bit state in `PSTATE.mm`, described in *PSTATE_mem_model (MM)* on page 74.

Writing a new value into `PSTATE.mm` causes subsequent memory reference instructions to be performed with the order constraints of the specified memory model.

SPARC V9 processors need not provide all three memory models; undefined values of PSTATE.mm have implementation-dependent effects.

**IMPL. DEP. #119:** The effect of writing an unimplemented memory mode designation into PSTATE.mm is implementation dependent.

Except when a trap enters RED_state, PSTATE.mm is left unchanged when a trap is entered and the old value is stacked. When RED_state is entered, the value of PSTATE.mm is set to TSO.

## 8.4.6 Hardware Primitives for Mutual Exclusion

In addition to providing memory-ordering primitives that allow programmers to construct mutual-exclusion mechanisms in software, SPARC V9 provides three hardware primitives for mutual exclusion:

- Compare and Swap (CASA and CASXA)
- Load Store Unsigned Byte (LDSTUB and LDSTUBA)
- Swap (SWAP and SWAPA)

Each of these instructions has the semantics of both a load and a store in all three memory models. They are all *atomic*, in the sense that no other store to the same location can be performed between the load and store elements of the instruction. All of the hardware mutual-exclusion operations conform to the memory models and may require barrier instructions to ensure proper data visibility.

When the hardware mutual-exclusion primitives address I/O locations, the results are implementation dependent. In addition, the atomicity of hardware mutual-exclusion primitives is guaranteed only for references to memory by processors and not when the memory location is simultaneously being addressed by an I/O device such as a channel or DMA.

### 8.4.6.1 Compare-and-Swap (CASA, CASXA)

Compare-and-swap is an atomic operation that compares a value in a virtual processor register to a value in memory and, if and only if they are equal, swaps the value in memory with the value in a second virtual processor register. Both 32-bit (CASA) and 64-bit (CASXA) operations are provided. The compare-and-swap operation is atomic in the sense that once it begins, no other virtual processor can access the memory location specified until the compare has completed and the swap (if any) has also completed and is potentially visible to all other virtual processors in the system.

Compare-and-swap is substantially more powerful than the other hardware synchronization primitives. It has an infinite consensus number; that is, it can resolve, in a wait-free fashion, an infinite number of contending processes. Because

of this property, compare-and-swap can be used to construct wait-free algorithms that do not require the use of locks. See Appendix J, *Programming with the Memory Models*, for examples.

#### 8.4.6.2 Swap (SWAP)

SWAP atomically exchanges the lower 32 bits in a virtual processor register with a word in memory. SWAP has a consensus number of two; that is, it cannot resolve more than two contending processes in a wait-free fashion.

#### 8.4.6.3 Load Store Unsigned Byte (LDSTUB)

LDSTUB loads a byte value from memory to a register and writes the value $FF_{16}$ into the addressed byte atomically. LDSTUB is the classic test-and-set instruction. Like SWAP, it has a consensus number of two and so cannot resolve more than two contending processes in a wait-free fashion.

### 8.4.7 Synchronizing Instruction and Data Memory

The SPARC V9 memory models do not require that instruction and data memory images be consistent at all times. The instruction and data memory images may become inconsistent if a program writes into the instruction stream. As a result, whenever instructions are modified by a program in a context where the data (that is, the instructions) in the memory and the data cache hierarchy may be inconsistent with instructions in the instruction cache hierarchy, some special programmatic action must be taken.

The FLUSH instruction will ensure consistency between the in-flight instruction stream and the data references in the virtual processor executing FLUSH. The programmer must ensure that the modification sequence is robust under multiple updates and concurrent execution. Since, in general, loads and stores may be performed out of order, appropriate MEMBAR and FLUSH instructions must be interspersed as needed to control the order in which the instruction data are mutated.

The FLUSH instruction ensures that subsequent instruction fetches from the doubleword target of the FLUSH by the virtual processor executing the FLUSH appear to execute after any loads, stores, and atomic load-stores issued by the virtual processor to that address prior to the FLUSH. FLUSH acts as a barrier for instruction fetches in the virtual processor on which it executes and has the properties of a store with respect to MEMBAR operations.

FLUSH has zero latency on the issuing virtual processor; the modified instruction stream is immediately available.[4]

4. SPARC V8 specified a five-instruction latency. Invalidation of instructions in the instruction cache during their execution is likely to force an instruction-cache fault.

**IMPL. DEP. #122:** The latency between the execution of FLUSH on one virtual processor and the point at which the modified instructions have replaced outdated instructions in a multiprocessor is implementation dependent.

---

**Programming Note –** Because FLUSH is designed to act on a doubleword and because, on some implementations, FLUSH may trap to system software, it is recommended that system software provide a user-callable service routine for flushing arbitrarily sized regions of memory. On some implementations, this routine would issue a series of FLUSH instructions; on others, it might issue a single trap to system software that would then flush the entire region.

---

On a SPARC JPS2 virtual processor:

- A FLUSH instruction flushes the virtual processor pipeline and synchronizes the virtual processor.
- Coherency between instruction and data memories is maintained with hardware; therefore, the address in the operands of a FLUSH instruction may be ignored (but must be supplied by software for SPARC V9 compatibility).

---

**Programming Note –** Although SPARC JPS2 virtual processors maintain coherency between instruction and data caches in hardware, SPARC V9 implementations in general are not required to do so (and some do not). Therefore, portable SPARC V9 software:
(1) must always assume that store instructions (except Block Store with Commit) do not coherently update I-cache(s);
(2) must, in every FLUSH instruction, supply the address of the instruction or instructions that were modified.

---

# Multithreaded Processing (MTP)

A SPARC JPS2 implementation may include multiple virtual processors on the same processor module to provide a dense, high throughput system. This may be achieved by having a combination of multiple physical processor cores and/or multiple strands (threads) per physical processor core. This chapter specifies a common interface between hardware and software for such products, referred to here as multithreaded processors (MTPs). It addresses issues common to MTPs, regardless of the microarchitecture of the individual physical processor cores.

There is a broad range of designs that would fall under the definition of MTP. The interface described in this chapter is intended to provide a set of common behaviors to enable operating system code and other privileged code to be common across SPARC JPS2 processors. This interface is not complete, as there is a range of implementation dependent features that will exist to configure and control these processors. It would be beneficial if this interface could be expanded over time to provide more functionality common across implementations.

## 9.1 Overview of MTP

The MTP Programming Model describes a set of privileged registers that are used for identification and configuration of MTPs. Equally important, the MTP Programming Model describes certain behavior that must be common for MTPs. The set of registers and the common behavior are covered in the following sections, grouped into a number of topics.

It is recommended that SPARC JPS2 processors that are not MTPs should implement a subset of the interface. This enables the processors to be more easily integrated into products that may also contain MTP parts. This also enables more consistent software to be deployed across all future products. Section 9.8, *Recommended Subset of MTP Interface for Non-MTP Parts*, provides the recommendation for non-MTP parts.

**Note –** Memory models for SPARC JPS2 processors do not change because of MTP.

## 9.1.1 MTP Definition

Defining a SPARC JPS2 MTP is critical to defining the scope of this standard interface. An MTP is defined by its external visible nature and not its internal organization. This definition is not fully consistent with the common hardware definition of MTP. The following section gives some background terminology followed by a description of the MTP definition.

### 9.1.1.1 Background Terminology

**Thread** The term *thread* is overused and ambiguous. Software and hardware have historically used it differently. From software's (operating system) perspective, the term thread refers to an entity that can be run on hardware, it is something that is scheduled and may or may not be actively running on hardware at any given time, and may migrate around the hardware of a system. From hardware's perspective, the term multithreaded processor refers to a processor that can run multiple software threads simultaneously. To avoid confusion the term thread is used exclusively in the manner that it is used by software, and specifically the operating system. A thread can be viewed in a practical sense as a Solaris process or lightweight process (LWP).

**Strand** The term *strand* is used to identify the hardware state used to hold a software thread in order to execute it. Strand is specifically the software visible architected state (PC, next PC, general-purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.) of a thread and any microarchitecture state required by hardware for its execution. "Strand" replaces the ambiguous term "hardware thread." The number of strands in a processor defines the number of threads that the operating system can schedule on that processor at any given time.

**Physical Core** The term *physical processor core*, or just *physical core*, is similar to the term fiber but represents a broader collection of hardware. A physical core includes an execution pipeline (fiber) and associated structures, such as caches, that are required for performing the execution of instructions from one or more software threads. A physical core contains one or more strands. The physical core provides the necessary resources for the threads on each strand to make forward progress at a reasonable rate. A multistranded physical core can execute multiple software threads either by time multiplexing or partitioning resources (or any combination thereof).

There is not a precise delineation of strand and physical core. And among different microarchitecture organizations the scope of the terms may vary. In general, in a given microarchitecture it will be apparent what constitutes a physical core. A physical core will be a highly integrated unit with a clearly

defined interface to further out levels of the memory hierarchy and the system interface unit. A physical core will contain a defined number of strands, that is, it will be able to handle that many software threads being scheduled on it at any given time.

**Processor** A *processor* is the unit on which a shared interface is provided to control the configuration and execution of a collection of strands. A processor contains one or more physical cores, each of which contains one or more strands. On a more physical side, a *processor* is a physical module that plugs into a system. A processor is expected to appear logically as a single agent on the system interconnect fabric.

Therefore, a simple processor that can only execute one thread at a time would be a processor with a single physical core which is single-stranded. A processor that follows the academic model of simultaneous multithreading (SMT) would be a processor with a single physical core where that physical core supports multiple strands in order to execute multiple threads at the same time (multistranded physical core). A processor that follows the academic model of a CMP would be a processor with multiple physical cores, each only supporting a single strand. A processor may also contain multiple physical cores, where each physical core is multistranded.

**Virtual Processor** The term *virtual processor core*, or *virtual processor*, is used to identify each strand in a processor. Each strand corresponds to a specific strand on a specific physical core where there may be multiple physical core each with multiple strands. In most respects a virtual processor appears to the system, and to the operating system software, as a processing unit equivalent to a traditional single-stranded processor. Each virtual processor has its own interrupt ID and the operating system can schedule independent threads on each virtual processor. How multiple virtual processors are achieved within a processor is an implementation issue, and as much as possible the software interface is independent of how multiple virtual processors are implemented. The term virtual processor is used in place of strand because of the common association of the term strand with multistranded physical cores.

### 9.1.1.2 MTP Definition

A SPARC JPS2 MTP is defined as a SPARC JPS2 processor that contains more than one virtual processor. In fact, the definition can be extended to also include SPARC JPS2 processors with only one virtual processor. This further definition actually applies to all SPARC JPS2 processors. The case of a single virtual processor is a special case and is intentionally left outside of the scope of this programming interface. Although, it is strongly recommended that all SPARC JPS2 processors, even ones with a single virtual processor, comply with the specification. Section 9.8, *Recommended Subset of MTP Interface for Non-MTP Parts*, provides the recommendation for the special case of a processor containing only one virtual processor. The interface is the same regardless if the virtual processors are provided by multiple physical cores, a single physical core with multiple strands, or even multiple physical cores each with multiple physical strands.

A *virtual processor core (virtual processor)* is a processing entity that can execute a software thread. A virtual processor has a number of key characteristics, and includes all the architecturally visible state to execute a thread (general-purpose registers, floating-point registers, process state, status registers, condition codes, etc.). Every virtual processor in a system has a unique interrupt address. The addressability of interrupts to individual virtual processors is a very important aspect of the MTP programming interface. The hardware must provide sufficient resources that every virtual processor within the processor makes forward progress at a reasonable rate.

All user-visible architected state of a processor is per virtual processor. The privileged architected state of a processor falls in a number of different categories. Some of the privileged architected state will be per virtual processor, so each virtual processor has its own copy. Some of the privileged architected state is per processor, so a single copy is visible and shared by all virtual processors. In some implementations there may also be privileged architected state that is per group of virtual processors, so there are multiple copies where each copy is shared by a non-overlapping subset of the virtual processors.

## 9.1.2 General MTP Behavior

In general, each virtual processor of an MTP behaves functionally as if it was an independent processor. This is an important aspect of MTPs because user code running on a virtual processor need not know whether or not that virtual processor is part of an MTP. At a high level, even most of the operating system privileged code treats virtual processors of an MTP as if each were an independent processor. Various software—boot, error, diagnostic, among others—must be aware of the MTP. This chapter deals chiefly with the interface this software has with an MTP.

Each virtual processor of an MTP obeys the same memory model semantics as if it was an independent processor. All multiprocessing and threading libraries and code must be able to operate on MTPs without modification.

There are significant performance implications of MTPs, especially when shared resources (such as caches) exist. The proximity of virtual processors will potentially mean drastically different communication costs for communicating between two virtual processors on the same MTP, as opposed to two virtual processors on different MTPs, adding another degree of nonuniform memory access (NUMA) to a system. For high performance, the operating system, and even some user applications, will want to program specifically for the NUMA nature of MTPs. There may also be important resource contention issues between virtual processors on the same MTP.

## 9.2 Accessing MTP Registers

A key part of the MTP Programming Model is a set of privileged registers. This section covers how these registers are organized and how these registers are accessed. The registers can be accessed by software running on one of the virtual processors of the MTP.

The MTP-specific registers can be accessed by privileged software running on one of the virtual processors as *ASI-mapped registers*. The SPARC instruction set provides a convenient way to map additional architected state through the use of address space identifiers (ASIs). This state is accessible through special load and store instructions that provide an ASI value and an address (virtual address). Certain address space identifier values are used to access main memory but with different behaviors than the default semantics of normal load and store operations. Other ASI values are used to access special state for configuration, diagnostics, or other uses. The MTP Programming Model defines a number of ASIs specifically for accessing the MTP-specific registers.

### 9.2.1 Types of MTP Registers

There are two main classes of MTP-specific registers: per-core registers and shared registers.

- For per-core MTP registers, there is a private copy of the register associated with each virtual processor.
- For shared MTP registers, there is a single copy of each register that is shared by all the virtual processors.

All the classes of MTP-specific registers can be accessed as ASI-mapped registers by privileged software running on one of the virtual processors. Software can access the per-core registers for the virtual processor it is running on as well as the shared registers. It is not possible to address (much less, read) the per-core registers of another virtual processor. The specific semantics for accessing the MTP registers through the ASI interface are described in Section 9.2.2, *Accessing MTP Registers Through ASI Interface*.

### 9.2.2 Accessing MTP Registers Through ASI Interface

Each MTP-specific register is accessible through an ASI address, a combination of address space identifier value and virtual address. All MTP registers are mapped into ASI values that are only accessible in privileged mode. The specific ASI number and virtual address of each MTP register is covered later in this document.

Each virtual processor can access the per-core MTP registers associated with that virtual processor. Accesses to these registers follow the sequential semantics on the virtual processor with which they are associated. It is the responsibility of the hardware to guarantee the sequential semantics on the accesses to these registers.

Each virtual processor can access all the shared MTP registers on its MTP. An update to a shared register from one virtual processor will be visible to all other cores. The ordering of accesses to shared registers from different cores is not defined, but there are a number of hardware rules that must be enforced.

- The hardware must guarantee that accesses to a shared register from the same virtual processor follow sequential semantics.
- The hardware must also guarantee that if multiple virtual processors attempt to store to the register at the same time, after the updates, the register contains the value from one of those stores. That is, stores to these registers must be performed atomically on all bits of the register.

Some registers can have additional hardware-enforced restrictions on updates.

All the MTP registers are 64-bit registers, although some of the bits of individual registers can be reserved or defined to a fixed value. *Reserved* register fields should always be written by software with values of those fields previously read from that register or with zeroes; they should read as zero in hardware. Software intended to run on future versions of MTPs should not assume that these fields will read as 0 or any other particular value. This software convention makes future expansion of the interface easier.

Only the `LDXA`, `LDDFA`, `STXA`, and `STDFA` instructions can be used to access the MTP registers. Only the Load extended from alternate space (`LDXA`) or Load double floating-point register from alternate space (`LDDFA`) instructions can be used to read MTP registers. Only the Store extended into alternate space (`STXA`) and the Store double floating-point register to alternate space (`STDFA`) instructions can be used to store to MTP registers. An attempt to access an MTP register with any other instruction results in a *data_access_exception* trap. Whether access with `LDDA`/`STDA` causes an exception trap is implementation-dependent (impl. dep. # 300, 301). For each MTP register there is a defined ASI number and virtual address for addressing the register.

# 9.3 MTP Identification

There are two per-core registers used for virtual processor identification. The first register identifies the virtual processor within the MTP. The second register identifies the unique interrupt address of the virtual processor within the entire system. There is also a range of ASI addresses reserved for implementation specific, per-core identification and/or configuration registers.

## 9.3.1 Register Specification

For each register defined in this document a table is provided that specifies the key attributes of the register. There are seven fields that specify a register:

1. **REGISTER ASI NAME**: The formal name of the register.
2. **ASI #**: The address space identifier number used for accessing the register from software running on the MTP.
3. **VA**: The virtual address used for accessing the register from software running on the MTP.
4. **SHARED?**: Is the register a “PER-CORE” or a “SHARED” register?
5. **ACCESS**: Is the register read-only, write-only or read/write?
6. **NOTE**: This field provides additional information when appropriate.

## 9.3.2 Core ID Register (`ASI_CORE_ID`)

The Core ID register, described in TABLE 9-1, is a read-only, per-core register that holds the ID value assigned by hardware to each implemented virtual processor. The ID value is unique within the MTP.

**TABLE 9-1** Core ID Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| `ASI_CORE_ID` | $63_{16}$ | $10_{16}$ | PER CORE | RD only | |

As shown in FIGURE 9-1, the Core ID register has two fields:

1. `core_id`, which represents this virtual processor’s number, as assigned by the hardware. The Core ID is encoded in 6 bits.

2. max_core_id, which is the bit-position index (bit number) of the most significant ‘1’ bit in the CORE_AVAILABLE register. This is the core ID of the highest-numbered implemented (but not necessarily enabled) virtual processor in this MTP.

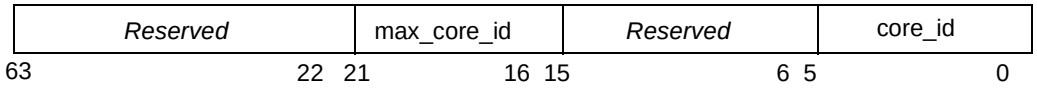



**FIGURE 9-1** Core ID Register

Many of the MTP-specific registers provide a bit mask wherein each bit corresponds to an individual virtual processor. For these registers, the core ID field indicates which bit of a bit mask corresponds to this specific virtual processor.

#### 9.3.2.1 Core Numbering Convention

The numbering of virtual processors may or may not be contiguous; system software may only assume that each core ID is unique within an MTP. In general, virtual processors should be numbered in a sequential, contiguous series starting with core number 0. When numbering the virtual processors within an MTP, this convention appears straight forward. There are cases, however, where this might not be so simple. This convention is recommended but not required.

In an MTP designed with many virtual processors, it is likely that partial-good chips – chips where a subset of the virtual processors and all the common area are good – would want to be salvaged and sold as MTPs with fewer virtual processors. In those cases, it would be preferable that the functional virtual processors be numbered as a contiguous series starting from 0 and the max_core_id field from the Core ID register be set to the highest numbered functional virtual processor. This requires some way to reprogram the virtual processor identity after manufacturing (such as with fuses). Reassigning identities to virtual processors after manufacturing might not be practical; if so, the functioning virtual processors may not be contiguously numbered.

### 9.3.3 Core Interrupt ID Register (ASI_INTR_ID)

This per-core register defined in TABLE 9-2 allows the software to assign a 16-bit interrupt ID to a virtual processor that is unique within the system. This is important to enable virtual processors to receive interrupts. The ID in this register is used by

other virtual processors (on the same and different MTPs) and other bus agents to address interrupts to this specific virtual processor. It can also be used by this core to identify the source of interrupts it issues to other virtual processors and bus agents.

**TABLE 9-2** Interrupt ID Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_INTR_ID | $63_{16}$ | $00_{16}$ | PER CORE | RD / WR | |

This register is Read / Write, Privileged access. It is expected to be changed only at boot or reconfiguration time.

As shown in FIGURE 9-2, the Core Interrupt ID register has only one field, a 16 bit interrupt ID field, int_id. Some implementations will not use all of the bits of the 16-bit interrupt ID. If an implementation uses less than 16 bits for int_id, the unused bits read as zero and writes to them are ignored.

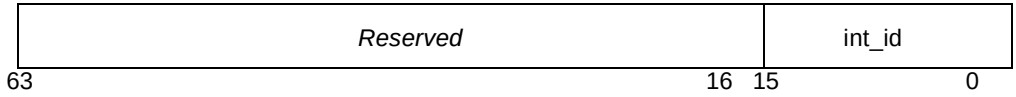



**FIGURE 9-2** Interrupt ID Register

### 9.3.3.1 Assigning Interrupt ID

**IMPL. DEP. #1100:** Whether any portion of the int_id field of the Core Interrupt ID register is read-only is implementation dependent.

When assigning the Interrupt ID to a virtual processor, the software must be aware of any interrupt routing conventions used in the system. Some portion of the ID might be required to follow a hardware convention to enable the interrupt to be correctly routed through the system interconnect. In some implementations, a part of the interrupt ID can be fixed by the processor to correspond to the core ID. This portion of the Interrupt ID can be read-only. Such requirements are both processor and system platform specific.

Each virtual processor in the MTP must have an interrupt ID that is unique within the system. If the Interrupt ID of multiple virtual processors in the same system are set to the same value, the behavior of the processor is undefined when an interrupt specifying that ID is sent or received.

### 9.3.3.2 Dispatching and Receiving Interrupts

The mechanisms used to dispatch and receive interrupts must work with the Interrupt ID register. An interrupt dispatch mechanism will have a way to specify the interrupt ID of the destination virtual processor the interrupt is to be delivered to. When a destination interrupt ID is specified, the interrupt must be delivered to

the virtual processor that has that ID in its Interrupt ID register. When using a legacy core to create an MTP, the existing interrupt dispatch and receive mechanism must be made compatible or replaced.

For legacy cores, the existing interrupt dispatch mechanism can be made to work with the Interrupt ID register. However the destination interrupt ID is specified, it must function that the interrupt is delivered to the virtual processor that has that ID in its Interrupt ID register. The existing mechanism used for identifying the interrupt ID of the virtual processor can be made to alias to the Interrupt ID field of the Interrupt ID register.

For legacy cores, the existing interrupt dispatch mechanism might need to be replaced to support MTPs. With MTPs, the number of virtual processors in the system will increase and the existing interrupt dispatch mechanism might not be able to address a sufficient number of virtual processors. In this case an alternative dispatch mechanism will be required. An example of such a mechanism would be an Interrupt Vector Dispatch Extension register, as described below.

### 9.3.3.3 Interrupt Vector Dispatch Extension Register (ASI_INTR_DISPATCH_EXT_W)

**IMPL. DEP. #1101:** Whether a SPARC JPS2 processor implements an Interrupt Vector Dispatch Extension register is implementation dependent

The Interrupt Vector Dispatch Extension register is a recommended alternative register to the Interrupt Vector Dispatch register. The reason for this new register is to increase the Interrupt ID from ten bits in legacy cores to 16 bits in MTP processors. The existing Interrupt Vector Dispatch register could also be preserved, for software compatibility. A write to either the Interrupt Vector Dispatch register or the Interrupt Vector Dispatch Extension register can be used to trigger interrupt delivery. The Interrupt Vector Dispatch Extension Register, like the Interrupt Vector Dispatch Register, is write-only.

**TABLE 9-3** Interrupt Dispatch Extension Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_INTR_DISPATCH_EXT_W | $77_{16}$ | see NOTE column | PER CORE | WR only | VA<13:0> = $78_{16}$<br>VA<63:14> specify target and source information as shown in FIGURE 9-3. |

The virtual address used when writing to the Interrupt Vector Dispatch Extension register is interpreted to determine the source ID, target ID (destination), and Busy / NACK Pair number. FIGURE 9-3 shows how the virtual address is interpreted.

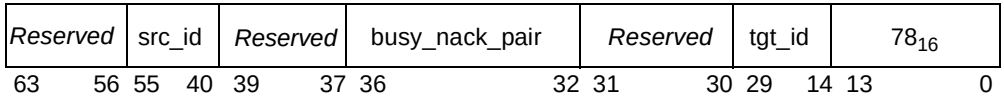



**FIGURE 9-3** Virtual Address of the Core Interrupt Vector Dispatch Extension Register

As shown in FIGURE 9-3, the virtual address of the Core Interrupt Vector Dispatch Extension register has three fields:

1. **`src_id`** – Specifies the core ID of the interrupt's source. This field is optional and implementation dependent. In many implementations this field will be ignored and the hardware will automatically use the virtual processor's interrupt ID as the source ID.
2. **`busy_nack_pair`** – Specifies which of the Busy / Nack pairs in the Core Interrupt Vector Status register to use for this interrupt.
3. **`tgt_id`** – Specifies the target virtual processor interrupt ID.

The data value written to the Interrupt Vector Dispatch Extension Register is ignored. The virtual address and the act of writing cause an interrupt to be sent.

**Extending the Interrupt Vector Receive Register.** If the Interrupt Vector Dispatch Extension Register is used to extend the interrupt ID field for sending interrupts, the Interrupt Vector Receive register must be extended to allow for a larger ID field. The existing Interrupt Vector Dispatch register (`ASI_INTR_RECEIVE`) at ASI# $49_{16}$ and VA $00_{16}$ can be used. The Source ID field in the data returned by reading the register can be simply extended by expanding the `sid_u` (upper bits of the source ID) from bits 10:6 to bits 16:6. This provides for a 16-bit interrupt ID to be returned by the Interrupt Vector Receive register.

### 9.3.3.4 Updating the Interrupt ID Register

It is expected that the interrupt ID register of virtual processors will be written once by software when a virtual processor is initially booted. It is assumed that at the time a virtual processor is being booted there will be no interrupt traffic in the system.

The latency from when software writes to the interrupt ID register and when the write takes effect is implementation dependent. Following the write with a `MEMBAR #sync` enforces that the write will be complete before any code executed on the virtual processor from after the `membar`.

The updates to the interrupt ID register will be atomic. If the value of the interrupt ID register is being written, the value observed at any time will be either the old value or the new value; no transient value will be observed. If an interrupt is issued to a virtual processor while its interrupt ID register is being updated, either addressed to its old or new interrupt ID, may or may not be received by the virtual processor. Once an interrupt is acknowledged for the new interrupt ID of a virtual processor, the virtual processor will not acknowledge any interrupts addressed to the old interrupt ID.

If an interrupt is issued to a system, addressed to an interrupt ID that does not match any virtual processors or other system agents, the interrupt will not be acknowledged and will cause an unmapped transaction error.

### 9.3.4 Reserved ASI Range

There is a range of ASI addresses reserved for implementation-specific, per-core identification and/or configuration registers. For ASI# $63_{16}$, virtual addresses of $40_{16}$ and above are reserved for implementation specific registers. These are intended for per-core registers required for either specific core implementations or for specific system architectures.

**TABLE 9-4** Reserved ASI Address Range

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| Reserved for implementation specific per-core registers | $63_{16}$ | $40_{16}$ and higher | PER CORE | Impl Depnd | |

## 9.4 Disabling and Parking Virtual Processors

The MTP programming model provides the ability to disable virtual processors and temporarily suspend (park) virtual processors. This section describes the interface for probing what virtual processors are available, enabled, and not suspended. This section also describes the interface for enabling/disabling virtual processors and parking/unparking virtual processors.

## 9.4.1 Core Available Register (ASI_CORE_AVAILABLE)

The Core Available register, shown in TABLE 9-5, is a shared register that indicates how many virtual processors are implemented in an MTP and what virtual processor numbers are assigned to them.

**TABLE 9-5** Core Available Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_CORE_AVAILABLE | $41_{16}$ | $00_{16}$ | SHARED | RD only | |

The Core Available register is a read-only register with a single 64-bit field (assuming a maximum of 64 virtual processors per MTP chip) in which each bit position corresponds to a virtual processor (see FIGURE 9-4). Bit 0 represents Virtual Processor 0; bit 63 represents Virtual Processor 63. If a bit in the register is asserted (1), the corresponding virtual processor is implemented and is functional in the MTP. If a bit in the register is not asserted (0), the corresponding virtual processor is not implemented or is malfunctional in the MTP. An implemented virtual processor is a virtual processor that can be enabled and used.

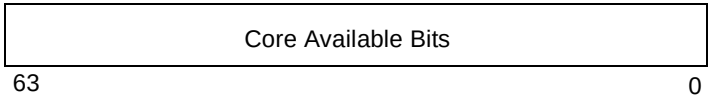



**FIGURE 9-4** Core Available Register

## 9.4.2 Enabling and Disabling Virtual Processors

The MTP Programming Model allows virtual processors to be enabled and disabled. Enabling or disabling a virtual processor is a heavyweight operation that requires a system reset (or equivalent reset that resets the entire MTP) for updates. A disabled virtual processor produces no architectural effects observable by other virtual processors, and do *not* participate in cache coherency. Any transaction issued to a disabled virtual processor, such as an interrupt, results in an “unmapped” reply or a time-out.

**IMPL. DEP. #1102:** Whether disabling a virtual processor reduces the power used by an MTP is implementation dependent.

### 9.4.2.1 Core Enable Status Register (ASI_CORE_ENABLE_STATUS)

The Core Enable Status register, described in TABLE 9-6, is a shared register that indicates whether each virtual processor is currently enabled. The register is a read-only register with a single 64-bit field (assuming a maximum of 64 virtual processors per MTP chip) in which each bit corresponds to a virtual processor.

**TABLE 9-6** Core Enable Status Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_CORE_ENABLE_STATUS | $41_{16}$ | $10_{16}$ | SHARED | RD only | |

As shown in FIGURE 9-5, bit 0 and bit 63 represents Virtual Processor 0 and Virtual Processor 63, respectively. If a bit in the register is asserted (1), the corresponding virtual processor is implemented and enabled. A virtual processor not implemented in an MTP, indicated as "not available" in the Core Available register, cannot be enabled and its corresponding enabled bit in this register will be 0. A virtual processor that is parked is still considered enabled.

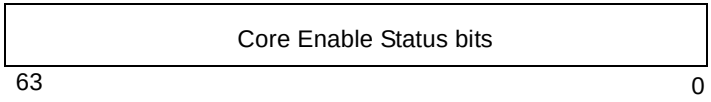



**FIGURE 9-5** Core Enable Status Register

**State After Reset.** The CORE_ENABLE_STATUS register changes only at the end of a system reset or at the end of an equivalent reset such as *power_on_reset*. At the end of such a reset, the contents of its CORE_ENABLE register are copied to CORE_ENABLE_STATUS register. This behavior is the same for *power_on_reset* and other system resets.

---

**Note –** In earlier specification drafts, Core Enable Status and ASI_CORE_ENABLE_STATUS were referred to as Core Enabled and ASI_CORE_ENABLED, respectively. Some JPS2 Extensions documents may still retain vestiges of the legacy terms "Core Enabled register" and "ASI_CORE_ENABLED."

---

### 9.4.2.2 Core Enable Register (ASI_CORE_ENABLE)

The Core Enable register is a shared register, used by software to enable/disable an MTP's virtual processors. A disabled virtual processor and any structures private to a disabled virtual processor behave as though they were not present. The disabling or enabling of a virtual processor takes effect only after a system reset (or equivalent reset). Specifically, the value of the CORE_ENABLE register takes effect at the end of a

system reset. The exact timing of the system reset may be implementation and system specific. The setup and hold time rules of the Core Enable register, with respect to the end of a system reset, is implementation dependent.

---

**Note –** For systems that use a system reset pin, the value of the core enable register takes effect when the system reset signal is de-asserted.

---

**TABLE 9-7** Core Enable Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_CORE_ENABLE | $41_{16}$ | $20_{16}$ | SHARED | RD/WR | Takes effect after reset |

As shown in FIGURE 9-6, the Core Enable register has only one 64-bit field (one bit per possible virtual processor), with bit *n* representing Virtual Processor *n*. A bit set to 1 means a virtual processor should be enabled after the next system reset and a bit set to 0 means a virtual processor should be disabled after the next reset. If a bit in the Core Available register is 0 (unavailable), hardware forces the corresponding bit in the Core Enable register to 0 and ignores attempts to write "1" to that bit.

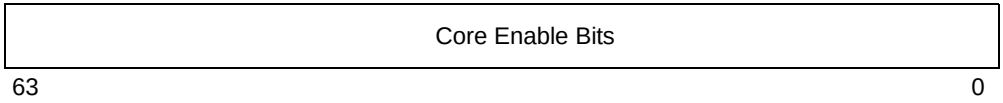



**FIGURE 9-6** CORE_ENABLE Register

**Restrictions on Updating `Core_Enable` Register.**

**IMPL. DEP. #1103:** Whether a restriction is provided to protect against all available virtual processors being disabled is implementation dependent.

**State After Reset.** The value of the CORE_ENABLE register is set to the value of the CORE_AVAILABLE register at the assertion of a power-on reset. The value of the CORE_ENABLE register remains unchanged during all other resets, including system resets or equivalent resets, unless explicitly updated by a service processor.

## 9.4.3 Parking and Unparking Virtual Processors

Parking is a way to temporarily suspend the operation of a virtual processor. Parked virtual processors can be later unparked to start them running again. The parking and unparking of virtual processors can be performed at arbitrary points in time and, unlike disabling a virtual processor, a system reset is not required. There may be an arbitrarily long, but bounded, delay from when a virtual processor is directed to park or unpark until the change takes effect. There is a Core Running Status register that can be used to determine if a virtual processor, that has been directed to park, has completed the process of becoming parked.

A parked virtual processor does not execute instructions and does not initiate any transactions on its own. A parked virtual processor does remain coherent with the system. To remain coherent, a parked virtual processor fully participates in cache coherency and can generate transactions in response to coherency requests from other virtual processors on the same or different MTP. When a virtual processor is unparked, it continues execution with the instruction that was next to be executed when the virtual processor was parked. It is transparent to the software running on a virtual processor that it was ever parked.

An interrupt to a parked virtual processor behaves the same as if the virtual processor was too busy to accept the interrupt. For example, if an interrupt buffer is available, the interrupt is ACKed and a trap is taken only when the virtual processor is unparked. If, however, no interrupt buffer is available, the interrupt is NACKed.

The STICK counter continues to count while a virtual processor is parked. The TICK counter will continue to count while a virtual processor is parked. Parking virtual processors is intended for critical diagnostic and recovery code. The interference with performance monitors using the TICK or STICK counters should not be a general issue. Using the TICK or STICK counter to detect the parking of a virtual processor is not recommended.

**IMPL. DEP. #1104:** Whether parking a virtual processor reduces the power used by an MTP is implementation dependent.

### 9.4.3.1 Core Running Register (ASI_CORE_RUNNING)

The Core Running register is a shared register, used by software to park and unpark selected MTP virtual processors. When a virtual processor is parked, the virtual processor stops executing new instructions and will not initiate transactions except in response to a coherency transaction initiated by another virtual processor. There may be an arbitrarily long, but bounded, delay from when the Core Running register is updated until the corresponding virtual processor(s) actually park or unpark.

The Core Running register can be accessed in a number of ways, depending on the virtual address used to access it (see TABLE 9-8). First, the register can be accessed for normal reading and writing (ASI_CORE_RUNNING_RW). Second, the register can be accessed as a write-only register where a write of 1 in a bit position sets the corresponding bits to 1 and a write of 0 in a bit position leaves the corresponding bit unchanged (ASI_CORE_RUNNING_WIS). Finally, the register can be accessed as a write-only register where a write of 1 in a bit position clears the corresponding bit 0 and a write of 0 in a bit position leaves the corresponding bit unchanged.

**TABLE 9-8** Core Running Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_CORE_RUNNING_RW | $41_{16}$ | $50_{16}$ | SHARED | RD / WR | General access |

**TABLE 9-8** Core Running Register *(Continued)*

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_CORE_RUNNING_W1S | $41_{16}$ | $60_{16}$ | SHARED | W1S | Write one to set bit(s) |
| ASI_CORE_RUNNING_W1C | $41_{16}$ | $68_{16}$ | SHARED | W1C | Write one to clear bit(s) |

To minimize the need for synchronization between virtual processors in writing the Core Running register, separate VAs are provided to set and clear the bits of this register. When writing to this register, there is a choice between writing a specific value to all bits and modifying individual bits. When a virtual processor parks itself, a write to the ASI_CORE_RUNNING_W1C should be used. When a virtual processor wants to become the only virtual processor active, it is more appropriate to write the desired value directly to ASI_CORE_RUNNING_RW. A direct write eliminates the need to perform a set and a clear operation to write a specific value to the register.

As shown in FIGURE 9-7, the Core Running register has only one 64-bit field (one bit per possible virtual processor), with bit *n* representing Virtual Processor *n*. A value of 1 in a bit position activates the corresponding virtual processor for normal execution, while a value of 0 in a bit position parks the corresponding virtual processor. If a bit in the Core Enabled register is 0 (not enabled), hardware forces the corresponding bit in the Core Running register to 0 and ignores attempts to write "1" to that bit.

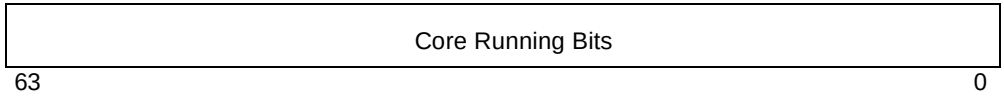



**FIGURE 9-7** Core Running Register

**Core Running Register – Updating.** When a virtual processor parks itself by updating the Core Running register and follows the update with a FLUSH instruction, the hardware will guarantee that no instruction after the FLUSH instruction will be executed until the virtual processor is unparked. The virtual address specified by the FLUSH instruction is not important. The FLUSH instruction may be either executed before parking takes effect or after the virtual processor is unparked. The FLUSH can, therefore, enable software to bound when parking takes effect in the case of a virtual processor parking itself.

**At Least One Virtual Processor Must Remain Unparked.** The hardware must enforce the restriction that an update to the Core Running register by software running on one of the virtual processors cannot cause all the enabled virtual processors to become parked. This restriction is important to avoid the dangerous case where all virtual processors become parked and there is no way to reactivate any of the virtual processors.

In the case that an update to the Core Running register would cause all virtual processors to become parked, the virtual processor performing the update must be automatically unparked by hardware. It is important that the virtual processor automatically unparked is the virtual processor that issued an update to the parking register. A virtual processor updating the Core Running register will be in a section of error diagnostic or other special code that is aware of the behavior and implications of parking. Arbitrarily unparking a virtual processor would be problematic because a virtual processor in the midst of running user code could become the only unparked virtual processor. If this were to happen, the only active virtual processor in the MTP would be unaware of the state of the MTP and would not know to check the running status of other virtual processors.

To demonstrate the complexity of this mechanism, it is worth noting some of the cases where all virtual processors can become parked. There are trivial cases where a virtual processor attempts to park all unparked virtual processors, including itself. In this case, the virtual processor updating the Core Running register remains unparked. More complicated cases are when multiple virtual processors attempt to park a subset of the virtual processors simultaneously. For example, assume an MTP with two virtual processors that both try to park the other virtual processor about the same time. Assume Virtual Processor 0 issues its update to the Core Running register first and sets Virtual Processor 1 to parked. There are two possible scenarios after this. First, the parking of Virtual Processor 1 take effect before it execute the instruction to park Virtual Processor 0. In this case, Virtual Processor 1 will be parked and then later, after unparking, it will continue on to park Virtual Processor 0. Second, although Virtual Processor 1 has been directed to park it continues to execute the instruction to park Virtual Processor 0. The update of Virtual Processor 1 would park all virtual processors, hence Virtual Processor 1 is automatically unparked. (Ironically, the loser of the race to park each other actually becomes the winner.)

**State after Reset.** On system reset (or equivalent reset that resets the entire MTP), Core Running is initialized by default such that all the virtual processors are parked, except for the lowest numbered enabled virtual processor. This provides an on-chip “boot master” virtual processor, reducing BootBus contention.

The initial value of Core Running is set at the assertion of a system reset based on the default set of virtual processors that is expected to be enabled after the reset. For power on resets, the value of the Core Running register is set so that the lowest-numbered available virtual processor is set to running and all other virtual processors are set to parked. For other system resets, the value of the Core Running register is set based on the value of the Core Enable register at the time of assertion of reset such that the lowest-numbered virtual processor set to be enabled is set to running and all other virtual processors are set to parked.

**Note –** For systems that use a system reset pin, the value of the Core Running register is updated upon assertion of the system reset signal.

### 9.4.3.2 Core Running Status Register (ASI_CORE_RUNNING_STATUS)

Since there is a delay from when a virtual processor is directed to park until it actually becomes parked, the Core Running Status register is provided to indicate when a virtual processor actually becomes parked. The Core Running Status register is a shared, read-only register (see TABLE 9-9) where each bit indicates if the corresponding virtual processor is active.

**TABLE 9-9** Core Running Status Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_CORE_RUNNING_STATUS | $41_{16}$ | $58_{16}$ | SHARED | RD only | |

As shown in FIGURE 9-8, the Core Running Status register has only one 64-bit field (one bit per possible virtual processor), with bit *n* representing Virtual Processor *n*. A value of 0 in a bit position corresponding to an enabled virtual processor indicates that a virtual processor is truly parked and will not execute any additional instructions or initiate any new transactions until it is unparked. All virtual processors that have a 0 in the Core Enabled register must have a 0 in the Running Status Register. A value of 1 in a bit position indicates that a virtual processor is active and can execute instructions and initiate transactions. All virtual processors that have a 1 in the Core Running register must have a 1 in the Core Running Status Register.

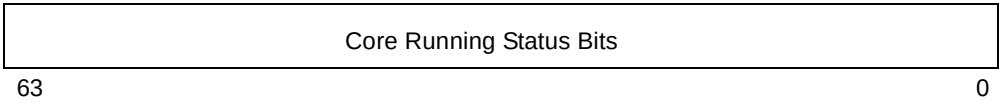



**FIGURE 9-8** Core Running Status Register

The intent of the Core Running Status register is to indicate when a virtual processor that has been directed to park is no longer executing any instructions or initiating any transactions (except in response to coherency transactions generated by other virtual processors) until it is unparked. The actual criteria used for indicating that a virtual processor is parked are implementation dependent. The hardware must guarantee that the Core Running Status register will make only a single transition (from 1 to 0) for a bit after the corresponding bit in the Core Running register has been changed from 1 to 0.

**State After Reset.** The value of the Core Running Status register is the same as the value of the Core Running register at the end of a system reset.

## 9.5 Reset and Trap Handling

Each reset is handled differently in an MTP. Some resets apply to all the virtual processors, some apply to an individual virtual processor, and some apply to an arbitrary subset. The following sections address how each type of reset is handled with respect to having multiple virtual processors integrated into a package. In general, the reset nomenclature used is consistent with SPARC JPS1 processors. Future processors may have a different classification of resets; if this is the case, the processors should extend this model appropriately.

Traps are virtual processor specific and are discussed in Section 9.5.4, *Traps*.

Initial value of MTP registers are discussed in Appendix O, *Reset, RED_state, and error_state*.

### 9.5.1 Per-Core Resets (SIR and WDR Resets)

The only resets that are limited to a single virtual processor are the resets internally generated by a virtual processor. Any SPARC JPS2 processor will have a number of resets of this class. For current SPARC processors, these are the software initiated reset (*SIR*) and the watchdog reset (*WDR*). These types of resets are generated by an individual virtual processor and are not propagated to the other virtual processors on an MTP.

### 9.5.2 Full-MTP Resets (System Reset)

There is a class of resets that are generated by an external agent and apply to all the virtual processors in an MTP. These include any reset that can be associated with fundamentally reconfiguring the MTP chip. Current SPARC processors have a System Reset, of which Power-On Reset is a special case. This is a reset that is required for certain reconfigurations of the processor. Future processors may have multiple resets that replace the single System Reset of current processors.

The Power-On and System resets (or their equivalents in future processors) are sent to all virtual processors in an MTP processor. As discussed in *Core Running Register (ASI_CORE_RUNNING)* on page 200, all the virtual processors except the lowest enabled virtual processor will be set, by default, to parked at the beginning of system reset. The unparked virtual processor is the default master core, which

should arbitrate for the BootBus (if multiple MTP chips share the same BootBus). The master core should unpark the other virtual processors at the proper time in the booting process.

## 9.5.3 Partial MTP Resets (XIR Reset)

There is a class of resets that are generated by an external agent and apply to an arbitrary subset of virtual processors within an MTP. The subset may be anything from all virtual processors to no virtual processors. Current SPARC JPS2 processors have, in addition to a system reset, an additional externally initiated reset called an XIR. This is a reset intended to reset a specific processor in a system, primarily for diagnostic and recovery purposes. Future processors may have multiple resets that replace the single XIR reset of current processors.

For this class of resets there must be a mechanism to specify which subset of virtual processors should be reset. There are two possible ways to specify the subset. The first way to specify the subset is to have a steering register that is set up ahead of time to specify the subset of virtual processors. For systems using an XIR reset, the XIR Steering register described in Section 9.5.3.1 on page 206 should be used.

The second way to specify the subset is to specify the subset concurrently with delivering the reset across the interface used for communicating the reset. This method would require that the interface used for communicating resets supports sending packets of information along with the resets.

For future systems that replace XIR reset with a different set of resets, there is a fixed set of rules for extending this MTP programming interface. Each reset of this class may use an interface where along with the reset comes the set of virtual processors to reset. The highest priority reset of this class, which does not have a subset come along with the reset, will use the XIR Steering register to determine the subset of virtual processors that will be reset. Subsequent lower-priority resets of this class, which do not have a subset come along with the reset, can either use the XIR Steering register or can have an additional steering register comparable to the XIR Steering register associated with each type of reset. These additional registers will have the same ASI number, $41_{16}$, of the XIR Steering register and will use a currently unassigned virtual address. Subsequent generations should be consistent with the virtual addresses used.

### 9.5.3.1 XIR Steering Register (ASI_XIR_STEERING)

The externally initiated reset (XIR) can be steered to an arbitrary subset of virtual processors under the control of the XIR steering register. The XIR Steering register, defined in TABLE 9-10, is a shared register used by software to control which MTP virtual processor(s) will get the XIR reset when the XIR is asserted for the processor module.

**TABLE 9-10** XIR Steering Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_XIR_STEERING | $41_{16}$ | $30_{16}$ | SHARED | RD / WR | General access |

**IMPL. DEP. #1105a:** Whether XIR_STEERING is a read-only register or read / write register is implementation dependent. If XIR_STEERING is read-only, writes to it are ignored and XIR_STEERING is set to steer XIRs to all available virtual processors (that is, XIR_STEERING reads identically to CORE_AVAILABLE).

As shown in FIGURE 9-9, the XIR Steering register has only one 64-bit field (one bit per virtual processor), with bit *n* representing virtual processor *n*. When an XIR is asserted for the processor module, each virtual processor with its corresponding bit in the XIR steering register set (value of 1) is directed to take an XIR reset. Virtual processors with the corresponding bit cleared (value of 0) continue execution, being unaware of the XIR. If a bit in the Core Enable Status register is 0 (not enabled), hardware forces the corresponding bit in the XIR Steering register to 0 and ignores attempts to write “1” to that bit.

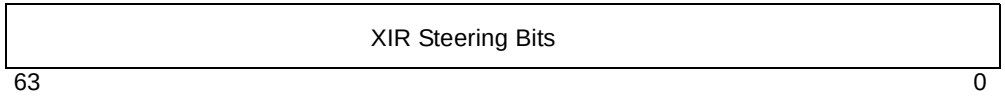



**FIGURE 9-9** XIR Steering Register

A virtual processor that is parked when it receives an XIR reset remains parked and will take the XIR reset immediately after being unparked. Software may, if it wishes, have the first action of a virtual processor that gets an XIR to unpark all other virtual processors. This will produce the effect of all the virtual processors being unparked and reset by an XIR.

At the end of a system reset (or equivalent resets that resets the entire MTP), the XIR Steering register is set to equal the value of the Core Enable Status register. This provides for a default so that all enabled virtual processors receive an XIR reset on assertion of XIR.

**State After Reset.** At the end of a system reset (or equivalent reset), the value of the XIR reset is equal to the value of the Core Enable register if the register is implemented as a read / write register. If the register is implemented as a read-only register, the value is unchanged on a system reset.

### 9.5.4 Traps

Most traps are associated with an action within a virtual processor, and these traps apply only to the associated virtual processor.

Traps, which are issued due to detected errors, are, for the most part, also maintained per virtual processor. If the error is in a structure private to a virtual processor, the error generates a trap for that virtual processor. Traps due to errors in structures that are shared between two or more virtual processors in an MTP are handled in one of two ways:

1. Shared structures, such as the bus interface unit, should maintain a virtual processor "tag" such that an error occurring on a request from a virtual processor can be reported back to that virtual processor. An example would be a bus unmapped error.

2. Traps due to errors on shared structures where the source cannot be traced to a requesting virtual processor are handled as described in Section 9.6.2, *Non-Core-Specific Error Reporting*. An example would be an ECC error on a copyback.

The ways for handling traps for non-core specific errors are implementation dependent and are described in the next section.

## 9.6 Error Handling in MTPs

Errors in a structure private to a virtual processor are reported to that virtual processor, using the error reporting mechanism of that virtual processor. These errors are considered core-specific.

An error in a shared structure is, whenever possible, reported to the virtual processor initiating the request that caused or detected the error. These errors are considered core-specific. Some errors in a shared structure cannot be attributed to a virtual processor, and are, therefore, noncore-specific.

The following sections describe how an MTP handles both core-specific and noncore-specific errors.

## 9.6.1 Core-Specific Error Reporting

Core-specific errors are reported to that virtual processor, using the virtual processor's error reporting mechanism. These errors consist of both synchronous and asynchronous errors. They also include errors that occur in shared structures. It is the responsibility of the error handling software to recognize the implication of errors in shared structures and take appropriate action.

For legacy cores used to create MTPs, there may be new types of errors. The error reporting mechanism may need to be extended appropriately to identify these new errors.

## 9.6.2 Non-Core-Specific Error Reporting

Non-core-specific errors are more complicated than core-specific errors. When a non-core-specific error occurs, it must be recorded; one of the virtual processors within the MTP must be trapped to deal with the error. Where to record the error and which virtual processor to trap is addressed in the following subsections.

By definition, noncore-specific errors are asynchronous errors (if they could be identified with an instruction they could be identified with a virtual processor) that occur in shared resources. Even within this set of asynchronous errors in shared structures, as many errors as possible should be attributed to a specific virtual processor and reported as a core-specific error.

### 9.6.2.1 Error Steering

When a noncore-specific error occurs in a shared resource, the error must be reported to one of the virtual processors that shares that resource. Error steering registers are used to determine which virtual processor will handle the error. Error steering registers are software configurable registers where software can specify which virtual processor should handle an error. That is, the error steering register defines to which virtual processor the error is reported and that virtual processor will be trapped to handle the error.

A SPARC JPS2 MTP implementation may have resources shared by all the virtual processors of the MTP, or private to only one virtual processor. A single error steering register is used and that error steering register follows the behavior of the `ASI_ERROR_STEERING` register defined below.

**General Guidelines for Error Steering Registers.** Error Steering registers are used to control which virtual processor handles non-core-specific errors. The specified virtual processor has the error recorded in its asynchronous error reporting mechanism (as appropriate) and takes an appropriate trap.

Error Steering registers specify a virtual processor by an encoded field that corresponds to the core ID of the targeted virtual processor. By using an encoded representation it is guaranteed that only one virtual processor can be specified. An error steering register should contain only one six-bit field, the `target_id` field, that encodes the core ID of the virtual processor that should be informed of noncore-specific errors. An implementation can implement a subset of the bits of `target_id` as long as the subset is sufficient to encode any of the corresponding core IDs that share the resource(s). If a subset of bits are implemented, the unused bits read as zero and writes to them are ignored.

It is the responsibility of the software to ensure that an Error Steering register identifies an appropriate virtual processor. If an error steering register identifies a virtual processor that is not available (or that does not share the corresponding resource), a noncore-specific error can either not be reported or it can be reported to another virtual processor, depending on how the hardware chooses to interpret invalid values in the register. If an error steering register identifies a virtual processor that is disabled, none of the enabled virtual processors will be affected by the reporting of a non-core specific error, but the behavior of specified disabled virtual processor is undefined (the fault status register of the disabled virtual processor may or may not be observed to have been updated). If an error steering register identifies a virtual processor that is parked, the noncore-specific error is reported to that virtual processor, which will take the appropriate trap but not until after it is unparked.

The timing of the update to an Error Steering register is not defined. If the store to the register is followed by a `MEMBAR` synchronization barrier, the completion of the barrier guarantees the completion of the update. When a non-core-specific error occurs, the appropriate error steering register is consulted. The error is reported to and an exception is generated in the virtual processor indicated by the Error Steering register. If a non-core-specific error occurs at the time the Error Steering register is being changed, the subsequent error report and the generated exception will occur together on the *same* virtual processor, either the one indicated by the old value in the Error Steering register or the one indicated by the new value. That is, for noncore-specific errors, the generation of an error report and an exception is atomic with respect to changes to the contents of the Error Steering register.

After a system reset (or equivalent reset that resets the entire MTP) the value in the `target_id` field of an Error Steering register should specify the lowest-numbered enabled virtual processor that corresponds to the resource(s) covered by the steering register.

**Error Steering Register (*`ASI_ERROR_STEERING`*).** The Error Steering register, defined in TABLE 9-11, is an Error Steering register for noncore-specific errors in resources shared by all virtual processors of the MTP. It is a shared register accessible from all virtual processors as well as being accessible from a service processor (if any). This register is used to control which virtual processor handles

corresponding non-core-specific errors. The specified virtual processor has the error recorded in its asynchronous error reporting mechanism and takes an appropriate trap.

The Error Steering register has only one six-bit field that encodes the core ID of the virtual processor that should be informed of noncore-specific errors. When an error that cannot be traced back to a virtual processor is detected, the error is recorded in, and a trap is sent to, the virtual processor identified by the Error Steering register.

**TABLE 9-11** Error Steering Register

| REGISTER ASI NAME | ASI # | VA | SHARED? | ACCESS | NOTE |
|---|---|---|---|---|---|
| ASI_ERROR_STEERING | $41_{16}$ | $40_{16}$ | SHARED | RD/WR | |

**IMPL. DEP. #1106a:** The number of implemented bits of ERROR_STEERING.target_id (see FIGURE 9-10) is implementation dependent, but must be sufficient to encode the highest implemented core ID. If fewer than six bits are implemented, the unused bits read as zero and writes to them are ignored.

If a subset of bits is implemented, the unused bits are read as zero and writes to them are ignored.

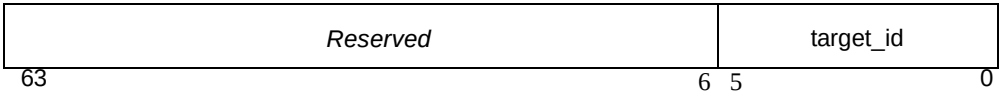



**FIGURE 9-10** Error Steering Register

### 9.6.2.2 Reporting Non-Core-Specific Errors

Before a trap can be generated for a noncore-specific error, the error must be recorded. Non-core-specific errors are recorded in the asynchronous error reporting mechanism of the virtual processor specified by the Error Steering register. The same asynchronous error reporting mechanism is used that is used for reporting core-specific errors. This reporting mechanism may require extending the virtual processor's asynchronous error reporting mechanism to enable it to record a larger set of errors.

Each asynchronous error is defined as either core-specific or non-core-specific. If the same error can occur as either a core-specific error or a non-core-specific error, it must be broken into two different errors for reporting purposes.

The type of trap sent to the virtual processor to handle a non-core-specific error is implementation-specific. A virtual processor can choose to use the same trap type used for corresponding core-specific asynchronous errors or it can choose to use a new trap type.

# 9.7 Additional MTP Software Interface

## 9.7.1 Diagnostic / RAS Registers

The MTP Software interface defines virtual processor disabling and parking that can be used for diagnostic and error recovery. The MTP software interface also defines how errors are reported in an MTP. Beyond this functionality no additional diagnostic or error recovery mechanisms are specified. It is up to the implementation to provide appropriate diagnostic and recovery mechanisms.

A future extension of the MTP Programming Model may include more common features for diagnostics and RAS. Increasing commonality without significantly limiting the implementation options is best.

## 9.7.2 Booting Support

Some of the registers previously described are used by the firmware for booting support. The Core Running register, described in *Core Running Register (ASI_CORE_RUNNING)* on page 200, is an example of such a register.

Only one of the virtual processors (the enabled virtual processor with the lowest `core_id`) is unparked after a system reset is asserted (or after initiation of equivalent reset that resets the entire MTP). If the service processor does not change the core parking register before the end of the reset, only this virtual processor fetches instructions after the reset. If the service processor wishes, it can configure any other, or all, virtual processors to be unparked. In this case, any unparked virtual processor starts fetching instructions after the reset.

For optimal booting, the software determines when virtual processors become unparked after reset. An expected behavior is to have only one virtual processor unparked when reset is removed. This virtual processor configures common registers and then unparks other virtual processors one at a time. This instance is only one possible boot sequence, and software can implement something else.

# 9.8 Recommended Subset of MTP Interface for Non-MTP Parts

New single virtual processor SPARC processors are recommended to implement a subset of the MTP interface. This enables the processors to be more easily integrated into products that may also contain MTP parts. This also enables more consistent software to be deployed across all future products.

These processors should implement all the registers, but many of the registers can be implemented as read-only registers with fixed hardware values where writes are ignored. The Interrupt ID register (ASI_INTR_ID) and the Scratchpad registers (ASI_SCRATCHPAD_*n*_REG) should be fully implemented. All the other registers can be implemented as read-only registers with fixed hardware values. The effect of write to these registers are implementation dependent (impl. dep. #305). TABLE 9-12 gives a summary of the recommended subset.

**TABLE 9-12** Recommended Subset for Non-MTP Parts

| Value | ASI Name | Type | VA | Note |
|---|---|---|---|---|
| $41_{16}$ | ASI_CORE_AVAILABLE | RD-only | $00_{16}$ | Read value of $01_{16}$ |
| $41_{16}$ | ASI_CORE_ENABLE_STATUS | RD-only | $10_{16}$ | Read value of $01_{16}$ |
| $41_{16}$ | ASI_CORE_ENABLE | RD-only | $20_{16}$ | Read value of $01_{16}$ |
| $41_{16}$ | ASI_XIR_STEERING | RD-only | $30_{16}$ | Read value of $01_{16}$ |
| $41_{16}$ | ASI_ERROR_STEERING | RD-only | $40_{16}$ | Read value of $00_{16}$ |
| $41_{16}$ | ASI_CORE_RUNNING_RW | RD-only | $50_{16}$ | Read value of $01_{16}$ |
| $41_{16}$ | ASI_CORE_RUNNING_STATUS | RD-only | $58_{16}$ | Read value of $01_{16}$ |
| $41_{16}$ | ASI_CORE_RUNNING_W1S | WR ignore | $60_{16}$ | Write ignore |
| $41_{16}$ | ASI_CORE_RUNNING_W1C | WR ignore | $68_{16}$ | Write ignore |
| $63_{16}$ | ASI_INTR_ID | Impl. Dep. | $00_{16}$ | Software assigned unique interrupt ID for core (read / write) |
| $63_{16}$ | ASI_CORE_ID | RD-only | $10_{16}$ | Read value of $00_{16}$ |

**Note –** A partially-working SPARC JPS2 MTP processor can implement a subset of, or the entire MTP specification.

# 9.9 Machine State Summary

Refer to TABLE 9-13 and TABLE 9-14 for a comprehensive machine state summary.

**TABLE 9-13** ASI Extensions

| Value | ASI Name | Shared? | Type | VA | Description |
|---|---|---|---|---|---|
| $41_{16}$ | ASI_CORE_AVAILABLE | SHARED | RD | $00_{16}$ | Bit mask of implemented virtual processors |
| $41_{16}$ | ASI_CORE_ENABLE_STATUS | SHARED | RD | $10_{16}$ | Bit mask of enabled virtual processors |
| $41_{16}$ | ASI_CORE_ENABLE | SHARED | RD/WR | $20_{16}$ | Bit mask of virtual processors to enable after next reset (read/write) |
| $41_{16}$ | ASI_XIR_STEERING | SHARED | RD/WR | $30_{16}$ | Bit mask of virtual processors to propagate XIR to (read/write) |
| $41_{16}$ | ASI_ERROR_STEERING | SHARED | RD/WR | $40_{16}$ | Specify ID of which virtual processor to trap on non-core-specific error (read/write) |
| $41_{16}$ | ASI_CORE_RUNNING_RW | SHARED | RD/WR | $50_{16}$ | Bit mask to control which virtual processors are running and which are parked (read/write). 1= active, 0 = parked |
| $41_{16}$ | ASI_CORE_RUNNING_STATUS | SHARED | RD | $58_{16}$ | Bit mask of virtual processors that are currently active. 1 = active, 0 = parked |
| $41_{16}$ | ASI_CORE_RUNNING_W1S | SHARED | W1S | $60_{16}$ | Bit mask to control which virtual processors are active and which are parked (write one to set) |
| $41_{16}$ | ASI_CORE_RUNNING_W1C | SHARED | W1C | $68_{16}$ | Bit mask to control which virtual processors are active and which are parked (write one to clear) |
| $63_{16}$ | ASI_INTR_ID | PER-CORE | RD/WR | $00_{16}$ | Software assigned unique interrupt ID for virtual processor (read/write) |
| $63_{16}$ | ASI_CORE_ID | PER-CORE | RD | $10_{16}$ | Hardware assigned ID for virtual processor (read-only) |
| $63_{16}$ | *Reserved* | PER-CORE | *Impl. Depnd,* | $40_{16}$ and higher | Reserved for implementation specific per-core registers |

**TABLE 9-14** Recommended ASI Extensions for Legacy Cores

| Value | ASI Name | Shared? | Type | VA | Description |
|---|---|---|---|---|---|
| $77_{16}$ | ASI_INTR_DISPATCH_EXT_W | PER CORE | WR only | VA<13:0> = $78_{16}$ | Alternative register to the Interrupt Vector Dispatch register used in many SPARC processors. This extension enables more virtual processors to be addressed in a system. |

FIGURE 9-11 illustrates MTP register state changes due to reset.

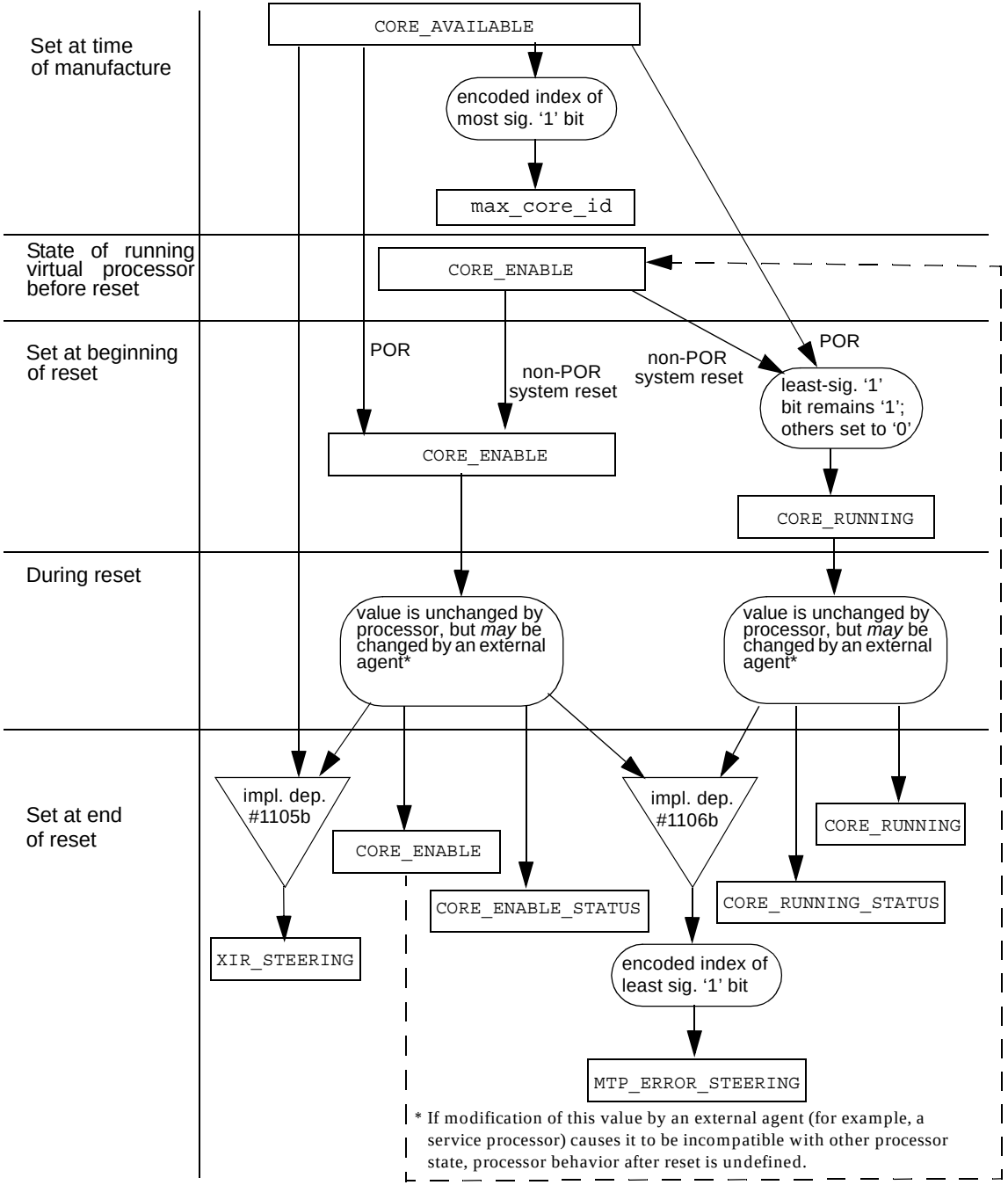



**FIGURE 9-11** MTP Register State Changes due to Reset

# 9.10 MTP State Transition

FIGURE 9-12 shows the MTP state transition for a SPARC JPS2 virtual processor.

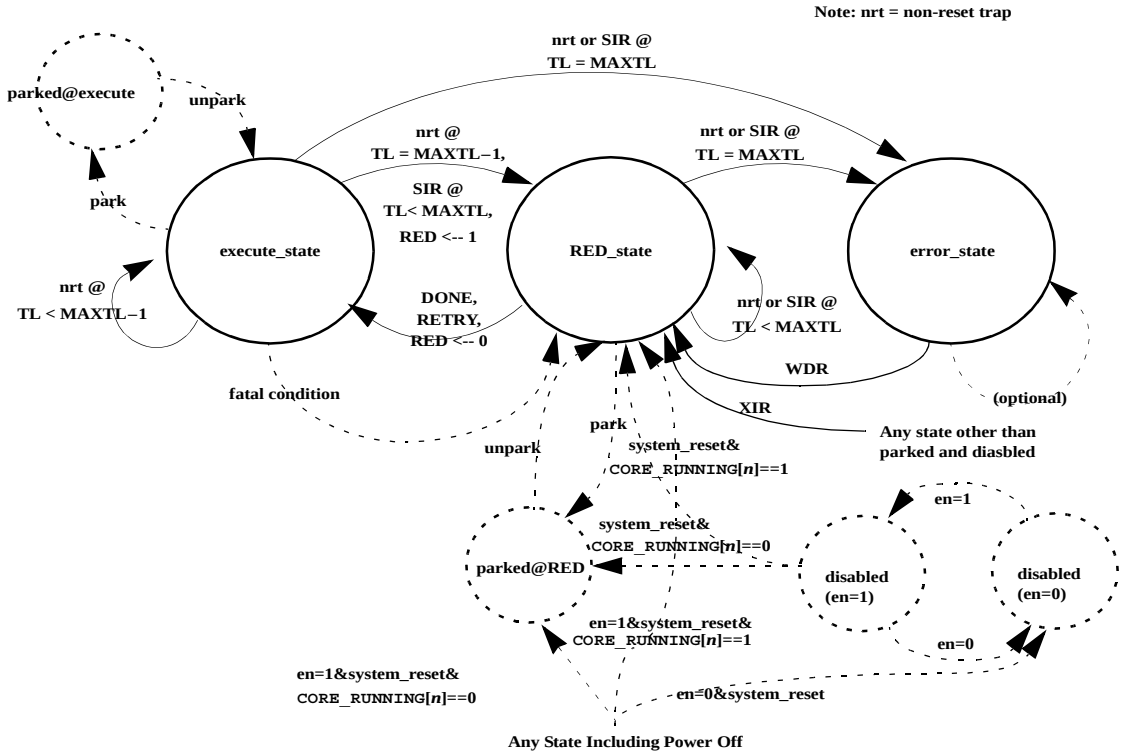



**FIGURE 9-12** MTP State Transition Diagram

# Instruction Definitions

The *SPARC Joint Programming Specification 2* extends the standard SPARC V9 instruction set with four classes of instructions:

- Low-power mode: `SHUTDOWN` (A.59)
- Enhanced graphics functionality: instructions for alignment (A.2); array handling (A.3); `BMASK` and `BSHUFFLE` (A.5); edge handling (A.12); logical operations on floating-point registers (A.33); and partitioned arithmetic and pixel manipulation (A.43 to A.47)
- Efficient memory access: partial store (A.42); short floating-point loads and stores (A.58); atomic load quadword (A.30); and block load and store (A.4)
- Efficient interval arithmetic: `SIAM` (A.55) and all instructions that reference `GSR.im`

Related instructions are grouped into subsections. Each subsection consists of these parts:

1. A table of the opcodes defined in the subsection with the values of the field(s) that uniquely identify the instruction(s).
2. An illustration of the applicable instruction format(s). In these illustrations a dash (—) indicates that the field is *reserved* for future versions of the architecture and shall be 0 in any instance of the instruction. If a conforming Sun SPARC implementation encounters nonzero values in these fields, its behavior is as defined in *Reserved Opcodes and Instruction Fields* on page 123. See Appendix I, *Extending the SPARC V9 Architecture*, for information about extending the SPARC V9 instruction set.
3. A list of the suggested assembly language syntax; the syntax notation is described in *Appendix G, Assembly Language Syntax*.
4. A description of the features, restrictions, and exception-causing conditions.
5. A list of exceptions that can occur as a consequence of attempting to execute the instruction(s). Exceptions due to an *instruction_access_error, instruction_access_exception*, *fast_instruction_access_MMU_miss*, *fast_ECC_error*†, *async_data_error*†, *ECC_error (corrected ECC_error)*, *WDR*,

and interrupts are not listed because they can occur on any instruction. A floating-point operation that is not implemented in hardware generates an *fp_exception_other* exception with `ftt` = *unimplemented_FPop* when executed. Non-floating-point instructions not implemented in hardware shall generate an *illegal_instruction* exception and therefore will not generate any of the other exceptions listed. The *illegal_instruction* exception is not listed because it can occur on any instruction that triggers an instruction breakpoint or contains an invalid field.

This appendix does not include any timing information (in either cycles or clock time), since timing is implementation dependent.

TABLE A-2 summarizes the instruction set; the instruction definitions follow the table. Within TABLE A-2, throughout this appendix, and in *Appendix E, Opcode Maps*, certain opcodes are marked with mnemonic superscripts. The superscripts and their meanings are defined in TABLE A-1.

**TABLE A-1** Opcode Superscripts

| Superscript | Meaning |
|---|---|
| D | Deprecated instruction |
| P | Privileged opcode |
| $P_{ASI}$ | Privileged action if bit 7 of the referenced ASI is 0 |
| $P_{ASR}$ | Privileged opcode if the referenced ASR register is privileged |
| $P_{NPT}$ | Privileged action if `PSTATE.priv` = 0 and (S)`TICK.npt` = 1 |
| $P_{PIC}$ | Privileged action if `PCR.priv` = 1 |
| $P_{PCR}$ | Privileged access to `PCR` (impl. dep. #250) |

**TABLE A-2** Instruction Set *(1 of 6)*

| Operation | Name | Page | Ext. to V9? |
|---|---|---|---|
| `ADD` (`ADDcc`) | Add (and modify condition codes) | 224 | |
| `ADDC` (`ADDCcc`) | Add with carry (and modify condition codes) | 224 | |
| `ALIGNADDRESS{_LITTLE}` | Calculate address for misaligned data | 226 | ✓ |
| `AND` (`ANDcc`) | And (and modify condition codes) | 291 | |
| `ANDN` (`ANDNcc`) | And not (and modify condition codes) | 291 | |
| `ARRAY(8,16,32)` | 3-D array addressing instructions | 228 | ✓ |
| `BPcc` | Branch on integer condition codes with prediction | 242 | |
| `Bicc`$^{D}$ | Branch on integer condition codes | 387 | |
| `BMASK` | Set the `GSR.mask` field | 235 | ✓ |

† Implementation-dependent exception; see *SPARC JPS2 Implementation-Dependent Traps* on page 167.

**TABLE A-2** Instruction Set *(2 of 6)*

| Operation | Name | Page | Ext. to V9? |
|---|---|---|---|
| BPr | Branch on contents of integer register with prediction | 237 | |
| BSHUFFLE | Permute bytes as specified by GSR.mask | 235 | ✓ |
| CALL | Call and link | 245 | |
| CASA$^{P_{ASI}}$ | Compare and swap word in alternate space | 246 | |
| CASXA$^{P_{ASI}}$ | Compare and swap doubleword in alternate space | 246 | |
| DONE$^{P}$ | Return from trap | 249 | |
| EDGE(8,16,32){L} | Edge handling instructions | 250 | ✓ |
| FABS(s,d,q) | Floating-point absolute value | 263 | |
| FADD(s,d,q) | Floating-point add | 253 | |
| FALIGNDATA | Perform data alignment for misaligned data | 226 | ✓ |
| FAND{S} | Logical AND operation | 288 | ✓ |
| FANDNOT(1,2){S} | Logical AND operation with one inverted source | 288 | ✓ |
| FBfcc$^{D}$ | Branch on floating-point condition codes | 384 | |
| FBPfcc | Branch on floating-point condition codes with prediction | 239 | |
| FCMP(s,d,q) | Floating-point compare | 255 | |
| FCMPE(s,d,q) | Floating-point compare (exception if unordered) | 255 | |
| FCMP(GT,LE,NE,EQ)(16,32) | Pixel compare operations | 323 | ✓ |
| FDIV(s,d,q) | Floating-point divide | 265 | |
| FdMULq | Floating-point multiply double to quad | 265 | |
| FEXPAND | Pixel expansion | 330 | ✓ |
| FiTO(s,d,q) | Convert integer to floating-point | 261 | |
| FLUSH | Flush instruction memory | 268 | |
| FLUSHW | Flush register windows | 270 | |
| FMOV(s,d,q) | Floating-point move | 263 | |
| FMOV(s,d,q)cc | Move floating-point register if condition is satisfied | 296 | |
| FMOVR(s,d,q) | Move f-p reg. if integer reg. contents satisfy condition | 302 | |
| FMUL(s,d,q) | Floating-point multiply | 265 | |
| FMUL8x16 | 8x16 partitioned product | 318 | ✓ |
| FMUL8x16(AU,AL) | 8x16 upper/lower α partitioned product | 319 | ✓ |
| FMUL8(SU,UL)x16 | 8x16 upper/lower partitioned product | 320 | ✓ |
| FMULD8(SU,UL)x16 | 8x16 upper/lower partitioned product | 321 | ✓ |
| FNAND{S} | Logical NAND operation | 288 | ✓ |
| FNEG(s,d,q) | Floating-point negate | 263 | |
| FNOR{S} | Logical NOR operation | 288 | ✓ |
| FNOT(1,2){S} | Copy negated source | 288 | ✓ |
| FPACK(16,32, FIX) | Pixel packing | 327, 328, 329 | ✓ |

**TABLE A-2** Instruction Set *(3 of 6)*

| Operation | Name | Page | Ext. to V9? |
|---|---|---|---|
| FPADD(16,32){S} | Pixel add (single) 16- or 32-bit | 315 | ✓ |
| FPMERGE | Pixel merge | 331 | ✓ |
| FONE{S} | One fill | 288 | ✓ |
| FOR{S} | Logical OR operation | 288 | ✓ |
| FORNOT(1,2){S} | Logical OR operation with one inverted source | 288 | ✓ |
| FPSUB(16,32){S} | Pixel subtract (single) 16- or 32-bit | 315 | ✓ |
| FsMULd | Floating-point multiply single to double | 265 | |
| FSQRT(s,d,q) | Floating-point square root | 267 | |
| FSRC(1,2){S} | Copy source | 288 | ✓ |
| F(s,d,q)TOi | Convert floating point to integer | 257 | |
| F(s,d,q)TO(s,d,q) | Convert between floating-point formats | 259 | |
| F(s,d,q)TOx | Convert floating point to 64-bit integer | 257 | |
| FSUB(s,d,q) | Floating-point subtract | 253 | |
| FXNOR{S} | Logical XNOR operation | 288 | ✓ |
| FXOR{S} | Logical XOR operation | 288 | ✓ |
| FxTO(s,d,q) | Convert 64-bit integer to floating-point | 261 | |
| FZERO{S} | Zero fill | 288 | ✓ |
| ILLTRAP | Illegal instruction | 271 | |
| IMPDEP2A | Implementation-dependent instructions | 272 | |
| IMPDEP2B | Implementation-dependent instructions (reserved) | 272 | |
| JMPL | Jump and link | 273 | |
| LDD$^{D}$ | Load integer doubleword | 394 | |
| LDDA$^{D,\ P_{ASI}}$ | Load integer doubleword from alternate space | 396 | |
| LDDA ASI_NUCLEUS_QUAD* | Load integer quadword, atomic | 283 | ✓ |
| LDDF | Load double floating-point | 274 | |
| LDDFA$^{P_{ASI}}$ | Load double floating-point from alternate space | 231 | |
| LDDFA ASI_BLK* | Block loads | 231 | ✓ |
| LDDFA ASI_FL* | Short floating point loads | 356 | ✓ |
| LDF | Load floating-point | 274 | |
| LDFA$^{P_{ASI}}$ | Load floating-point from alternate space | 274 | |
| LDFSR$^{D}$ | Load floating-point state register lower | 393 | |
| LDQF | Load quad floating-point | 274 | |
| LDQFA$^{P_{ASI}}$ | Load quad floating-point from alternate space | 274 | |
| LDSB | Load signed byte | 279 | |
| LDSBA$^{P_{ASI}}$ | Load signed byte from alternate space | 281 | |
| LDSH | Load signed halfword | 279 | |
| LDSHA$^{P_{ASI}}$ | Load signed halfword from alternate space | 281 | |

**TABLE A-2** Instruction Set *(4 of 6)*

| Operation | Name | Page | Ext. to V9? |
|---|---|---|---|
| LDSTUB | Load-store unsigned byte | 285 | |
| LDSTUBA$^{P_{ASI}}$ | Load-store unsigned byte in alternate space | 286 | |
| LDSW | Load signed word | 279 | |
| LDSWA$^{P_{ASI}}$ | Load signed word from alternate space | 281 | |
| LDUB | Load unsigned byte | 279 | |
| LDUBA$^{P_{ASI}}$ | Load unsigned byte from alternate space | 281 | |
| LDUH | Load unsigned halfword | 279 | |
| LDUHA$^{P_{ASI}}$ | Load unsigned halfword from alternate space | 281 | |
| LDUW | Load unsigned word | 279 | |
| LDUWA$^{P_{ASI}}$ | Load unsigned word from alternate space | 281 | |
| LDX | Load extended | 279 | |
| LDXA$^{P_{ASI}}$ | Load extended from alternate space | 281 | |
| LDXFSR | Load floating-point state register | 274 | |
| MEMBAR | Memory barrier | 293 | |
| MOV*cc* | Move integer register if condition is satisfied | 304 | |
| MOV*r* | Move integer register on contents of integer register | 309 | |
| MULS*cc*$^{D}$ | Multiply step (and modify condition codes) | 400 | |
| MULX | Multiply 64-bit integers | 311 | |
| NOP | No operation | 312 | |
| OR (OR*cc*) | Inclusive-or (and modify condition codes) | 291 | |
| ORN (ORN*cc*) | Inclusive-or not (and modify condition codes) | 291 | |
| PDIST | Pixel component distance | 325 | ✓ |
| POPC | Population count | 332 | |
| PREFETCH | Prefetch data | 334 | |
| PREFETCHA$^{P_{ASI}}$ | Prefetch data from alternate space | 334 | |
| RDASI | Read ASI register | 343 | |
| RDASR$^{P_{ASR}}$ | Read ancillary state register | 343 | |
| RDCCR | Read condition codes register | 343 | |
| RDDCR$^{P}$ | Read dispatch control register | 343 | |
| RDFPRS | Read floating-point registers state register | 343 | |
| RDGSR | Read graphic status register | 343 | |
| RDPC | Read program counter | 343 | |
| RDPCR$^{P_{PCR}}$ | Read performance control register | 343 | |
| RDPIC$^{P_{PIC}}$ | Read performance instrumentation counters | 343 | |
| RDPR$^{P}$ | Read privileged register | 341 | |
| RDSOFTINT$^{P}$ | Read per-virtual processor soft interrupt register | 343 | |

**TABLE A-2** Instruction Set *(5 of 6)*

| Operation | Name | Page | Ext. to V9? |
|---|---|---|---|
| RDSTICK$^{P_{NPT}}$ | Read system TICK register | 343 | |
| RDSTICK_CMPR$^{P}$ | Read system TICK compare register | 343 | |
| RDTICK$^{P_{NPT}}$ | Read TICK register | 343 | |
| RDTICK_CMPR$^{P}$ | Read TICK compare register | 343 | |
| RDY$^{D}$ | Read Y register | 343 | |
| RESTORE | Restore caller's window | 348 | |
| RESTORED$^{P}$ | Window has been restored | 351 | |
| RETRY$^{P}$ | Return from trap and retry | 249 | |
| RETURN | Return | 346 | |
| SAVE | Save caller's window | 348 | |
| SAVED$^{P}$ | Window has been saved | 351 | |
| SDIV$^{D}$ (SDIVcc$^{D}$) | 32-bit signed integer divide (and modify condition codes) | 390 | |
| SDIVX | 64-bit signed integer divide | 311 | |
| SETHI | Set high 22 bits of low word of integer register | 353 | |
| SHUTDOWN | Shut down the virtual processor | 358 | ✓ |
| SIAM | Set Interval Arithmetic Mode | 352 | ✓ |
| SIR | Software-initiated reset | 359 | |
| SLL | Shift left logical | 354 | |
| SLLX | Shift left logical, extended | 354 | |
| SMUL$^{D}$ (SMULcc$^{D}$) | Signed integer multiply (and modify condition codes) | 398 | |
| SRA | Shift right arithmetic | 354 | |
| SRAX | Shift right arithmetic, extended | 354 | |
| SRL | Shift right logical | 354 | |
| SRLX | Shift right logical, extended | 354 | |
| STB | Store byte | 366 | |
| STBA$^{P_{ASI}}$ | Store byte into alternate space | 368 | |
| STBAR$^{D}$ | Store barrier | 403 | |
| STD$^{D}$ | Store doubleword | 366 | |
| STDA$^{D, P_{ASI}}$ | Store doubleword into alternate space | 408 | |
| STDF | Store double floating-point | 360 | |
| STDFA$^{P_{ASI}}$ | Store double floating-point into alternate space | 363 | |
| STDFA ASI_BLK* | Block stores | 231 | ✓ |
| STDFA ASI_FL* | Short floating point stores | 356 | ✓ |
| STDFA ASI_PST* | Partial Store instructions | 313 | ✓ |
| STF | Store floating-point | 360 | |
| STFA$^{P_{ASI}}$ | Store floating-point into alternate space | 363 | |
| STFSR$^{D}$ | Store floating-point state register | 404 | |

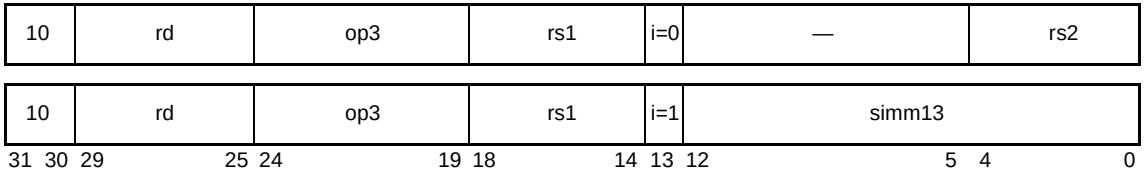
**TABLE A-2** Instruction Set *(6 of 6)*

| Operation | Name | Page | Ext. to V9? |
|---|---|---|---|
| STH | Store halfword | 404 | |
| STHA$^{P_{ASI}}$ | Store halfword into alternate space | 368 | |
| STQF | Store quad floating-point | 360 | |
| STQFA$^{P_{ASI}}$ | Store quad floating-point into alternate space | 363 | |
| STW | Store word | 366 | |
| STWA$^{P_{ASI}}$ | Store word into alternate space | 368 | |
| STX | Store extended | 366 | |
| STXA$^{P_{ASI}}$ | Store extended into alternate space | 368 | |
| STXFSR | Store extended floating-point state register | 360 | |
| SUB (SUBcc) | Subtract (and modify condition codes) | 370 | |
| SUBC (SUBCcc) | Subtract with carry (and modify condition codes) | 370 | |
| SWAP$^{D}$ | Swap integer register with memory | 410 | |
| SWAPA$^{D, P_{ASI}}$ | Swap integer register with memory in alternate space | 412 | |
| TADDcc (TADDccTV$^{D}$) | Tagged add and modify condition codes (trap on overflow) | 372, 414 | |
| Tcc | Trap on integer condition codes | 374 | |
| TSUBcc (TSUBccTV$^{D}$) | Tagged subtract and modify condition codes (trap on overflow) | 373, 416 | |
| UDIV$^{D}$ (UDIVcc$^{D}$) | Unsigned integer divide (and modify condition codes) | 390 | |
| UDIVX | 64-bit unsigned integer divide | 311 | |
| UMUL$^{D}$ (UMULcc$^{D}$) | Unsigned integer multiply (and modify condition codes) | 398 | |
| WRASI | Write ASI register | 380 | |
| WRASR$^{P_{ASR}}$ | Write ancillary state register | 380 | |
| WRCCR | Write condition codes register | 380 | |
| WRDCR$^{P}$ | Write dispatch control register | 380 | |
| WRFPRS | Write floating-point registers state register | 380 | |
| WRGSR | Write graphic status register | 380 | |
| WRPCR$^{P_{PCR}}$ | Write performance control register | 380 | |
| WRPIC$^{P_{PIC}}$ | Write performance instrumentation counters register | 380 | |
| WRPR$^{P}$ | Write privileged register | 377 | |
| WRSOFTINT$^{P}$ | Write per-virtual processor soft interrupt register | 380 | |
| WRSOFTINT_CLR$^{P}$ | Clear bits of per-virtual processor soft interrupt register | 380 | |
| WRSOFTINT_SET$^{P}$ | Set bits of per-virtual processor soft interrupt register | 380 | |
| WRTICK_CMPR$^{P}$ | Write TICK compare register | 380 | |
| WRSTICK$^{P}$ | Write System TICK register | 380 | |
| WRSTICK_CMPR$^{P}$ | Write System TICK compare register | 380 | |
| WRY$^{D}$ | Write Y register | 418 | |
| XNOR (XNORcc) | Exclusive-nor (and modify condition codes) | 291 | |
| XOR (XORcc) | Exclusive-or (and modify condition codes) | 291 | |

# A.1 Add

| Opcode | Op3 | Operation |
|---|---|---|
| ADD | 00 0000 | Add |
| ADDcc | 01 0000 | Add and modify cc's |
| ADDC | 00 1000 | Add with Carry |
| ADDCcc | 01 1000 | Add with Carry and modify cc's |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| add | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| addcc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| addc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| addccc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |

*Description:* ADD and ADDcc compute "R[rs1] + R[rs2]" if ***i*** = 0, or "R[rs1] + sign_ext(simm13)" if ***i*** = 1, and write the sum into R[rd].

ADDC and ADDCcc ("ADD with carry") also add the CCR register's 32-bit carry (icc.c) bit; that is, they compute "R[rs1] + R[rs2] + icc.c" or "R[rs1] + sign_ext(simm13) + icc.c" and write the sum into R[rd].

ADDcc and ADDCcc modify the integer condition codes (CCR.icc and CCR.xcc). Overflow occurs on addition if both operands have the same sign and the sign of the sum is different.

**Programming Note –** ADDC and ADDCcc read the 32-bit condition codes' carry bit (CCR.icc.c), not the 64-bit condition codes' carry bit (CCR.xcc.c).

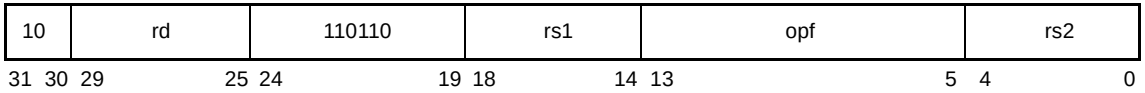
**JPS Compatibility Note –** ADDC and ADDCcc were named ADDX and ADDXcc, respectively, in SPARC V8.

*Exceptions:* None

# A.2 Alignment Instructions (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| ALIGNADDRESS | 0 0001 1000 | Calculate address for misaligned data access |
| ALIGNADDRESS_LITTLE | 0 0001 1010 | Calculate address for misaligned data access little-endian |
| FALIGNDATA | 0 0100 1000 | Perform data alignment for misaligned data |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| alignaddr | $reg_{rs1}$, $regr_{s2}$, $reg_{rd}$ |
| alignaddrl | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| faligndata | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

*Description:* ALIGNADDRESS adds two integer values, R[rs1] and R[rs2], and stores the result (with the least significant 3 bits forced to 0) in the integer register R[rd]. The least significant 3 bits of the result are stored in the GSR.align field.

ALIGNADDRESS_LITTLE is the same as ALIGNADDRESS except that the 2's complement of the least significant 3 bits of the result is stored in GSR.align.

**Note –** ALIGNADDR_LITTLE generates the opposite-endian byte ordering for a subsequent FALIGNDATA operation.

FALIGNDATA concatenates the two 64-bit floating-point registers specified by rs1 and rs2 to form a 128-bit (16-byte) intermediate value. The contents of the first source operand form the more-significant 8 bytes of the intermediate value, and the contents of the second source operand form the less significant 8 bytes of the intermediate value. Bytes in the intermediate value are numbered from most significant (byte 0) to least significant (byte 15). Eight bytes are extracted from the

intermediate value and stored in the 64-bit floating-point destination register specified by `rd`. `GSR.align` specifies the number of the most significant byte to extract (and, therefore, the least significant byte extracted from the intermediate value is numbered `GSR.align+7`).

A byte-aligned 64-bit load can be performed as shown in CODE EXAMPLE A-1.

**CODE EXAMPLE A-1** Byte-Aligned 64-Bit Load

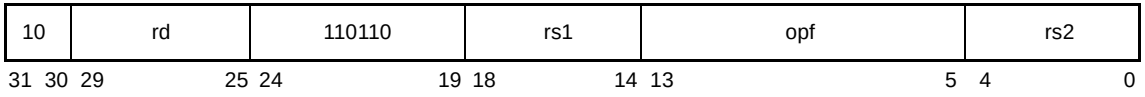
```
alignaddr   Address, Offset, Address
ldd         [Address], %f0
ldd         [Address + 8], %f2
faligndata %f0, %f2, %f4
```

*Exceptions:* *fp_disabled*

# A.3 Three-Dimensional Array Addressing Instructions (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| ARRAY8 | 0 0001 0000 | Convert 8-bit 3D address to blocked byte address |
| ARRAY16 | 0 0001 0010 | Convert 16-bit 3D address to blocked byte address |
| ARRAY32 | 0 0001 0100 | Convert 32-bit 3D address to blocked byte address |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| array8 | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| array16 | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| array32 | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |

*Description* These instructions convert three-dimensional (3D) fixed-point addresses contained in R[rs1] to a blocked-byte address; they store the result in R[rd]. Fixed-point addresses typically are used for address interpolation for planar reformatting operations. Blocking is performed at the 64-byte level to maximize external cache block reuse, and at the 64-Kbyte level to maximize TLB entry reuse, regardless of the orientation of the address interpolation. These instructions specify an element size of 8 bits (ARRAY8), 16 bits (ARRAY16), or 32 bits (ARRAY32). The second operand, R[rs2], specifies the power-of-2 size of the X and Y dimensions of a 3D image array. The legal values for rs2 and their meanings are shown in TABLE A-3. Illegal values produce undefined results in the destination register, R[rd].

**TABLE A-3** 3D R[rs2] Array X/Y Dimensions

| r[rs2] value | Number of elements |
|---|---|
| 0 | 64 |
| 1 | 128 |

**TABLE A-3** 3D R[rs2] Array X/Y Dimensions *(Continued)*

| r[rs2] value | Number of elements |
|---|---|
| 2 | 256 |
| 3 | 512 |
| 4 | 1024 |
| 5 | 2048 |

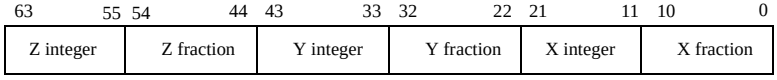



**FIGURE A-1** Three-Dimensional Array Fixed-Point Address Format

The integer parts of X, Y, and Z are converted to the following blocked-address formats.

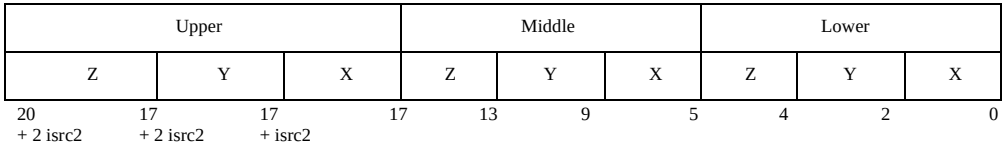



**FIGURE A-2** Three-Dimensional Array Blocked-Address Format (Array8)

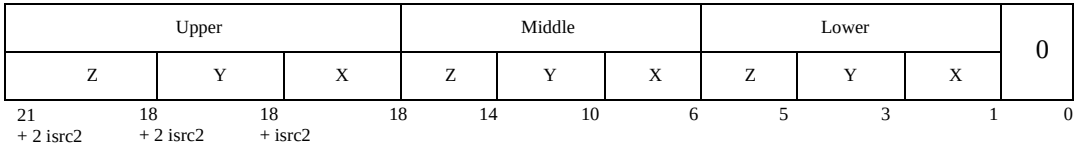



**FIGURE A-3** Three-Dimensional Array Blocked-Address Format (Array16)

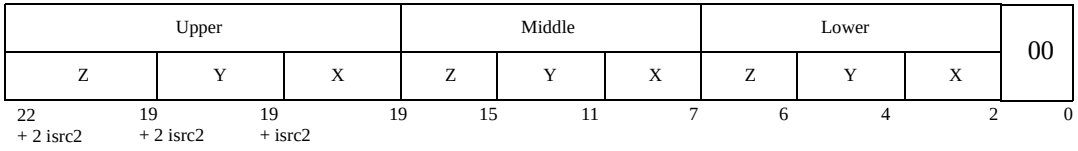



**FIGURE A-4** Three Dimensional Array Blocked-Address Format (Array32)

The bits above Z upper are set to 0. The number of zeroes in the least significant bits is determined by the element size. An element size of 8 bits has no zeroes, an element size of 16 bits has one zero, and an element size of 32 bits has two zeroes. Bits in X and Y above the size specified by `R[rs2]` are ignored.

The code fragment in CODE EXAMPLE A-2 shows assembly of components along an interpolated line at the rate of one component per clock.

**CODE EXAMPLE A-2** Assembly of Components Along an Interpolated Line

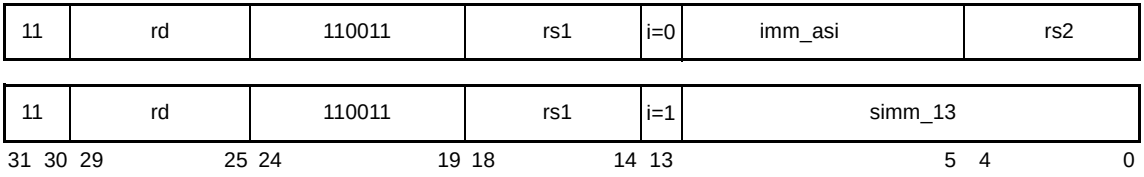
```
add         Addr, DeltaAddr, Addr
array8      Addr, %g0, bAddr
ldda        [bAddr] ASI_FL8_PRIMARY, data
faligndata data, accum, accum
```

*Exceptions* None

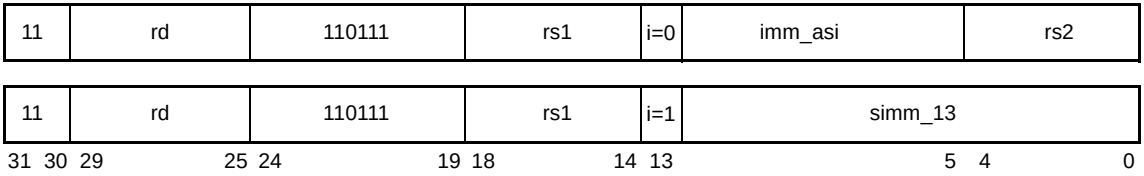
# A.4 Block Load and Store (VIS I)

| Opcode | imm_asi | ASI Value | Operation |
|---|---|---|---|
| LDDFA<br>STDFA | ASI_BLK_AIUP | $70_{16}$ | 64-byte block load / store from / to primary address space, user privilege |
| LDDFA<br>STDFA | ASI_BLK_AIUS | $71_{16}$ | 64-byte block load / store from / to secondary address space, user privilege |
| LDDFA<br>STDFA | ASI_BLK_AIUPL | $78_{16}$ | 64-byte block load / store from / to primary address space, little-endian, user privilege |
| LDDFA<br>STDFA | ASI_BLK_AIUSL | $79_{16}$ | 64-byte block load / store from / to secondary address space, little-endian, user privilege |
| LDDFA<br>STDFA | ASI_BLK_P | $F0_{16}$ | 64-byte block load / store from / to primary address space |
| LDDFA<br>STDFA | ASI_BLK_S | $F1_{16}$ | 64-byte block load / store from / to secondary address space |
| LDDFA<br>STDFA | ASI_BLK_PL | $F8_{16}$ | 64-byte block load / store from / to primary address space, little-endian |
| LDDFA<br>STDFA | ASI_BLK_SL | $F9_{16}$ | 64-byte block load / store from / to secondary address space, little-endian |
| STDFA | ASI_BLK_COMMIT_P | $E0_{16}$ | 64-byte block commit store to primary address space |
| STDFA | ASI_BLK_COMMIT_S | $E1_{16}$ | 64-byte block commit store to secondary address space |

*Format (3) LDDFA*



*Format (3)* `STDFA`




| Assembly Language Syntax | |
|---|---|
| `ldda` | [*reg_addr*] *imm_asi*, *freg*$_{rd}$ |
| `ldda` | [*reg_plus_imm*] `%asi`, *freg*$_{rd}$ |
| `stda` | *freg*$_{rd}$, [*reg_addr*] *imm_asi* |
| `stda` | *freg*$_{rd}$, [*reg_plus_imm*] `%asi` |

*Description*

A block load or store instruction uses an `LDDFA` or `STDFA` instruction combined with a block transfer ASI. Block transfer ASIs allow block loads and stores to be performed accessing the same address space as normal loads and stores. Little-endian ASIs (those with an 'L' suffix) access data in little-endian format; otherwise, the access is assumed to be big-endian. Byte swapping is performed separately for each of the eight double-precision registers used by the instruction. Endianness does not matter if these instructions are only being used for a block copy operation.

A block store with commit forces the data to be written to memory and invalidates copies in all caches present. As a result, a block store with commit maintains coherency with the I-cache.† It does not, however, flush instructions that have already been fetched into the pipeline before executing the modified code. If a block store with commit is used to write modified instructions, a `FLUSH` instruction must still be executed to guarantee that the instruction pipeline is flushed. (See *Synchronizing Instruction and Data Memory* on page 183 for more information.)

`LDDFA` with a block transfer ASI loads 64 bytes of data from a 64-byte aligned memory area into the eight double-precision floating-point registers specified by `rd`. The lowest-addressed eight bytes in memory are loaded into the lowest-numbered double-precision destination register. An *illegal_instruction* exception occurs if the floating-point registers are not aligned on an eight-double-precision register boundary. The least significant 6 bits of the memory address must be 0 or a *mem_address_not_aligned* exception occurs.

`STDFA` with a block transfer ASI stores data from the eight double-precision floating-point registers specified by `rs1` to a 64-byte-aligned memory area. The lowest-addressed eight bytes in memory are stored from the lowest-numbered double-

† Although all stores on JPS2 virtual processors coherently update the instruction cache (see page 183), stores (other than Block Store with Commit) on SPARC V9 implementations in general do *not* maintain coherency between instruction and data caches.

precision `rd`. An *illegal_instruction* exception occurs if the floating-point registers are not aligned on an eight-register boundary. The least significant 6 bits of the memory address must be 0 or a *mem_address_not_aligned* exception occurs.

ASIs $E0_{16}$ and $E1_{16}$ are only used for block store-with-commit operations; they are not used for block load operations. See *Block Load and Store ASIs* on page 582 for more information.

---

**Programming Note –** Block load does not provide register dependency interlocks, as ordinary load instructions do.

---

Before block load data can be referenced, a second block load (to a different set of registers) or a `MEMBAR #Sync` must be performed. If a second block load is used to synchronize against returning data, the virtual processor will continue execution before all data has been returned. The programmer is then responsible for scheduling instructions so registers are only used when they become valid.

To determine when data is valid, the programmer must count instruction groups containing FP operate instructions (not FP loads or stores). The lowest-numbered destination register of the first block load may be referenced in the first instruction group following the second block load, using an FP operate instruction only.

The second-lowest-numbered destination register of the first block load may be referenced in the second instruction group containing an FP operate instruction, and so on.

If this block-load/block-load synchronization mechanism is used, the initial reference to the block load data must be an FP operate instruction (not an FP store), and only instruction groups with FP operate instructions are counted when determining block load data availability.

If these rules are violated, data from before or after the block load may be returned by a reference to any of the block load's destination registers.

If a `MEMBAR #Sync` is used to synchronize on block load data, there are no restrictions on data usage, although performance will be lower than if block-load/block-load synchronization is used. No other `MEMBAR`s can be used to provide data synchronization for block load.

FP operate instructions can be issued in a single instruction group with FP stores. If block-load/block-load synchronization is used, FP operates and FP stores can be interlaced. This allows an FP operate instruction, such as `FMOVD` or `FALIGNDATA`, to reference the returning data before using the data in any FP store (normal store or block store).

The virtual processor also continues execution, without register interlocks, before all the store data for block stores are transferred from the register file.

If store source registers are overwritten before the next block store or `MEMBAR #Sync` instruction, then the following rule must be observed: The first register can be overwritten in the same instruction group as the block store, the second register can be overwritten in the instruction group following the block store, and so on. If this rule is violated, the block store may use the old or the new (overwritten) data.

When determining correctness for a code sample, note that JPS2 implementations may interlock more than required above. For example, there may be partial register interlocks, such as on the lowest-number register.

Code that does not meet the above constraints may appear to work on a particular implementation. However, to be portable across all SPARC JPS2 implementations, all of the above rules should be followed.

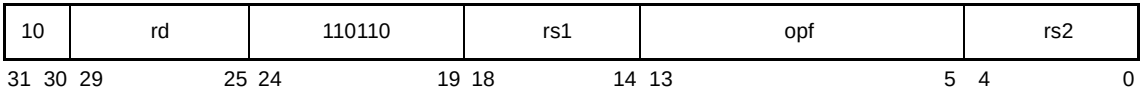
*Exceptions*

*fp_disabled*
*PA_watchpoint* (recognized on only the first 8 bytes of a transfer)
*VA_watchpoint* (recognized on only the first 8 bytes of a transfer)
*illegal_instruction* (misaligned `rd`)
*mem_address_not_aligned*
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection* (Block Store only)

# A.5 Byte Mask and Shuffle Instructions (VIS II)

| Opcode | opf | Operation |
|---|---|---|
| BMASK | 0 0001 1001 | Set the GSR.mask field in preparation for a following BSHUFFLE instruction |
| BSHUFFLE | 0 0100 1100 | Permute bytes as specified by GSR.mask |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| bmask | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| bshuffle | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

*Description:* BMASK adds two integer registers, R[rs1] and R[rs2], and stores the result in the integer register R[rd]. The least significant 32 bits of the result are stored in the GSR.mask field.

BSHUFFLE concatenates the two 64-bit floating-point registers specified by rs1 (more-significant half) and rs2 (less significant half) to form a 16-byte value. Bytes in the concatenated value are numbered from most significant to least significant, with the most significant byte being byte 0. BSHUFFLE extracts 8 of those 16 bytes and stores the result in the 64-bit floating-point register specified by rd. Bytes in the rd register are also numbered from most to least significant, with the most significant being byte 0. The following table indicates which source byte is extracted from the concatenated value for each byte in rd.

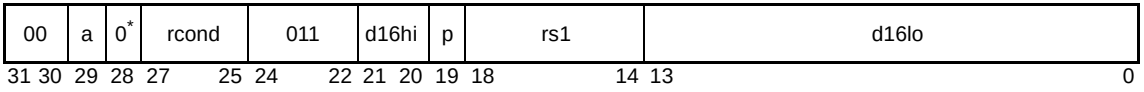
| Destination Byte (in R[rd]) | Source Byte |
|---|---|
| 0 (most significant) | (R[rs1] ▯ R[rs2])[GSR.mask<31:28>] |
| 1 | (R[rs1] ▯ R[rs2])[GSR.mask<27:24>] |
| 2 | (R[rs1] ▯ R[rs2])[GSR.mask<23:20>] |
| 3 | (R[rs1] ▯ R[rs2])[GSR.mask<19:16>] |
| 4 | (R[rs1] ▯ R[rs2])[GSR.mask<15:12>] |
| 5 | (R[rs1] ▯ R[rs2])[GSR.mask<11:8>] |
| 6 | (R[rs1] ▯ R[rs2])[GSR.mask<7:4>] |
| 7 (least significant) | (R[rs1] ▯ R[rs2])[GSR.mask<3:0>] |

*Exceptions:* *fp_disabled*

# A.6 Branch on Integer Register with Prediction (BPr)

| Opcode | rcond | Operation | RegisterContents Test |
|---|---|---|---|
| — | 000 | *Reserved* | — |
| BRZ | 001 | Branch on Register Zero | R[rs1] = 0 |
| BRLEZ | 010 | Branch on Register Less Than or Equal to Zero | R[rs1] ≤ 0 |
| BRLZ | 011 | Branch on Register Less Than Zero | R[rs1] < 0 |
| — | 100 | *Reserved* | — |
| BRNZ | 101 | Branch on Register Not Zero | R[rs1] ≠ 0 |
| BRGZ | 110 | Branch on Register Greater Than Zero | R[rs1] > 0 |
| BRGEZ | 111 | Branch on Register Greater Than or Equal to Zero | R[rs1] ≥ 0 |

*Format (2)*




* Although SPARC implementations should cause an *illegal_instruction* exception when bit 28 = 1, many early implementations ignored the value of this bit and executed the opcode as a BPr instruction even if bit 28 = 1.

| Assembly Language Syntax | |
|---|---|
| brz{,a}{,pt\|,pn} | $reg_{rs1}$, *label* |
| brlez{,a}{,pt\|,pn} | $reg_{rs1}$, *label* |
| brlz{,a}{,pt\|,pn} | $reg_{rs1}$, *label* |
| brnz{,a}{,pt\|,pn} | $reg_{rs1}$, *label* |
| brgz{,a}{,pt\|,pn} | $reg_{rs1}$, *label* |
| brgez{,a}{,pt\|,pn} | $reg_{rs1}$, *label* |

**Programming Note –** To set the annul bit for `BPr` instructions, append "`,a`" to the opcode mnemonic. For example, use "`brz,a %i3,`*label*." In the preceding table, braces signify that the "`,a`" is optional. To set the branch prediction bit `p`, append either "`,pt`" for predict taken or "`,pn`" for predict not taken to the opcode mnemonic. If neither "`,pt`" nor "`,pn`" is specified, the assembler shall default to "`,pt`".

*Description*

These instructions branch based on the contents of `R[rs1]`. They treat the register contents as a signed integer value.

A `BPr` instruction examines all 64 bits of `R[rs1]` according to the `rcond` field of the instruction, producing either a `TRUE` or `FALSE` result. If `TRUE`, the branch is taken; that is, the instruction causes a `PC`-relative, delayed control transfer to the address "`PC` + (4 * `sign_ext(d16hi || d16lo)`)." If `FALSE`, the branch is not taken.

If the branch is taken, the delay instruction is always executed, regardless of the value of the annul bit. If the branch is not taken and the annul bit (`a`) is 1, the delay instruction is annulled (not executed).

The predict bit (`p`) gives the hardware a hint about whether the branch is expected to be taken. A 1 in the `p` bit indicates that the branch is expected to be taken; a 0 indicates that the branch is expected not to be taken.

Annulment, delay instructions, prediction, and delayed control transfers are described further in Chapter 6, *Instructions.*

**V9 Compatibility Note –** If this instruction is implemented by tagging each register value with an N (negative) bit and Z (zero) bit, use the table below to determine if `rcond` is `TRUE`:

| Branch | Test |
|---|---|
| `BRNZ` | **not** Z |
| `BRZ` | Z |
| `BRGEZ` | **not** N |
| `BRLZ` | N |
| `BRLEZ` | N **or** Z |
| `BRGZ` | **not** (N **or** Z) |

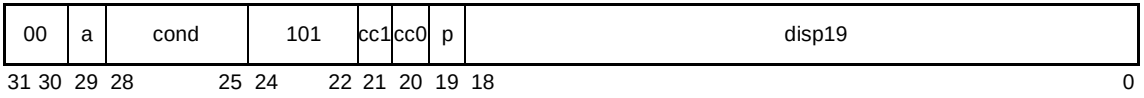
*Exceptions*

*illegal_instruction* (if `rcond` = $000_2$ or $100_2$)

# A.7 Branch on Floating-Point Condition Codes with Prediction (FBPfcc)

| Opcode | cond | Operation | *fcc* Test |
|---|---|---|---|
| FBPA | 1000 | Branch Always | 1 |
| FBPN | 0000 | Branch Never | 0 |
| FBPU | 0111 | Branch on Unordered | U |
| FBPG | 0110 | Branch on Greater | G |
| FBPUG | 0101 | Branch on Unordered or Greater | G **or** U |
| FBPL | 0100 | Branch on Less | L |
| FBPUL | 0011 | Branch on Unordered or Less | L **or** U |
| FBPLG | 0010 | Branch on Less or Greater | L **or** G |
| FBPNE | 0001 | Branch on Not Equal | L **or** G **or** U |
| FBPE | 1001 | Branch on Equal | E |
| FBPUE | 1010 | Branch on Unordered or Equal | E **or** U |
| FBPGE | 1011 | Branch on Greater or Equal | E **or** G |
| FBPUGE | 1100 | Branch on Unordered or Greater or Equal | E **or** G **or** U |
| FBPLE | 1101 | Branch on Less or Equal | E **or** L |
| FBPULE | 1110 | Branch on Unordered or Less or Equal | E **or** L **or** U |
| FBPO | 1111 | Branch on Ordered | E **or** L **or** G |

*Format (2)*



| cc1 cc0 | Condition Code |
|---|---|
| 00 | fcc*0* |
| 01 | fcc*1* |
| 10 | fcc*2* |
| 11 | fcc*3* |

| Assembly Language Syntax | | |
|---|---|---|
| fba{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbn{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbu{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbg{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbug{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbl{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbul{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fblg{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbne{,a}{,pt\|,pn} | %fcc*n*, *label* | (*synonym*: fbnz) |
| fbe{,a}{,pt\|,pn} | %fcc*n*, *label* | (*synonym*: fbz) |
| fbue{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbge{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbuge{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fble{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbule{,a}{,pt\|,pn} | %fcc*n*, *label* | |
| fbo{,a}{,pt\|,pn} | %fcc*n*, *label* | |

**Programming Note –** To set the annul bit for FBPfcc instructions, append ",a" to the opcode mnemonic. For example, use "fbl,a %fcc3,label." In the preceding table, braces signify that the ",a" is optional. To set the branch prediction bit, append either ",pt" (for predict taken) or "pn" (for predict not taken) to the opcode mnemonic. If neither ",pt" nor ",pn" is specified, the assembler shall default to ",pt". To select the appropriate floating-point condition code, include "%fcc0", "%fcc1", "%fcc2", or "%fcc3" before the label.

## Branch on Floating-Point Condition Codes with Prediction (FBPfcc)

*Description:* Unconditional branches and Fcc-conditional branches are described below.

- **Unconditional branches (`FBPA`, `FBPN`)** — If its annul field is 0, an `FBPN` (Floating-Point Branch Never with Prediction) instruction acts like a NOP. If the Branch Never's annul field is 0, the following (delay) instruction is executed; if the annul field is 1, the following instruction is annulled (not executed). In no case does an `FBPN` cause a transfer of control to take place.

  `FBPA` (Floating-Point Branch Always with Prediction) causes an unconditional `PC`-relative, delayed control transfer to the address "`PC` + (4 × `sign_ext(disp19)`)." If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

- **Fcc-conditional branches** — Conditional `FBPfcc` instructions (except `FBPA` and `FBPN`) evaluate one of the four floating-point condition codes (`fcc0`, `fcc1`, `fcc2`, `fcc3`) as selected by `cc0` and `cc1`, according to the `cond` field of the instruction, producing either a `TRUE` or `FALSE` result. If `TRUE`, the branch is taken, that is, the instruction causes a `PC`-relative, delayed control transfer to the address "`PC` + (4 × `sign_ext(disp19)`)." If `FALSE`, the branch is not taken.

  If a conditional branch is taken, the delay instruction is always executed, regardless of the value of the annul field. If a conditional branch is not taken and the `a` (annul) field is 1, the delay instruction is annulled (not executed). **Note**: The annul bit has a *different* effect on conditional branches than it does on unconditional branches.

  The predict bit (`p`) gives the hardware a hint about whether the branch is expected to be taken. A 1 in the `p` bit indicates that the branch is expected to be taken. A 0 indicates that the branch is expected not to be taken.

  Annulment, delay instructions, and delayed control transfers are described further in Chapter 6, *Instructions*.

If `FPRS.fef` = 0 or `PSTATE.pef` = 0, or if an FPU is not present, an `FBPfcc` instruction is not executed and instead, an *fp_disabled* exception is generated.

---

**JPS Compatibility Note –** Unlike SPARC V8, SPARC V9 does not require an instruction between a floating-point compare operation and a floating-point branch (`FBfcc`, `FBPfcc`).

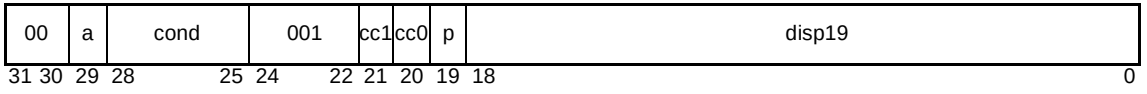
---

*Exceptions* *fp_disabled*

# A.8 Branch on Integer Condition Codes with Prediction (BPcc)

| Opcode | cond | Operation | *icc* Test |
|---|---|---|---|
| BPA | 1000 | Branch Always | 1 |
| BPN | 0000 | Branch Never | 0 |
| BPNE | 1001 | Branch on Not Equal | **not** Z |
| BPE | 0001 | Branch on Equal | Z |
| BPG | 1010 | Branch on Greater | **not** (Z **or** (N **xor** V)) |
| BPLE | 0010 | Branch on Less or Equal | Z **or** (N **xor** V) |
| BPGE | 1011 | Branch on Greater or Equal | **not** (N **xor** V) |
| BPL | 0011 | Branch on Less | N **xor** V |
| BPGU | 1100 | Branch on Greater Unsigned | **not** (C **or** Z) |
| BPLEU | 0100 | Branch on Less or Equal Unsigned | C **or** Z |
| BPCC | 1101 | Branch on Carry Clear (Greater Than or Equal, Unsigned) | **not** C |
| BPCS | 0101 | Branch on Carry Set (Less than, Unsigned) | C |
| BPPOS | 1110 | Branch on Positive | **not** N |
| BPNEG | 0110 | Branch on Negative | N |
| BPVC | 1111 | Branch on Overflow Clear | **not** V |
| BPVS | 0111 | Branch on Overflow Set | V |

*Format (2)*




| cc1 cc0 | Condition Code |
|---|---|
| 00 | *icc* |
| 01 | — |
| 10 | *xcc* |
| 11 | — |

| Assembly Language Syntax | | |
|---|---|---|
| `ba{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bn{,a}{,pt|,pn}` | *i_or_x_cc, label* | (or: `iprefetch` label) |
| `bne{,a}{,pt|,pn}` | *i_or_x_cc, label* | (*synonym*: `bnz`) |
| `be{,a}{,pt|,pn}` | *i_or_x_cc, label* | (*synonym*: `bz`) |
| `bg{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `ble{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bge{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bl{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bgu{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bleu{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bcc{,a}{,pt|,pn}` | *i_or_x_cc, label* | (*synonym*: `bgeu`) |
| `bcs{,a}{,pt|,pn}` | *i_or_x_cc, label* | (*synonym*: `blu`) |
| `bpos{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bneg{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bvc{,a}{,pt|,pn}` | *i_or_x_cc, label* | |
| `bvs{,a}{,pt|,pn}` | *i_or_x_cc, label* | |

**Programming Note –** To set the annul bit for `BPcc` instructions, append "`,a`" to the opcode mnemonic. For example, use `bgu,a %icc,label`. Braces in the preceding table signify that the "`,a`" is optional. To set the branch prediction bit, append to an opcode mnemonic either "`,pt`" for predict taken or "`,pn`" for predict not taken. If neither "`,pt`" nor "`,pn`" is specified, the assembler shall default to "`,pt`". To select the appropriate integer condition code, include "`%icc`" or "`%xcc`" before the label.

*Description:* Unconditional branches and conditional branches are described below:

- **Unconditional branches (`BPA`, `BPN`)** — A `BPN` (Branch Never with Prediction) instruction for this branch type (`op2` = 1) is used in SPARC V9 as an instruction prefetch; that is, the effective address (`PC` + (4 × `sign_ext(disp19)`)) specifies an address of an instruction that is expected to be executed soon. If the Branch Never's annul field is 1, then the following (delay) instruction is annulled (not executed). If the annul field is 0, then the following instruction is executed. In no case does a Branch Never cause a transfer of control to take place.

BPA (Branch Always with Prediction) causes an unconditional PC-relative, delayed control transfer to the address "PC + (4 × sign_ext(disp19))." If the annul field of the branch instruction is 1, then the delay instruction is annulled (not executed). If the annul field is 0, then the delay instruction is executed.

- **Conditional branches** — Conditional BPcc instructions (except BPA and BPN) evaluate one of the two integer condition codes (icc or xcc), as selected by cc0 and cc1, according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken; that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 × sign_ext(disp19))." If FALSE, the branch is not taken.

  If a conditional branch is taken, the delay instruction is always executed regardless of the value of the annul field. If a conditional branch is not taken and the a (annul) field is 1, the delay instruction is annulled (not executed). **Note**: The annul bit has a *different* effect for conditional branches than it does for unconditional branches.

  The predict bit (p) is used to give the hardware a hint about whether the branch is expected to be taken. A 1 in the p bit indicates that the branch is expected to be taken; a 0 indicates that the branch is expected not to be taken.

  Annulment, delay instructions, prediction, and delayed control transfers are described further in Chapter 6, *Instructions.*

*Exceptions* *illegal_instruction* (*cc1* □ *cc0* = $01_2$ or $11_2$)

# A.9 Call and Link

| Opcode | op | Operation |
|---|---|---|
| CALL | 01 | Call and Link |

*Format (1)*


| Assembly Language Syntax | |
|---|---|
| call | *label* |

*Description* The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address PC + (4 × sign_ext(disp30)). Since the word displacement (disp30) field is 30 bits wide, the target address lies within a range of $-2^{31}$ to $+2^{31} - 4$ bytes. The PC-relative displacement is formed by sign-extending the 30-bit word displacement field to 62 bits and appending two low-order zeroes to obtain a 64-bit byte displacement.

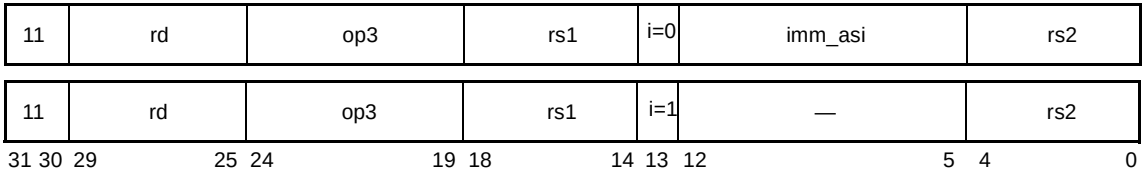
The CALL instruction also writes the value of PC, which contains the address of the CALL, into R[15] (out register 7).

*Exceptions* None

# A.10 Compare and Swap

| Opcode | op3 | Operation |
|---|---|---|
| CASA$^{P_{ASI}}$ | 11 1100 | Compare and Swap Word from Alternate Space |
| CASXA$^{P_{ASI}}$ | 11 1110 | Compare and Swap Extended from Alternate Space |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| `casa` | [$reg_{rs1}$] *imm_asi*, $reg_{rs2}$, $reg_{rd}$ |
| `casa` | [$reg_{rs1}$] `%asi`, $reg_{rs2}$, $reg_{rd}$ |
| `casxa` | [$reg_{rs1}$] *imm_asi*, $reg_{rs2}$, $reg_{rd}$ |
| `casxa` | [$reg_{rs1}$] `%asi`, $reg_{rs2}$, $reg_{rd}$ |

*Description* Concurrent processes use these instructions for synchronization and memory updates. Uses of compare-and-swap include spin-lock operations, updates of shared counters, and updates of linked-list pointers. The last two can use wait-free (nonlocking) protocols.

The `CASXA` instruction compares the value in register `R[rs2]` with the doubleword in memory pointed to by the doubleword address in `R[rs1]`. If the values are equal, the value in `R[rd]` is swapped with the doubleword pointed to by the doubleword address in `R[rs1]`. If the values are not equal, the contents of the doubleword pointed to by `R[rs1]` replaces the value in `R[rd]`, but the memory location remains unchanged.

The `CASA` instruction compares the low-order 32 bits of register `R[rs2]` with a word in memory pointed to by the word address in `R[rs1]`. If the values are equal, then the low-order 32 bits of register `R[rd]` are swapped with the contents of the memory word pointed to by the address in `R[rs1]` and the high-order 32 bits of

register R[rd] are set to 0. If the values are not equal, the memory location remains unchanged, but the zero-extended contents of the memory word pointed to by R[rs1] replace the low-order 32 bits of R[rd] and the high-order 32 bits of register R[rd] are set to 0.

A compare-and-swap instruction comprises three operations: a load, a compare, and a swap. The overall instruction is atomic; that is, no intervening interrupts or deferred traps are recognized by the virtual processor and no intervening update resulting from a compare-and-swap, swap, load, load-store unsigned byte, or store instruction to the doubleword containing the addressed location, or any portion of it, is performed by the memory system.

A compare-and-swap operation does *not* imply any memory barrier semantics. When compare-and-swap is used for synchronization, the same consideration should be given to memory barriers as if a load, store, or swap instruction were used.

A compare-and-swap operation behaves as if it performs a store, either of a new value from R[rd] or of the previous value in memory. The addressed location must be writable, even if the values in memory and R[rs2] are not equal.

If i = 0, the address space of the memory location is specified in the imm_asi field; if i = 1, the address space is specified in the ASI register.

A *mem_address_not_aligned* exception is generated if the address in R[rs1] is not properly aligned. CASXA and CASA cause a *privileged_action* exception if PSTATE.PRIV = 0 and bit 7 of the ASI is 0.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120).

---

**JPS Compatibility Note –** An implementation might cause an exception because of an error during the store memory access, even though there was no error during the load memory access.

---

**Programming Note –** Compare and Swap (CAS) and Compare and Swap Extended (CASX) synthetic instructions are available for “big endian” memory accesses. Compare and Swap Little (CASL) and Compare and Swap Extended Little (CASXL) synthetic instructions are available for “little endian” memory accesses. See *Synthetic Instructions* on page 514 for the syntax of these synthetic instructions.

---

The compare-and-swap instructions do not affect the condition codes.

Accesses to a noncacheable page generates a *data_access_exception*.

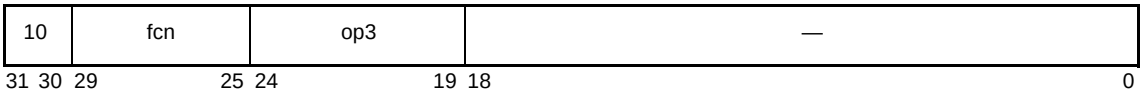
*Exceptions*

*privileged_action*
*mem_address_not_aligned*
*data_access_exception*
*data_access_error*
*VA_watchpoint*
*PA_watchpoint*
*fast_data_access_MMU_miss*
*fast_data_access_protection*

# A.11 DONE and RETRY

| Opcode | op3 | fcn | Operation |
|---|---|---|---|
| DONE[P] | 11 1110 | 0 | Return from Trap (skip trapped instruction) |
| RETRY[P] | 11 1110 | 1 | Return from Trap (retry trapped instruction) |
| — | 11 1110 | 2–31 | *Reserved* |

*Format (3)*




| Assembly Language Syntax |
|---|
| `done` |
| `retry` |

*Description*

The DONE and RETRY instructions restore the saved state from TSTATE (CWP, ASI, CCR, and PSTATE), set PC and nPC, and decrement TL.

The RETRY instruction resumes execution with the trapped instruction by setting PC←TPC[TL] (the saved value of PC on trap) and nPC←TNPC[TL] (the saved value of nPC on trap).

The DONE instruction skips the trapped instruction by setting PC←TNPC[TL] and nPC←TNPC[TL] +4.

Execution of a DONE or RETRY instruction in the delay slot of a control-transfer instruction produces undefined results.

**Programming Note –** Use the DONE and RETRY instructions to return from privileged trap handlers.

The result of an attempt to write a reserved combination of ag, ig, and mg bit values to the PSTATE register by copying them from TSTATE during a DONE or RETRY instruction is implementation dependent. (impl. dep. #308)

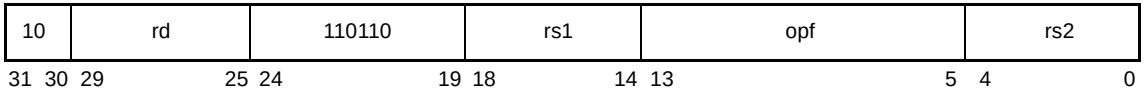
*Exceptions*

*privileged_opcode*
*illegal_instruction* (if TL = 0 or fcn = 2–31)

# A.12 Edge Handling Instructions (VIS I, II)

| Opcode | opf | Operation |
|---|---|---|
| EDGE8 | 0 0000 0000 | Eight 8-bit edge boundary processing |
| EDGE8N | 0 0000 0001 | Eight 8-bit edge boundary processing, no CC |
| EDGE8L | 0 0000 0010 | Eight 8-bit edge boundary processing little-endian |
| EDGE8LN | 0 0000 0011 | Eight 8-bit edge boundary processing, little-endian, no CC |
| EDGE16 | 0 0000 0100 | Four 16-bit edge boundary processing |
| EDGE16N | 0 0000 0101 | Four 16-bit edge boundary processing, no CC |
| EDGE16L | 0 0000 0110 | Four 16-bit edge boundary processing little-endian |
| EDGE16LN | 0 0000 0111 | Four 16-bit edge boundary processing, little-endian, no CC |
| EDGE32 | 0 0000 1000 | Two 32-bit edge boundary processing |
| EDGE32N | 0 0000 1001 | Two 32-bit edge boundary processing, no CC |
| EDGE32L | 0 0000 1010 | Two 32-bit edge boundary processing little-endian |
| EDGE32LN | 0 0000 1011 | Two 32-bit edge boundary processing, little-endian, no CC |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| edge8 | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge8n | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge8l | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge8ln | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge16 | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge16n | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge16l | $reg_{rs1}$, $re_{grs2}$, $reg_{rd}$ |
| edge16ln | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge32 | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge32n | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge32l | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |
| edge32ln | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ |

*Description* These instructions handle the boundary conditions for parallel pixel scan line loops, where src1 is the address of the next pixel to render and src2 is the address of the last pixel in the scan line.

EDGE8L(N), EDGE16L(N), and EDGE32L(N) are little-endian versions of EDGE8(N), EDGE16(N), and EDGE32(N). They produce an edge mask that is bit-reversed from their big-endian counterparts but are otherwise identical. This makes the mask consistent with the mask produced by the graphics compare operations (see *Pixel Compare (VIS I)* on page 323) and with the Partial Store instruction (see *Partial Store (VIS I)* on page 313) on little-endian data.

A 2-bit (EDGE32), 4-bit (EDGE16), or 8-bit (EDGE8) pixel mask is stored in the least significant bits of R[rd]. The mask is computed from left and right edge masks as follows:

1. The left edge mask is computed from the 3 least significant bits (LSBs) of R[rs1], and the right edge mask is computed from the 3 LSBs of R[s2], according to TABLE A-4 (TABLE A-5 for little-endian byte ordering).
2. If a 32-bit address masking is disabled (PSTATE.am = 0, 64-bit addressing) and the upper 61 bits of R[rs1] are equal to the corresponding bits in R[rs2], R[rd] is set to the right edge mask ANDed with the left edge mask.
3. If 32-bit address masking is enabled (PSTATE.am = 1, 32-bit addressing) and bits 31:3 of R[rs1] match bits 31:3 of R[rs2], R[rd] is set to the right edge mask ANDed with the left edge mask.
4. Otherwise, R[rd] is set to the left edge mask.

The integer condition codes are set per the rules of the SUBCC instruction with the same operands (see A.65, *Subtract*, on page 370).

The EDGE(8,16,32)(L)N instructions do not set the integer condition codes.

*Exceptions* None

**TABLE A-4** Edge Mask Specification

| Edge Size | A2–A0 | Left Edge | Right Edge |
|---|---|---|---|
| 8 | 000 | 1111 1111 | 1000 0000 |
| 8 | 001 | 0111 1111 | 1100 0000 |
| 8 | 010 | 0011 1111 | 1110 0000 |
| 8 | 011 | 0001 1111 | 1111 0000 |
| 8 | 100 | 0000 1111 | 1111 1000 |
| 8 | 101 | 0000 0111 | 1111 1100 |

**TABLE A-4** Edge Mask Specification *(Continued)*

| Edge Size | A2–A0 | Left Edge | Right Edge |
|---|---|---|---|
| 8 | 110 | 0000 0011 | 1111 1110 |
| 8 | 111 | 0000 0001 | 1111 1111 |
| 16 | 00x | 1111 | 1000 |
| 16 | 01x | 0111 | 1100 |
| 16 | 10x | 0011 | 1110 |
| 16 | 11x | 0001 | 1111 |
| 32 | 0xx | 11 | 10 |
| 32 | 1xx | 01 | 11 |

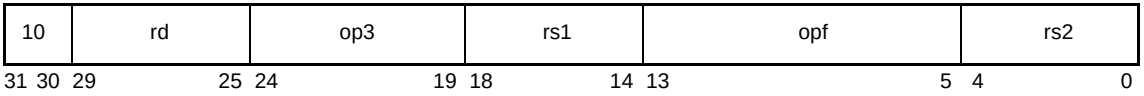
**TABLE A-5** Edge Mask Specification (Little-Endian)

| Edge Size | A2–A0 | Left Edge | Right Edge |
|---|---|---|---|
| 8 | 000 | 1111 1111 | 0000 0001 |
| 8 | 001 | 1111 1110 | 0000 0011 |
| 8 | 010 | 1111 1100 | 0000 0111 |
| 8 | 011 | 1111 1000 | 0000 1111 |
| 8 | 100 | 1111 0000 | 0001 1111 |
| 8 | 101 | 1110 0000 | 0011 1111 |
| 8 | 110 | 1100 0000 | 0111 1111 |
| 8 | 111 | 1000 0000 | 1111 1111 |
| 16 | 00x | 1111 | 0001 |
| 16 | 01x | 1110 | 0011 |
| 16 | 10x | 1100 | 0111 |
| 16 | 11x | 1000 | 1111 |
| 32 | 0xx | 11 | 01 |
| 32 | 1xx | 10 | 11 |

# A.13 Floating-Point Add and Subtract

| Opcode | op3 | opf | Operation |
|---|---|---|---|
| FADDs | 11 0100 | 0 0100 0001 | Add Single |
| FADDd | 11 0100 | 0 0100 0010 | Add Double |
| FADDq | 11 0100 | 0 0100 0011 | Add Quad |
| FSUBs | 11 0100 | 0 0100 0101 | Subtract Single |
| FSUBd | 11 0100 | 0 0100 0110 | Subtract Double |
| FSUBq | 11 0100 | 0 0100 0111 | Subtract Quad |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fadds | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| faddd | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| faddq | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fsubs | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fsubd | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fsubq | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

*Description*

The floating-point add instructions add the floating-point register(s) specified by the `rs1` field and the floating-point register(s) specified by the `rs2` field. The instructions then write the sum into the floating-point register(s) specified by the `rd` field.

The floating-point subtract instructions subtract the floating-point register(s) specified by the `rs2` field from the floating-point register(s) specified by the `rs1` field. The instructions then write the difference into the floating-point register(s) specified by the `rd` field.

Rounding is performed as specified by the FSR.rd field.

---

**Notes –** 1) SPARC JPS2 processors do not implement in hardware the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with ftt = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

2) For FADDs, FADDd, FSUBs, FSUBd, an *fp_exception_other* with ftt = *unfinished_FPop* can occur if the add/subtract operation detects certain unusual conditions.

---

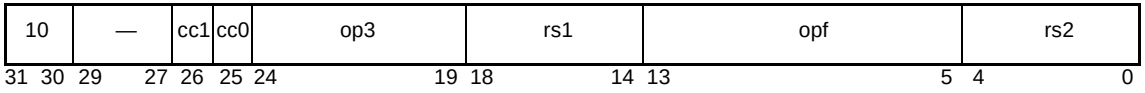
*Exceptions*

*fp_disabled*
*fp_exception_ieee_754* (OF, UF, NX, NV)
*fp_exception_other* (ftt = *unimplemented_FPop* (FADDq and FSUBq only))
*fp_exception_other* (ftt = *unifinished_FPop* (FADDs, FADDd, FSUBs, FSUBd only))

# A.14 Floating-Point Compare

| Opcode | op3 | opf | Operation |
|---|---|---|---|
| FCMPs | 11 0101 | 0 0101 0001 | Compare Single |
| FCMPd | 11 0101 | 0 0101 0010 | Compare Double |
| FCMPq | 11 0101 | 0 0101 0011 | Compare Quad |
| FCMPEs | 11 0101 | 0 0101 0101 | Compare Single and Exception if Unordered |
| FCMPEd | 11 0101 | 0 0101 0110 | Compare Double and Exception if Unordered |
| FCMPEq | 11 0101 | 0 0101 0111 | Compare Quad and Exception if Unordered |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fcmps | %fccn, $freg_{rs1}$, $freg_{rs2}$ |
| fcmpd | %fccn, $freg_{rs1}$, $freg_{rs2}$ |
| fcmpq | %fccn, $freg_{rs1}$, $freg_{rs2}$ |
| fcmpes | %fccn, $freg_{rs1}$, $freg_{rs2}$ |
| fcmped | %fccn, $freg_{rs1}$, $freg_{rs2}$ |
| fcmpeq | %fccn, $freg_{rs1}$, $freg_{rs2}$ |

| cc1 cc0 | Condition Code |
|---|---|
| 00 | fcc0 |
| 01 | fcc1 |
| 10 | fcc2 |
| 11 | fcc3 |

*Description*

These instructions compare the floating-point register(s) specified by the `rs1` field with the floating-point register(s) specified by the `rs2` field, and set the selected floating-point condition code (`fcc`*n*) as shown below.

| fcc value | Relation |
|---|---|
| 0 | $freg_{rs1} = freg_{rs2}$ |
| 1 | $freg_{rs1} < freg_{rs2}$ |
| 2 | $freg_{rs1} > freg_{rs2}$ |
| 3 | $freg_{rs1}$ ? $freg_{rs2}$ (unordered) |

The "?" in the preceding table means that the comparison is unordered. The unordered condition occurs when one or both of the operands to the compare is a signalling or quiet NaN.

The "compare and cause exception if unordered" (`FCMPEs`, `FCMPEd`, and `FCMPEq`) instructions cause an invalid (NV) exception if either operand is a NaN.

`FCMP` causes an invalid (NV) exception if either operand is a signalling NaN.

---

**V9 Compatibility Note –** Unlike SPARC V8, SPARC V9 does not require an instruction between a floating-point compare operation and a floating-point branch (`FBfcc`, `FBPfcc`).

SPARC V8 floating-point compare instructions are required to have a 0 in the `R[rd]` field. In SPARC V9, bits 26 and 25 of the `R[rd]` field specify the floating-point condition code to be set. Legal SPARC V8 code will work on SPARC V9 because the zeroes in the `R[rd]` field are interpreted as `fcc0` and the `FBfcc` instruction branches according to `fcc0`.

---

**Note –** SPARC JPS2 instructions do not implement in hardware the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = unimplemented_FPop), which causes a trap. Supervisor software then emulates these instructions.

---

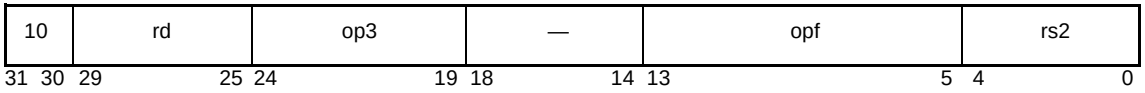
*Exceptions*

*fp_disabled*
*fp_exception_ieee_754* (NV)
*fp_exception_other* (`ftt` = unimplemented_FPop (`FCMPq`, `FCMPEq` only))

# A.15 Convert Floating-Point to Integer

| Opcode | op3 | opf | Operation |
|---|---|---|---|
| FsTOx | 11 0100 | 0 1000 0001 | Convert Single to 64-bit Integer |
| FdTOx | 11 0100 | 0 1000 0010 | Convert Double to 64-bit Integer |
| FqTOx | 11 0100 | 0 1000 0011 | Convert Quad to 64-bit Integer |
| FsTOi | 11 0100 | 0 1101 0001 | Convert Single to 32-bit Integer |
| FdTOi | 11 0100 | 0 1101 0010 | Convert Double to 32-bit Integer |
| FqTOi | 11 0100 | 0 1101 0011 | Convert Quad to 32-bit Integer |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fstox | $freg_{rs2}$, $freg_{rd}$ |
| fdtox | $freg_{rs2}$, $freg_{rd}$ |
| fqtox | $freg_{rs2}$, $freg_{rd}$ |
| fstoi | $freg_{rs2}$, $freg_{rd}$ |
| fdtoi | $freg_{rs2}$, $freg_{rd}$ |
| fqtoi | $freg_{rs2}$, $freg_{rd}$ |

*Description:* FsTOx, FdTOx, and FqTOx convert the floating-point operand in the floating-point register(s) specified by rs2 to a 64-bit integer in the floating-point register(s) specified by rd.

FsTOi, FdTOi, and FqTOi convert the floating-point operand in the floating-point register(s) specified by rs2 to a 32-bit integer in the floating-point register specified by rd.

The result is always rounded toward zero; that is, the rounding direction (rd) field of the FSR register is ignored.

If the floating-point operand's value is too large to be converted to an integer of the specified size or is a NaN or infinity, then an *fp_exception_ieee_754* "invalid" exception occurs. The value written into the floating-point register(s) specified by `rd` in these cases is as defined in *Integer Overflow Definition* on page 423.

---

**Note –** SPARC JPS2 instructions do not implement in hardware the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = unimplemented_FPop), which causes a trap. Supervisor software then emulates these instructions.

---

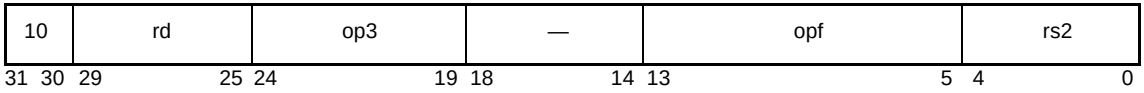
*Exceptions*

*fp_disabled*
*fp_exception_ieee_754* (NV, NX)
*fp_exception_other* (`ftt` = unimplemented_FPop (`FqTOi`, `FqTOx` only))

# A.16 Convert Between Floating-Point Formats

| Opcode | op3 | opf | Operation |
|---|---|---|---|
| FsTOd | 11 0100 | 0 1100 1001 | Convert Single to Double |
| FsTOq | 11 0100 | 0 1100 1101 | Convert Single to Quad |
| FdTOs | 11 0100 | 0 1100 0110 | Convert Double to Single |
| FdTOq | 11 0100 | 0 1100 1110 | Convert Double to Quad |
| FqTOs | 11 0100 | 0 1100 0111 | Convert Quad to Single |
| FqTOd | 11 0100 | 0 1100 1011 | Convert Quad to Double |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fstod | *freg*$_{rs2}$, *freg*$_{rd}$ |
| fstoq | *freg*$_{rs2}$, *freg*$_{rd}$ |
| fdtos | *freg*$_{rs2}$, *freg*$_{rd}$ |
| fdtoq | *freg*$_{rs2}$, *freg*$_{rd}$ |
| fqtos | *freg*$_{rs2}$, *freg*$_{rd}$ |
| fqtod | *freg*$_{rs2}$, *freg*$_{rd}$ |

*Description:* These instructions convert the floating-point operand in the floating-point register(s) specified by rs2 to a floating-point number in the destination format. They write the result into the floating-point register(s) specified by rd.

Rounding is performed as specified by the FSR.rd field.

FqTOd, FqTOs, and FdTOs (the “narrowing” conversion instructions) can raise OF, UF, and NX exceptions. FdTOq, FsTOq, and FsTOd (the “widening” conversion instructions) cannot.

Any of these six instructions can trigger an NV exception if the source operand is a signalling NaN.

*Untrapped Result in Different Format from Operands* on page 420 defines the rules for converting NaNs from one floating-point format to another.

---

**Notes –** 1) SPARC JPS2 instructions do not implement in hardware the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = unimplemented_FPop), which causes a trap. Supervisor software then emulates these instructions.

2) For `FdTOs` and `FsTOd`, an *fp_exception_other* with `ftt` = unfinished_FPop can occur if the convert operation detects certain unusual conditions.

---

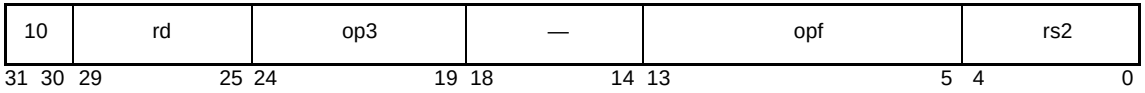
*Exceptions*

*fp_disabled*
*fp_exception_ieee_754* (OF, UF, NV, NX)
*fp_exception_other* (`ftt` = unimplemented_FPop (`FsTOq`, `FdTOq`, `FqTOs`, `FqTOd` only))
*fp_exception_other* (`ftt` = unfinished_FPop (`FdTOs` and `FsTOd` only))

# A.17 Convert Integer to Floating-Point

| Opcode | op3 | opf | Operation |
|---|---|---|---|
| FxTOs | 11 0100 | 0 1000 0100 | Convert 64-bit Integer to Single |
| FxTOd | 11 0100 | 0 1000 1000 | Convert 64-bit Integer to Double |
| FxTOq | 11 0100 | 0 1000 1100 | Convert 64-bit Integer to Quad |
| FiTOs | 11 0100 | 0 1100 0100 | Convert 32-bit Integer to Single |
| FiTOd | 11 0100 | 0 1100 1000 | Convert 32-bit Integer to Double |
| FiTOq | 11 0100 | 0 1100 1100 | Convert 32-bit Integer to Quad |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fxtos | $freg_{rs2}$, $freg_{rd}$ |
| fxtod | $freg_{rs2}$, $freg_{rd}$ |
| fxtoq | $freg_{rs2}$, $freg_{rd}$ |
| fitos | $freg_{rs2}$, $freg_{rd}$ |
| fitod | $freg_{rs2}$, $freg_{rd}$ |
| fitoq | $freg_{rs2}$, $freg_{rd}$ |

*Description*

FxTOs, FxTOd, and FxTOq convert the 64-bit signed integer operand in the floating-point registers specified by rs2 into a floating-point number in the destination format.

FiTOs, FiTOd, and FiTOq convert the 32-bit signed integer operand in floating-point register(s) specified by rs2 into a floating-point number in the destination format. All write their result into the floating-point register(s) specified by rd.

FiTOs, FxTOs, and FxTOd round as specified by the FSR.rd field.

**Note –** SPARC JPS2 instructions do not implement in hardware the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = unimplemented_FPop), which causes a trap. Supervisor software then emulates these instructions.

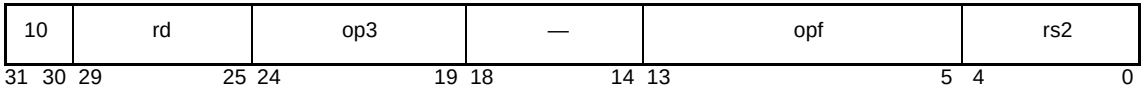
*Exceptions*

*fp_disabled*
*fp_exception_ieee_754* (NX (`FiTOs`, `FxTOs`, `FxTOd` only))
*fp_exception_other* (`ftt` = unimplemented_FPop (`FiTOq`, `FxTOq` only))

# A.18 Floating-Point Move

| Opcode | op3 | opf | Operation |
|---|---|---|---|
| FMOVs | 11 0100 | 0 0000 0001 | Move Single |
| FMOVd | 11 0100 | 0 0000 0010 | Move Double |
| FMOVq | 11 0100 | 0 0000 0011 | Move Quad |
| FNEGs | 11 0100 | 0 0000 0101 | Negate Single |
| FNEGd | 11 0100 | 0 0000 0110 | Negate Double |
| FNEGq | 11 0100 | 0 0000 0111 | Negate Quad |
| FABSs | 11 0100 | 0 0000 1001 | Absolute Value Single |
| FABSd | 11 0100 | 0 0000 1010 | Absolute Value Double |
| FABSq | 11 0100 | 0 0000 1011 | Absolute Value Quad |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fmovs | $freg_{rs2}$, $freg_{rd}$ |
| fmovd | $freg_{rs2}$, $freg_{rd}$ |
| fmovq | $freg_{rs2}$, $freg_{rd}$ |
| fnegs | $freg_{rs2}$, $freg_{rd}$ |
| fnegd | $freg_{rs2}$, $freg_{rd}$ |
| fnegq | $freg_{rs2}$, $freg_{rd}$ |
| fabss | $freg_{rs2}$, $freg_{rd}$ |
| fabsd | $freg_{rs2}$, $freg_{rd}$ |
| fabsq | $freg_{rs2}$, $freg_{rd}$ |

*Description*

The single-precision versions of these instructions copy the contents of a single-precision floating-point register to the destination. The double-precision versions copy the contents of a double-precision floating-point register to the destination. The quad-precision versions copy a quad-precision value in floating-point registers to the destination.

`FMOV` copies the source to the destination unaltered.

`FNEG` copies the source to the destination with the sign bit complemented.

`FABS` copies the source to the destination with the sign bit cleared.

These instructions do not round.

---

**Note –** SPARC JPS2 instructions do not implement in hardware the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = unimplemented_FPop), which causes a trap. Supervisor software then emulates these instructions.

---

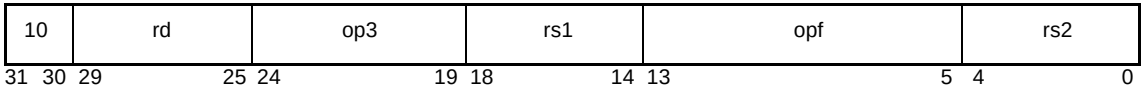
*Exceptions*

*fp_disabled*
*fp_exception_other* (`ftt` = unimplemented_FPop (`FMOVq`, `FNEGq`, `FABSq` only))

# A.19 Floating-Point Multiply and Divide

| Opcode | op3 | opf | Operation |
|---|---|---|---|
| FMULs | 11 0100 | 0 0100 1001 | Multiply Single |
| FMULd | 11 0100 | 0 0100 1010 | Multiply Double |
| FMULq | 11 0100 | 0 0100 1011 | Multiply Quad |
| FsMULd | 11 0100 | 0 0110 1001 | Multiply Single to Double |
| FdMULq | 11 0100 | 0 0110 1110 | Multiply Double to Quad |
| FDIVs | 11 0100 | 0 0100 1101 | Divide Single |
| FDIVd | 11 0100 | 0 0100 1110 | Divide Double |
| FDIVq | 11 0100 | 0 0100 1111 | Divide Quad |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fmuls | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fmuld | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fmulq | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fsmuld | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fdmulq | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fdivs | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fdivd | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fdivq | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

*Description* The floating-point multiply instructions multiply the contents of the floating-point register(s) specified by the `rs1` field by the contents of the floating-point register(s) specified by the `rs2` field. The instructions then write the product into the floating-point register(s) specified by the `rd` field.

The FsMULd instruction provides the exact double-precision product of two single-precision operands, without underflow, overflow, or rounding error. Similarly, FdMULq provides the exact quad-precision product of two double-precision operands.

The floating-point divide instructions divide the contents of the floating-point register(s) specified by the rs1 field by the contents of the floating-point register(s) specified by the rs2 field. The instructions then write the quotient into the floating-point register(s) specified by the rd field.

Rounding is performed as specified by the FSR.rd field.

---

**Notes –** 1) SPARC JPS2 instructions do not implement in hardware the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with ftt = unimplemented_FPop), which causes a trap. Supervisor software then emulates these instructions.

2) For FDIVs and FDIVd, an *fp_exception_other* with ftt = unfinished_FPop can occur if the divide unit detects certain unusual conditions.

---

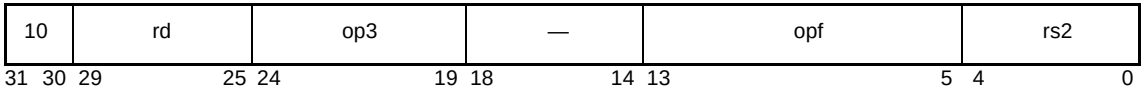
*Exceptions*

*fp_disabled*
*fp_exception_ieee_754* (of, uf, dz (FDIV only), nv, nx)
*fp_exception_other* (ftt = unimplemented_FPop (FMULq, FdMULq, FDIVq)
*fp_exception_other* (ftt = unifinished_FPop (FMULs, FMULd, FSMULd, FDIVs, FDIV))

# A.20 Floating-Point Square Root

| Opcode | op3 | opf | Operation |
|---|---|---|---|
| FSQRTs | 11 0100 | 0 0010 1001 | Square Root Single |
| FSQRTd | 11 0100 | 0 0010 1010 | Square Root Double |
| FSQRTq | 11 0100 | 0 0010 1011 | Square Root Quad |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fsqrts | $freg_{rs2}$, $freg_{rd}$ |
| fsqrtd | $freg_{rs2}$, $freg_{rd}$ |
| fsqrtq | $freg_{rs2}$, $freg_{rd}$ |

*Description*

These SPARC V9 instructions generate the square root of the floating-point operand in the floating-point register(s) specified by the rs2 field and place the result in the destination floating-point register(s) specified by the rd field. Rounding is performed as specified by the FSR.rd field.

**Note –** SPARC JPS2 instructions do not implement in hardware the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with ftt = unimplemented_FPop), which causes a trap. Supervisor software then emulates these instructions.

For FSQRTs and FSQRTd an *fp_exception_other* (with ftt = unfinished_FPop) can occur if the operand to the square root is positive subnormalized. See *FSR_floating-point_trap_type (ftt)* on page 56 for additional details.

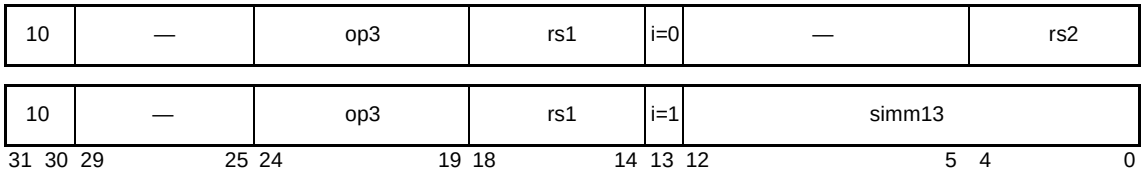
*Exceptions*

*fp_disabled*
*fp_exception_ieee_754* (*IEEE_754_exception* (NV, NX))
*fp_exception_other* (unimplemented_FPop) (Quad forms)
*fp_exception_other* (unfinished_FPop) (FSQRTs, FSQRTd)

# A.21 Flush Instruction Memory

| Opcode | op3 | Operation |
|---|---|---|
| FLUSH | 11 1011 | Flush Instruction Memory |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| flush | *address* |

*Description*

FLUSH ensures that the doubleword specified as the effective address is consistent across any local caches and, in a multiprocessor system, will eventually become consistent everywhere.

In the following discussion $P_{FLUSH}$ refers to the virtual processor that executed the FLUSH instruction.

FLUSH ensures that instruction fetches from the specified effective address by $P_{FLUSH}$ appear to execute after any loads, stores, and atomic load-stores to that address issued by $P_{FLUSH}$ prior to the FLUSH. In a multiprocessor system, FLUSH also ensures that these values will eventually become visible to the instruction fetches of all other virtual processors. FLUSH behaves as if it were a store with respect to MEMBAR-induced orderings. See A.35, *Memory Barrier*.

The effective address operand for the FLUSH instruction is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + sign_ext(simm13)" if i = 1. The least significant two address bits of the effective address are unused and should be supplied as zeroes by software. Bit 2 of the address is ignored because FLUSH operates on at least a doubleword.

**Programming Notes –**

1. Typically, FLUSH is used in self-modifying code. See H.1.6, *Self-Modifying Code*, for information about use of the FLUSH instruction in portable self-modifying code. The use of self-modifying code is discouraged.
2. The order in which memory is modified can be controlled by means of FLUSH and MEMBAR instructions interspersed appropriately between stores and atomic load-stores. FLUSH is needed only between a store and a subsequent instruction fetch from the modified location. When multiple processes may concurrently modify live (that is, potentially executing) code, the programmer must ensure that the order of update maintains the program in a semantically correct form at all times.
3. The memory model guarantees in a uniprocessor that *data* loads observe the results of the most recent store, even if there is no intervening FLUSH.
4. FLUSH may be time consuming.
5. In a multiprocessor system, the time it takes for a FLUSH to take effect is implementation dependent. No mechanism is provided to ensure or test completion.
6. Because FLUSH is designed to act on a doubleword and because, on some implementations, FLUSH may trap to system software, system software should provide a user-callable service routine for flushing arbitrarily sized regions of memory. On some implementations, this routine would issue a series of FLUSH instructions; on others, it might issue a single trap to system software that would then flush the entire region.

**IMPL. DEP. #42:** If FLUSH is not implemented in hardware, it causes an *illegal_instruction* exception and the function of FLUSH is performed by system software. Whether FLUSH traps is implementation dependent.

**V9 Compatibility Note –** The effect of a FLUSH instruction as observed from the virtual processor on which FLUSH executes is immediate. Other virtual processors in a multiprocessor system eventually will see the effect of the FLUSH, but the latency is implementation dependent.

On a SPARC JPS2 virtual processor:

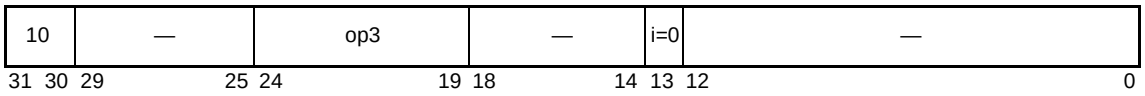
- A FLUSH instruction flushes the virtual processor pipeline and synchronizes the virtual processor.
- Coherency between instruction and data memories is maintained by hardware; therefore, a JPS2 implementation may ignore the address in a FLUSH instruction's operands. However, for portability across all SPARC V9 implementations, software must supply the target effective address in FLUSH instructions.

*Exceptions* None

# A.22 Flush Register Windows

| Opcode | op3 | Operation |
|---|---|---|
| FLUSHW | 10 1011 | Flush Register Windows |

*Format (3)*




| Assembly Language Syntax |
|---|
| flushw |

*Description*

FLUSHW causes all active register windows except the current window to be flushed to memory at locations determined by privileged software. FLUSHW behaves as a NOP if there are no active windows other than the current window. At the completion of the FLUSHW instruction, the only active register window is the current one.

**Programming Note –** The FLUSHW instruction can be used by application software to switch memory stacks or to examine register contents for previous stack frames.

FLUSHW acts as a NOP if CANSAVE = NWINDOWS – 2. Otherwise, there is more than one active window, so FLUSHW causes a spill exception. The trap vector for the spill exception is based on the contents of OTHERWIN and WSTATE. The spill trap handler is invoked with the CWP set to the window to be spilled (that is, (CWP + CANSAVE + 2) **mod** NWINDOWS). See *Register Window Management Instructions* on page 118.

**Programming Note –** Typically, the spill handler saves a window on a memory stack and returns to reexecute the FLUSHW instruction. Thus, FLUSHW traps and reexecutes until all active windows other than the current window have been spilled.

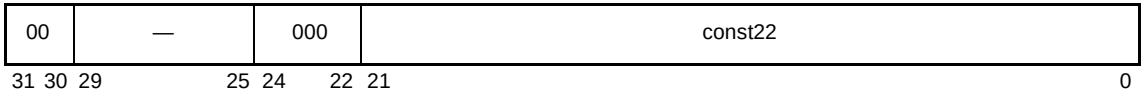
*Exceptions*

*spill_n_normal*
*spill_n_other*

# A.23 Illegal Instruction Trap

| Opcode | op | op2 | Operation |
|---|---|---|---|
| ILLTRAP | 00 | 000 | *illegal_instruction* trap |

*Format (2)*




| Assembly Language Syntax | |
|---|---|
| illtrap | *const22* |

*Description*

The ILLTRAP instruction causes an *illegal_instruction* exception. The const22 value is ignored by the hardware; specifically, this field is *not* reserved by the architecture for any future use.

**V9 Compatibility Note –** Except for its name, this instruction is identical to the SPARC V8 UNIMP instruction.

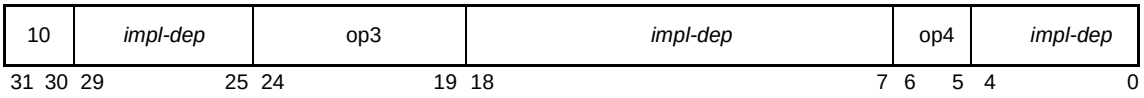
*Exceptions*

*illegal_instruction*

# A.24 Implementation-Dependent Instructions

| Opcode | op3 | op4 | Operation |
|---|---|---|---|
| IMPDEP1 | 11 0110 | (any) | Implementation-Dependent Instruction 1 |
| IMPDEP2A | 11 0111 | 0 | Implementation-Dependent Instruction 2A |
| IMPDEP2B | 11 0111 | 1, 2, 3 | Implementation-Dependent Instruction 2B (FMADD, FMSUB, FNMADD, FNMSUB) |

*Format (3)*




*Description*

**IMPL. DEP. #106:** The IMPDEP2A opcode space is completely implementation dependent. Implementation-dependent aspects of IMPDEP2A instructions include their operation, the interpretation of bits 29–25, 18–7, and 4–0 in their encodings, and which (if any) exceptions they may cause.

IMPDEP2B instructions are implementation dependent but may only be used to implement FMADD, FMSUB, FNMADD, and FNMSUB instructions (as described in the SPARC JPS2 Extensions document for the SPARC64 VI processor). These instructions are expected to become part of Commonality in a future JPS.

See I.2, *Implementation-Dependent and Reserved Opcodes*, for information about extending the SPARC V9 instruction set by means of the implementation-dependent instructions.

**V9 Compatibility Note –** IMPDEP2A and IMPDEP2B are subsets of the SPARC V9 IMPDEP2 opcode space. The IMPDEP1 opcode space from SPARC V9 is occupied by various VIS instructions in JPS2, so is no longer available for implementation-dependent uses.

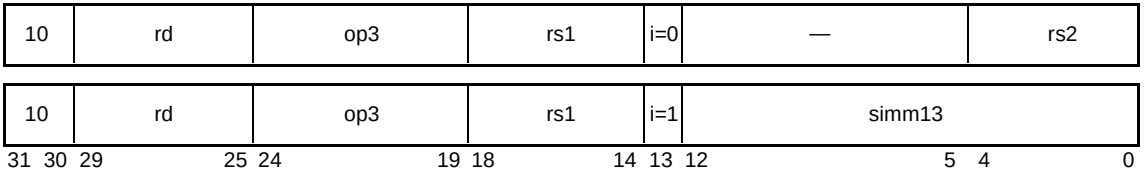
*Exceptions*

implementation-dependent (IMPDEP2A, IMPDEP2B)

# A.25 Jump and Link

| Opcode | op3 | Operation |
|---|---|---|
| JMPL | 11 1000 | Jump and Link |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| jmpl | *address*, $reg_{rd}$ |

*Description*

The JMPL instruction causes a register-indirect delayed control transfer to the address given by "R[rs1] + R[rs2]" if i field = 0, or "R[rs1] + sign_ext(simm13)" if i = 1.

The JMPL instruction copies the PC, which contains the address of the JMPL instruction, into register R[rd].

If either of the low-order two bits of the jump address is nonzero, a *mem_address_not_aligned* exception occurs.

**Programming Note –** A JMPL instruction with rd = 15 functions as a register-indirect call using the standard link register.

JMPL with rd = 0 can be used to return from a subroutine. The typical return address is "R[31] + 8," if a nonleaf routine (one that uses the SAVE instruction) is entered by a CALL instruction, or "R[15] + 8" if a leaf routine (one that does not use the SAVE instruction) is entered by a CALL instruction or by a JMPL instruction with rd = 15.

*Exceptions*

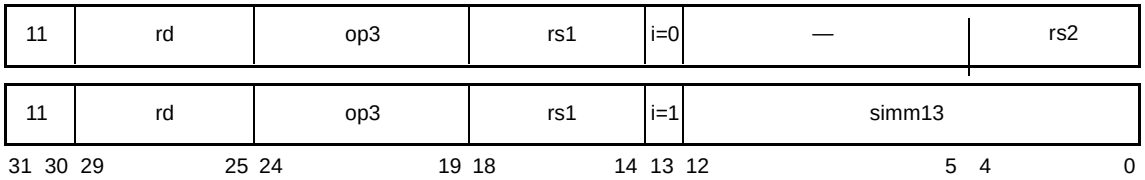
*mem_address_not_aligned*

# A.26 Load Floating-Point

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| LDF | 10 0000 | 0–31 | Load Floating-Point Register |
| LDDF | 10 0011 | † | Load Double Floating-Point Register |
| LDQF | 10 0010 | † | Load Quad Floating-Point Register |
| LDXFSR | 10 0001 | 1 | Load Floating-Point State Register |
| — | 10 0001 | 2–31 | *Reserved* |

† Encoded floating-point register value, as described on page 51.

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| ld | [*address*], $freg_{rd}$ |
| ldd | [*address*], $freg_{rd}$ |
| ldq | [*address*], $freg_{rd}$ |
| ldx | [*address*], %fsr |

*Description:* The load single floating-point instruction (LDF) copies a word from memory into f[rd].

The load doubleword floating-point instruction (LDDF) copies a word-aligned doubleword from memory into a double-precision floating-point register.

The load quad floating-point instruction (LDQF) copies a word-aligned quad-word from memory into a quad-precision floating-point register.

The load floating-point state register instruction (LDXFSR) waits for all FPop instructions that have not finished execution to complete and then loads a doubleword from memory into the FSR.

Load floating-point instructions access the primary address space (ASI = $80_{16}$) if TL=0. The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + sign_ext(simm13)" if i = 1.

LDF causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned. LDXFSR causes a *mem_address_not_aligned* exception if the address is not doubleword aligned. If the floating-point unit is not enabled (per FPRS.fef and PSTATE.pef) or if no FPU is present, then a load floating-point instruction causes an *fp_disabled* exception.

LDDF requires only word alignment. However, if the effective address is word aligned but not doubleword aligned, LDDF on a JPS2 virtual processor will cause an *LDDF_mem_address_not_aligned* exception and trap handler software must emulate the LDDF instruction and return (impl. dep. #109a).

In a JPS2 implementation, LDQF always causes an *fp_exception_other* trap (with FSR.ftt = unimplemented_FPop) and trap handler software emulates the LDQF. A JPS2 never generates an *LDQF_mem_address_not_aligned* exception (impl. dep. #111a).

---

**Programming Note –** In SPARC V8, some compilers issued sequences of single-precision loads when they could not determine that doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned loads is expected to be fast, we recommend that compilers issue sets of single-precision loads only when they can determine that doubleword or quadword operands are *not* properly aligned.

---

**IMPL. DEP. #44a:** If a load floating-point instruction traps with any type of access error, the contents of the destination floating-point register(s) remain unchanged or are undefined.

*Exceptions*

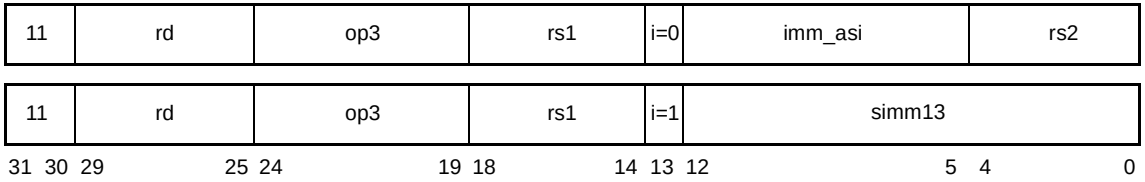
*illegal_instruction* (op3 = $21_{16}$ and rd = 2–31)
*fp_disabled*
*LDDF_mem_address_not_aligned* (LDDF only)
*LDQF_mem_address_not_aligned* (LDQF only) (not used in JPS2)
*mem_address_not_aligned*
*data_access_exception*
*PA_watchpoint*
*VA_watchpoint*
*data_access_error*
*fast_data_access_MMU_miss*

# A.27 Load Floating-Point from Alternate Space

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| LDFA^PASI | 11 0000 | 0–31 | Load Floating-Point Register from Alternate Space |
| LDDFA^PASI | 11 0011 | † | Load Double Floating-Point Register from Alternate Space |
| LDQFA^PASI | 11 0010 | † | Load Quad Floating-Point Register from Alternate Space |

† Encoded floating-point register value, as described in *Floating-Point Register Number Encoding* on page 51.

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| `lda` | [*regaddr*] *imm_asi*, *freg*$_{rd}$ |
| `lda` | [*reg_plus_imm*] `%asi`, *freg*$_{rd}$ |
| `ldda` | [*regaddr*] *imm_asi*, *freg*$_{rd}$ |
| `ldda` | [*reg_plus_imm*] `%asi`, *freg*$_{rd}$ |
| `ldqa` | [*regaddr*] *imm_asi*, *freg*$_{rd}$ |
| `ldqa` | [*reg_plus_imm*] `%asi`, *freg*$_{rd}$ |

*Description:* The load single floating-point from alternate space instruction (`LDFA`) copies a word from memory into `F[rd]`.

The load double floating-point from alternate space instruction (`LDDFA`) copies a word-aligned doubleword from memory into a double-precision floating-point register.

The load quad floating-point from alternate space instruction (`LDQFA`) copies a word-aligned quadword from memory into a quad-precision floating-point register.

## Load Floating-Point from Alternate Space

Load floating-point from alternate space instructions contain the address space identifier (ASI) to be used for the load in the `imm_asi` field if `i` = 0, or in the ASI register if `i` = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "`R[rs1] + R[rs2]`" if `i` = 0, or "`R[rs1] + sign_ext(simm13)`" if `i` = 1.

`LDFA` causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned. If the floating-point unit is not enabled (per `FPRS.fef` and `PSTATE.pef`) or if no FPU is present, then load floating-point from alternate space instructions cause an *fp_disabled* exception.

`LDDFA` with certain target ASIs is defined to be a 64-byte block-load instruction. See *Block Load and Store (VIS I)* on page 231 for details.

`LDDFA` with certain target ASIs is defined to be a Short Floating-point Load instruction. See *Short Floating-Point Load and Store (VIS I)* on page 356 for details.

---

**V9 Compatibility Note –** `LDFA`, `LDDFA`, and `LDQFA` cause a *privileged_action* exception if `PSTATE.priv` = 0 and bit 7 of the ASI is 0.

---

`LDDFA` requires only word alignment. However, if the effective address is word aligned but not doubleword aligned, `LDDFA` on a JPS2 virtual processor will cause an *LDDF_mem_address_not_aligned* exception and trap handler software must emulate the `LDDF` instruction and return (impl. dep. #109b).

In a JPS2 implementation, `LDQFA` always causes an *fp_exception_other* trap (with `FSR.ftt` = unimplemented_FPop) and trap handler software emulates the `LDQFA`. A JPS2 implementation never generates an *LDQF_mem_address_not_aligned* exception (impl. dep. #111b).

---

**Programming Note –** In SPARC V8, some compilers issued sequences of single-precision loads when they could not determine that doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned loads is expected to be fast, compilers should issue sets of single-precision loads only when they can determine that doubleword or quadword operands are *not* properly aligned.

---

**IMPL. DEP. #44b:** If a load floating-point instruction traps with any type of access error, the destination floating-point register(s) remain unchanged or are undefined.

*Exceptions*

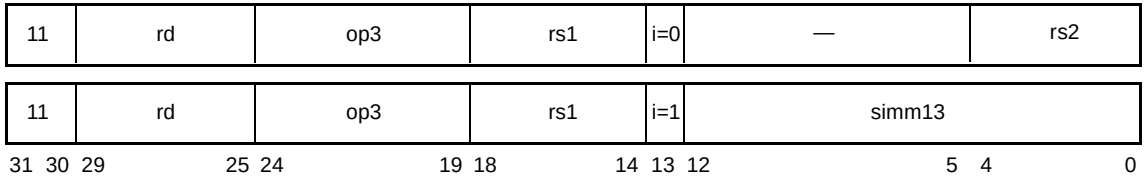
*illegal_instruction* (`LDQFA` only)
*fp_disabled*
*LDDF_mem_address_not_aligned* (`LDDFA` only)
*LDQF_mem_address_not_aligned* (`LDQFA` only) (not used in JPS2)
*mem_address_not_aligned*
*privileged_action*

*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*VA_watchpoint*
*PA_watchpoint*

# A.28 Load Integer

| Opcode | op3 | Operation |
|---|---|---|
| LDSB | 00 1001 | Load Signed Byte |
| LDSH | 00 1010 | Load Signed Halfword |
| LDSW | 00 1000 | Load Signed Word |
| LDUB | 00 0001 | Load Unsigned Byte |
| LDUH | 00 0010 | Load Unsigned Halfword |
| LDUW | 00 0000 | Load Unsigned Word |
| LDX | 00 1011 | Load Extended Word |

*Format (3)*




| Assembly Language Syntax | | |
|---|---|---|
| ldsb | [*address*], $reg_{rd}$ | |
| ldsh | [*address*], $reg_{rd}$ | |
| ldsw | [*address*], $reg_{rd}$ | |
| ldub | [*address*], $reg_{rd}$ | |
| lduh | [*address*], $reg_{rd}$ | |
| lduw | [*address*], $reg_{rd}$ | *(synonym:* ld*)* |
| ldx | [*address*], $reg_{rd}$ | |

*Description:* The load integer instructions copy a byte, a halfword, a word, or an extended word from memory. All copy the fetched value into R[rd]. A fetched byte, halfword, or word is right-justified in the destination register R[rd]; it is either sign-extended or zero-filled on the left, depending on whether the opcode specifies a signed or unsigned operation, respectively.

Load integer instructions access the primary address space (ASI = $80_{16}$) if TL = 0. The effective address is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + sign_ext(simm13)" if i = 1.

A successful load (notably, load extended) instruction operates atomically.

LDUH and LDSH cause a *mem_address_not_aligned* exception if the address is not halfword aligned. LDUW and LDSW cause a *mem_address_not_aligned* exception if the effective address is not word aligned. LDX causes a *mem_address_not_aligned* exception if the address is not doubleword aligned.

---

**V9 Compatibility Note –** The SPARC V8 LD instruction has been renamed LDUW in SPARC V9. The LDSW instruction is new in SPARC V9.

---

Note that the load integer doubleword instruction is deprecated; see A.71.5, *Load Integer Doubleword*, on page 394.

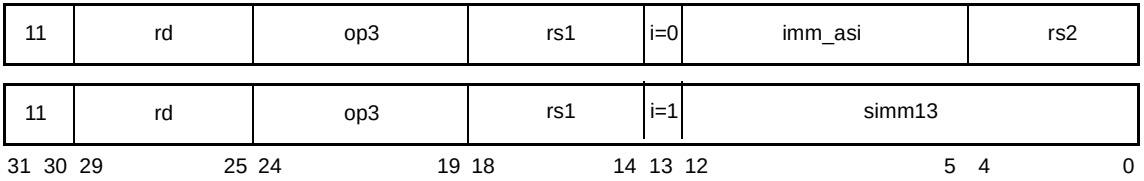
*Exceptions*

*mem_address_not_aligned* (all except LDSB, LDUB)
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*VA_watchpoint*
*PA_watchpoint*

# A.29 Load Integer from Alternate Space

| Opcode | op3 | Operation |
|---|---|---|
| LDSBA$^{P_{ASI}}$ | 01 1001 | Load Signed Byte from Alternate Space |
| LDSHA$^{P_{ASI}}$ | 01 1010 | Load Signed Halfword from Alternate Space |
| LDSWA$^{P_{ASI}}$ | 01 1000 | Load Signed Word from Alternate Space |
| LDUBA$^{P_{ASI}}$ | 01 0001 | Load Unsigned Byte from Alternate Space |
| LDUHA$^{P_{ASI}}$ | 01 0010 | Load Unsigned Halfword from Alternate Space |
| LDUWA$^{P_{ASI}}$ | 01 0000 | Load Unsigned Word from Alternate Space |
| LDXA$^{P_{ASI}}$ | 01 1011 | Load Extended Word from Alternate Space |

*Format (3)*




| Assembly Language Syntax | | |
|---|---|---|
| `ldsba` | [*regaddr*] *imm_asi*, *reg* $_{rd}$ | |
| `ldsha` | [*regaddr*] *imm_asi*, *reg* $_{rd}$ | |
| `ldswa` | [*regaddr*] *imm_asi*, *reg* $_{rd}$ | |
| `lduba` | [*regaddr*] *imm_asi*, *reg* $_{rd}$ | |
| `lduha` | [*regaddr*] *imm_asi*, *reg* $_{rd}$ | |
| `lduwa` | [*regaddr*] *imm_asi*, *reg* $_{rd}$ | (*synonym:* `lda`) |
| `ldxa` | [*regaddr*] *imm_asi*, *reg* $_{rd}$ | |
| `ldsba` | [*reg_plus_imm*] `%asi`, *reg* $_{rd}$ | |
| `ldsha` | [*reg_plus_imm*] `%asi`, *reg* $_{rd}$ | |
| `ldswa` | [*reg_plus_imm*] `%asi`, *reg* $_{rd}$ | |
| `lduba` | [*reg_plus_imm*] `%asi`, *reg* $_{rd}$ | |
| `lduha` | [*reg_plus_imm*] `%asi`, *reg* $_{rd}$ | |
| `lduwa` | [*reg_plus_imm*] `%asi`, *reg* $_{rd}$ | (*synonym:* `lda`) |
| `ldxa` | [*reg_plus_imm*] `%asi`, *reg* $_{rd}$ | |

*Description*

The load integer from alternate space instructions copy a byte, a halfword, a word, or an extended word from memory. All copy the fetched value into `R[rd]`. A fetched byte, halfword, or word is right-justified in the destination register `R[rd]`; it is either sign-extended or zero-filled on the left, depending on whether the opcode specifies a signed or unsigned operation, respectively.

The load integer from alternate space instructions contain the address space identifier (ASI) to be used for the load in the `imm_asi` field if `i` = 0, or in the ASI register if `i` = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "`R[rs1] + R[rs2]`" if `i` = 0, or "`R[rs1] + sign_ext(simm13)`" if `i` = 1.

A successful load (notably, load extended) instruction operates atomically.

`LDUHA` and `LDSHA` cause a *mem_address_not_aligned* exception if the address is not halfword aligned. `LDUWA` and `LDSWA` cause a *mem_address_not_aligned* exception if the effective address is not word aligned; `LDXA` causes a *mem_address_not_aligned* exception if the address is not doubleword aligned.

These instructions cause a *privileged_action* exception if `PSTATE.priv` = 0 and bit 7 of the ASI is 0.

Note that the load integer doubleword from alternate space instruction is deprecated; see A.71.6, *Load Integer Doubleword from Alternate Space*, on page 396.

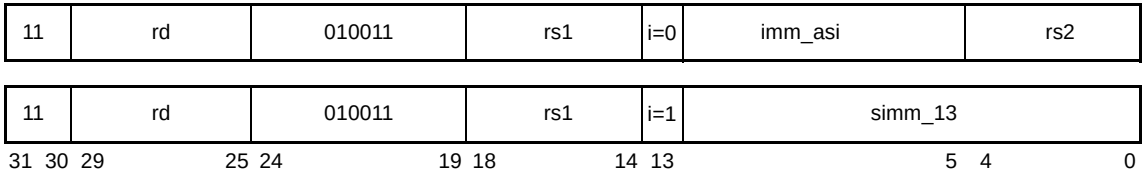
*Exceptions*

*privileged_action*
*mem_address_not_aligned* (all except `LDSBA` and `LDUBA`)
*data_access_exception*
*PA_watchpoint*
*VA_watchpoint*
*fast_data_access_MMU_miss*
*data_access_error*

# A.30 Load Quadword, Atomic (VIS I)

| Opcode | imm_asi | ASI Value | Operation |
|---|---|---|---|
| LDDA | ASI_QUAD_LDD | $24_{16}$ | 128-bit atomic load |
| LDDA | ASI_QUAD_LDD_L | $2C_{16}$ | 128-bit atomic load, little-endian |
| LDDA | ASI_QUAD_LDD_PHYS | $34_{16}$ | 128-bit atomic load, physical address |
| LDDA | ASI_QUAD_LDD_PHYS_L | $3C_{16}$ | 128-bit atomic load, physical address, little-endian |

*Format (3) LDDA*




| Assembly Language Syntax | |
|---|---|
| ldda | [*reg_addr*] *imm_asi*, $reg_{rd}$ |
| ldda | [*reg_plus_imm*] %asi, $reg_{rd}$ |

*Description*

The Load Quadword Atomic (LDQ_Atomic) instruction atomically reads a 128-bit data item. LDQ_Atomic is encoded using ASI $24_{16}$, $2C_{16}$, $34_{16}$, or $3C_{16}$ with the LDDA instruction. It is intended to be used by a TLB miss handler to access TSB entries without requiring software locks. The data are placed in an even/odd pair of 64-bit registers. The lowest-address 64 bits are placed in the even register; the highest-address 64 bits are placed in the odd-numbered register.

**JPS Compatibility Note –** ASIs $24_{16}$ and $2C_{16}$ perform an access using a virtual address from the nucleus context, while ASIs $34_{16}$ and $3C_{16}$ use a physical address.

In addition to the usual traps for LDDA using a privileged ASI, a *data_access_exception* trap occurs for an access to a noncacheable page or if a quadword-load ASI is used with any instruction other than LDDA. A *mem_address_not_aligned* trap is taken if the access is not aligned on a 16-byte boundary.

**IMPL. DEP. #306:** The trap type generated when Load Quadword, Atomic is issued to an address that is not mapped to cacheable memory space is implementation dependent.

A Load Quadword Atomic instruction that performs a Little-Endian access behaves as if it comprises two 64-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

---

**JPS Compatibility Note –** To reduce the number of locked pages in dTLB, a new ASI load instruction, atomic quad load physical (`ldda ASI_QUAD_LDD_PHYS`), was added.

---

Atomic quad load physical allows a full TTE entry (128 bits, tag and data) in TSB to be read directly with PA, bypassing the VA-to-PA translation. In today's dTLB miss handler, a TTE entry is read using two ldx instructions. ASIs 3416 and 3C16 are not translated by the MMU and addresses provided are interpreted directly as physical addresses.

Since quad load with these ASIs bypasses the dMMU, the physical address is set equal to the truncated virtual address, that is, PA[42:0]=VA[42:0]. Internally in hardware, the physical page attribute bits of these ASIs are hardcoded (not coming from DCU Control Register) as follows:

CP = 1, CV = 0, IE = 0, E = 0, P = 0, W = 0, NFO = 0, Size = 8 K

Note that (CP, CV) = 10 means it is cacheable in L2-cache, W-cache, and P-cache, but not D-cache (since D-cache is VA-indexed). Therefore, this atomic quad load physical instruction can only be used with cacheable PA.

Semantically, `ASI_QUAD_LDD_PHYS` is like a combination of `ASI_NUCLEUS_QUAD_LDD` and `ASI_PHYS_USE_EC`.

An illegal_instruction occurs if an odd "rd" register number is used. If non-privileged software tries to use this ASI, a privileged_action exception occurs. If the physical address of the data referenced matches the watchpoint register (`ASI_dMMU_PA_WATCHPOINT_REG`), the *PA_watchpoint* exception occurs.

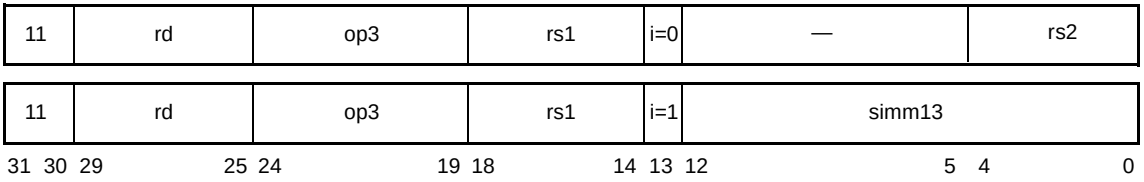
*Exceptions:*

*privileged_action*
*PA_watchpoint* (recognized on only the first 8 bytes of an access)
*VA_watchpoint* (recognized on only the first 8 bytes of an access)
*illegal_instruction* (misaligned `rd`)
*mem_address_not_aligned*
*data_access_exception* (an attempt to access a page marked as noncacheable)
*data_access_error*
*fast_data_access_MMU_miss*

# A.31 Load-Store Unsigned Byte

| Opcode | op3 | Operation |
|---|---|---|
| LDSTUB | 00 1101 | Load-Store Unsigned Byte |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| ldstub | [*address*], $reg_{rd}$ |

*Description*

The load-store unsigned byte instruction copies a byte from memory into R[rd], then rewrites the addressed byte in memory to all 1's. The fetched byte is right-justified in the destination register R[rd] and zero-filled on the left.

The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing LDSTUB, LDSTUBA, CASA, CASXA, SWAP, or SWAPA instructions addressing all or parts of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + sign_ext(simm13)" if i = 1.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120).

Accesses to a noncacheable page generates a *data_access_exception*.

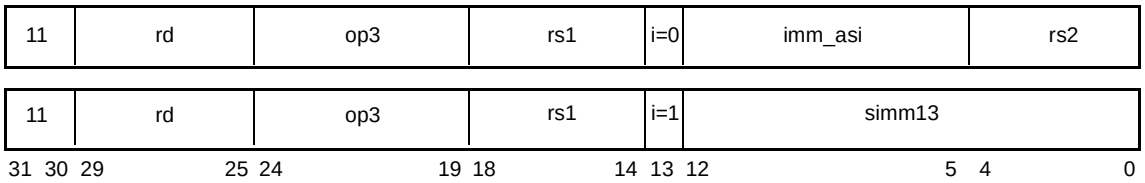
*Exceptions*

*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*VA_watchpoint*
*PA_watchpoint*

# A.32 Load-Store Unsigned Byte to Alternate Space

| Opcode | op3 | Operation |
|---|---|---|
| LDSTUBA$^{PASI}$ | 01 1101 | Load-Store Unsigned Byte into Alternate Space |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| `ldstuba` | [*regaddr*] *imm_asi*, $reg_{rd}$ |
| `ldstuba` | [*reg_plus_imm*] `%asi`, $reg_{rd}$ |

*Description*

The load-store unsigned byte into alternate space instruction copies a byte from memory into `R[rd]`, then rewrites the addressed byte in memory to all 1's. The fetched byte is right-justified in the destination register `R[rd]` and zero-filled on the left.

The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing `LDSTUB`, `LDSTUBA`, `CASA`, `CASXA`, `SWAP`, or `SWAPA` instructions addressing all or parts of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

`LDSTUBA` contains the address space identifier (ASI) to be used for the load in the `imm_asi` field if `i` = 0, or in the ASI register if `i` = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address is "`R[rs1]` + `R[rs2]`" if `i` = 0, or "`R[rs1]` + `sign_ext(simm13)`" if `i` = 1.

`LDSTUBA` causes a *privileged_action* exception if `PSTATE.priv` = 0 and bit 7 of the ASI is 0.

For information about the coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses, see Appendix F of the JPS2 Extensions documents.

Accesses to a noncacheable page generates a *data_access_exception*.

*Exceptions*

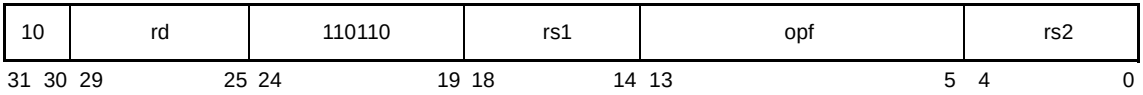
*privileged_action*
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*VA_watchpoint*
*PA_watchpoint*

# A.33 Logical Operate Instructions (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| FZERO | 0 0110 0000 | Zero fill |
| FZEROS | 0 0110 0001 | Zero fill, single precision |
| FONE | 0 0111 1110 | One fill |
| FONES | 0 0111 1111 | One fill, single precision |
| FSRC1 | 0 0111 0100 | Copy src1 |
| FSRC1S | 0 0111 0101 | Copy src1, single precision |
| FSRC2 | 0 0111 1000 | Copy src2 |
| FSRC2S | 0 0111 1001 | Copy src2, single precision |
| FNOT1 | 0 0110 1010 | Negate (1's complement) src1 |
| FNOT1S | 0 0110 1011 | Negate (1's complement) src1, single precision |
| FNOT2 | 0 0110 0110 | Negate (1's complement) src2 |
| FNOT2S | 0 0110 0111 | Negate (1's complement) src2, single precision |
| FOR | 0 0111 1100 | Logical OR |
| FORS | 0 0111 1101 | Logical OR, single precision |
| FNOR | 0 0110 0010 | Logical NOR |
| FNORS | 0 0110 0011 | Logical NOR, single precision |
| FAND | 0 0111 0000 | Logical AND |
| FANDS | 0 0111 0001 | Logical AND, single precision |
| FNAND | 0 0110 1110 | Logical NAND |
| FNANDS | 0 0110 1111 | Logical NAND, single precision |
| FXOR | 0 0110 1100 | Logical XOR |
| FXORS | 0 0110 1101 | Logical XOR, single precision |
| FXNOR | 0 0111 0010 | Logical XNOR |
| FXNORS | 0 0111 0011 | Logical XNOR, single precision |
| FORNOT1 | 0 0111 1010 | Negated src1 OR src2 |
| FORNOT1S | 0 0111 1011 | Negated src1 OR src2, single precision |
| FORNOT2 | 0 0111 0110 | src1 OR negated src2 |
| FORNOT2S | 0 0111 0111 | src1 OR negated src2, single precision |
| FANDNOT1 | 0 0110 1000 | Negated src1 AND src2 |
| FANDNOT1S | 0 0110 1001 | Negated src1 AND src2, single precision |

| Opcode | opf | Operation |
|---|---|---|
| FANDNOT2 | 0 0110 0100 | src1 AND negated src2 |
| FANDNOT2S | 0 0110 0101 | src1 AND negated src2, single precision |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fzero | $freg_{rd}$ |
| fzeros | $freg_{rd}$ |
| fone | $freg_{rd}$ |
| fones | $freg_{rd}$ |
| fsrc1 | $freg_{rs1}$, $freg_{rd}$ |
| fsrc1s | $freg_{rs1}$, $freg_{rd}$ |
| fsrc2 | $freg_{rs2}$, $freg_{rd}$ |
| fsrc2s | $freg_{rs2}$, $freg_{rd}$ |
| fnot1 | $freg_{rs1}$, $freg_{rd}$ |
| fnot1s | $freg_{rs1}$, $freg_{rd}$ |
| fnot2 | $freg_{rs2}$, $freg_{rd}$ |
| fnot2s | $freg_{rs2}$, $freg_{rd}$ |
| for | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fors | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fnor | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fnors | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fand | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fand | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fnands | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fnands | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fxor | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fxors | $fregrs1$, $freg_{rs2}$, $freg_{rd}$ |
| fxnor | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fxnors | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fornot1 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

| Assembly Language Syntax | |
|---|---|
| `fornot1s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| `fornot2` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| `fornot2s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| `fandnot1` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| `fandnot1s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| `fandnot2` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| `fandnot2s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

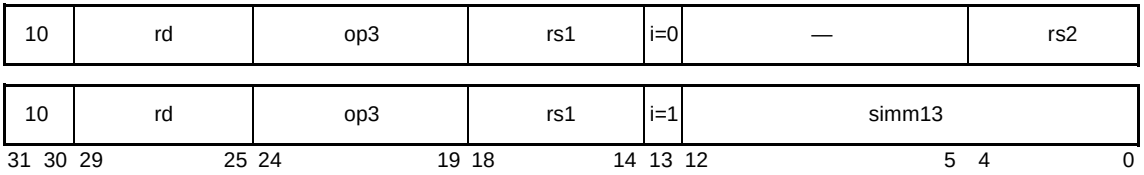
*Description* The standard 64-bit versions of these instructions perform one of sixteen 64-bit logical operations between the 64-bit floating-point registers specified by `rs1` and `rs2`. The result is stored in the 64-bit floating-point destination register specified by `rd`. The 32-bit (single-precision) version of these instructions perform 32-bit logical operations.

*Exceptions* *fp_disabled*

# A.34 Logical Operations

| Opcode | op3 | Operation |
|---|---|---|
| AND | 00 0001 | And |
| ANDcc | 01 0001 | And and modify cc's |
| ANDN | 00 0101 | And Not |
| ANDNcc | 01 0101 | And Not and modify cc's |
| OR | 00 0010 | Inclusive Or |
| ORcc | 01 0010 | Inclusive Or and modify cc's |
| ORN | 00 0110 | Inclusive Or Not |
| ORNcc | 01 0110 | Inclusive Or Not and modify cc's |
| XOR | 00 0011 | Exclusive Or |
| XORcc | 01 0011 | Exclusive Or and modify cc's |
| XNOR | 00 0111 | Exclusive Nor |
| XNORcc | 01 0111 | Exclusive Nor and modify cc's |

*Format (3)*



| Assembly Language Syntax | |
|---|---|
| `and` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `andcc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `andn` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `andncc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `or` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `orcc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `orn` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `orncc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `xor` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `xorcc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `xnor` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| `xnorcc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |

*Description*

These instructions implement bitwise logical operations. They compute "`R[rs1]` **op** `R[rs2]`" if `i` = 0, or "`R[rs1]` **op** `sign_ext(simm13)`" if `i` = 1, and write the result into `R[rd]`.

`ANDcc`, `ANDNcc`, `ORcc`, `ORNcc`, `XORcc`, and `XNORcc` modify the integer condition codes (`icc` and `xcc`). They set the condition codes as follows:

- `icc.v`, `icc.c`, `xcc.v`, and `xcc.c` are set to 0
- `icc.n` is copied from bit 31 of the result
- `xcc.n` is copied from bit 63 of the result
- `icc.z` is set to 1 if bits 31:0 of the result are zero (otherwise to 0)
- `xcc.z` is set to 1 if all 64 bits of the result are zero (otherwise to 0)

`ANDN`, `ANDNcc`, `ORN`, and `ORNcc` logically negate their second operand before applying the main (AND or OR) operation.

**Programming Note –** `XNOR` and `XNORcc` are identical to the XOR-Not and XOR-Not-cc logical operations, respectively.

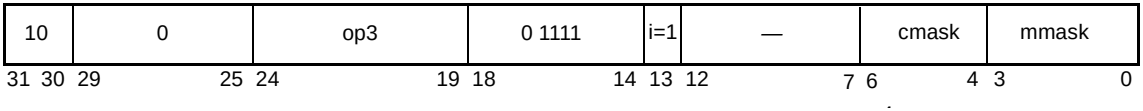
*Exceptions*

None

# A.35 Memory Barrier

| Opcode | op3 | Operation |
|---|---|---|
| MEMBAR | 10 1000 | Memory Barrier |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| membar | *membar_mask* |

*Description*

The memory barrier instruction, MEMBAR, has two complementary functions: to express order constraints between memory references and to provide explicit control of memory-reference completion. The membar_mask field in the suggested assembly language is the concatenation of the cmask and mmask instruction fields.

MEMBAR introduces an order constraint between classes of memory references appearing before the MEMBAR and memory references following it in a program. The particular classes of memory references are specified by the mmask field. Memory references are classified as loads (including load instructions LDSTUB(A), SWAP(A), CASA, and CASXA and stores (including store instructions LDSTUB(A), SWAP(A), CASA, CASXA, and FLUSH). The mmask field specifies the classes of memory references subject to ordering, as described below. MEMBAR applies to all memory operations in all address spaces referenced by the issuing virtual processor, but it has no effect on memory references by other virtual processors. When the cmask field is nonzero, completion as well as order constraints are imposed, and the order imposed can be more stringent than that specifiable by the mmask field alone.

A load has been performed when the value loaded has been transmitted from memory and cannot be modified by another virtual processor. A store has been performed when the value stored has become visible, that is, when the previous value can no longer be read by any virtual processor. In specifying the effect of MEMBAR, instructions are considered to be executed as if they were processed in a strictly sequential fashion, with each instruction completed before the next has begun.

The mmask field is encoded in bits 3 through 0 of the instruction. TABLE A-6 specifies the order constraint that each bit of mmask (selected when set to 1) imposes on memory references appearing before and after the MEMBAR. From zero to four mask bits may be selected in the mmask field.

**TABLE A-6** MEMBAR mmask Encodings

| Mask Bit | Name | Description |
|---|---|---|
| mmask<3> | #StoreStore | The effects of all stores appearing prior to the MEMBAR instruction must be visible to all virtual processors before the effect of any stores following the MEMBAR. Equivalent to the deprecated STBAR instruction. |
| mmask<2> | #LoadStore | All loads appearing prior to the MEMBAR instruction must have been performed before the effects of any stores following the MEMBAR are visible to any other virtual processor. |
| mmask<1> | #StoreLoad | The effects of all stores appearing prior to the MEMBAR instruction must be visible to all virtual processors before loads following the MEMBAR may be performed. |
| mmask<0> | #LoadLoad | All loads appearing prior to the MEMBAR instruction must have been performed before any loads following the MEMBAR may be performed. |

The cmask field is encoded in bits 6 through 4 of the instruction. Bits in the cmask field, described in TABLE A-7, specify additional constraints on the order of memory references and the processing of instructions. If cmask is zero, then MEMBAR enforces the partial ordering specified by the mmask field; if cmask is nonzero, then completion and partial order constraints are applied.

**TABLE A-7** MEMBAR cmask Encodings

| Mask Bit | Function | Name | Description |
|---|---|---|---|
| cmask[2] | Synchronization barrier | #Sync | All operations (including nonmemory reference operations) appearing prior to the MEMBAR must have been performed and the effects of any exceptions be visible before any instruction after the MEMBAR may be initiated. |
| cmask[1] | Memory issue barrier | #MemIssue | All memory reference operations appearing prior to the MEMBAR must have been performed before any memory operation after the MEMBAR may be initiated. |
| cmask[0] | Lookaside barrier | #Lookaside | A store appearing prior to the MEMBAR must complete before any load following the MEMBAR referencing the same address can be initiated. |

For information on the use of MEMBAR, see 8.4.3, *MEMBAR Instruction*, and Appendix J, *Programming with the Memory Models*. For additional information about the memory models themselves, see Chapter 8, *Memory Models*.

The encoding of MEMBAR is identical to that of the RDASR instruction, except that rs1 = 15, rd = 0, and i = 1.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120).

---

**V9 Compatibility Note –** `MEMBAR` with `mmask` = $8_{16}$ and `cmask` = $0_{16}$ ("`membar #StoreStore`") is identical in function to the SPARC V8 `STBAR` instruction, which is deprecated.

---

*Exceptions* None

# A.36 Move Floating-Point Register on Condition (FMOVcc)

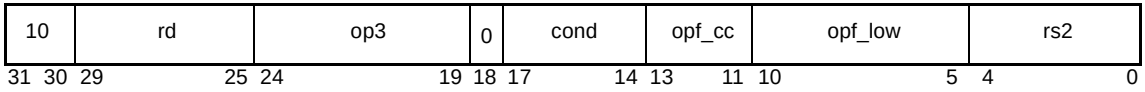
*For Integer Condition Codes*

| Opcode | op3 | cond | Operation | *icc*/*xcc* Test |
|---|---|---|---|---|
| FMOVA | 11 0101 | 1000 | Move Always | 1 |
| FMOVN | 11 0101 | 0000 | Move Never | 0 |
| FMOVNE | 11 0101 | 1001 | Move if Not Equal | **not** Z |
| FMOVE | 11 0101 | 0001 | Move if Equal | Z |
| FMOVG | 11 0101 | 1010 | Move if Greater | **not** (Z **or** (N **xor** V)) |
| FMOVLE | 11 0101 | 0010 | Move if Less or Equal | Z **or** (N **xor** V) |
| FMOVGE | 11 0101 | 1011 | Move if Greater or Equal | **not** (N **xor** V) |
| FMOVL | 11 0101 | 0011 | Move if Less | N **xor** V |
| FMOVGU | 11 0101 | 1100 | Move if Greater Unsigned | **not** (C **or** Z) |
| FMOVLEU | 11 0101 | 0100 | Move if Less or Equal Unsigned | (C **or** Z) |
| FMOVCC | 11 0101 | 1101 | Move if Carry Clear (Greater or Equal, Unsigned) | **not** C |
| FMOVCS | 11 0101 | 0101 | Move if Carry Set (Less than, Unsigned) | C |
| FMOVPOS | 11 0101 | 1110 | Move if Positive | **not** N |
| FMOVNEG | 11 0101 | 0110 | Move if Negative | N |
| FMOVVC | 11 0101 | 1111 | Move if Overflow Clear | **not** V |
| FMOVVS | 11 0101 | 0111 | Move if Overflow Set | V |

*For Floating-Point Condition Codes*

| Opcode | op3 | cond | Operation | *fcc* Test |
|---|---|---|---|---|
| FMOVFA | 11 0101 | 1000 | Move Always | 1 |
| FMOVFN | 11 0101 | 0000 | Move Never | 0 |
| FMOVFU | 11 0101 | 0111 | Move if Unordered | U |
| FMOVFG | 11 0101 | 0110 | Move if Greater | G |
| FMOVFUG | 11 0101 | 0101 | Move if Unordered or Greater | G **or** U |
| FMOVFL | 11 0101 | 0100 | Move if Less | L |
| FMOVFUL | 11 0101 | 0011 | Move if Unordered or Less | L **or** U |
| FMOVFLG | 11 0101 | 0010 | Move if Less or Greater | L **or** G |
| FMOVFNE | 11 0101 | 0001 | Move if Not Equal | L **or** G **or** U |
| FMOVFE | 11 0101 | 1001 | Move if Equal | E |
| FMOVFUE | 11 0101 | 1010 | Move if Unordered or Equal | E **or** U |
| FMOVFGE | 11 0101 | 1011 | Move if Greater or Equal | E **or** G |
| FMOVFUGE | 11 0101 | 1100 | Move if Unordered or Greater or Equal | E **or** G **or** U |
| FMOVFLE | 11 0101 | 1101 | Move if Less or Equal | E **or** L |
| FMOVFULE | 11 0101 | 1110 | Move if Unordered or Less or Equal | E **or** L **or** U |
| FMOVFO | 11 0101 | 1111 | Move if Ordered | E **or** L **or** G |

*Format (4)*



*Encoding of the opf_cc Field (also see* TABLE E-10 *on page 464)*

| opf_cc | Condition Code |
|---|---|
| 000 | `fcc0` |
| 001 | `fcc1` |
| 010 | `fcc2` |
| 011 | `fcc3` |
| 100 | `icc` |
| 101 | — |
| 110 | `xcc` |
| 111 | — |

*Encoding of opf Field (opf_cc  opf_low)*

| Instruction | Variation | opf_cc | opf_low | opf |
|---|---|---|---|---|
| `FMOVScc` | `%fcc`*n,rs2,rd* | 0*nn* | 00 0001 | 0 *nn*00 0001 |
| `FMOVDcc` | `%fcc`*n,rs2,rd* | 0*nn* | 00 0010 | 0 *nn*00 0010 |
| `FMOVQcc` | `%fcc`*n,rs2,rd* | 0*nn* | 00 0011 | 0 *nn*00 0011 |
| `FMOVScc` | `%icc`, *rs2,rd* | 100 | 00 0001 | 1 0000 0001 |
| `FMOVDcc` | `%icc`, *rs2,rd* | 100 | 00 0010 | 1 0000 0010 |
| `FMOVQcc` | `%icc`, *rs2,rd* | 100 | 00 0011 | 1 0000 0011 |
| `FMOVScc` | `%xcc`, *rs2,rd* | 110 | 00 0001 | 1 1000 0001 |
| `FMOVDcc` | `%xcc`, *rs2,rd* | 110 | 00 0010 | 1 1000 0010 |
| `FMOVQcc` | `%xcc`, *rs2,rd* | 110 | 00 0011 | 1 1000 0011 |

*For Integer Condition Codes*

| Assembly Language Syntax | | |
|---|---|---|
| `fmov{s,d,q}a` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}n` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}ne` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | (*synonyms*: `fmov{s,d,q}nz`) |
| `fmov{s,d,q}e` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | (*synonyms*: `fmov{s,d,q}z`) |
| `fmov{s,d,q}g` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}le` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}ge` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}l` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}gu` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}leu` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}cc` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | (*synonyms*: `fmov{s,d,q}geu`) |
| `fmov{s,d,q}cs` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | (*synonyms*: `fmov{s,d,q}lu`) |
| `fmov{s,d,q}pos` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}neg` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}vc` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}vs` | *i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$ | |

**Programming Note –** To select the appropriate condition code, include `%icc` or `%xcc` before the registers.

*For Floating-Point Condition Codes*

| Assembly Language Syntax | | |
|---|---|---|
| `fmov{s,d,q}a` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}n` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}u` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}g` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}ug` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}l` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}ul` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}lg` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}ne` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | (*synonyms:* `fmov{s,d,q}nz`) |
| `fmov{s,d,q}e` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | (*synonyms:* `fmov{s,d,q}z`) |
| `fmov{s,d,q}ue` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}ge` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}uge` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}le` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}ule` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |
| `fmov{s,d,q}o` | `%fccn`, $freg_{rs2}$, $freg_{rd}$ | |

*Description*

These instructions copy the floating-point register(s) specified by `rs2` to the floating-point register(s) specified by `rd` if the condition indicated by the `cond` field is satisfied by the selected condition code. The condition code used is specified by the `opf_cc` field of the instruction. If the condition is `FALSE`, then the destination register(s) are not changed.

These instructions do not modify any condition codes.

**Programming Note –** Branches cause the performance of most implementations to degrade significantly. Frequently, the `MOVcc` and `FMOVcc` instructions can be used to avoid branches. For example, the following C language segment:

```
double A, B, X;
if (A > B) then X  = 1.03; else X  = 0.0;
```

can be coded as

```
! assume A is in %f0; B is in %f2; %xx points to constant area
   ldd       [%xx+C_1.03],%f4   ! X  = 1.03
```

```
      fcmpd     %fcc3,%f0,%f2       ! A > B
      fble ,a   %fcc3,label
      ! following only executed if the branch is taken
      fsubd     %f4,%f4,%f4         ! X  = 0.0
   label:...
```

This code takes four instructions including a branch.

With FMOVcc, this could be coded as

```
      ldd       [%xx+C_1.03],%f4    ! X  = 1.03
      fsubd     %f4,%f4,%f6         ! X'  = 0.0
      fcmpd     %fcc3,%f0,%f2       ! A > B
      fmovdle   %fcc3,%f6,%f4       ! X  = 0.0
```

This code also takes four instructions but requires no branches and may boost performance significantly. Use MOVcc and FMOVcc instead of branches wherever these instructions would improve performance.

---

*Exceptions*

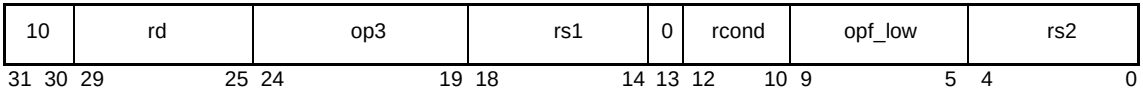
*fp_disabled*

*fp_exception_other* (ftt = unimplemented_FPop (opf_cc = $101_2$ or $111_2$ and quad forms))

# A.37 Move Floating-Point Register on Integer Register Condition (FMOVr)

| Opcode | op3 | rcond | Operation | Test |
|---|---|---|---|---|
| — | 11 0101 | 000 | *Reserved* | — |
| FMOVRZ | 11 0101 | 001 | Move if Register Zero | R[rs1] = 0 |
| FMOVRLEZ | 11 0101 | 010 | Move if Register Less Than or Equal to Zero | R[rs1] ≤ 0 |
| FMOVRLZ | 11 0101 | 011 | Move if Register Less Than Zero | R[rs1] < 0 |
| — | 11 0101 | 100 | *Reserved* | — |
| FMOVRNZ | 11 0101 | 101 | Move if Register Not Zero | R[rs1] ≠ 0 |
| FMOVRGZ | 11 0101 | 110 | Move if Register Greater Than Zero | R[rs1] > 0 |
| FMOVRGEZ | 11 0101 | 111 | Move if Register Greater Than or Equal to Zero | R[rs1] ≥ 0 |

*Format (4)*




*Encoding of* `opf_low` *Field*

| Instruction variation | opf_low |
|---|---|
| FMOVRS*rcond* *rs1, rs2, rd* | 0 0101 |
| FMOVRD*rcond* *rs1, rs2, rd* | 0 0110 |
| FMOVRQ*rcond* *rs1, rs2, rd* | 0 0111 |

| Assembly Language Syntax | | |
|---|---|---|
| `fmovr{s,d,q}e` | $reg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | (*synonym:* `fmovr{s,d,q}z`) |
| `fmovr{s,d,q}lez` | $reg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | |
| `fmovr{s,d,q}lz` | $reg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | |
| `fmovr{s,d,q}ne` | $reg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | (*synonym:* `fmovr{s,d,q}nz`) |
| `fmovr{s,d,q}gz` | $reg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | |
| `fmovr{s,d,q}gez` | $reg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | |

*Description*

If the contents of integer register `R[rs1]` satisfy the condition specified in the `rcond` field, these instructions copy the contents of the floating-point register(s) specified by the `rs2` field to the floating-point register(s) specified by the `rd` field. If the contents of `R[rs1]` do not satisfy the condition, the floating-point register(s) specified by the `rd` field are not modified.

These instructions treat the integer register contents as a signed integer value; they do not modify any condition codes.

**V9 Compatibility Note –** If this instruction is implemented by tagging each register value with an N (negative) and a Z (zero) bit, use the following table to determine whether `rcond` is `TRUE`:

| Branch | Test |
|---|---|
| `FMOVRNZ` | **not** Z |
| `FMOVRZ` | Z |
| `FMOVGEZ` | **not** N |
| `FMOVRLZ` | N |
| `FMOVRLEZ` | N **or** Z |
| `FMOVRGZ` | N **nor** Z |

*Exceptions*

*fp_disabled*
*fp_exception_other* (unimplemented_FPop (`rcond` = $000_2$ or $100_2$ and quad forms))

# A.38 Move Integer Register on Condition (MOVcc)

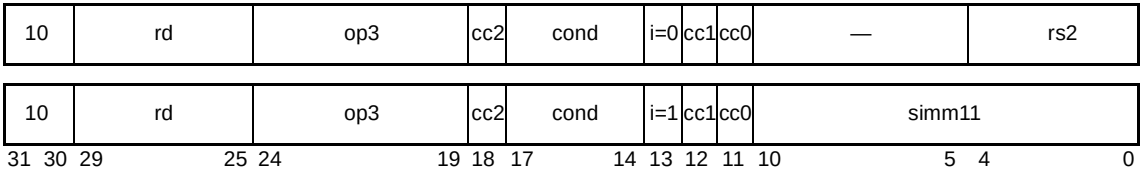
*For Integer Condition Codes*

| Opcode | op3 | cond | Operation | icc/xcc Test |
|---|---|---|---|---|
| MOVA | 10 1100 | 1000 | Move Always | 1 |
| MOVN | 10 1100 | 0000 | Move Never | 0 |
| MOVNE | 10 1100 | 1001 | Move if Not Equal | **not** Z |
| MOVE | 10 1100 | 0001 | Move if Equal | Z |
| MOVG | 10 1100 | 1010 | Move if Greater | **not** (Z **or** (N **xor** V)) |
| MOVLE | 10 1100 | 0010 | Move if Less or Equal | Z **or** (N **xor** V) |
| MOVGE | 10 1100 | 1011 | Move if Greater or Equal | **not** (N **xor** V) |
| MOVL | 10 1100 | 0011 | Move if Less | N **xor** V |
| MOVGU | 10 1100 | 1100 | Move if Greater Unsigned | **not** (C **or** Z) |
| MOVLEU | 10 1100 | 0100 | Move if Less or Equal Unsigned | (C **or** Z) |
| MOVCC | 10 1100 | 1101 | Move if Carry Clear (Greater or Equal, Unsigned) | **not** C |
| MOVCS | 10 1100 | 0101 | Move if Carry Set (Less than, Unsigned) | C |
| MOVPOS | 10 1100 | 1110 | Move if Positive | **not** N |
| MOVNEG | 10 1100 | 0110 | Move if Negative | N |
| MOVVC | 10 1100 | 1111 | Move if Overflow Clear | **not** V |
| MOVVS | 10 1100 | 0111 | Move if Overflow Set | V |

*For Floating-Point Condition Codes*

| Opcode | op3 | cond | Operation | fcc Test |
|---|---|---|---|---|
| MOVFA | 10 1100 | 1000 | Move Always | 1 |
| MOVFN | 10 1100 | 0000 | Move Never | 0 |
| MOVFU | 10 1100 | 0111 | Move if Unordered | U |
| MOVFG | 10 1100 | 0110 | Move if Greater | G |
| MOVFUG | 10 1100 | 0101 | Move if Unordered or Greater | G **or** U |
| MOVFL | 10 1100 | 0100 | Move if Less | L |
| MOVFUL | 10 1100 | 0011 | Move if Unordered or Less | L **or** U |
| MOVFLG | 10 1100 | 0010 | Move if Less or Greater | L **or** G |
| MOVFNE | 10 1100 | 0001 | Move if Not Equal | L **or** G **or** U |
| MOVFE | 10 1100 | 1001 | Move if Equal | E |
| MOVFUE | 10 1100 | 1010 | Move if Unordered or Equal | E **or** U |
| MOVFGE | 10 1100 | 1011 | Move if Greater or Equal | E **or** G |
| MOVFUGE | 10 1100 | 1100 | Move if Unordered or Greater or Equal | E **or** G **or** U |
| MOVFLE | 10 1100 | 1101 | Move if Less or Equal | E **or** L |
| MOVFULE | 10 1100 | 1110 | Move if Unordered or Less or Equal | E **or** L **or** U |
| MOVFO | 10 1100 | 1111 | Move if Ordered | E **or** L **or** G |

*Format (4)*



| cc2 | cc1 | cc0 | Condition Code |
|---|---|---|---|
| 0 | 0 | 0 | fcc0 |
| 0 | 0 | 1 | fcc1 |
| 0 | 1 | 0 | fcc2 |
| 0 | 1 | 1 | fcc3 |
| 1 | 0 | 0 | icc |
| 1 | 0 | 1 | *Reserved* |
| 1 | 1 | 0 | xcc |
| 1 | 1 | 1 | *Reserved* |

*For Integer Condition Codes*

| Assembly Language Syntax | | |
|---|---|---|
| `mova` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movn` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movne` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | (*synonym:* `movnz`) |
| `move` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | (*synonym:* `movz`) |
| `movg` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movle` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movge` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movl` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movgu` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movleu` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movcc` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | (*synonym:* `movgeu`) |
| `movcs` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | (*synonym:* `movlu`) |
| `movpos` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movneg` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movvc` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |
| `movvs` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | |

**Programming Note –** To select the appropriate condition code, include `%icc` or `%xcc` before the register or immediate field.

*For Floating-Point Condition Codes*

| Assembly Language Syntax | | |
|---|---|---|
| `mova` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movn` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movu` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movg` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movug` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movl` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movul` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movlg` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movne` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | (*synonym:* `movnz`) |
| `move` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | (*synonym:* `movz`) |
| `movue` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movge` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movuge` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movle` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movule` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |
| `movo` | `%fccn`, *reg_or_imm11*, $reg_{rd}$ | |

**Programming Note –** To select the appropriate condition code, include `%fcc0`, `%fcc1`, `%fcc2`, or `%fcc3` before the register or immediate field.

*Description* These instructions test to see if `cond` is `TRUE` for the selected condition codes. If so, they copy the value in `R[rs2]` if `i` field = 0, or "`sign_ext(simm11)`" if `i` = 1 into `R[rd]`. The condition code used is specified by the `cc2`, `cc1`, and `cc0` fields of the instruction. If the condition is `FALSE`, then `R[rd]` is not changed.

## Move Integer Register on Condition (MOVcc)

These instructions copy an integer register to another integer register if the condition is TRUE. The condition code that is used to determine whether the move will occur can be either integer condition code (icc or xcc) or any floating-point condition code (fcc0, fcc1, fcc2, or fcc3).

These instructions do not modify any condition codes.

---

**Programming Note –** Branches cause the performance of many implementations to degrade significantly. Frequently, the MOVcc and FMOVcc instructions can be used to avoid branches. For example, the C language if-then-else statement

---

```
if (A > B) then X = 1; else X = 0;
```

can be coded as

```
    cmp       %i0,%i2
    bg,a      %xcc,label
    or        %g0,1,%i3          ! X  = 1
    or        %g0,0,%i3          ! X  = 0
label:...
```

This takes four instructions including a branch. With MOVcc this could be coded as

```
    cmp       %i0,%i2
    or        %g0,1,%i3          ! assume X = 1
    movle     %xcc,0,%i3         ! overwrite with X = 0
```

This approach takes only three instructions and no branches and may boost performance significantly. Use MOVcc and FMOVcc instead of branches wherever these instructions would increase performance.

---

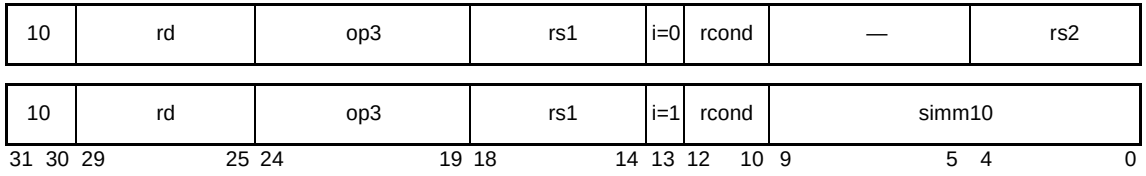
*Exceptions*

*illegal_instruction* (cc2 || cc1 || cc0 = $101_2$ or $111_2$)
*fp_disabled* (cc2 || cc1 || cc0 = $000_2$, $001_2$, $010_2$, or $011_2$ and the FPU is disabled)

# A.39 Move Integer Register on Register Condition (MOVr)

| Opcode | op3 | rcond | Operation | Test |
|---|---|---|---|---|
| — | 10 1111 | 000 | *Reserved* | — |
| MOVRZ | 10 1111 | 001 | Move if Register Zero | R[rs1] = 0 |
| MOVRLEZ | 10 1111 | 010 | Move if Register Less Than or Equal to Zero | R[rs1] ≤ 0 |
| MOVRLZ | 10 1111 | 011 | Move if Register Less Than Zero | R[rs1] < 0 |
| — | 10 1111 | 100 | *Reserved* | — |
| MOVRNZ | 10 1111 | 101 | Move if Register Not Zero | R[rs1] ≠ 0 |
| MOVRGZ | 10 1111 | 110 | Move if Register Greater Than Zero | R[rs1] > 0 |
| MOVRGEZ | 10 1111 | 111 | Move if Register Greater Than or Equal to Zero | R[rs1] ≥ 0 |

*Format (3)*




| Assembly Language Syntax | | |
|---|---|---|
| movrz | *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | (*synonym:* movre) |
| movrlez | *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | |
| movrlz | *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | |
| movrnz | *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | (*synonym:* movrne) |
| movrgz | *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | |
| movrgez | *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | |

## Move Integer Register on Register Condition (MOVr)

*Description* If the contents of integer register `R[rs1]` satisfy the condition specified in the `rcond` field, these instructions copy `R[rs2]` (if `i` = 0) or `sign_ext(simm10)` (if `i` = 1) into `R[rd]`. If the contents of `R[rs1]` do not satisfy the condition, then `R[rd]` is not modified. These instructions treat the register contents as a signed integer value; they do not modify any condition codes.

---

**V9 Compatibility Note –** If this instruction is implemented by tagging each register value with an `n` (negative) and a `z` (zero) bit, use the table below to determine if `rcond` is `TRUE`.

| Move | Test |
|---|---|
| `MOVRNZ` | **not** Z |
| `MOVRZ` | Z |
| `MOVRGEZ` | **not** N |
| `MOVRLZ` | N |
| `MOVRLEZ` | N **or** Z |
| `MOVRGZ` | N **nor** Z |

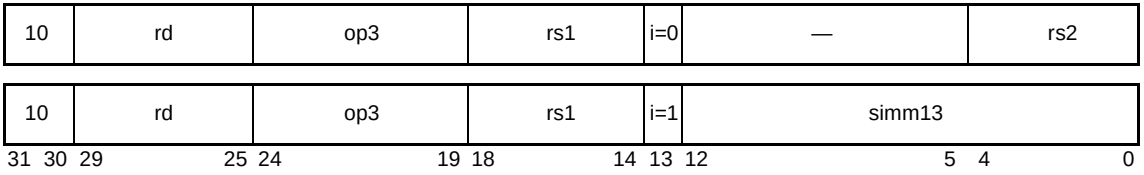
---

*Exceptions* *illegal_instruction* (`rcond` = $000_2$ or $100_2$)

# A.40 Multiply and Divide (64-bit)

| Opcode | op3 | Operation |
|---|---|---|
| MULX | 00 1001 | Multiply (signed or unsigned) |
| SDIVX | 10 1101 | Signed Divide |
| UDIVX | 00 1101 | Unsigned Divide |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| mulx | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| sdivx | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| udivx | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |

*Description*

MULX computes "R[rs1] × R[rs2]" if i = 0 or "R[rs1] × sign_ext(simm13)" if i = 1, and writes the 64-bit product into R[rd]. MULX can be used to calculate the 64-bit product for signed or unsigned operands (the product is the same).

SDIVX and UDIVX compute "R[rs1] ÷ R[rs2]" if i = 0 or "R[rs1] ÷ sign_ext(simm13)" if i = 1, and write the 64-bit result into R[rd]. SDIVX operates on the operands as signed integers and produces a corresponding signed result. UDIVX operates on the operands as unsigned integers and produces a corresponding unsigned result.

For SDIVX, if the largest negative number is divided by –1, the result should be the largest negative number. That is:

$8000\ 0000\ 0000\ 0000_{16} \div FFFF\ FFFF\ FFFF\ FFFF_{16} = 8000\ 0000\ 0000\ 0000_{16}$.

These instructions do not modify any condition codes.

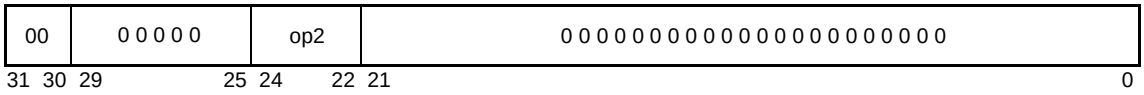
*Exceptions*

*division_by_zero*

# A.41 No Operation

| Opcode | op2 | Operation |
|---|---|---|
| NOP | 100 | No Operation |

*Format (2)*




| Assembly Language Syntax |
|---|
| nop |

*Description* The NOP instruction changes no program-visible state (except that of the PC and nPC).

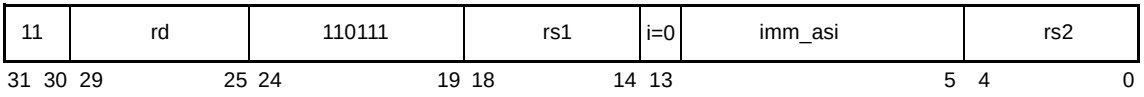
NOP is a special case of the SETHI instruction, with imm22 = 0 and rd = 0.

*Exceptions* None

# A.42 Partial Store (VIS I)

| Opcode | imm_asi | ASI Value | Operation |
|---|---|---|---|
| STDFA | ASI_PST8_P | $C0_{16}$ | Eight 8-bit conditional stores to primary address space |
| STDFA | ASI_PST8_S | $C1_{16}$ | Eight 8-bit conditional stores to secondary address space |
| STDFA | ASI_PST8_PL | $C8_{16}$ | Eight 8-bit conditional stores to primary address space, little-endian |
| STDFA | ASI_PST8_SL | $C9_{16}$ | Eight 8-bit conditional stores to secondary address space, little-endian |
| STDFA | ASI_PST16_P | $C2_{16}$ | Four 16-bit conditional stores to primary address space |
| STDFA | ASI_PST16_S | $C3_{16}$ | Four 16-bit conditional stores to secondary address space |
| STDFA | ASI_PST16_PL | $CA_{16}$ | Four 16-bit conditional stores to primary address space, little-endian |
| STDFA | ASI_PST16_SL | $CB_{16}$ | Four 16-bit conditional stores to secondary address space, little-endian |
| STDFA | ASI_PST32_P | $C4_{16}$ | Two 32-bit conditional stores to primary address space |
| STDFA | ASI_PST32_S | $C5_{16}$ | Two 32-bit conditional stores to secondary address space |
| STDFA | ASI_PST32_PL | $CC_{16}$ | Two 32-bit conditional stores to primary address space, little-endian |
| STDFA | ASI_PST32_SL | $CD_{16}$ | Two 32-bit conditional stores to secondary address space, little-endian |

*Format (3)*




| Assembly Language Syntax[1] | |
|---|---|
| stda | *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] *imm_asi* |

1. The original assembly language syntax for a Partial Store instruction ("stda *freg*$_{rd}$, [*reg*$_{rs1}$] *reg*$_{rs2}$, *imm_asi*") has been deprecated because of inconsistency with the rest of the SPARC assembly language. Over time, assemblers will support the new syntax for this instruction. In the meantime, some assemblers may recognize only the original syntax.

*Description*

The partial store instructions are selected by one of the partial store ASIs with the STDFA instruction.

Two 32-bit, four 16-bit, or eight 8-bit values from the 64-bit floating-point register specified by rd are conditionally stored at the address specified by R[rs1], using the mask specified in R[rs2]. The value in R[rs2] has the same format as the result specified by the pixel compare instructions (see *Pixel Compare (VIS I)* on page 323). The most significant bit of the mask (not the entire register) corresponds to the most significant part of the floating-point register specified by rd. The data is stored in little-endian form in memory if the ASI name has an "L" suffix; otherwise, it is stored in big-endian format.

A *mem_address_not_aligned* exception is generated if the operand address is not 8-byte aligned

A Partial Store instruction can cause a virtual (or physical) watchpoint exception when the following conditions are met:

- The virtual (physical) address in R[rs1] matches the address in the VA (PA) Data Watchpoint Register.
- The byte store mask in R[rs2] indicates that a byte is to be stored.
- The Virtual (Physical) Data Watchpoint Mask in DCUCR indicates that one or more of the bytes to be stored at the watched address is being watched.

**IMPL. DEP. #249:** For a Partial Store instruction, the following aspects of data watchpoints are implementation dependent: (a) whether data watchpoint logic examines the byte store mask in R[rs2] or it conservatively behaves as if every Partial Store always stores all 8 bytes, and (b) whether data watchpoint logic examines individual bits in the Virtual (Physical) Data Watchpoint Mask in DCUCR to determine which bytes are being watched or (when the Watchpoint Mask is nonzero) it conservatively behaves as if all 8 bytes are being watched.

ASIs $C0_{16}$-$C5_{16}$ and $C8_{16}$-$CD_{16}$ are only used for partial store operations. In particular, they should not be used with the LDDFA instruction. See *Partial Store ASIs* on page 582 for more information.

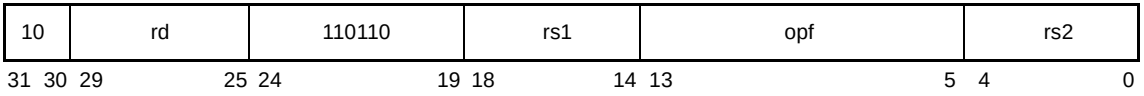
*Exceptions*

*fp_disabled*
*illegal_instruction* (when i = 1: no immediate mode is supported.)
*PA_watchpoint* (see text)
*VA_watchpoint* (see text)
*mem_address_not_aligned*
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*

# A.43 Partitioned Add/Subtract Instructions (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| FPADD16 | 0 0101 0000 | Four 16-bit Add |
| FPADD16S | 0 0101 0001 | Two 16-bit Add |
| FPADD32 | 0 0101 0010 | Two 32-bit Add |
| FPADD32S | 0 0101 0011 | One 32-bit Add |
| FPSUB16 | 0 0101 0100 | Four 16-bit Subtract |
| FPSUB16S | 0 0101 0101 | Two 16-bit Subtract |
| FPSUB32 | 0 0101 0110 | Two 32-bit Subtract |
| FPSUB32S | 0 0101 0111 | One 32-bit Subtract |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fpadd16 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fpadd16s | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fpadd32 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fpadd32s | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fpsub16 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fpsub16s | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fpsub32 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fpsub32s | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

*Description*

The standard versions of these instructions perform four 16-bit or two 32-bit partitioned adds or subtracts between the corresponding fixed-point values contained in the source operands (the 64-bit floating-point registers specified by `rs1` and `rs2`). For subtraction, the second operand is subtracted from the first operand. The result is placed in the 64-bit destination register specified by `rd`.

The single-precision versions of these instructions (`FPADD16S`, `FPSUB16S`, `FPADD32S`, `FPSUB32S`) perform two 16-bit or one 32-bit partitioned add(s) or subtract(s); only the low 32-bits of the destination register are affected.

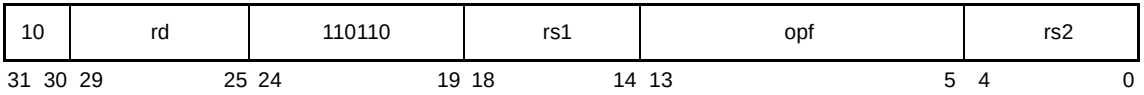
*Exceptions*

*fp_disabled*

# A.44 Partitioned Multiply Instructions (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| FMUL8x16 | 0 0011 0001 | 8-bit x 16-bit Partitioned Product |
| FMUL8x16AU | 0 0011 0011 | 8-bit x 16-bit Upper α Partitioned Product |
| FMUL8x16AL | 0 0011 0101 | 8-bit x 16-bit Upper α Partitioned Product |
| FMUL8SUx16 | 0 0011 0110 | Upper 8-bit x 16-bit Partitioned Product |
| FMUL8ULx16 | 0 0011 0111 | Lower Unsigned 8-bit x 16-bit Partitioned Product |
| FMULD8SUx16 | 0 0011 1000 | Upper 8-bit x 16-bit Partitioned Product |
| FMULD8ULx16 | 0 0011 1001 | Lower Unsigned 8-bit x 16-bit Partitioned Product |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fmul8x16 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fmul8x16au | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fmul8x16al | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fmul8sux16 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fmul8ulx16 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fmuld8sux16 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fmuld8ulx16 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

*Description*

**Note –** For good performance, the result of a partitioned multiply should not be used as the source operand of a 32-bit VIS graphics instruction in the next three instruction groups.

**Programming Note –** When software emulates an 8-bit unsigned by 16-bit signed multiply, the unsigned value must be zero-extended and the 16-bit value sign-extended before the multiplication.

The following sections describe the versions of partitioned multiplies.

*Exceptions* *fp_disabled*

## A.44.1 FMUL8x16 Instruction

`FMUL8x16` multiplies each unsigned 8-bit value (for example, a pixel component) in the 32-bit floating-point register `f[rs1]` by the corresponding (signed) 16-bit fixed-point integer in the 64-bit floating-point register specified by `rs2`. It rounds the 24-bit product (assuming binary point between bits 7 and 8) and stores the most significant 16 bits of the result into the corresponding 16-bit field in the 64-bit floating-point destination register specified by `rd`. FIGURE A-5 illustrates the operation.

**Note –** This instruction treats the pixel component values as fixed-point with the binary point to the left of the most significant bit. Typically, this operation is used with filter coefficients as the fixed-point `rs2` value and image data as the `rs1` pixel value. Appropriate scaling of the coefficient allows various fixed-point scaling to be realized.

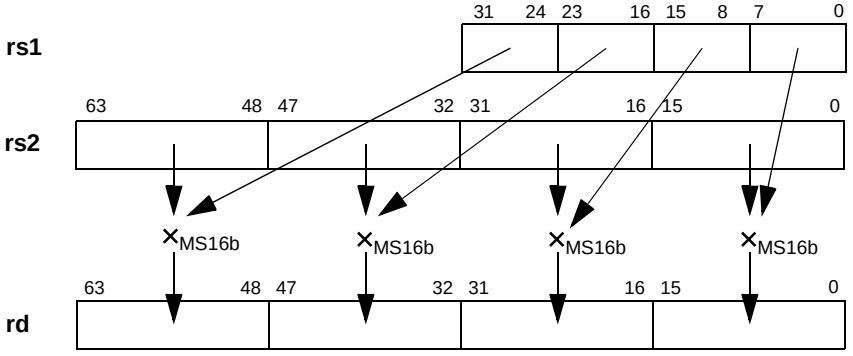



**FIGURE A-5** `FMUL8x16` Operation

## A.44.2 FMUL8x16AU Instruction

FMUL8x16AU is the same as FMUL8x16, except that one 16-bit fixed-point value is used as the multiplier for all four multiplies. This multiplier is the most significant ("upper") 16 bits of the 32-bit register f[rs2] (typically an α pixel component value). FIGURE A-6 illustrates the operation.

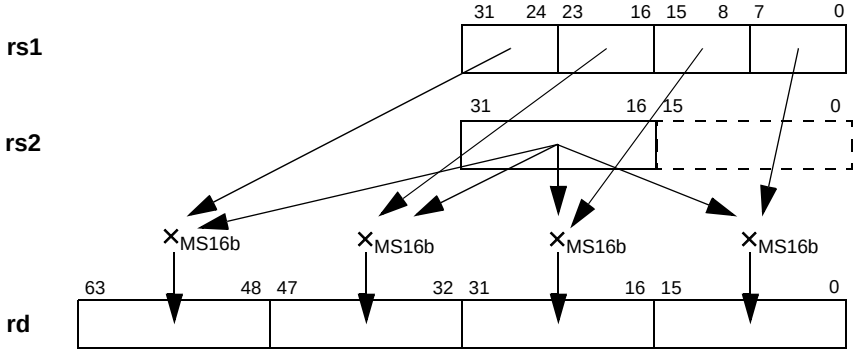



**FIGURE A-6** FMUL8x16AU Operation

## A.44.3 FMUL8x16AL Instruction

FMUL8x16AL is the same as FMUL8x16AU, except that the *least* significant ("lower") 16 bits of the 32-bit register f[rs2] register are used as the multiplier. FIGURE A-7 illustrates the operation.

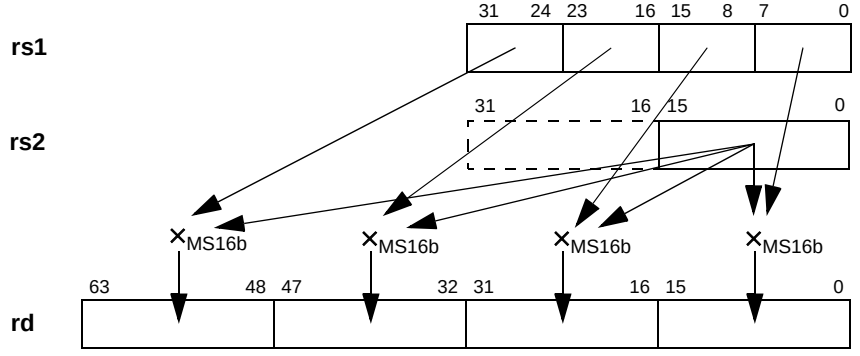



**FIGURE A-7** FMUL8x16AL Operation

## A.44.4 FMUL8SUx16 Instruction

`FMUL8SUx16` multiplies the most significant (“upper”) 8 bits of each 16-bit signed value in the 64-bit floating-point register specified by `rs1` by the corresponding signed, 16-bit, fixed-point, signed integer in the 64-bit floating-point register specified by `rs2`. It rounds the 24-bit product toward the nearest representable value and then stores the most significant 16 bits of the result into the corresponding 16-bit field of the 64-bit floating-point destination register specified by `rd`. If the product is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE A-8 illustrates the operation.

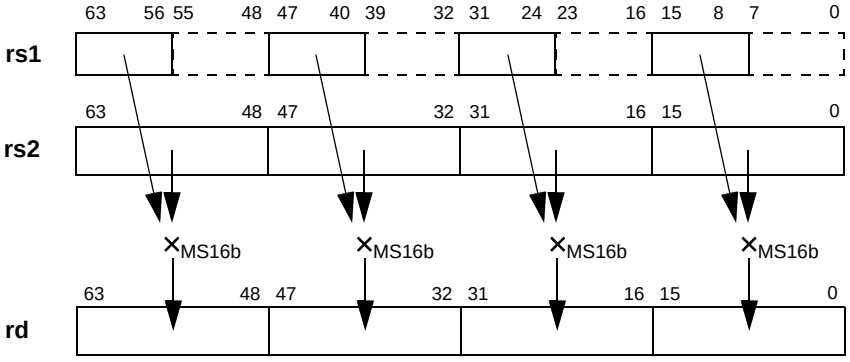



**FIGURE A-8** `FMUL8SUx16` Operation

## A.44.5 FMUL8ULx16 Instruction

`FMUL8ULx16` multiplies the unsigned least significant (“lower”) 8 bits of each 16-bit value in the 64-bit floating-point register specified by `rs1` by the corresponding fixed-point signed 16-bit integer in the 64-bit floating-point register specified by `rs2`. Each 24-bit product is sign-extended to 32 bits. The most significant (“upper”) 16 bits of the sign-extended value are rounded to nearest and then stored in the corresponding 16-bit field of the 64-bit floating-point destination register specified by `rd`. If the result is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE A-9 illustrates the operation; CODE EXAMPLE A-4 exemplifies the operation.

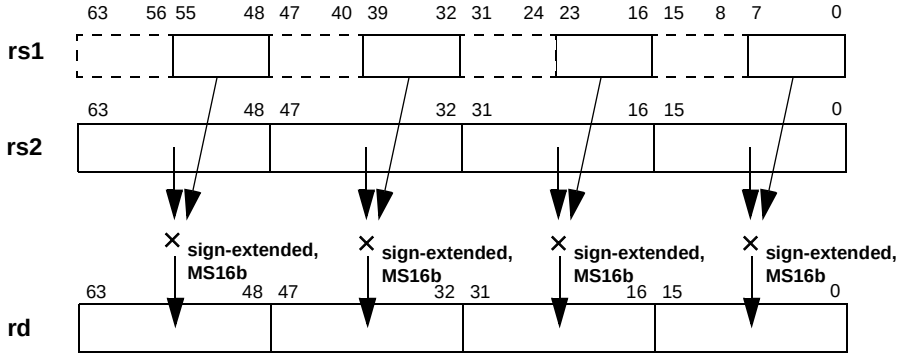



**FIGURE A-9** FMUL8ULx16 Operation

**CODE EXAMPLE A-3** 16-bit x 16-bit → 16-bit Multiply

```
fmul8sux16    %f0, %f1, %f2
fmul8ulx16    %f0, %f1, %f3
fpadd16       %f2, %f3, %f4
```

## A.44.6 FMULD8SUx16 Instruction

FMULD8SUx16 multiplies the most significant ("upper") 8 bits of each 16-bit signed value in f[rs1] by the corresponding signed 16-bit fixed-point value in f[rs2]. Each 24-bit product is shifted left by 8 bits to generate a 32-bit result, which is then stored in the 64-bit floating-point register specified by rd. FIGURE A-10 illustrates the operation.

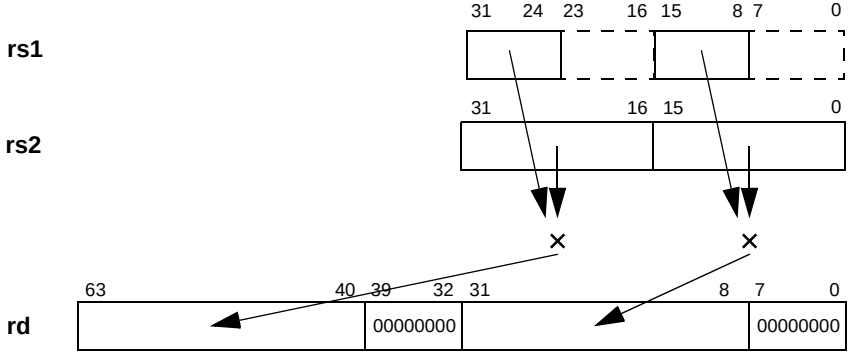



**FIGURE A-10** FMULD8SUx16 Operation

## A.44.7 FMULD8ULx16 Instruction

FMULD8ULx16 multiplies the unsigned least significant ("lower") 8 bits of each 16-bit value in f[rs1] by the corresponding 16-bit fixed-point signed integer in f[rs2]. Each 24-bit product is sign-extended to 32 bits and stored in the corresponding half of the 64-bit floating-point register specified by rd. FIGURE A-11 illustrates the operation; CODE EXAMPLE A-4 exemplifies the operation.

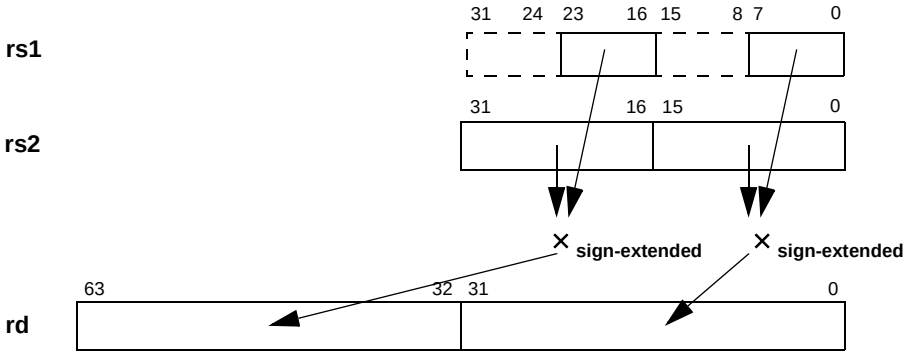



**FIGURE A-11** FMULD8ULx16 Operation

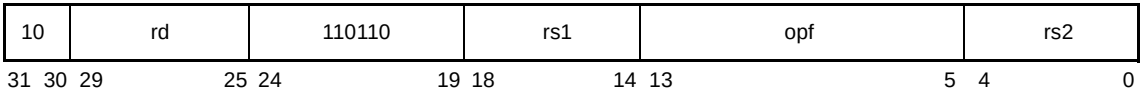
**CODE EXAMPLE A-4** 16-bit x 16-bit → 32-bit Multiply

```
fmuld8sux16   %f0, %f1, %f2
fmuld8ulx16   %f0, %f1, %f3
fpadd32       %f2, %f3, %f4
```

# A.45 Pixel Compare (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| FCMPGT16 | 0 0010 1000 | Four 16-bit Compares; set rd if src1 > src2 |
| FCMPGT32 | 0 0010 1100 | Two 32-bit Compares; set rd if src1 > src2 |
| FCMPLE16 | 0 0010 0000 | Four 16-bit Compares; set rd if src1 ≤ src2 |
| FCMPLE32 | 0 0010 0100 | Two 32-bit Compares; set rd if src1 ≤ src2 |
| FCMPNE16 | 0 0010 0010 | Four 16-bit Compares; set rd if src1 ≠ src2 |
| FCMPNE32 | 0 0010 0110 | Two 32-bit Compares; set rd if src1 ≠ src2 |
| FCMPEQ16 | 0 0010 1010 | Four 16-bit Compares; set rd if src1 = src2 |
| FCMPEQ32 | 0 0010 1110 | Two 32-bit Compares; set rd if src1 = src2 |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fcmpgt16 | $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ |
| fcmpgt32 | $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ |
| fcmple16 | $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ |
| fcmple32 | $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ |
| fcmpne16 | $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ |
| fcmpne32 | $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ |
| fcmpeq16 | $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ |
| fcmpeq32 | $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ |

*Description* Either four 16-bit or two 32-bit fixed-point values in the 64-bit floating-point source registers specified by rs1 and rs2 are compared. The 4-bit or 2-bit results are stored in the least significant bits in the integer destination register R[rd] as well as GSR.gcc. Whether GSR.gcc bits are implemented is implementation dependent (impl. dep. #303). Signed comparisons are used. Bit 0 of R[rd] corresponds to the least significant 16-bit or 32-bit comparison.

For FCMPGT, each bit in the result is set if the corresponding value in the first source operand is greater than the value in the second source operand. Less-than comparisons are made by swapping the operands.

For FCMPLE, each bit in the result is set if the corresponding value in the first source operand is less than or equal to the value in the second source operand. Greater-than-or-equal comparisons are made by swapping the operands.

For FCMPEQ, each bit in the result is set if the corresponding value in the first source operand is equal to the value in the second source operand.

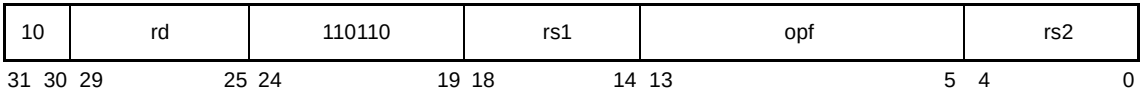
For FCMPNE, each bit in the result is set if the corresponding value in the first source operand is not equal to the value in the second source operand.

*Exceptions* *fp_disabled*

# A.46 Pixel Component Distance (PDIST) (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| PDIST | 0 0011 1110 | Distance between eight 8-bit components |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| pdist | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

*Description* Eight unsigned 8-bit values are contained in the 64-bit floating-point source registers specified by `rs1` and `rs2`. The corresponding 8-bit values in the source registers are subtracted (that is, the second source operand from the first source operand). The sum of the absolute value of each difference is added to the integer in the 64-bit floating-point destination register specified by `rd`. The result is stored in the destination register. Typically, this instruction is used for motion estimation in video compression algorithms.

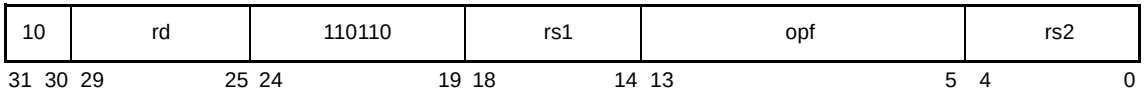
**Note –** For good performance, the `rd` operand of `PDIST` should not reference the result of a non-`PDIST` instruction in the five previously executed instruction groups.

*Exceptions* *fp_disabled*

# A.47 Pixel Formatting (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| FPACK16 | 0 0011 1011 | Four 16-bit packs into 8 unsigned bits |
| FPACK32 | 0 0011 1010 | Two 32-bit packs into 8 unsigned bit |
| FPACKFIX | 0 0011 1101 | Four 16-bit packs into 16 signed bits |
| FEXPAND | 0 0100 1101 | Four 16-bit expands |
| FPMERGE | 0 0100 1011 | Two 32-bit merges |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| fpack16 | $freg_{rs2}$, $freg_{rd}$ |
| fpack32 | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |
| fpackfix | $freg_{rs2}$, $freg_{rd}$ |
| fexpand | $freg_{rs2}$, $freg_{rd}$ |
| fpmerge | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ |

*Description*

The FPACK instructions convert multiple values in a source register to a lower-precision fixed or pixel format and stores the resulting values in the destination register. Input values are clipped to the dynamic range of the output format. Packing applies a scale factor from GSR.scale to allow flexible positioning of the binary point.

FEXPAND performs the inverse of the FPACK16 operation.

FPMERGE interleaves four 8-bit values from each of two 32-bit registers into a single 64-bit destination register.

*Exceptions*

*fp_disabled*

## A.47.1 FPACK16

FPACK16 takes four 16-bit fixed values from the 64-bit floating-point register specified by rs2, scales, truncates, and clips them into four 8-bit unsigned integers, and stores the results in the 32-bit destination register, f[rd]. FIGURE A-12 illustrates the FPACK16 operation.

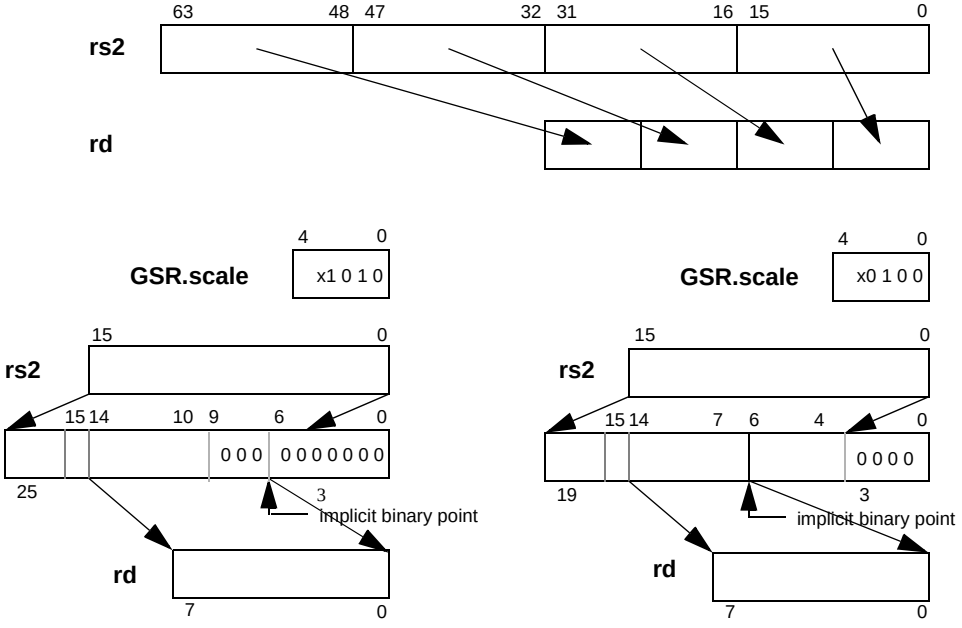



**FIGURE A-12** FPACK16 Operation

---

**Note –** FPACK16 ignores the most significant bit of GSR.scale (GSR.scale<4>).

---

This operation is carried out as follows:

1. Left-shift the value from the 64-bit floating-point register specified by rs2 by the number of bits specified in GSR.scale while maintaining clipping information.
2. Truncate and clip to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 7 and 6 for each 16-bit word). Truncation converts the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative (that is, its most significant bit is set), 0 is returned as the clipped value. If the value is greater than 255, then 255 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.
3. Store the result in the corresponding byte in the 32-bit destination register, f[rd].

## A.47.2 FPACK32

FPACK32 takes two 32-bit fixed values from the second source operand (the 64-bit floating-point register specified by rs2) and scales, truncates, and clips them into two 8-bit unsigned integers. The two 8-bit integers are merged at the corresponding least significant byte positions with each 32-bit word in the 64-bit floating-point register specified by rs1, left-shifted by 8 bits. The 64-bit result is stored in the 64-bit floating-point register specified by rd. Thus, successive FPACK32 instructions can assemble two pixels by using three or four pairs of 32-bit fixed values. FIGURE A-13 illustrates the FPACK32 operation.

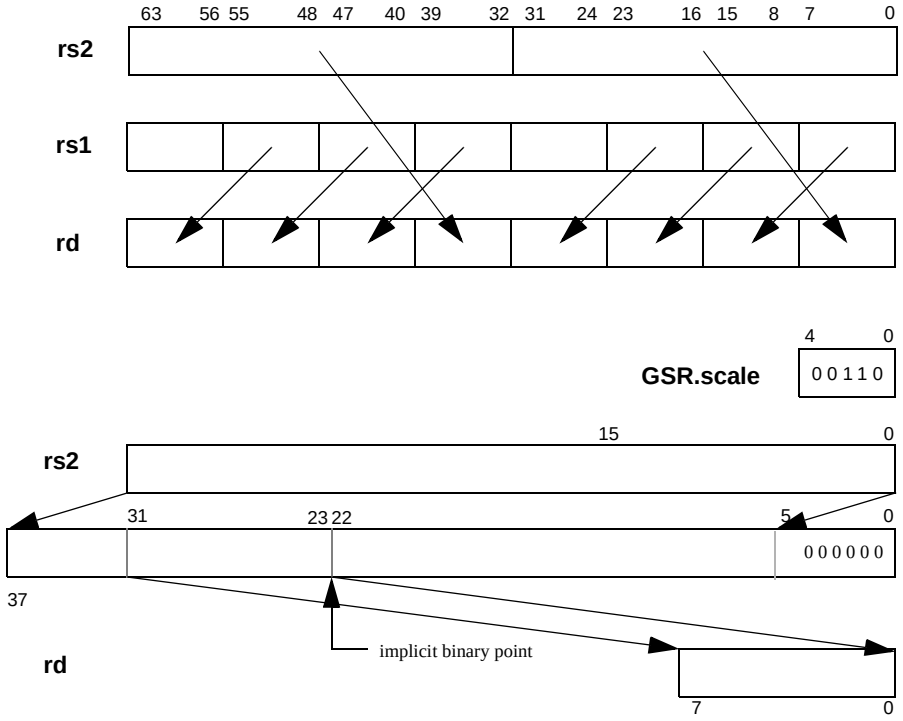



**FIGURE A-13** FPACK32 Operation

This operation is carried out as follows:

1. Left-shift each 32-bit value from the second source operand by the number of bits specified in GSR.scale, while maintaining clipping information.

2. For each 32-bit value, truncate and clip to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 23 and 22 for each 32-bit word). Truncation converts the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative

(that is, MSB is set), then 0 is returned as the clipped value. If the value is greater than 255, then 255 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.

3. Left-shift each 32-bit value from the first source operand (the 64-bit floating-point register specified by `rs1`) by 8 bits.
4. Merge the two clipped 8-bit unsigned values into the corresponding least significant byte positions in the left-shifted value from the second source operand.
5. Store the result in the `rd` register.

## A.47.3 FPACKFIX

`FPACKFIX` takes two 32-bit fixed values from the 64-bit floating-point register specified by `rs2`, scales, truncates, and clips them into two 16-bit unsigned integers, and then stores the result in the 32-bit destination register `f[rd]`. FIGURE A-14 illustrates the `FPACKFIX` operation.

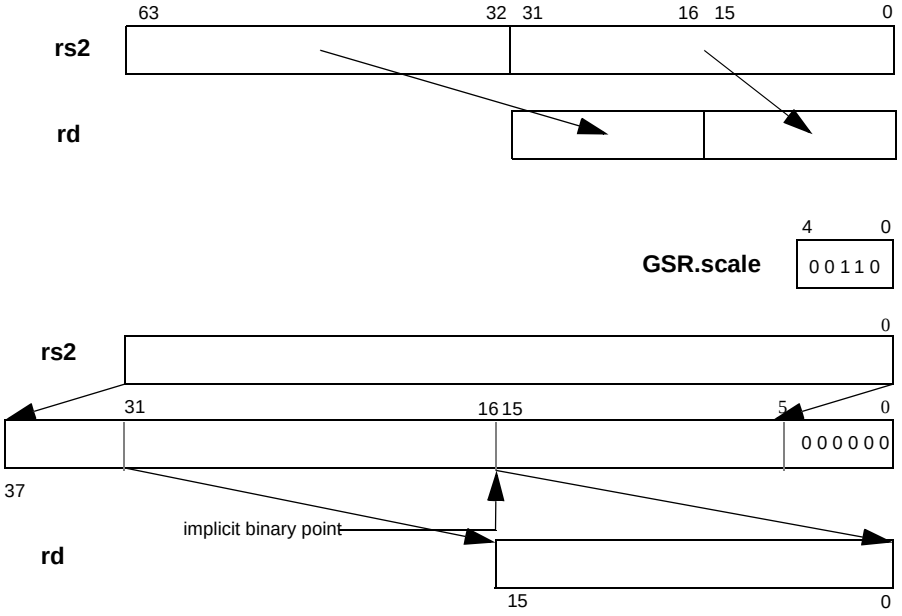



**FIGURE A-14** `FPACKFIX` Operation

This operation is carried out as follows:

1. Left-shift each 32-bit value from the source operand (the 64-bit floating-point register specified by `rs2`) by the number of bits specified in `GSR.scale` while maintaining clipping information.
2. For each 32-bit value, truncate and clip to a 16-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 16 and 15 for each 32-bit word). Truncation converts the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is less than –32768, then –32768 is returned as the clipped value. If the value is greater than 32767, then 32767 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.
3. Store the result in the 32-bit destination register `f[rd]`.

## A.47.4 FEXPAND

`FEXPAND` takes four 8-bit unsigned integers from `f[rs2]`, converts each integer to a 16-bit fixed-point value, and stores the four resulting 16-bit values in a 64-bit floating-point register specified by `rd`. FIGURE A-15 illustrates the operation.

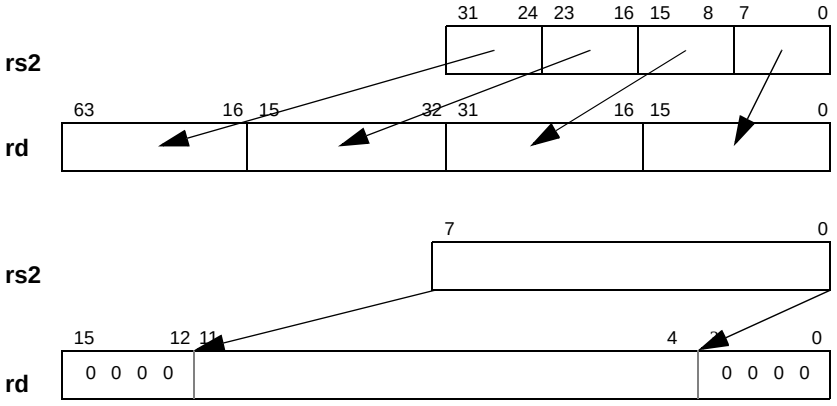



**FIGURE A-15** `FEXPAND` Operation

This operation is carried out as follows:

1. Left-shift each 8-bit value by 4 and zero-extend the results to a 16-bit fixed value.
2. Store the result in the destination register.

## A.47.5 FPMERGE

FPMERGE interleaves four corresponding 8-bit unsigned values in f[rs1] and f[rs2] to produce a 64-bit value in the 64-bit floating-point destination register specified by rd. This instruction converts from packed to planar representation when it is applied twice in succession; for example, R1G1B1A1, R3G3B3A3 → R1R3G1G3A1A3 → R1R2R3R4G1G2G3G4.

FPMERGE also converts from planar to packed when it is applied twice in succession; for example, R1R2R3R4, B1B2B3B4 → R1B1R2B2R3B3R4B4 → R1G1B1A1R2G2B2A2.

FIGURE A-16 illustrates the operation.

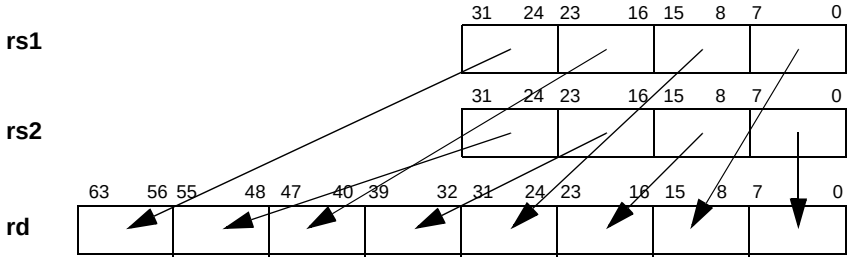
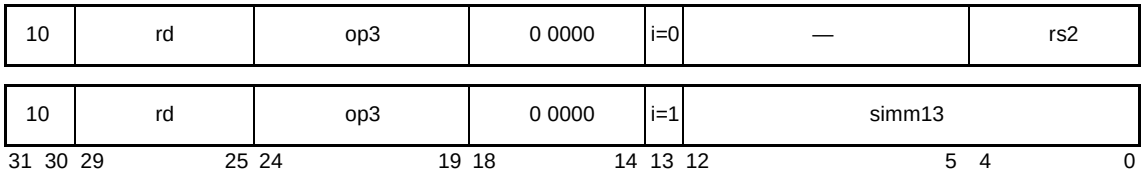



**FIGURE A-16** FPMERGE Operation

# A.48 Population Count

| Opcode | op3 | Operation |
|---|---|---|
| POPC | 10 1110 | Population Count |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| popc | *reg_or_imm*, $reg_{rd}$ |

*Description* POPC counts the number of one bits in R[rs2] if i = 0, or the number of one bits in sign_ext(simm13) if i = 1, and stores the count in R[rd]. This instruction does not modify the condition codes. **Note**: No JPS2 implementation implements this instruction in hardware; instead it generates an *illegal_instruction* exception. The instruction is emulated in supervisor software.

---

**V9 Compatibility Note –** Instruction bits 18 through 14 must be zero for POPC. Other encodings of this field (rs1) may be used in future versions of the SPARC architecture for other instructions.

---

**Programming Note –** POPC can be used to "find first bit set" in a register. A C program illustrating how POPC can be used for this purpose follows:

```
int ffs(zz)/* finds first 1 bit, counting from the LSB */
unsigned zz;
{
    return popc ( zz ^ (~ (-zz)));/* for nonzero zz */
}
```

Inline assembly language code for ffs() is

```
neg   %IN, %M_IN            ! -zz(2's complement)
xnor  %IN, %M_IN, %TEMP     ! ^ ~ -zz (exclusive nor)
popc  %TEMP,%RESULT         ! result  = popc(zz ^ ~ -zz)
movrz %IN,%g0,%RESULT       ! %RESULT should be 0 for %IN=0
```

where IN, M_IN, TEMP, and RESULT are integer registers.

---

*Example*

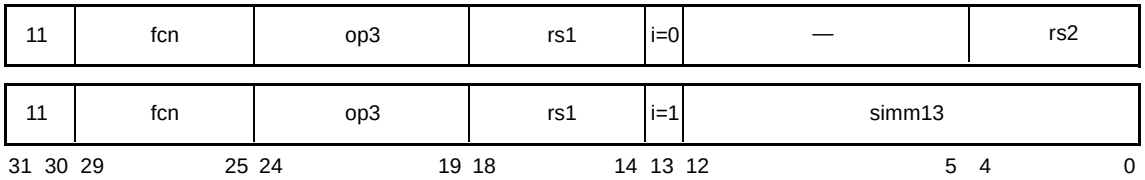
```
IN            = ...00101000! 1st 1 bit from rt is 4th bit
-IN           = ...11011000
~ -IN         = ...00100111
IN ^ ~ -IN    = ...00001111
popc(IN ^ ~ -IN) = 4
```

*Exceptions* *illegal_instruction*

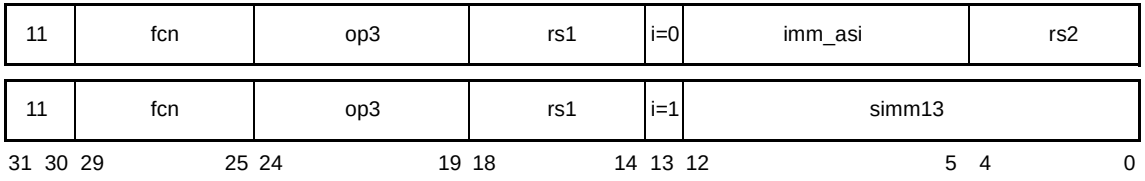
# A.49 Prefetch Data

| Opcode | op3 | Operation |
|---|---|---|
| PREFETCH | 10 1101 | Prefetch Data |
| PREFETCHA$^{P_{ASI}}$ | 11 1101 | Prefetch Data from Alternate Space |

*Format (3) PREFETCH*




*Format (3) PREFETCHA*




| fcn | SPARC JPS2 Prefetch Function |
|---|---|
| 0 | Prefetch for several reads |
| 1 | Prefetch for one read |
| 2 | Prefetch for several writes and possibly reads |
| 3 | Prefetch for one write |
| 4 | Prefetch page |
| 5–15 ($05_{16}$–$0F_{16}$) | Reserved |
| 16–19 ($10_{16}$–$13_{16}$) | Implementation dependent |
| 20 ($14_{16}$) | Strong Prefetch for several reads |
| 21 ($15_{16}$) | Strong Prefetch for one read |
| 22 ($16_{16}$) | Strong Prefetch for several writes and possibly reads |
| 23 ($17_{16}$) | Strong Prefetch for one write |
| 24-31 ($18_{16}$–$1F_{16}$) | Implementation dependent |

# Prefetch Data

| Assembly Language Syntax | |
|---|---|
| `prefetch` | [*address*], *prefetch_fcn* |
| `prefetcha` | [*regaddr*] *imm_asi*, *prefetch_fcn* |
| `prefetcha` | [*reg_plus_imm*] `%asi`, *prefetch_fcn* |

*Description*

In nonprivileged code, a prefetch instruction has the same observable effect as a NOP; its execution is nonblocking and cannot cause an observable trap. In particular, a prefetch instruction shall not trap if it is applied to an illegal or nonexistent memory address.

**IMPL. DEP. #103a:** Whether the execution of a `PREFETCH` instruction has observable effects in privileged code is implementation dependent.

**IMPL. DEP. #103b:** Whether the execution of a `PREFETCH` instruction can cause a *data_access_mmu_miss* exception is implementation dependent.

Whether `PREFETCH` always succeeds when the MMU is disabled is implementation dependent (impl. dep. #117).

---

**V9 Compatibility Note –** Any effects of prefetch in privileged code should be reasonable (for example, in handling ECC errors, no page prefetching is allowed within code that handles page faults). The benefits of prefetching should be available to most privileged code.

---

Execution of a prefetch instruction initiates data movement (or preparation for future data movement or address mapping) to reduce the latency of subsequent loads and stores to the specified address range.

A successful prefetch initiates movement of a block of data containing the addressed byte from memory toward the virtual processor. In SPARC JPS2, the block of data is one 64-byte cache line.

**IMPL. DEP. #103c:** The size and alignment in memory of the data block is implementation dependent; the minimum size is 64 bytes and the minimum alignment is a 64-byte boundary.

---

**Programming Note –** Software may prefetch 64 bytes beginning at an arbitrary address *address* by issuing the instructions

```
prefetch    [address], prefetch_fcn
prefetch    [address + 63], prefetch_fcn
```

---

**V9 Compatibility Note –** Prefetching may be used to help manage memory cache(s). A prefetch from a nonprefetchable location has no effect. It is up to memory management hardware to determine how locations are identified as not prefetchable.

Prefetch instructions that do *not* load from an alternate address space access the primary address space (`ASI_PRIMARY{_LITTLE}`). Prefetch instructions that *do* load from an alternate address space contain the address space identifier (ASI) to be used for the load in the `imm_asi` field if `i` = 0, or in the ASI register if `i` = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "`R[rs1]` + `R[rs2]`" if `i` = 0, or "`R[rs1]` + `sign_ext(simm13)`" if `i` = 1.

Variants of the prefetch instruction can be used to prepare the memory system for different types of accesses.

**IMPL. DEP. #103d:** An implementation may implement none, some, or all of these variants. A variant not implemented shall execute as a NOP. An implemented variant may support its full semantics or just the simple common-case prefetching semantics.

**IMPL. DEP. #103f:** Whether an attempt to reference a restricted ASI ($< 80_{16}$) by a `PREFETCHA` instruction while in nonprivileged mode (`PSTATE.priv` = 0) causes a *privileged_action* exception or executes as a NOP is implementation dependent.

## A.49.1 SPARC V9 Prefetch Variants

The prefetch variant is selected by the `fcn` field of the instruction. `fcn` values 5–15 are reserved for future extensions of the architecture, and `PREFETCH` `fcn` values of 16–19 and 24–31 are implementation dependent in SPARC JPS2.

Each prefetch variant reflects an intent on the part of the compiler or programmer. This is different from other instructions in SPARC V9 (except `BPN`), all of which specify specific actions. An implementation may implement a prefetch variant by any technique, as long as the intent of the variant is achieved.

The prefetch instruction is designed to treat the common cases as well. The variants are intended to provide scalability for future improvements in both hardware and compilers. If a variant is implemented, it should have the effects described below. In case some of the variants listed below are implemented and some are not, a recommended overloading of the unimplemented variants is provided in the SPARC V9 specification.

### A.49.1.1 Prefetch for Several Reads (fcn = 0)

The intent of this variant is to cause movement of data into the data cache nearest the virtual processor, with “reasonable” efforts made to obtain the data.

---

**Programming Note –** The intended use of this variant is in streaming relatively small amounts of data into the primary data cache of the virtual processor.

---

### A.49.1.2 Prefetch for One Read (fcn = 1)

The data to be read from the given address is expected to be read once and not reused (read or written) soon after that. Use of this PREFETCH variant indicates that, if possible, the data cache should be minimally disturbed by the data read from the given address.

---

**Programming Note –** The intended use of this variant is in streaming medium amounts of data into the virtual processor without disturbing the data in the primary data cache memory.

---

### A.49.1.3 Prefetch for Several Writes (and Possibly Reads) (fcn = 2)

The intent of this variant is to cause movement of data in preparation for writing.

---

**Programming Note –** An example use of this variant is to initialize a cache line in preparation for a partial write.

---

---

**V9 Compatibility Note –** On a multiprocessor system, this variant indicates that exclusive ownership of the addressed data is needed, so it may have the additional effect of obtaining exclusive ownership of the addressed cache line.

---

### A.49.1.4 Prefetch for One Write (fcn = 3)

This variant indicates that, if possible, the data cache should be minimally disturbed by the data written to this address, because those data are not expected to be reused (read or written) soon after they have been written once.

### A.49.1.5 Prefetch Page (fcn = 4)

In a virtual memory system, the intended action of this variant is for the supervisor software or hardware to initiate asynchronous mapping of the referenced virtual address, assuming that it is legal to do so.

---

**Programming Note –** The desire is to avoid a later page fault for the given address, or at least to shorten the latency of a page fault.

---

In a non-virtual-memory system or if the addressed page is already mapped, this variant has no effect.

The referenced page need not be mapped when the instruction completes. Loads and stores issued before the page is mapped should block just as they would if the prefetch had never been issued. When the activity associated with the mapping has completed, the loads and stores may proceed.

---

**JPS Compatibility Note –** An example of mapping activity is DMA from secondary storage.

Use of this variant may be disabled or restricted in privileged code that is not permitted to cause page faults.

SPARC JPS2 treats this variant as a NOP; no operation is performed.

---

## A.49.2 SPARC JPS2 Prefetch Variants (fcn = 20–23)

These values are available for implementations to use. An implementation shall treat any unimplemented prefetch fcn values as NOPs (impl. dep. #103).

### A.49.2.1 Strong Prefetch for Several Reads (fcn =20)

The intent of this variant is the same as for Prefetch for Several Reads (fcn = 0) except this variant may cause an exception if access causes a TLB miss.

### A.49.2.2 Strong Prefetch for One Read (fcn = 21)

The intent of this variant is the same as for Prefetch for One Read (fcn = 1) except this variant may cause an exception if this access causes a TLB miss.

### A.49.2.3 Strong Prefetch for Several Writes (fcn = 22)

The intent of this variant is the same as for Prefetch for Several Writes (fcn = 2) except this variant may cause an exception if this access causes TLB miss.

### A.49.2.4 Strong Prefetch for One Write (fcn = 23)

The intent of this variant is the same as for Prefetch for One Write (fcn = 3) except this variant may cause an exception if this access causes a TLB miss.

## A.49.3 Implementation-Dependent Prefetch Variants (fcn = 16–19, 24–31)

fcns 16-19 (impl. dep. #103e) and 24-31 are implementation dependent.

---

**JPS Compatibility Note –** It is desirable to avoid conflicting uses of the same prefetch function code on different implementation; the following is a list of function codes that are either implemented in JPS2 implementations or are expected to be used in future implementations, and their respective uses:

| fcn | Expected Use |
|---|---|
| 16 | Prefetch Invalidate |
| 17 | NOP |
| 18 | NOP |
| 19 | NOP |
| 20 | Strong Prefetch for read |
| 21 | Strong Prefetch for read |
| 22 | Strong Prefetch for write |
| 23 | Strong Prefetch for write |
| 24 | Invalidate Cache Entry |
| 25 | NOP |
| 26 | NOP |
| 27 | NOP |
| 28 | NOP |
| 29 | NOP |
| 30 | NOP |
| 31 | NOP |

---

Please refer to Implementation Supplements for details.

## A.49.4 General Comments

There is no variant of PREFETCH for instruction prefetching. Instruction prefetching should be encoded with the Branch Never (BPN) form of the BPcc instruction (see A.8, *Branch on Integer Condition Codes with Prediction (BPcc)*, on page 242).

One error to avoid in thinking about prefetch instructions is that they should have "no cost to execute." As long as the cost of executing a prefetch instruction is well less than one-third the cost of a cache miss, use of prefetching is a net win. It does not appear that prefetching causes a significant number of useless fetches from memory, though it may increase the rate of *useful* fetches (and hence the bandwidth), because it more efficiently overlaps computing with fetching.

---

**Programming Note –** A SPARC V9 compiler that generates PREFETCH instructions should generate each of the variants where it is most appropriate. The overloadings suggested in the previous Implementation Note ensure that such code will be portable and reasonably efficient across a range of hardware configurations.

---

---

**V9 Compatibility Note –** The Prefetch for One Read and Prefetch for One Write variants assume the existence of a "bypass cache," so that the bulk of the "real cache" remains undisturbed. If such a bypass cache is used, it should be large enough to properly shield the virtual processor from memory latency. Such a cache should probably be small, highly associative, and use a FIFO replacement policy.

---

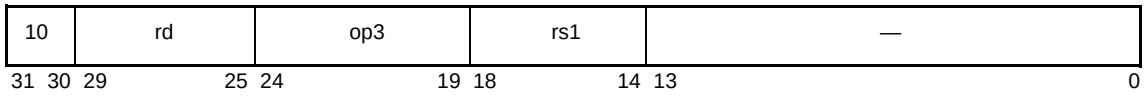
*Exceptions*

*illegal_instruction* (fcn = 5–15)
*fast_data_access_MMU_miss* (fcn = 20–23)
*privileged_action* (PREFETCHA with PSTATE.priv = 0 and ASI < $80_{16}$ (impl. dep. #103f))

# A.50 Read Privileged Register

| Opcode | op3 | Operation |
|---|---|---|
| RDPR[P] | 10 1010 | Read Privileged Register |

*Format (3)*




| rs1 | Privileged Register |
|---|---|
| 0 | TPC |
| 1 | TNPC |
| 2 | TSTATE |
| 3 | TT |
| 4 | TICK |
| 5 | TBA |
| 6 | PSTATE |
| 7 | TL |
| 8 | PIL |
| 9 | CWP |
| 10 | CANSAVE |
| 11 | CANRESTORE |
| 12 | CLEANWIN |
| 13 | OTHERWIN |
| 14 | WSTATE |
| 15 | FQ |
| 16–30 | — |
| 31 | VER |

| Assembly Language Syntax | |
|---|---|
| rdpr | %tpc, *reg$_{rd}$* |
| rdpr | %tnpc, *reg$_{rd}$* |
| rdpr | %tstate, *reg$_{rd}$* |
| rdpr | %tt, *reg$_{rd}$* |
| rdpr | %tick, *reg$_{rd}$* |
| rdpr | %tba, *reg$_{rd}$* |
| rdpr | %pstate, *reg$_{rd}$* |
| rdpr | %tl, *reg$_{rd}$* |
| rdpr | %pil, *reg$_{rd}$* |
| rdpr | %cwp, *reg$_{rd}$* |
| rdpr | %cansave, *reg$_{rd}$* |
| rdpr | %canrestore, *reg$_{rd}$* |
| rdpr | %cleanwin, *reg$_{rd}$* |
| rdpr | %otherwin, *reg$_{rd}$* |
| rdpr | %wstate, *reg$_{rd}$* |
| rdpr | %fq, *reg$_{rd}$* |
| rdpr | %ver, *reg$_{rd}$* |

*Description*

The rs1 field in the instruction determines the privileged register that is read. There are MAXTL copies of the TPC, TNPC, TT, and TSTATE registers. A read from one of these registers returns the value in the register indexed by the current value in the trap level register (TL). A read of TPC, TNPC, TT, or TSTATE when the trap level is zero (TL = 0) causes an *illegal_instruction* exception.

RDPR instructions with rs1 in the range 16–30 are reserved; executing an RDPR instruction with rs1 in that range causes an *illegal_instruction* exception.

**Programming Note –** On an implementation with precise floating-point traps, the address of a trapping instruction will be in the TPC[TL] register when the trap code begins execution. On an implementation with deferred floating-point traps, the address of the trapping instruction might be a value obtained from the FQ.

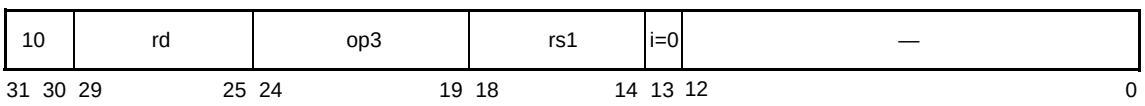
*Exceptions*

*privileged_opcode*
*illegal_instruction* ((rs1 = 16–30) or ((rs1 ≤ 3) and (TL = 0)))

# A.51 Read State Register

| Opcode | op3 | rs1 | Operation |
|---|---|---|---|
| RDY[D] | 10 1000 | 0 | Read Y Register; deprecated (see A.71.9) |
| — | 10 1000 | 1 | *Reserved* |
| RDCCR | 10 1000 | 2 | Read Condition Codes Register |
| RDASI | 10 1000 | 3 | Read ASI Register |
| RDTICK[P_NPT] | 10 1000 | 4 | Read Tick Register |
| RDPC | 10 1000 | 5 | Read Program Counter |
| RDFPRS | 10 1000 | 6 | Read Floating-Point Registers Status Register |
| — | 10 1000 | 7–14 | *Reserved* |
| *See text* | 10 1000 | 15 | STBAR, MEMBAR, or *Reserved*; see text |
| RDASR | 10 1000 | 16-31 | Read non-SPARC V9 ASRs |
| RDPCR[P_PCR] | | 16 | Read Performance Control Registers (PCR) |
| RDPIC[P_PIC] | | 17 | Read Performance Instrumentation Counters (PIC) |
| RDDCR[P] | | 18 | Read Dispatch Control Register (DCR) |
| RDGSR | | 19 | Read Graphic Status Register (GSR) |
| — | | 20–21 | *Implementation dependent (impl. dep. #8, 9)* |
| RDSOFTINT[P] | | 22 | Read per-virtual processor Soft Interrupt Register |
| RDTICK_CMPR[P] | | 23 | Read Tick Compare Register |
| RDSTICK[P_NPT] | | 24 | Read System TICK Register |
| RDSTICK_CMPR[P] | | 25 | Read System TICK Compare Register |
| — | | 26-31 | *Implementation dependent (impl. dep. #8, 9)* |

*Format (3)*



| Assembly Language Syntax | |
|---|---|
| rd | %ccr, *reg$_{rd}$* |
| rd | %asi, *reg$_{rd}$* |
| rd | %tick, *reg$_{rd}$* |
| rd | %pc, *reg$_{rd}$* |
| rd | %fprs, *reg$_{rd}$* |
| rd | %pcr, *reg$_{rd}$* |
| rd | %pic, *reg$_{rd}$* |
| rd | %dcr, *reg$_{rd}$* |
| rd | %gsr, *reg$_{rd}$* |
| rd | %softint, *reg$_{rd}$* |
| rd | %tick_cmpr, *reg$_{rd}$* |
| rd | %sys_tick, *reg$_{rd}$* |
| rd | %sys_tick_cmpr, *reg$_{rd}$* |

*Description*

These instructions read the state register specified by rs1 into R[rd].

Values 7–14 of rs1 are reserved for future versions of the architecture. A Read State Register instruction with rs1 = 15, rd = 0, and i = 0 is defined to be a (deprecated) STBAR instruction (see A.71.10, *Store Barrier*, on page 403).

An instruction using the RDASR opcode with rs1 = 15, rd = 0, and i = 1 is defined to be a MEMBAR instruction (see page 293). RDASR with rs1 = 15 and rd ≠ 0 is reserved for future versions of the architecture; it causes an *illegal_instruction* exception.

The RDASR opcode with rs1 ≠ 15 and i = 1 is reserved for future versions of the architecture; it causes an *illegal_instruction* exception.

For RDPC, the high-order 32 bits of the PC value stored in R[rd] are implementation dependent when PSTATE.am = 1 (impl. dep. #125).

RDFPRS waits for all pending FPops and loads of floating-point registers to complete before reading the FPRS register.

RDGSR causes an *fp_disabled* exception if PSTATE.pef = 0 or FPRS.fef = 0.

RDTICK causes a *privileged_action* exception if PSTATE.priv = 0 and TICK.npt = 1. RDSTICK causes a *privileged_action* exception if PSTATE.priv = 0 and STICK.npt = 1.

RDPIC causes a *privileged_action* exception if PSTATE.priv = 0 and PCR.priv = 1.

RDPCR causes an exception due to access privilege violation under implementation-dependent circumstances (impl. dep. #250).

---

**Note –** See Section I.1, *Read/Write Ancillary State Registers (ASRs)*, for a discussion of extending the SPARC V9 instruction set using read / write ASR instructions.

---

**V9 Compatibility Note –** Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers. See *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, a document available from SPARC International, for information on implemented ancillary state registers.

---

**V9 Compatibility Note –** The SPARC V8 RDPSR, RDWIM, and RDTBR instructions do not exist in SPARC V9 since the PSR, WIM, and TBR registers do not exist in SPARC V9.

---

*Exceptions*

*privileged_opcode*(RDDCR, RDSOFTINT, RDTICK_CMPR, RDSTICK, RDSTICK_CMPR, and RDPCR (impl. dep. #250))

*illegal_instruction*(RDASR with rs1 = 1 or 7–14;
RDASR with rs1 = 15 and rd ≠ 0;
RDASR with rs1 ≠ 15 and i = 1;
RDASR with rs1 = 20–21, 26–31)

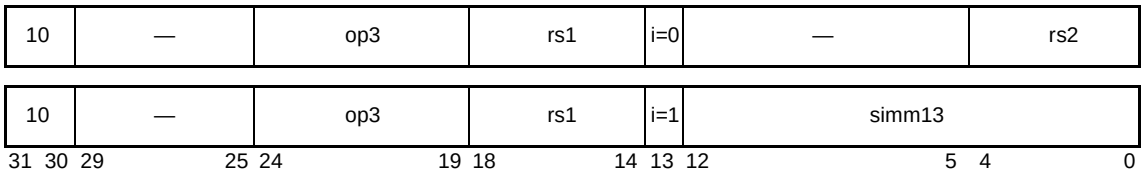
*privileged_action* (RDTICK with PSTATE.priv = 0 and TICK.npt = 1;
RDPIC with PSTATE.priv = 0 and PCR.priv = 1;
RDSTICK with PSTATE.priv = 0 and STICK.npt = 1;
RDPCR (impl. dep. #250))

*fp_disabled* (RDGSR with PSTATE.pef = 0 or FPRS.fef = 0)

# A.52 RETURN

| Opcode | op3 | Operation |
|---|---|---|
| RETURN | 11 1001 | Return |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| `return` | *address* |

*Description*

The `RETURN` instruction causes a delayed transfer of control to the target address and has the window semantics of a `RESTORE` instruction; that is, it restores the register window prior to the last `SAVE` instruction. The target address is "`R[rs1] + R[rs2]`" if `i` = 0, or "`R[rs1] + sign_ext(simm13)`" if `i` = 1. Registers `R[rs1]` and `R[rs2]` come from the *old* window.

The `RETURN` instruction may cause an exception. It may cause a *window_fill* exception as part of its `RESTORE` semantics, or it may cause a *mem_address_not_aligned* exception if either of the two low-order bits of the target address is nonzero.

**Programming Note –** To reexecute the trapped instruction when returning from a user trap handler, use the `RETURN` instruction in the delay slot of a `JMPL` instruction, for example:

```
jmpl    %l6,%g0   | Trapped PC supplied to user trap handler
return  %l7       | Trapped nPC supplied to user trap handler
```

**Programming Note –** A routine that uses a register window may be structured either as

```
  save      %sp,-framesize, %sp
  . . .
  ret                 | Same as jmpl %i7+8, %g0
  restore             | Something useful like "restore
                      | %o2,%l2,%o0"
```

or as

```
  save      %sp,-framesize, %sp
. . .
  return    %i7+8
  nop                 | Could do some useful work in the caller's
                      | window, e.g., "or %o1, %o2,%o0"
```

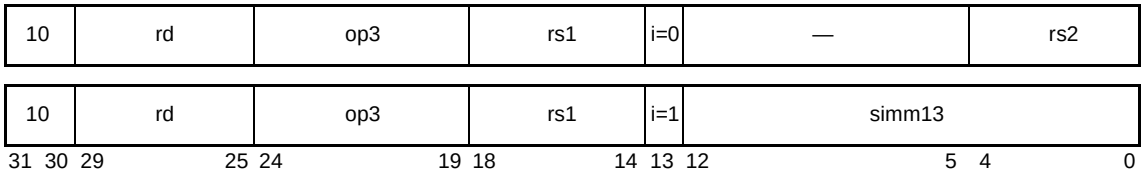
*Exceptions*

*mem_address_not_aligned*
*fill_n_normal* ($n$ = 0–7)
*fill_n_other* ($n$ = 0–7)

# A.53 SAVE and RESTORE

| Opcode | op3 | Operation |
|---|---|---|
| SAVE | 11 1100 | Save Caller's Window |
| RESTORE | 11 1101 | Restore Caller's Window |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| save | $reg_{rs1}$, *reg_or_imm*, $reg_{rd}$ |
| restore | $reg_{rs1}$, *reg_or_imm*, $reg_{rd}$ |

*Description (Effect on Nonprivileged State)*

The SAVE instruction provides the routine executing it with a new register window. The *out* registers from the old window become the *in* registers of the new window. The contents of the out and the local registers in the new window are zero or contain values from the executing process; that is, the process sees a clean window.

The RESTORE instruction restores the register window saved by the last SAVE instruction executed by the current process. The in registers of the old window become the out registers of the new window. The in and local registers in the new window contain the previous values.

Furthermore, if and only if a spill or fill trap is not generated, SAVE and RESTORE behave like normal ADD instructions, except that the source operands R[rs1] or R[rs2] are read from the *old* window (that is, the window addressed by the original CWP) and the sum is written into R[rd] of the *new* window (that is, the window addressed by the new CWP).

**Note**: CWP arithmetic is performed modulo the number of implemented windows, NWINDOWS.

**Programming Notes –** Typically, if a SAVE (RESTORE) instruction traps, the spill (fill) trap handler returns to the trapped instruction to reexecute it. So, although the ADD operation is not performed the first time (when the instruction traps), it is performed the second time the instruction executes. The same applies to changing the CWP.

The SAVE instruction can be used to atomically allocate a new window in the register file and a new software stack frame in memory. See H.1.2, *Leaf-Procedure Optimization*, for details.

There is a performance trade-off to consider between using SAVE/RESTORE and saving and restoring selected registers explicitly.

### *Description (Effect on Privileged State)*

If the SAVE instruction does not trap, it increments the CWP (**mod** NWINDOWS) to provide a new register window and updates the state of the register windows by decrementing CANSAVE and incrementing CANRESTORE.

If the new register window is occupied (that is, CANSAVE = 0), a spill trap is generated. The trap vector for the spill trap is based on the value of OTHERWIN and WSTATE. The spill trap handler is invoked with the CWP set to point to the window to be spilled (that is, old CWP + 2).

If CANSAVE ≠ 0, the SAVE instruction checks whether the new window needs to be cleaned. It causes a *clean_window* trap if the number of unused clean windows is zero, that is, (CLEANWIN – CANRESTORE) = 0. The *clean_window* trap handler is invoked with the CWP set to point to the window to be cleaned (that is, old CWP + 1).

If the RESTORE instruction does not trap, it decrements the CWP (**mod** NWINDOWS) to restore the register window that was in use prior to the last SAVE instruction executed by the current process. It also updates the state of the register windows by decrementing CANRESTORE and incrementing CANSAVE.

If the register window to be restored has been spilled (CANRESTORE = 0), then a fill trap is generated. The trap vector for the fill trap is based on the values of OTHERWIN and WSTATE, as described in *Trap Type for Spill/Fill Traps* on page 145. The fill trap handler is invoked with CWP set to point to the window to be filled, that is, old CWP – 1.

**Programming Note –** The vectoring of spill and fill traps can be controlled by setting the value of the OTHERWIN and WSTATE registers appropriately. For details, see H.2.3.1, *Splitting the Register Windows*, on page 535.

The spill (fill) handler normally will end with a SAVED (RESTORED) instruction followed by a RETRY instruction.

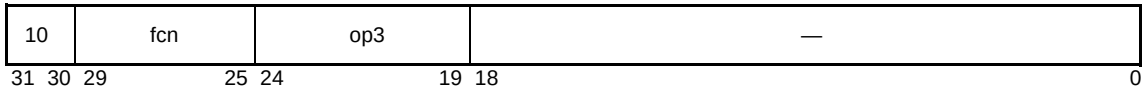
*Exceptions*

*clean_window* (SAVE only)
*fill_n_normal* (RESTORE only, $n = 0–7$)
*fill_n_other* (RESTORE only, $n = 0–7$)
*spill_n_normal* (SAVE only, $n = 0–7$)
*spill_n_other* (SAVE only, $n = 0–7$)

# A.54 SAVED and RESTORED

| Opcode | op3 | fcn | Operation |
|---|---|---|---|
| SAVED[P] | 11 0001 | 0 | Window has been saved |
| RESTORED[P] | 11 0001 | 1 | Window has been restored |
| — | 11 0001 | 2–31 | *Reserved* |

*Format (3)*




| Assembly Language Syntax |
|---|
| `saved` |
| `restored` |

*Description*

SAVED and RESTORED adjust the state of the register-windows control registers.

SAVED increments CANSAVE. If OTHERWIN = 0, SAVED decrements CANRESTORE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

RESTORED increments CANRESTORE. If CLEANWIN < (NWINDOWS–1), then RESTORED increments CLEANWIN. If OTHERWIN = 0, it decrements CANSAVE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

**Programming Notes –** The spill (fill) handlers use the SAVED (RESTORED) instruction to indicate that a window has been spilled (filled) successfully. See H.2.2, *Example Code for Spill Handler*, for details.

Normal privileged software would probably not do a SAVED or RESTORED from trap level zero (TL = 0). However, it is not illegal to do so and doing so does not cause a trap.

Executing a SAVED (RESTORED) instruction outside of a window spill (fill) trap handler is likely to create an inconsistent window state. Hardware will not signal an exception, however, since maintaining a consistent window state is the responsibility of privileged software.

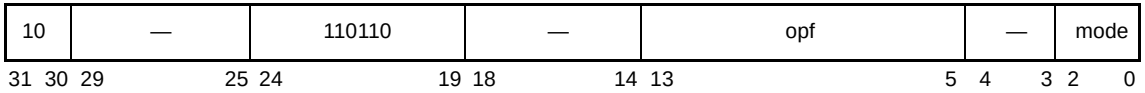
*Exceptions*

*privileged_opcode*
*illegal_instruction* (fcn = 2–31)

# A.55 Set Interval Arithmetic Mode (VIS II)

| Opcode | opf | Operation |
|---|---|---|
| SIAM | 0 1000 0001 | Set the interval arithmetic mode fields in the GSR |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| siam | *mode* |

*Description* The SIAM instruction sets the GSR.im and GSR.irnd fields as follows:

GSR.im = mode<2>

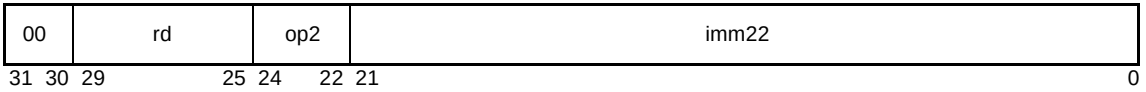
GSR.irnd = mode<1:0>

*Exceptions* *fp_disabled*

# A.56 SETHI

| Opcode | op2 | Operation |
|---|---|---|
| SETHI | 100 | Set High 22 Bits of Low Word |

*Format (2)*




| Assembly Language Syntax | |
|---|---|
| sethi | *const22*, *reg*$_{rd}$ |
| sethi | %hi (*value*), *reg*$_{rd}$ |

*Description*

SETHI zeroes the least significant 10 bits and the most significant 32 bits of R[rd] and replaces bits 31 through 10 of R[rd] with the value from its imm22 field.

SETHI does not affect the condition codes.

Some SETHI instructions with rd = 0 have special uses:

- rd = 0 and imm22 = 0: defined to be a NOP instruction (described in A.41)
- rd = 0 and imm22 ≠ 0 may be used to trigger hardware performance counters in some JPS2 implementations (for details, see Appendix Q in each JPS2 Extensions document).

**Programming Note –** The most common form of 64-bit constant generation is creating stack offsets whose magnitude is less than $2^{32}$. The code below can be used to create the constant 0000 0000 ABCD $1234_{16}$:

```
sethi   %hi(0xabcd1234),%o0
or      %o0, 0x234, %o0
```

The following code shows how to create a negative constant. **Note**: The immediate field of the xor instruction is sign extended and can be used to get 1's in all of the upper 32 bits. For example, to set the negative constant FFFF FFFF ABCD $1234_{16}$:

```
sethi   %hi(0x5432edcb),%o0  ! note 0x5432EDCB, not 0xABCD1234
xor     %o0, 0x1e34, %o0     ! part of imm. overlaps upper bits
```

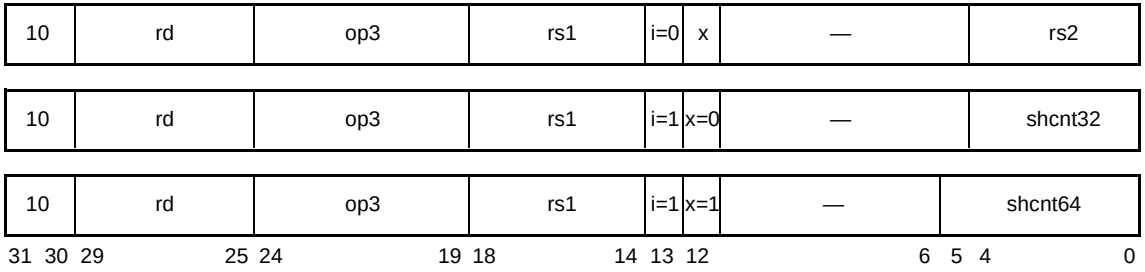
*Exceptions*

None

# A.57 Shift

| Opcode | op3 | x | Operation |
|---|---|---|---|
| SLL | 10 0101 | 0 | Shift Left Logical – 32 bits |
| SRL | 10 0110 | 0 | Shift Right Logical – 32 bits |
| SRA | 10 0111 | 0 | Shift Right Arithmetic– 32 bits |
| SLLX | 10 0101 | 1 | Shift Left Logical – 64 bits |
| SRLX | 10 0110 | 1 | Shift Right Logical – 64 bits |
| SRAX | 10 0111 | 1 | Shift Right Arithmetic – 64 bits |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| sll | *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ |
| srl | *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ |
| sra | *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ |
| sllx | *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ |
| srlx | *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ |
| srax | *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ |

*Description*

When `i` = 0 and `x` = 0, the shift count is the least significant five bits of `R[rs2]`.
When `i` = 0 and `x` = 1, the shift count is the least significant six bits of `R[rs2]`.
When `i` = 1 and `x` = 0, the shift count is the immediate value specified in bits 0 through 4 of the instruction.
When `i` = 1 and `x` = 1, the shift count is the immediate value specified in bits 0 through 5 of the instruction.

TABLE A-8 shows the shift count encodings for all values of `i` and `x`.

**TABLE A-8** Shift Count Encodings

| i | x | Shift Count |
|---|---|---|
| 0 | 0 | bits 4–0 of `R[rs2]` |
| 0 | 1 | bits 5–0 of `R[rs2]` |
| 1 | 0 | bits 4–0 of instruction |
| 1 | 1 | bits 5–0 of instruction |

`SLL` and `SLLX` shift all 64 bits of the value in `R[rs1]` left by the number of bits specified by the shift count, replacing the vacated positions with zeroes, and write the shifted result to `R[rd]`.

`SRL` shifts the low 32 bits of the value in `R[rs1]` right by the number of bits specified by the shift count. Zeroes are shifted into bit 31. The upper 32 bits are set to zero, and the result is written to `R[rd]`.

`SRLX` shifts all 64 bits of the value in `R[rs1]` right by the number of bits specified by the shift count. Zeroes are shifted into the vacated high-order bit positions, and the shifted result is written to `R[rd]`.

`SRA` shifts the low 32 bits of the value in `R[rs1]` right by the number of bits specified by the shift count and replaces the vacated positions with bit 31 of `R[rs1]`. The high-order 32 bits of the result are all set with bit 31 of `R[rs1]`, and the result is written to `R[rd]`.

`SRAX` shifts all 64 bits of the value in `R[rs1]` right by the number of bits specified by the shift count and replaces the vacated positions with bit 63 of `R[rs1]`. The shifted result is written to `R[rd]`.

No shift occurs when the shift count is 0, but the high-order bits are affected by the 32-bit shifts as noted above.

These instructions do not modify the condition codes.

---

**Programming Notes –** "Arithmetic left shift by 1 (and calculate overflow)" can be effected with the `ADDcc` instruction.

The instruction "`sra rs1,0,rd`" can be used to convert a 32-bit value to 64 bits, with sign extension into the upper word. "`srl rs1,0,rd`" can be used to clear the upper 32 bits of `R[rd]`.

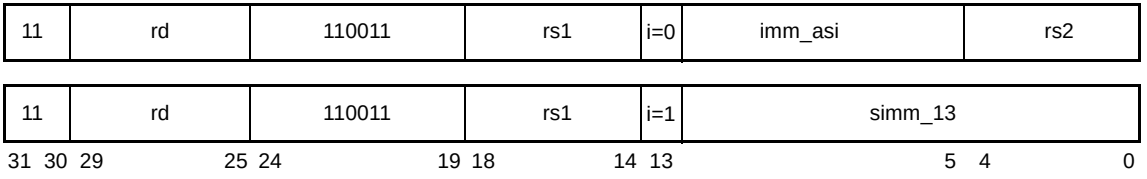
---

*Exceptions* None

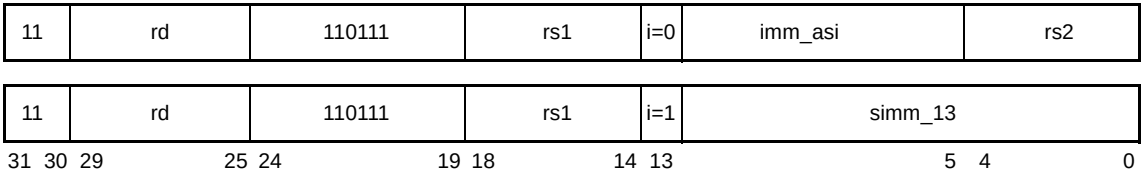
# A.58 Short Floating-Point Load and Store (VIS I)

| Opcode | imm_asi | ASI Value | Operation |
|---|---|---|---|
| LDDFA<br>STDFA | ASI_FL8_P | $D0_{16}$ | 8-bit load / store from / to primary address space |
| LDDFA<br>STDFA | ASI_FL8_S | $D1_{16}$ | 8-bit load / store from / to secondary address space |
| LDDFA<br>STDFA | ASI_FL8_PL | $D8_{16}$ | 8-bit load / store from / to primary address space, little-endian |
| LDDFA<br>STDFA | ASI_FL8_SL | $D9_{16}$ | 8-bit load / store from / to secondary address space, little-endian |
| LDDFA<br>STDFA | ASI_FL16_P | $D2_{16}$ | 16-bit load / store from / to primary address space |
| LDDFA<br>STDFA | ASI_FL16_S | $D3_{16}$ | 16-bit load / store from / to secondary address space |
| LDDFA<br>STDFA | ASI_FL16_PL | $DA_{16}$ | 16-bit load / store from / to primary address space, little-endian |
| LDDFA<br>STDFA | ASI_FL16_SL | $DB_{16}$ | 16-bit load / store from / to secondary address space, little-endian |

*Format (3) LDDFA*




*Format (3) STDFA*



| Assembly Language Syntax | |
|---|---|
| `ldda` | [*reg_addr*] *imm_asi*, $freg_{rd}$ |
| `ldda` | [*reg_plus_imm*] `%asi`, $freg_{rd}$ |
| `stda` | $freg_{rd}$, [*reg_addr*] *imm_asi* |
| `stda` | $freg_{rd}$, [*reg_plus_imm*] `%asi` |

*Description*

Short floating-point load and store instructions are selected by means of one of the short ASIs with the `LDDFA` and `STDFA` instructions.

These ASIs allow 8- and 16-bit loads or stores to be performed to/from the floating-point registers. Eight-bit loads can be performed to arbitrary byte addresses. For 16-bit loads, the least significant bit of the address must be 0 or a *mem_address_not_aligned* trap is taken. Short loads are zero-extended to the full floating-point register. Short stores access the low-order 8 or 16 bits of the register.

Little-endian ASIs transfer data in little-endian format in memory; otherwise, memory is assumed to be big-endian. Short loads and stores are typically used with the `FALIGNDATA` instruction (see *Alignment Instructions (VIS I)* on page 226) to assemble or store 64 bits on noncontiguous components.

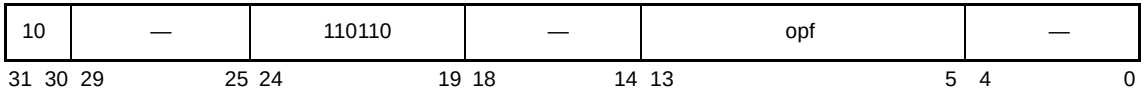
*Exceptions*

*fp_disabled*
*PA_watchpoint*
*VA_watchpoint*
*mem_address_not_aligned* (odd memory address for a 16-bit load or store)
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection* (Short Floating-Point Store only)

# A.59 SHUTDOWN (VIS I)

| Opcode | opf | Operation |
|---|---|---|
| SHUTDOWN[P] | 0 1000 0000 | Shut down to enter power-down mode |

*Format (3)*




| Assembly Language Syntax |
|---|
| shutdown |

*Description*

SHUTDOWN is a privileged instruction that may be used to bring the virtual processor or its containing system into a low-power state in an orderly manner. It has no effect on software-visible virtual processor state.

The SHUTDOWN instruction waits for all outstanding transactions to be completed, thereby leaving the caches and other internal registers in a clean state. It then enters a mode in which the virtual processor consumes substantially less power.

The SHUTDOWN instruction is intended to enter a low power mode (such as Energy Star).

Because SHUTDOWN is a privileged instruction, an attempt to execute it while in nonprivileged mode causes a *privileged_opcode* trap.

**IMPL. DEP. #206:** It is implementation dependent whether SHUTDOWN functions as described above or whether in nonprivileged mode it acts as a NOP in a given implementation.

**Note –** In privileged mode, SHUTDOWN acts as NOP on SPARC JPS2 implementations.

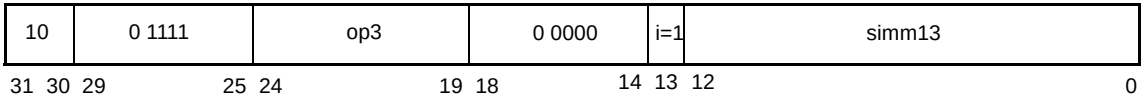
*Exceptions*

*privileged_opcode*

# A.60 Software-Initiated Reset

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| SIR | 11 0000 | 15 | Software-Initiated Reset |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| sir | *simm13* |

*Description*

On SPARC V9 systems, SIR is used to generate a software-initiated reset (SIR). As with other traps, a software-initiated reset performs different actions when TL = MAXTL than it does when TL < MAXTL.

See *Software-Initiated Reset (SIR) Traps* on page 158 for more information about software-initiated resets.

When executed in nonprivileged mode, SIR acts with no visible effect (as a NOP) (impl. dep. #116).

*Exceptions*

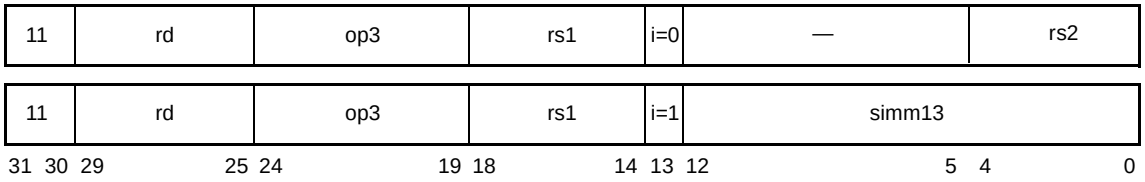
*software_initiated_reset*

# A.61 Store Floating-Point

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| STF | 10 0100 | 0–31 | Store Floating-Point Register |
| STDF | 10 0111 | † | Store Double Floating-Point Register |
| STQF | 10 0110 | † | Store Quad Floating-Point Register |
| STXFSR | 10 0101 | 1 | Store Floating-Point State Register |
| — | 10 0101 | 2–31 | *Reserved* |

† Encoded floating-point register value, as described on page 51.

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| st | *freg*$_{rd}$, [*address*] |
| std | *freg*$_{rd}$, [*address*] |
| stq | *freg*$_{rd}$, [*address*] |
| stx | %fsr, [*address*] |

*Description*

The store single floating-point instruction (STF) copies f[rd] into memory.

The store double floating-point instruction (STDF) copies a doubleword from a double floating-point register into a word-aligned doubleword in memory.

The store quad floating-point instruction (STQF) copies the contents of a quad floating-point register into a word-aligned quadword in memory.

The store floating-point state register instruction (STXFSR) waits for any currently executing FPop instructions to complete, and then it writes all 64 bits of the FSR into memory.

STXFSR zeroes FSR.ftt after writing the FSR to memory.

**V9 Compatibility Note –** FSR.ftt should not be zeroed until it is known that the store will not cause a precise trap.

The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + sign_ext(simm13)" if i = 1.

STF causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned. STXFSR causes a *mem_address_not_aligned* exception if the address is not doubleword aligned. If the floating-point unit is not enabled for the source register rd (per FPRS.fef and PSTATE.pef) or if the FPU is not present, then a store floating-point instruction causes an *fp_disabled* exception.

**IMPL. DEP. #110a:** STDF requires only word alignment in memory. If the effective address is word aligned but not doubleword aligned, it may cause an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the STDF instruction and return.

In a JPS2 implementation, STQF always causes an *fp_exception_other* trap (with FSR.ftt = unimplemented_FPop) and trap handler software emulates the STQF. A JPS2 implementation never generates an *STQF_mem_address_not_aligned* exception (impl. dep. #112a).

**V9 Compatibility Note –** A floating-point operation that is not implemented in hardware shall generate an *fp_exception_other* exception with ftt = unimplemented_FPop when executed. Other instructions not implemented in hardware shall generate an *illegal_instruction* exception and therefore will not generate any of the other exceptions listed.

**Programming Note –** In SPARC V8, some compilers issued sets of single-precision stores when they could not determine that double- or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned stores is expected to be fast, it is recommended that compilers issue sets of single-precision stores only when they can determine that double- or quadword operands are *not* properly aligned.

*Exceptions*

*illegal_instruction* (op3 = $25_{16}$ and rd = 2–31)
*fp_disabled*
*mem_address_not_aligned*
*STDF_mem_address_not_aligned* (STDF only)
*STQF_mem_address_not_aligned* (STQF only) (not used in JPS2)
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*

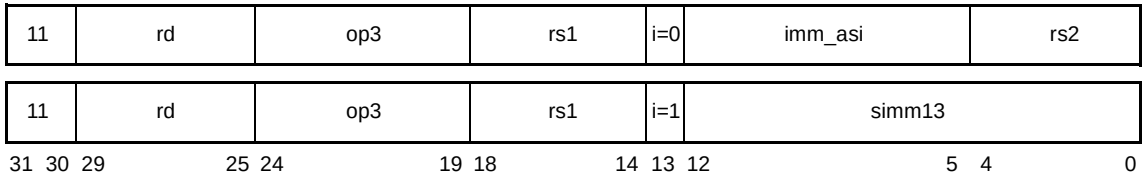
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.62 Store Floating-Point into Alternate Space

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| STFA$^{P_{ASI}}$ | 11 0100 | 0–31 | Store Floating-Point Register to Alternate Space |
| STDFA$^{P_{ASI}}$ | 11 0111 | † | Store Double Floating-Point Register to Alternate Space |
| STQFA$^{P_{ASI}}$ | 11 0110 | † | Store Quad Floating-Point Register to Alternate Space |

† Encoded floating-point register value, as described on page 51.

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| sta | *freg$_{rd}$, [regaddr] imm_asi* |
| sta | *freg$_{rd}$, [reg_plus_imm]* %asi |
| stda | *freg$_{rd}$, [regaddr] imm_asi* |
| stda | *freg$_{rd}$, [reg_plus_imm]* %asi |
| stqa | *freg$_{rd}$, [regaddr] imm_asi* |
| stqa | *freg$_{rd}$, [reg_plus_imm]* %asi |

*Description*

The store single floating-point into alternate space instruction (STFA) copies `f[rd]` into memory.

The store double floating-point into alternate space instruction (STDFA) copies a doubleword from a double floating-point register into a word-aligned doubleword in memory.

The store quad floating-point into alternate space instruction (STQFA) copies the contents of a quad floating-point register into a word-aligned quadword in memory.

Store floating-point into alternate space instructions contain the address space identifier (ASI) to be used for the load in the `imm_asi` field if `i` = 0 or in the ASI register if `i` = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "`R[rs1]` + `R[rs2]`" if `i` = 0, or "`R[rs1]` + `sign_ext(simm13)`" if `i` = 1.

`STFA` causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned. If the floating-point unit is not enabled for the source register `rd` (per `FPRS.fef` and `PSTATE.pef`) or if the FPU is not present, store floating-point into alternate space instructions cause an *fp_disabled* exception.

---

**V9 Compatibility Note –** This check is not made for `STQFA`. `STFA` and `STDFA` cause a *privileged_action* exception if `PSTATE.priv` = 0 and bit 7 of the ASI is 0.

---

`STDFA` with certain target ASIs is defined to be a 64-byte block-store instruction. See *Block Load and Store (VIS I)* on page 231 for details.

`STDFA` with certain target ASIs is defined to be a Partial Store instruction. See *Partial Store (VIS I)* on page 313 for details.

`STDFA` with certain target ASIs is defined to be a Short Floating-point Store instruction. See *Short Floating-Point Load and Store (VIS I)* on page 356 for details.

**IMPL. DEP. #110b:** `STDFA` requires only word alignment in memory. If the effective address is word aligned but not doubleword aligned, it may cause an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the `STDFA` instruction and return.

In a JPS2 implementation, `STQFA` always causes an *fp_exception_other* trap (with `FSR.ftt` = unimplemented_FPop) and trap handler software emulates the `STQFA`. A JPS2 implementation never generates an *STQF_mem_address_not_aligned* exception (impl. dep. #112b).

---

**Programming Note –** In SPARC V8, some compilers issued sets of single-precision stores when they could not determine that double- or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned stores is expected to be fast, we recommend that compilers issue sets of single-precision stores only when they can determine that double- or quadword operands are *not* properly aligned.

---

*Exceptions*

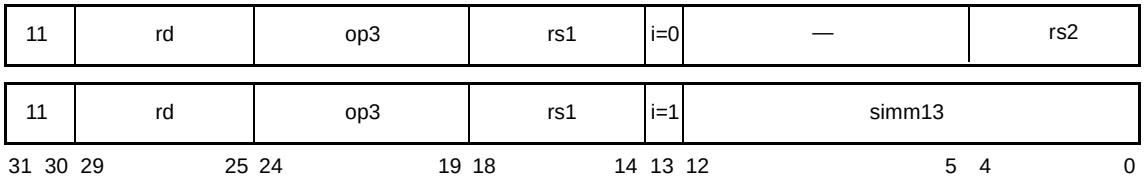
*fp_disabled*
*mem_address_not_aligned*
*STDF_mem_address_not_aligned* (`STDFA` only)
*STQF_mem_address_not_aligned* (`STQFA` only) (not used in JPS2)
*privileged_action*
*data_access_exception*

*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.63 Store Integer

| Opcode | op3 | Operation |
|---|---|---|
| STB | 00 0101 | Store Byte |
| STH | 00 0110 | Store Halfword |
| STW | 00 0100 | Store Word |
| STX | 00 1110 | Store Extended Word |

*Format (3)*




| Assembly Language Syntax | | |
|---|---|---|
| stb | *reg*$_{rd}$, [*address*] | (*synonyms*: stub, stsb) |
| sth | *reg*$_{rd}$, [*address*] | (*synonyms*: stuh, stsh) |
| stw | *reg*$_{rd}$, [*address*] | (*synonyms*: st, stuw, stsw) |
| stx | *reg*$_{rd}$, [*address*] | |

*Description*

The store integer instructions (except store doubleword) copy the whole extended (64-bit) integer, the less significant word, the least significant halfword, or the least significant byte of R[rd] into memory.

The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + sign_ext(simm13)" if i = 1.

A successful store (notably, store extended) instruction operates atomically.

STH causes a *mem_address_not_aligned* exception if the effective address is not halfword aligned. STW causes a *mem_address_not_aligned* exception if the effective address is not word aligned. STX causes a *mem_address_not_aligned* exception if the effective address is not doubleword aligned.

With respect to little-endian memory, a STD instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.

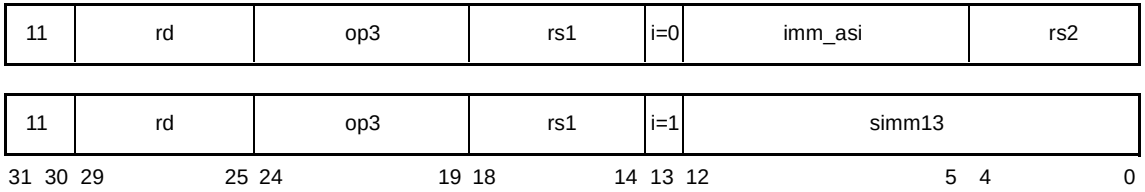
*Exceptions*

*mem_address_not_aligned* (all except STB)
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.64 Store Integer into Alternate Space

| Opcode | op3 | Operation |
|---|---|---|
| STBA$^{P_{ASI}}$ | 01 0101 | Store Byte into Alternate Space |
| STHA$^{P_{ASI}}$ | 01 0110 | Store Halfword into Alternate Space |
| STWA$^{P_{ASI}}$ | 01 0100 | Store Word into Alternate Space |
| STXA$^{P_{ASI}}$ | 01 1110 | Store Extended Word into Alternate Space |

*Format (3)*




| Assembly Language Syntax | | |
|---|---|---|
| `stba` | *reg$_{rd}$*, [*regaddr*] *imm_asi* | (*synonyms*: `stuba`, `stsba`) |
| `stha` | *reg$_{rd}$*, [*regaddr*] *imm_asi* | (*synonyms*: `stuha`, `stsha`) |
| `stwa` | *reg$_{rd}$*, [*regaddr*] *imm_asi* | (*synonyms*: `sta`, `stuwa`, `stswa`) |
| `stxa` | *reg$_{rd}$*, [*regaddr*] *imm_asi* | |
| `stba` | *reg$_{rd}$*, [*reg_plus_imm*] `%asi` | (*synonyms*: `stuba`, `stsba`) |
| `stha` | *reg$_{rd}$*, [*reg_plus_imm*] `%asi` | (*synonyms*: `stuha`, `stsha`) |
| `stwa` | *reg$_{rd}$*, [*reg_plus_imm*] `%asi` | (*synonyms*: `sta`, `stuwa`, `stswa`) |
| `stxa` | *reg$_{rd}$*, [*reg_plus_imm*] `%asi` | |

*Description* The store integer into alternate space instructions copy the whole extended (64-bit) integer, the less significant word, the least significant halfword, or the least significant byte of `R[rd]` into memory.

## Store Integer into Alternate Space

Store integer to alternate space instructions contain the address space identifier (ASI) to be used for the store in the `imm_asi` field if `i` = 0, or in the ASI register if `i` = 1. The access is privileged if bit 7 of the `asi` is 0; otherwise, it is not privileged. The effective address for these instructions is "`R[rs1]` + `R[rs2]`" if `i` = 0, or "`R[rs1]` + sign_ext(`simm13`)" if `i` = 1.

A successful store (notably, store extended) instruction operates atomically.

`STHA` causes a *mem_address_not_aligned* exception if the effective address is not halfword aligned. `STWA` causes a *mem_address_not_aligned* exception if the effective address is not word aligned. `STXA` causes a *mem_address_not_aligned* exception if the effective address is not doubleword aligned.

A store integer into alternate space instruction causes a *privileged_action* exception if `PSTATE.priv` = 0 and bit 7 of the ASI is 0.

With respect to little-endian memory, a `STDA` instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.

---

**V9 Compatibility Note –** The SPARC V8 `STA` instruction is renamed `STWA` in SPARC V9.

---

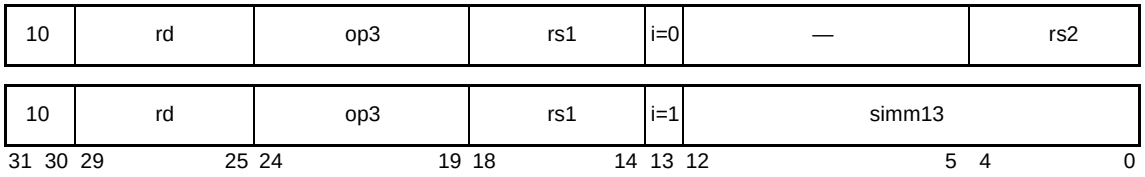
*Exceptions*

*privileged_action*
*mem_address_not_aligned* (all except `STBA`)
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.65 Subtract

| Opcode | op3 | Operation |
|---|---|---|
| SUB | 00 0100 | Subtract |
| SUBcc | 01 0100 | Subtract and modify cc's |
| SUBC | 00 1100 | Subtract with Carry |
| SUBCcc | 01 1100 | Subtract with Carry and modify cc's |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| sub | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| subcc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| subc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| subccc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |

*Description*

These instructions compute "R[rs1] – R[rs2]" if i = 0, or "R[rs1] – sign_ext(simm13)" if i = 1, and write the difference into R[rd].

SUBC and SUBCcc ("SUBtract with carry") also subtract the CCR register's 32-bit carry (icc.c) bit; that is, they compute "R[rs1] – R[rs2] – icc.c" or "R[rs1] – sign_ext(simm13) –icc.c," and write the difference into R[rd].

SUBcc and SUBCcc modify the integer condition codes (CCR.icc and ccr.xcc). A 32-bit overflow (CCR.icc.v) occurs on subtraction if bit 31 (the sign) of the operands differs and bit 31 (the sign) of the difference differs from R[rs1]<31>. A 64-bit overflow (CCR.xcc.v) occurs on subtraction if bit 63 (the sign) of the operands differs and bit 63 (the sign) of the difference differs from R[rs1]<63>.

**Programming Notes–** A `SUBcc` with `rd` = 0 can be used to effect a signed or unsigned integer comparison. See the `cmp` synthetic instruction in Appendix G, *Assembly Language Syntax*.

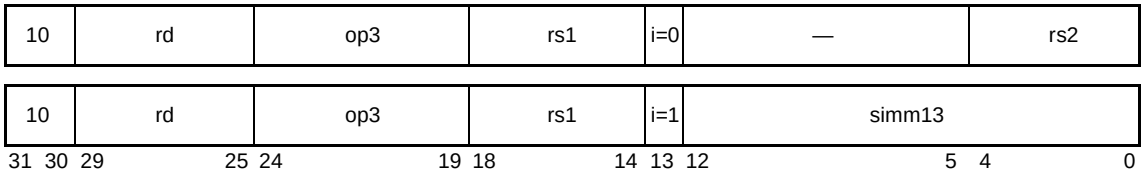
`SUBC` and `SUBCcc` read the 32-bit condition codes' carry bit (`ccr.icc.c`), not the 64-bit condition codes' carry bit (`ccr.xcc.c`).

*Exceptions* None

# A.66 Tagged Add

| Opcode | op3 | Operation |
|---|---|---|
| TADDcc | 10 0000 | Tagged Add and modify cc's |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| taddcc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |

*Description*

This instruction computes a sum that is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + sign_ext(simm13)" if i = 1.

TADDcc modifies the integer condition codes (icc and xcc).

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the addition generates 32-bit arithmetic overflow (that is, both operands have the same value in bit 31 and bit 31 of the sum is different).

If a TADDcc causes a tag overflow, the 32-bit overflow bit (ccr.icc.v) is set to 1; if TADDcc does not cause a tag overflow, ccr.icc.v is set to 0.

In either case, the remaining integer condition codes (both the other ccr.icc bits and all the ccr.xcc bits) are also updated as they would be for a normal ADD instruction. In particular, the setting of the ccr.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). ccr.xcc.v is set based on the 64-bit arithmetic overflow condition, like a normal 64-bit add.

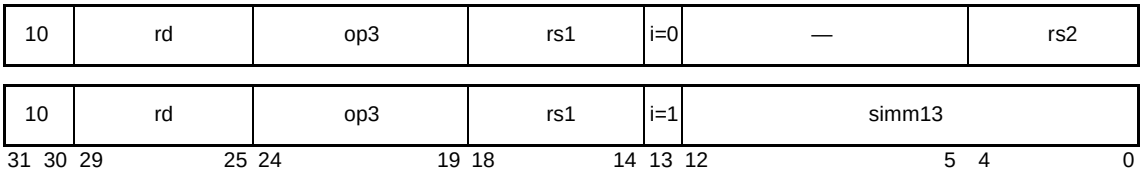
*Exceptions*

None

# A.67 Tagged Subtract

| Opcode | op3 | Operation |
|---|---|---|
| TSUBcc | 10 0001 | Tagged Subtract and modify cc's |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| tsubcc | *$reg_{rs1}$, reg_or_imm, $reg_{rd}$* |

*Description*

This instruction computes "R[rs1] – R[rs2]" if i = 0, or "R[rs1] – sign_ext(simm13)" if i = 1.

TSUBcc modifies the integer condition codes (icc and xcc).

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the subtraction generates 32-bit arithmetic overflow; that is, the operands have different values in bit 31 (the 32-bit sign bit) and the sign of the 32-bit difference in bit 31 differs from bit 31 of R[rs1].

If a TSUBcc causes a tag overflow, the 32-bit overflow bit (ccr.icc.v) is set to 1; if TSUBcc does not cause a tag overflow, ccr.icc.v is set to 0.

In either case, the remaining integer condition codes (both the other ccr.icc bits and all the ccr.xcc bits) are also updated as they would be for a normal subtract instruction. In particular, the setting of the ccr.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). ccr.xcc.v is set based on the 64-bit arithmetic overflow condition, like a normal 64-bit subtract.

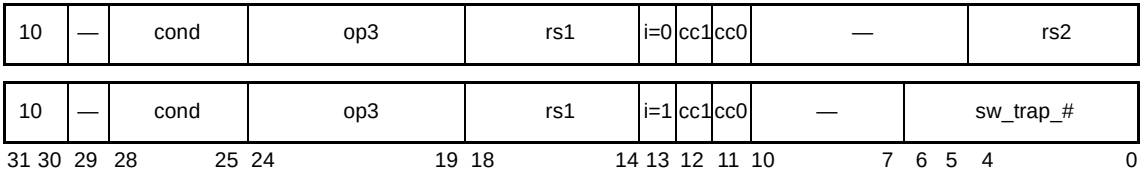
*Exceptions*

None

# A.68 Trap on Integer Condition Codes (Tcc)

| Opcode | op3 | cond | Operation | *icc* Test |
|---|---|---|---|---|
| TA | 11 1010 | 1000 | Trap Always | 1 |
| TN | 11 1010 | 0000 | Trap Never | 0 |
| TNE | 11 1010 | 1001 | Trap on Not Equal | not Z |
| TE | 11 1010 | 0001 | Trap on Equal | Z |
| TG | 11 1010 | 1010 | Trap on Greater | **not** (Z **or** (N **xor** V)) |
| TLE | 11 1010 | 0010 | Trap on Less or Equal | Z **or** (N **xor** V) |
| TGE | 11 1010 | 1011 | Trap on Greater or Equal | **not** (N **xor** V) |
| TL | 11 1010 | 0011 | Trap on Less | N **xor** V |
| TGU | 11 1010 | 1100 | Trap on Greater Unsigned | **not** (C **or** Z) |
| TLEU | 11 1010 | 0100 | Trap on Less or Equal Unsigned | (C **or** Z) |
| TCC | 11 1010 | 1101 | Trap on Carry Clear (Greater than or Equal, Unsigned) | **not** C |
| TCS | 11 1010 | 0101 | Trap on Carry Set (Less Than, Unsigned) | C |
| TPOS | 11 1010 | 1110 | Trap on Positive or zero | **not** N |
| TNEG | 11 1010 | 0110 | Trap on Negative | N |
| TVC | 11 1010 | 1111 | Trap on Overflow Clear | **not** V |
| TVS | 11 1010 | 0111 | Trap on Overflow Set | V |

*Format (4)*



| cc1 | cc0 | Condition Codes |
|---|---|---|
| 0 | 0 | icc |
| 0 | 1 | — |
| 1 | 0 | xcc |
| 1 | 1 | — |

| Assembly Language Syntax | | |
|---|---|---|
| ta | *i_or_x_cc, software_trap_number* | |
| tn | *i_or_x_cc, software_trap_number* | |
| tne | *i_or_x_cc, software_trap_number* | (*synonym*: tnz) |
| te | *i_or_x_cc, software_trap_number* | (*synonym*: tz) |
| tg | *i_or_x_cc, software_trap_number* | |
| tle | *i_or_x_cc, software_trap_number* | |
| tge | *i_or_x_cc, software_trap_number* | |
| tl | *i_or_x_cc, software_trap_number* | |
| tgu | *i_or_x_cc, software_trap_number* | |
| tleu | *i_or_x_cc, software_trap_number* | |
| tcc | *i_or_x_cc, software_trap_number* | (*synonym*: tgeu) |
| tcs | *i_or_x_cc, software_trap_number* | (*synonym*: tlu) |
| tpos | *i_or_x_cc, software_trap_number* | |
| tneg | *i_or_x_cc, software_trap_number* | |
| tvc | *i_or_x_cc, software_trap_number* | |
| tvs | *i_or_x_cc, software_trap_number* | |

*Description*

The Tcc instruction evaluates the selected integer condition codes (icc or xcc) according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE and no higher-priority exceptions or interrupt requests are pending, then a *trap_instruction* exception is generated. If FALSE, a *trap_instruction* exception does not occur and the instruction behaves like a NOP.

The software trap number is specified by the least significant seven bits of "R[rs1] + R[rs2]" if i = 0, or the least significant seven bits of "R[rs1] + sw_trap_#" if i = 1.

When i = 1, bits 7 through 10 are reserved and should be supplied as zeroes by software. When i = 0, bits 5 through 10 are reserved, the most significant 57 bits of "R[rs1] + R[rs2]" are unused, and both should be supplied as zeroes by software.

*Description (Effect on Privileged State)*

If a *trap_instruction* traps, 256 plus the software trap number is written into TT[TL]. Then the trap is taken and the virtual processor performs the normal trap entry procedure, as described in Chapter 7, *Traps*.

---

**Programming Note –** Tcc can be used to implement breakpointing, tracing, and calls to supervisor software. It can also be used for runtime checks, such as out-of-range array indexes, integer overflow, and so on.

---

---

**SPARC V8 Compatibility Note –** Tcc is upward compatible with the SPARC V8 Ticc instruction, with one qualification: a Ticc with i = 1 and simm13 < 0 may execute differently on a SPARC V9 processor. Use of the i = 1 form of Ticc is believed to be rare in SPARC V8 software, and simm13 < 0 is probably not used at all, so it is believed that, in practice, full software compatibility will be achieved.

---

*Exceptions*

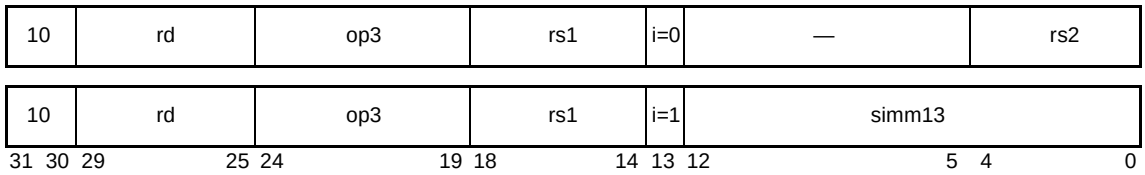
*trap_instruction*

*illegal_instruction* (cc1 ▯ cc0 = $01_2$ or $11_2$, or reserved fields nonzero)

# A.69 Write Privileged Register

| Opcode | op3 | Operation |
|---|---|---|
| WRPR[P] | 11 0010 | Write Privileged Register |

*Format (3)*




| rd | Privileged Register |
|---|---|
| 0 | TPC |
| 1 | TNPC |
| 2 | TSTATE |
| 3 | TT |
| 4 | TICK |
| 5 | TBA |
| 6 | PSTATE |
| 7 | TL |
| 8 | PIL |
| 9 | CWP |
| 10 | CANSAVE |
| 11 | CANRESTORE |
| 12 | CLEANWIN |
| 13 | OTHERWIN |
| 14 | WSTATE |
| 15–31 | *Reserved* |

| Assembly Language Syntax | |
|---|---|
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %tpc |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %tnpc |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %tstate |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %tt |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %tick |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %tba |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %pstate |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %tl |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %pil |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %cwp |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %cansave |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %canrestore |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %cleanwin |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %otherwin |
| wrpr | *reg*$_{rs1}$, *reg_or_imm*, %wstate |

*Description*

This instruction stores the value "R[rs1] **xor** R[rs2]" if i = 0, or "R[rs1] **xor** sign_ext(simm13)" if i = 1 to the writable fields of the specified privileged state register. **Note:** The operation is exclusive-or.

The rd field in the instruction determines the privileged register that is written. There are at least four copies of the TPC, TNPC, TT, and TSTATE registers, one for each trap level. A write to one of these registers sets the register indexed by the current value in the trap-level register (TL). A write to TPC, TNPC, TT, or TSTATE when the trap level is zero (TL = 0) causes an *illegal_instruction* exception.

A WRPR of TL does not cause a trap or return from trap; it does not alter any other machine state.

**Programming Note –** A WRPR of TL can be used to read the values of TPC, TNPC, and TSTATE for any trap level; however, take care that traps do not occur while the TL register is modified.

The WRPR instruction is a *non*-delayed-write instruction. The instruction immediately following the WRPR observes any changes made to virtual processor state made by the WRPR.

An attempt to write a reserved combination of ag, mg, and ig bit values to the PSTATE register using a WRPR instruction causes an *illegal_instruction* exception.

WRPR instructions with rd in the range 15–31 are reserved for future versions of the architecture; executing a WRPR instruction with rd in that range causes an *illegal_instruction* exception.

---

**V9 Compatibility Note –** Some WRPR instructions could serialize the virtual processor in some implementations. See specific JPS2 Extensions documents for applicability and details.

---

*Exceptions*

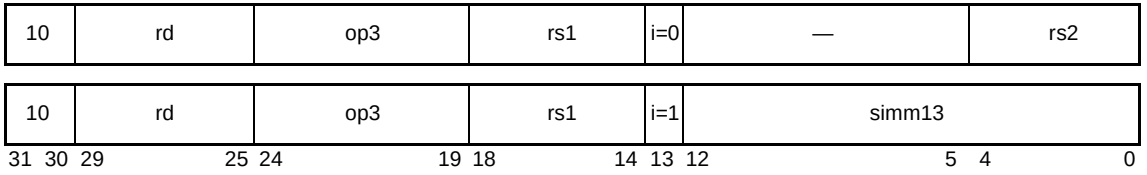
*privileged_opcode*

*illegal_instruction* ((rd = 15–31) or ((rd ≤ 3) and (TL = 0)), or
WRPR to PSTATE of a reserved combination of ag, mg, and ig bit values)

# A.70 Write State Register

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| WRY[D] | 11 0000 | 0 | Write Y register; deprecated (see A.71.18) |
| — | 11 0000 | 1 | *Reserved* |
| WRCCR | 11 0000 | 2 | Write Condition Codes Register |
| WRASI | 11 0000 | 3 | Write ASI Register |
| — | 11 0000 | 4, 5 | *Reserved* |
| WRFPRS | 11 0000 | 6 | Write Floating-Point Registers Status Register |
| — | 11 0000 | 7–14 | *Reserved* |
| — | 11 0000 | 15 | Software-initiated reset (see A.60) |
| WRASR | 11 0000 | 16–31 | Write non-SPARC V9 ASRs |
| WRPCR[P PCR] | | 16 | Write Performance Control Registers (PCR) |
| WRPIC[P PIC] | | 17 | Write Performance Instrumentation Counters (PIC) |
| WRDCR[P] | | 18 | Write Dispatch Control Register (DCR) |
| WRGSR | | 19 | Write Graphic Status Register (GSR) |
| WRSOFTINT_SET[P] | | 20 | Set bits of per-virtual processor Soft Interrupt Register |
| WRSOFTINT_CLR[P] | | 21 | Clear bits of per-virtual processor Soft Interrupt Register |
| WRSOFTINT[P] | | 22 | Write per-virtual processor Soft Interrupt Register |
| WRTICK_CMPR[P] | | 23 | Write Tick Compare Register |
| WRSTICK[P] | | 24 | Write System TICK Register |
| WRSTICK_CMPR[P] | | 25 | Write System TICK Compare Register |
| — | | 26–31 | *Implementation dependent (impl. dep. #8, 9)* |

*Format (3)*



| Assembly Language Syntax | | |
|---|---|---|
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %ccr |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %asi |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %fprs |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %pcr |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %pic |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %dcr |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %gsr |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %set_softint |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %clear_softint |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %softint |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %tick_cmpr |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %sys_tick |
| wr | *reg*$_{rs1}$, *reg_or_imm*, | %sys_tick_cmpr |

*Description*

These instructions store the value "R[rs1] **xor** R[rs2]" if i = 0, or "R[rs1] **xor** sign_ext(simm13)" if i = 1, to the writable fields of the specified state register. **Note**: The operation is exclusive-or.

WRASR writes a value to the ancillary state register (ASR) indicated by rd. The operation performed to generate the value written may be rd dependent or implementation dependent (see below). A WRASR instruction is indicated by op = *2*, rd = ≥ 16, and op3 = $30_{16}$.

**IMPL. DEP. #48:** WRASR instructions with rd in the range 26–31 are available for implementation-dependent uses (impl. dep. #8). For a WRASR instruction with rd in the range 26–31, the following are implementation dependent: the interpretation of bits 18:0 in the instruction, the operation(s) performed (for example, **xor**) to generate the value written to the ASR, whether the instruction is privileged (impl. dep. #9), and whether the instruction causes an *illegal_instruction* exception.

The WRASR opcode for rd = 15, rs1 = 0, and i = 1 is used for the software-initiated reset (SIR) instruction (see A.60).

The WRCCR, WRFPRS, and WRASI instructions are *not* delayed-write instructions. The instruction immediately following a WRCCR, WRFPRS, or WRASIR observes the new value of the CCR, FPRS, or ASI register.

WRFPRS waits for any pending floating-point operations to complete before writing the FPRS register.

WRGSR causes an *fp_disabled* trap if PSTATE.pef = 0 or FPRS.fef = 0.

WRPIC causes a *privileged_action* exception if PSTATE.priv = 0 and PCR.priv = 1.

WRPCR causes an exception due to access privilege violation under implementation-dependent circumstances (impl. dep. #250).

See *State Registers* on page 86 for details of the ASR registers.

---

**Note –** See I.1, *Read/Write Ancillary State Registers (ASRs)*, for a discussion of extending the SPARC V9 instruction set by means of read / write ASR instructions.

---

**V9 Compatibility Note –** Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers.

---

**V9 Compatibility Note –** The SPARC V8 WRIER, WRPSR, WRWIM, and WRTBR instructions do not exist in SPARC V9 because the IER, PSR, TBR, and WIM registers do not exist in SPARC V9.

---

**V9 Compatibility Note –** Some WRASR instructions could serialize the virtual processor in some implementations. See specific JPS2 Extensions documents for applicability and details.

---

*Exceptions*

*software_initiated_reset* (rd = 15, rs1 = 0, and i = 1 only)
*privileged_opcode* (WRDCR, WRSOFTINT_SET, WRSOFTINT_CLR, WRSOFTINT, WRTICK_CMPR, WRSTICK, WRSTICK_CMPR, and WRPCR (impl. dep. #250))
*illegal_instruction* (WRASR with rd = 1, 4, 5, 7–14, 26-31; WRASR with rd = 15 and rs1 ≠ 0 or i ≠ 1)
*privileged_action* (WRPIC with PSTATE.priv = 0 and PCR.priv = 1, WRPCR (impl. dep. #250))
*fp_disabled* (WRGSR with PSTATE.pef = 0 or FPRS.fef = 0)

# A.71 Deprecated Instructions

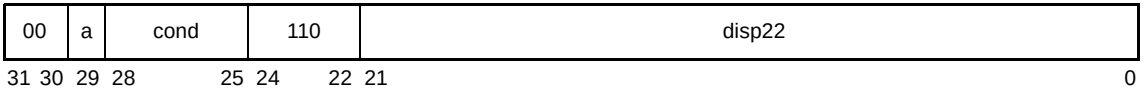
The following instructions are deprecated; they are provided only for compatibility with previous versions of the architecture. They should not be used in new SPARC V9 software. For each deprecated instruction, we recommend the instruction to be used instead. Please see TABLE A-2 on page 218 for the page number at which you can find a description of the preferred instruction.

## A.71.1 Branch on Floating-Point Condition Codes (FBfcc)

The `FBfcc` instructions are deprecated. Use the `FBPfcc` instructions instead.

| Opcode | cond | Operation | *fcc* Test |
|---|---|---|---|
| `FBA`$^{D}$ | 1000 | Branch Always | 1 |
| `FBN`$^{D}$ | 0000 | Branch Never | 0 |
| `FBU`$^{D}$ | 0111 | Branch on Unordered | U |
| `FBG`$^{D}$ | 0110 | Branch on Greater | G |
| `FBUG`$^{D}$ | 0101 | Branch on Unordered or Greater | G **or** U |
| `FBL`$^{D}$ | 0100 | Branch on Less | L |
| `FBUL`$^{D}$ | 0011 | Branch on Unordered or Less | L **or** U |
| `FBLG`$^{D}$ | 0010 | Branch on Less or Greater | L **or** G |
| `FBNE`$^{D}$ | 0001 | Branch on Not Equal | L **or** G **or** U |
| `FBE`$^{D}$ | 1001 | Branch on Equal | E |
| `FBUE`$^{D}$ | 1010 | Branch on Unordered or Equal | E **or** U |
| `FBGE`$^{D}$ | 1011 | Branch on Greater or Equal | E **or** G |
| `FBUGE`$^{D}$ | 1100 | Branch on Unordered or Greater or Equal | E **or** G **or** U |
| `FBLE`$^{D}$ | 1101 | Branch on Less or Equal | E **or** L |
| `FBULE`$^{D}$ | 1110 | Branch on Unordered or Less or Equal | E **or** L **or** U |
| `FBO`$^{D}$ | 1111 | Branch on Ordered | E **or** L **or** G |

*Format (2)*



| Assembly Language Syntax | | |
|---|---|---|
| `fba{,a}` | *label* | |
| `fbn{,a}` | *label* | |
| `fbu{,a}` | *label* | |
| `fbg{,a}` | *label* | |
| `fbug{,a}` | *label* | |
| `fbl{,a}` | *label* | |
| `fbul{,a}` | *label* | |
| `fblg{,a}` | *label* | |
| `fbne{,a}` | *label* | (*synonym*: `fbnz`) |
| `fbe{,a}` | *label* | (*synonym*: `fbz`) |
| `fbue{,a}` | *label* | |
| `fbge{,a}` | *label* | |
| `fbuge{,a}` | *label* | |
| `fble{,a}` | *label* | |
| `fbule{,a}` | *label* | |
| `fbo{,a}` | *label* | |

**Programming Note –** To set the annul bit for `FBfcc` instructions, append "`,a`" to the opcode mnemonic. For example, use "`fbl,a` *label*." In the preceding table, braces around "`,a`" signify that "`,a`" is optional.

*Description:* Unconditional and `Fcc` branches are described below:

- **Unconditional branches (FBA, FBN)** — If its annul field is 0, an `FBN` (Branch Never) instruction acts like a NOP. If its annul field is 1, the following (delay) instruction is annulled (not executed) when the `FBN` is executed. In neither case does a transfer of control take place.

  `FBA` (Branch Always) causes a `PC`-relative, delayed control transfer to the address "`PC` + (4 × `sign_ext(disp22)`)," regardless of the value of the floating-point condition code bits. If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

- **Fcc-conditional branches** — Conditional FBfcc instructions (except FBA and FBN) evaluate floating-point condition code zero (fcc0) according to the cond field of the instruction. Such evaluation produces either a TRUE or FALSE result. If TRUE, the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 × sign_ext(disp22))." If FALSE, the branch is not taken.

  If a conditional branch is taken, the delay instruction is always executed, regardless of the value of the annul field. If a conditional branch is not taken and the a (annul) field is 1, the delay instruction is annulled (not executed).

---

**Note –** The annul bit has a *different* effect on conditional branches than it does on unconditional branches.

---

Annulment, delay instructions, and delayed control transfers are described further in Chapter 6 of **Commonality**.

---

**V9 Compatibility Note –** Unlike SPARC V8, SPARC V9 does not require an instruction between a floating-point compare operation and a floating-point branch (FBfcc, FBPfcc).

---

If FPRS.fef = 0 or PSTATE.pef = 0, or if an FPU is not present, the FBfcc instruction is not executed and instead generates an *fp_disabled* exception.

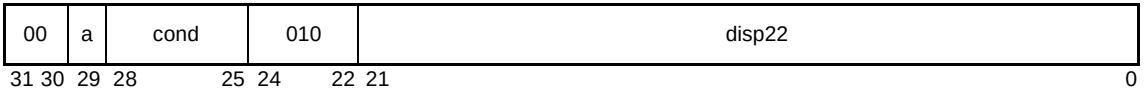
*Exceptions* *fp_disabled*

## A.71.2 Branch on Integer Condition Codes (Bicc)

Use the `BPcc` instructions in place of `Bicc` instructions.

| Opcode | cond | Operation | *icc* Test |
|---|---|---|---|
| `BA`$^{D}$ | 1000 | Branch Always | 1 |
| `BN`$^{D}$ | 0000 | Branch Never | 0 |
| `BNE`$^{D}$ | 1001 | Branch on Not Equal | **not** Z |
| `BE`$^{D}$ | 0001 | Branch on Equal | Z |
| `BG`$^{D}$ | 1010 | Branch on Greater | **not** (Z **or** (N **xor** V)) |
| `BLE`$^{D}$ | 0010 | Branch on Less or Equal | Z **or** (N **xor** V) |
| `BGE`$^{D}$ | 1011 | Branch on Greater or Equal | **not** (N **xor** V) |
| `BL`$^{D}$ | 0011 | Branch on Less | N **xor** V |
| `BGU`$^{D}$ | 1100 | Branch on Greater Unsigned | **not** (C **or** Z) |
| `BLEU`$^{D}$ | 0100 | Branch on Less or Equal Unsigned | C **or** Z |
| `BCC`$^{D}$ | 1101 | Branch on Carry Clear (Greater Than or Equal, Unsigned) | **not** C |
| `BCS`$^{D}$ | 0101 | Branch on Carry Set (Less Than, Unsigned) | C |
| `BPOS`$^{D}$ | 1110 | Branch on Positive | **not** N |
| `BNEG`$^{D}$ | 0110 | Branch on Negative | N |
| `BVC`$^{D}$ | 1111 | Branch on Overflow Clear | **not** V |
| `BVS`$^{D}$ | 0111 | Branch on Overflow Set | V |

*Format (2)*



| Assembly Language Syntax | | |
|---|---|---|
| `ba{,a}` | *label* | |
| `bn{,a}` | *label* | |
| `bne{,a}` | *label* | (*synonym*: `bnz`) |
| `be{,a}` | *label* | (*synonym*: `bz`) |
| `bg{,a}` | *label* | |
| `ble{,a}` | *label* | |
| `bge{,a}` | *label* | |
| `bl{,a}` | *label* | |
| `bgu{,a}` | *label* | |
| `bleu{,a}` | *label* | |
| `bcc{,a}` | *label* | (*synonym*: `bgeu`) |
| `bcs{,a}` | *label* | (*synonym*: `blu`) |
| `bpos{,a}` | *label* | |
| `bneg{,a}` | *label* | |
| `bvc{,a}` | *label* | |
| `bvs{,a}` | *label* | |

**Programming Note –** To set the annul bit for `Bicc` instructions, append "`,a`" to the opcode mnemonic. For example, use "`bgu,a` *label*." In the preceding table, braces signify that the "`,a`" is optional.

*Description*

Unconditional branches and icc-conditional branches are described below:

- **Unconditional branches (BA, BN)** — If its annul field is 0, a `BN` (Branch Never) instruction is treated as a NOP. If its annul field is 1, the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

  `BA` (Branch Always) causes an unconditional `PC`-relative, delayed control transfer to the address "`PC` + (4 × `sign_ext(disp22)`)." If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

- **Icc-conditional branches** — Conditional `Bicc` instructions (all except `BA` and `BN`) evaluate the 32-bit integer condition codes (`icc`), according to the `cond` field of the instruction, producing either a `TRUE` or `FALSE` result. If `TRUE`, the branch is taken, that is, the instruction causes a `PC`-relative, delayed control transfer to the address "`PC` + (4 × `sign_ext(disp22)`)." If `FALSE`, the branch is not taken.

  If a conditional branch is taken, the delay instruction is always executed regardless of the value of the annul field. If a conditional branch is not taken and the `a` (annul) field is 1, the delay instruction is annulled (not executed).

---

**Note –** The annul bit has a *different* effect on conditional branches than it does on unconditional branches.

---

Annulment, delay instructions, and delayed control transfers are described further in Chapter 6, *Instructions*.

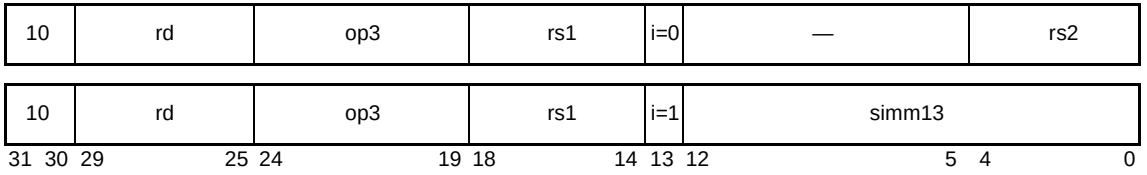
*Exceptions* None

# A.71.3 Divide (64-bit / 32-bit)

The UDIV, UDIVcc, SDIV, and SDIVcc instructions are deprecated. Use the UDIVX and SDIVX instructions instead.

| Opcode | op3 | Operation |
|---|---|---|
| UDIV[D] | 00 1110 | Unsigned Integer Divide |
| SDIV[D] | 00 1111 | Signed Integer Divide |
| UDIVcc[D] | 01 1110 | Unsigned Integer Divide and modify cc's |
| SDIVcc[D] | 01 1111 | Signed Integer Divide and modify cc's |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| udiv | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| sdiv | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| udivcc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| sdivcc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |

*Description*

The divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If i = 0, they compute "(Y □ r[rs1]<31:0>) ÷ r[rs2]<31:0>." Otherwise (that is, if i = 1), the divide instructions compute "(Y □ r[rs1]<31:0>) ÷ (sign_ext(simm13)<31:0>)." In either case, if overflow does not occur, the less significant 32 bits of the integer quotient are sign- or zero-extended to 64 bits and are written into r[rd].

The contents of the Y register are undefined after any 64-bit by 32-bit integer divide operation.

*Unsigned Divide*

Unsigned divide (UDIV, UDIVcc) assumes an unsigned integer doubleword dividend (Y || r[rs1]<31:0>) and an unsigned integer word divisor r[rs2<31:0>] or (sign_ext(simm13)<31:0>) and computes an unsigned integer word quotient (r[rd]). Immediate values in simm13 are in the ranges 0 to $2^{12}-1$ and $2^{32}-2^{12}$ to $2^{32}-1$ for unsigned divide instructions.

Unsigned division rounds an inexact rational quotient toward zero.

---

**Programming Note –** The *rational quotient* is the infinitely precise result quotient. It includes both the integer part and the fractional part of the result. For example, the rational quotient of 11/4 = 2.75 (integer part = 2, fractional part = .75).

---

The result of an unsigned divide instruction can overflow the less significant 32 bits of the destination register r[rd] under certain conditions. When overflow occurs, the largest appropriate unsigned integer is returned as the quotient in r[rd]. The condition under which overflow occurs and the value returned in r[rd] under this condition are specified in TABLE A-9.

**TABLE A-9** UDIV / UDIVcc Overflow Detection and Value Returned

| Condition Under Which Overflow Occurs | Value Returned in r[rd] |
|---|---|
| Rational quotient ≥ $2^{32}$ | $2^{32}-1$ ($0000\ 0000\ FFFF\ FFFF_{16}$) |

When no overflow occurs, the 32-bit result is zero-extended to 64 bits and written into register r[rd].

UDIV does not affect the condition code bits. UDIVcc writes the integer condition code bits as shown in the following table. Note that negative (N) and zero (Z) are set according to the value of r[rd] after it has been set to reflect overflow, if any.

| Bit | UDIVcc |
|---|---|
| icc.n | Set if r[rd]<31> = 1 |
| icc.z | Set if r[rd]<31:0> = 0 |
| icc.v | Set if overflow (*per* TABLE A-9) |
| icc.c | Zero |
| xcc.n | Set if r[rd]<63> = 1 |
| xcc.z | Set if r[rd]<63:0> = 0 |
| xcc.v | Zero |
| xcc.c | Zero |

*Signed Divide*

Signed divide (SDIV, SDIVcc) assumes a signed integer doubleword dividend (Y ▯ *lower 32 bits of* r[rs1]) and a signed integer word divisor (*lower 32 bits of* r[rs2] or *lower 32 bits of* sign_ext(simm13)) and computes a signed integer word quotient (r[rd]).

Signed division rounds an inexact quotient toward zero. For example, –7 ÷ 4 equals the rational quotient of –1.75, which rounds to –1 (not –2) when rounding toward zero.

The result of a signed divide can overflow the low-order 32 bits of the destination register r[rd] under certain conditions. When overflow occurs, the largest appropriate signed integer is returned as the quotient in r[rd]. The conditions under which overflow occurs and the value returned in r[rd] under those conditions are specified in TABLE A-10.

**TABLE A-10** SDIV / SDIVcc Overflow Detection and Value Returned

| Condition Under Which Overflow Occurs | Value Returned in *r[rd]* |
|---|---|
| Rational quotient $\geq 2^{31}$ | $2^{31} - 1$ (0000 0000 7FFF $FFFF_{16}$) |
| Rational quotient $\leq -2^{31} - 1$ | $-2^{31}$ (FFFF FFFF 8000 $0000_{16}$) |

When no overflow occurs, the 32-bit result is sign-extended to 64 bits and written into register r[rd].

SDIV does not affect the condition code bits. SDIVcc writes the integer condition code bits as shown in the following table. Note that negative (N) and zero (Z) are set according to the value of r[rd] after it has been set to reflect overflow, if any.

| Bit | SDIVcc |
|---|---|
| icc.n | Set if r[rd]<31> = 1 |
| icc.z | Set if r[rd]<31:0> = 0 |
| icc.v | Set if overflow (*per* TABLE A-10) |
| icc.c | Zero |
| xcc.n | Set if r[rd]<63]> = 1 |
| xcc.z | Set if r[rd]<63:0> = 0 |
| xcc.v | Zero |
| xcc.c | Zero |

*Exceptions*

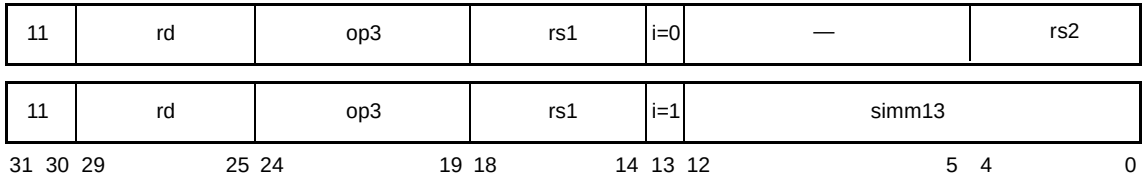
*division_by_zero*

## A.71.4 Load Floating-Point Status Register

The LDFSR instruction is deprecated. Use the LDXFSR instruction instead.

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| LDFSR[D] | 10 0001 | 0 | Load Floating-Point State Register Lower |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| ld | [*address*], %fsr |

*Description*

The load floating-point state register lower instruction (LDFSR) waits for all FPop instructions that have not finished execution to complete and then loads a word from memory into the less significant 32 bits of the FSR. The upper 32 bits of FSR are unaffected by LDFSR.

LDFSR causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned.

**V9 Compatibility Note –** SPARC V9 supports two different instructions to load the FSR: the SPARC V8 LDFSR instruction is defined to load only the less significant 32 bits of the FSR, whereas LDXFSR allows SPARC V9 programs to load all 64 bits of the FSR.

*Exceptions*

*mem_address_not_aligned*
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

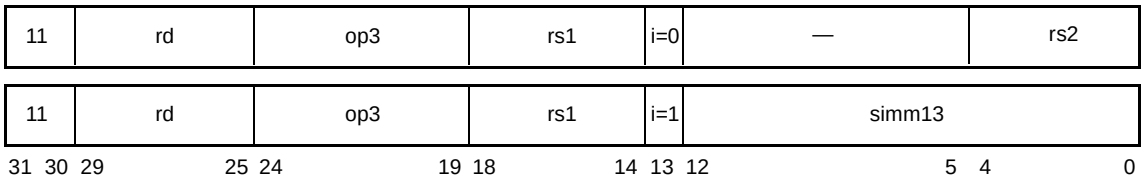
# A.71.5 Load Integer Doubleword

The LDD instruction is deprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC V9 software. Use the LDX instruction instead.

Please refer to A.28 on page 279 for the current load integer instructions.

| Opcode | op3 | Operation |
|---|---|---|
| LDD[D] | 00 0011 | Load doubleword |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| ldd | [*address*], *reg*$_{rd}$ |

*Description:* The load doubleword integer instruction (LDD) copies a doubleword from memory into an r-register pair. The word at the effective memory address is copied into the even r register. The word at the effective memory address + 4 is copied into the following odd-numbered r register. The upper 32 bits of both the even-numbered and odd-numbered r registers are zero-filled.

**Note –** A load doubleword with rd = 0 modifies only r[1].

The least significant bit of the rd field in an LDD instruction is unused and should always be set to 0 by software. An attempt to execute a load doubleword instruction that refers to a misaligned (odd-numbered) destination register causes an *illegal_instruction* exception.

With respect to little endian memory, an LDD instruction behaves as if it is comprises two 32-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

Load integer doubleword instructions access the primary address space (ASI = $80_{16}$). The effective address is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

A successful load doubleword instruction operates atomically.

**IMPL. DEP. #107a:** It is implementation dependent whether LDD is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_LDD* exception.

---

**Programming Note –** LDD is provided for compatibility with SPARC V8. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties.

---

*Exceptions*

*illegal_instruction* (LDD with odd rd)
*unimplemented_LDD*
*mem_address_not_aligned*
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

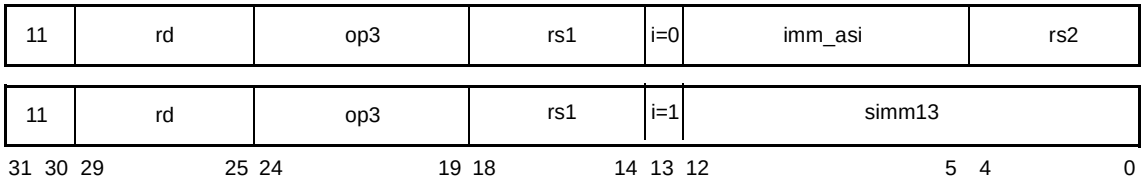
# A.71.6 Load Integer Doubleword from Alternate Space

The LDDA instruction is deprecated. Use the LDXA instruction in its place.

Please refer to A.29 on page 281 for current load integer from alternate space instructions.

| Opcode | op3 | Operation |
|---|---|---|
| $\texttt{LDDA}^{D,\ P_{ASI}}$ | 01 0011 | Load Doubleword from Alternate Space |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| `ldda` | [*regaddr*] *imm_asi*, *reg* $_{rd}$ |
| `ldda` | [*reg_plus_imm*] `%asi`, *reg* $_{rd}$ |

*Description*

The load doubleword integer from alternate space instruction (LDDA) copies a doubleword from memory into an r-register pair. The word at the effective memory address is copied into the even r register. The word at the effective memory address + 4 is copied into the following odd-numbered r register. The upper 32 bits of both the even-numbered and odd-numbered r registers are zero-filled.

**Note –** A load doubleword with rd = 0 modifies only r[1].

The least significant bit of the rd field in an LDDA instruction is unused and should always be set to 0 by software. An attempt to execute a load doubleword instruction that refers to a misaligned (odd-numbered) destination register causes an *illegal_instruction* exception.

With respect to little endian memory, an LDDA instruction behaves as if it is composed of two 32-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

The load integer doubleword from alternate space instructions contain the address space identifier (ASI) to be used for the load in the imm_asi field if i = 0, or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

A successful load doubleword instruction operates atomically.

LDDA causes a *mem_address_not_aligned* exception if the address is not doubleword aligned.

These instructions cause a *privileged_action* exception if PSTATE.priv = 0 and bit 7 of the ASI is 0.

LDDA with ASI = $24_{16}$, $2C_{16}$, $34_{16}$, or $3C_{16}$ is defined to be a Load Quadword Atomic instruction, which is *not* deprecated. See A.30 on page 283 for details.

**IMPL. DEP. #107b:** It is implementation dependent whether LDDA is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_LDD* exception.

Nontranslating ASIs should only be read using LDXA (not LDDA) instructions. Although reading from a nontranslating ASI using LDDA will work on a JPS2 processor, use of LDDA is discouraged because it may not work on non-JPS2 SPARC V9 processors or on future JPS processors.

**IMPL. DEP. #300:** If an LDDA referencing a nontranslating ASI is executed, it is implementation dependent whether the LDDA executes without error or generates a *data_access_exception* exception.

---

**Programming Note –** LDDA is provided for compatibility with SPARC V8. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties.

If LDDA is emulated in software, an LDXA instruction should be used for the memory access in order to preserve atomicity.

---

*Exceptions*

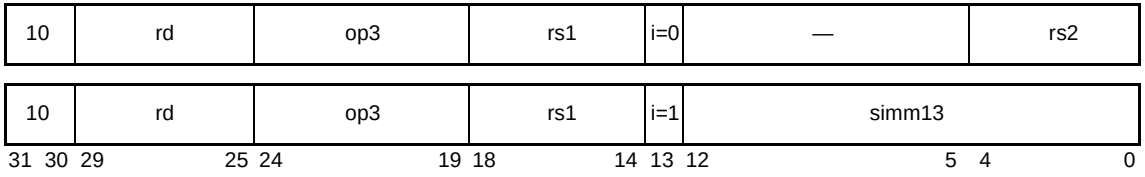
*privileged_action*
*illegal_instruction* (LDDA with odd rd)
*mem_address_not_aligned*
*data_access_exception*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.71.7 Multiply (32-bit)

The UMUL, UMULcc, SMUL, and SMULcc instructions are deprecated. Use the MULX instruction instead.

| Opcode | op3 | Operation |
|---|---|---|
| UMUL[D] | 00 1010 | Unsigned Integer Multiply |
| SMUL[D] | 00 1011 | Signed Integer Multiply |
| UMULcc[D] | 01 1010 | Unsigned Integer Multiply and modify cc's |
| SMULcc[D] | 01 1011 | Signed Integer Multiply and modify cc's |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| umul | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| smul | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| umulcc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |
| smulcc | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |

*Description*

The multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute "r[rs1]<31:0> × r[rs2]<31:0>" if i = 0, or "r[rs1]<31:0> × sign_ext(simm13)<31:0>" if i = 1. They write the 32 most significant bits of the product into the Y register and all 64 bits of the product into r[rd].

Unsigned multiply instructions (UMUL, UMULcc) operate on unsigned integer word operands and compute an unsigned integer doubleword product. Signed multiply instructions (SMUL, SMULcc) operate on signed integer word operands and compute a signed integer doubleword product.

UMUL and SMUL do not affect the condition code bits. UMULcc and SMULcc write the integer condition code bits, icc and xcc, as shown in TABLE A-11. **Note:** 32-bit negative (icc.n) and zero (icc.z) condition codes are set according to the *less* significant word of the product, not according to the full 64-bit result.

**TABLE A-11** UMULcc / SMULcc Condition Code Settings

| Bit | UMULcc / SMULcc |
|---|---|
| icc.n | Set if product<31> = 1 |
| icc.z | Set if product<31:0>= 0 |
| icc.v | 0 |
| icc.c | 0 |
| xcc.n | Set if product<63> = 1 |
| xcc.z | Set if product<63:0> = 0 |
| xcc.v | 0 |
| xcc.c | 0 |

**Programming Notes –** 32-bit overflow after UMUL/UMULcc is indicated by Y ≠ 0.

32-bit overflow after SMUL/SMULcc is indicated by Y ≠ (r[rd] >> 31), where ">>" indicates 32-bit arithmetic right-shift.

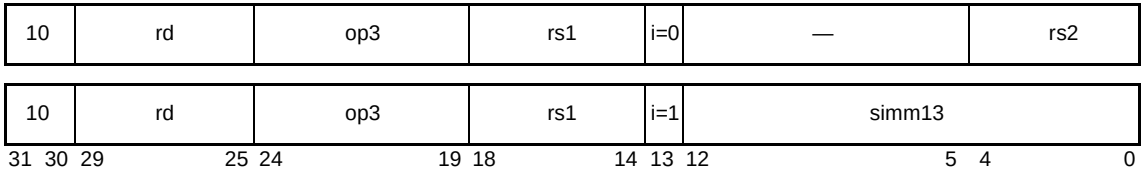
*Exceptions* None

# A.71.8 Multiply Step

The MULScc instruction is deprecated. Use the MULX instruction instead.

| Opcode | op3 | Operation |
|---|---|---|
| MULScc[D] | 10 0100 | Multiply Step and modify cc's |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| mulscc | $reg_{rs1}$, *reg_or_imm*, $reg_{rd}$ |

*Description*

MULScc treats the less significant 32 bits of both r[rs1] and the Y register as a single 64-bit, right-shiftable doubleword register. The least significant bit of r[rs1] is treated as if it were adjacent to bit 31 of the Y register. The MULScc instruction adds, based on the least significant bit of Y.

Multiplication assumes that the Y register initially contains the multiplier, r[rs1] contains the most significant bits of the product, and r[rs2] contains the multiplicand. Upon completion of the multiplication, the Y register contains the least significant bits of the product.

**Note**: A standard MULScc instruction has rs1 = rd.

MULScc operates as follows:

1. The multiplicand is r[rs2] if i = 0, or sign_ext(simm13) if i = 1.
2. A 32-bit value is computed by shifting r[rs1] right by one bit with "ccr.icc.n **xor** ccr.icc.v" replacing bit 31 of r[rs1]. (This is the proper sign for the previous partial product.)
3. If the least significant bit of Y = 1, the shifted value from step (2) and the multiplicand are added. If the least significant bit of the Y = 0, then 0 is added to the shifted value from step (2).

4. The sum from step (3) is written into `r[rd]`. The upper 32 bits of `r[rd]` are undefined. The integer condition codes are updated according to the addition performed in step (3). The values of the extended condition codes are undefined.

5. The `Y` register is shifted right by one bit, with the least significant bit of the unshifted `r[rs1]` replacing bit 31of `Y`.

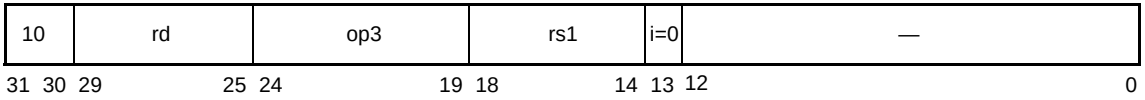
*Exceptions* None

# A.71.9 Read Y Register

The RDY instruction from the Read State Register instructions (A.71.9 on page 402) is deprecated. It is recommended that all instructions that reference the Y register be avoided.

| Opcode | op3 | rs1 | Operation |
|---|---|---|---|
| RDY[D] | 10 1000 | 0 | Read Y Register |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| rd | %y, $reg_{rd}$ |

*Description* This instruction reads the Y register into r[rd].

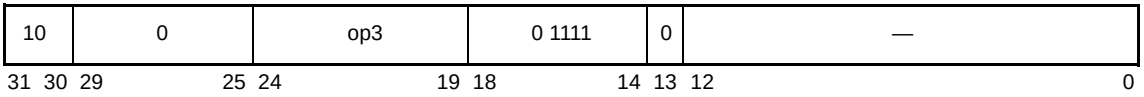
*Exceptions* None

# A.71.10 Store Barrier

The STBAR instruction is deprecated. Use the MEMBAR instruction instead.

| Opcode | op3 | Operation |
|---|---|---|
| STBAR[D] | 10 1000 | Store Barrier |

*Format (3)*




| Assembly Language Syntax |
|---|
| stbar |

*Description* The store barrier instruction (STBAR) forces *all* store and atomic load-store operations issued by a virtual processor prior to the STBAR to complete their effects on memory before *any* store or atomic load-store operations issued by that virtual processor subsequent to the STBAR are executed by memory.

**Note**: The encoding of STBAR is identical to that of the RDASR instruction except that rs1 = 15 and rd = 0, and it is identical to that of the MEMBAR instruction except that bit 13 (i) = 0.

**V9 Compatibility Note –** STBAR is identical in function to a MEMBAR instruction with mmask = $8_{16}$. STBAR is retained for compatibility with SPARC V8.

**V9 Compatibility Note –** For correctness, it is sufficient for a virtual processor to stop issuing new store and atomic load-store operations when an STBAR is encountered and to resume after all stores have completed and are observed in memory by all virtual processors. More efficient implementations may take advantage of the fact that the virtual processor is allowed to issue store and load-store operations after the STBAR, as long as those operations are guaranteed not to become visible before all the earlier stores and atomic load-stores have become visible to all virtual processors.

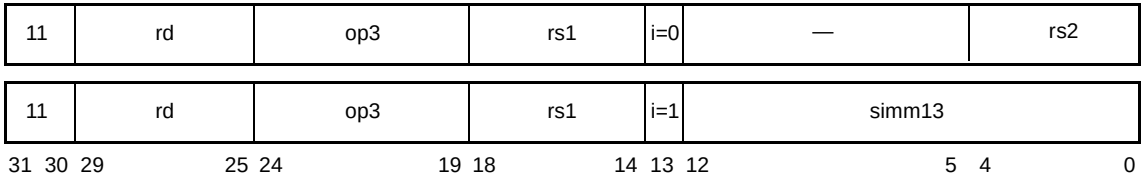
*Exceptions* None

# A.71.11 Store Floating-Point Status Register Lower

The STFSR instruction is deprecated. Use the STXFSR instruction instead.

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| STFSR$^{D}$ | 10 0101 | 0 | Store Floating-Point State Register Lower |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| st | %fsr, [*address*] |

*Description*

The store floating-point state register lower instruction (STFSR) waits for any currently executing FPop instructions to complete, and then it writes the less significant 32 bits of the FSR into memory.

**V9 Compatibility Note –** SPARC V9 needs two store-FSR instructions, since the SPARC V8 STFSR instruction is defined to store only 32 bits of the FSR into memory. STXFSR allows SPARC V9 programs to store all 64 bits of the FSR.

STFSR zeroes FSR.ftt after writing the FSR to memory.

**V9 Compatibility Note –** FSR.ftt should not be zeroed until it is known that the store will not cause a precise trap.

The effective address for this instruction is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

STFSR causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned.

*Exceptions*

*illegal_instruction* (op3 = $25_{16}$ and rd = 2–31)
*fp_disabled*
*mem_address_not_aligned*

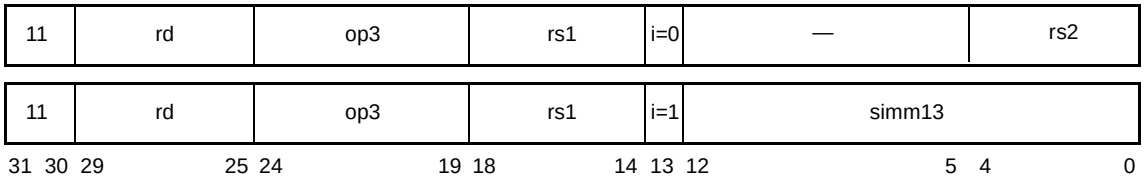
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.71.12 Store Integer Doubleword

The STD instruction is deprecated. Use the STX instruction instead.

| Opcode | op3 | Operation |
|---|---|---|
| STD[D] | 00 0111 | Store Doubleword |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| std | *reg*$_{rd}$, [*address*] |

*Description*

The store doubleword integer instruction (STD) copies two words from an r register pair into memory. The least significant 32 bits of the even-numbered r register are written into memory at the effective address, and the least significant 32 bits of the following odd-numbered r register are written into memory at the "effective address + 4."

The least significant bit of the rd field of a store doubleword instruction is unused and should always be set to 0 by software. An attempt to execute a store doubleword instruction that refers to a misaligned (odd-numbered) rd causes an *illegal_instruction* exception.

The effective address for this instruction is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

A successful store doubleword instruction operates atomically.

STD causes a *mem_address_not_aligned* exception if the effective address is not doubleword aligned.

**IMPL. DEP. #108a:** It is implementation dependent whether STD is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_STD* exception.

With respect to little-endian memory, a STD instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.

**Programming Notes –** STD is provided for compatibility with SPARC V8. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties. Therefore, software should avoid using STD.

If STD is emulated in software, STX should be used to preserve atomicity.

*Exceptions*

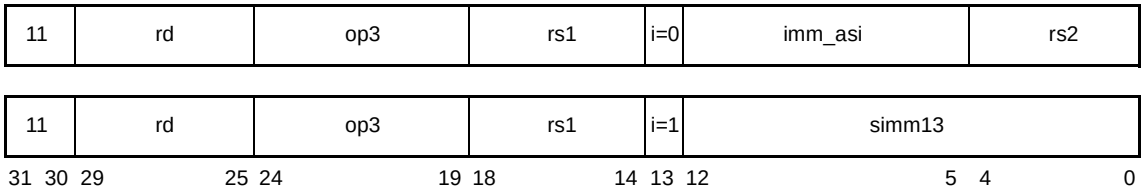
*illegal_instruction* (STD with odd *rd*)
*unimplemented_STD*
*mem_address_not_aligned* (all except STB)
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.71.13 Store Integer Doubleword into Alternate Space

The STDA instruction is deprecated. Instead, use the STXA instruction.

| Opcode | op3 | Operation |
|---|---|---|
| STDA[D, PASI] | 01 0111 | Store Doubleword into Alternate Space |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| stda | *reg*$_{rd}$, [*reg_plus_imm*] %asi |

*Description*

The store doubleword integer instruction (STDA) copies two words from an r register pair into memory. The least significant 32 bits of the even-numbered r register are written into memory at the effective address, and the least significant 32 bits of the following odd-numbered r register are written into memory at the "effective address + 4."

The least significant bit of the rd field of a store doubleword instruction is unused and should always be set to 0 by software. An attempt to execute a store doubleword instruction that refers to a misaligned (odd-numbered) rd causes an *illegal_instruction* exception.

Store integer doubleword to alternate space instructions contain the address space identifier (ASI) to be used for the store in the imm_asi field if i = 0, or in the ASI register if i = 1. The access is privileged if bit 7 of the asi is 0; otherwise, it is not privileged. The effective address for these instructions is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

A successful store doubleword instruction operates atomically.

STDA causes a *mem_address_not_aligned* exception if the effective address is not doubleword aligned.

A store integer into alternate space instruction causes a *privileged_action* exception if PSTATE.priv = 0 and bit 7 of the ASI is 0.

With respect to little-endian memory, a STDA instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.

**IMPL. DEP. #108b:** It is implementation dependent whether STDA is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_STD* exception.

Nontranslating ASIs should only be written using STXA (not STDA) instructions. Although writing to a nontranslating ASI using STDA will work on a JPS2 processor, use of STDA is discouraged because it may not work on non-JPS2 SPARC V9 processors or on future JPS processors.

**IMPL. DEP. #301:** If an STDA referencing a nontranslating ASI is executed, it is implementation dependent whether the STDA executes without error or generates a *data_access_exception* exception.

---

**Programming Note –** STDA is provided for compatibility with SPARC V8. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties. Therefore, software should avoid using STDA.

If STDA is emulated in software, STXA should be used to preserve atomicity.

---

*Exceptions*

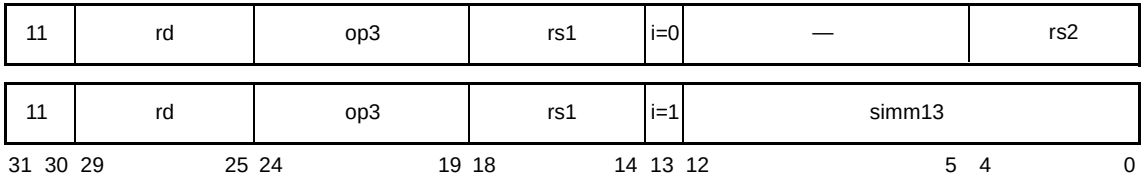
*illegal_instruction* (STDA with odd rd)
*privileged_action*
*mem_address_not_aligned*
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.71.14 Swap Register with Memory

The SWAP instruction is deprecated. Use the CASA or CASXA instruction in its place.

| Opcode | op3 | Operation |
|---|---|---|
| SWAP[D] | 00 1111 | Swap Register with Memory |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| swap | [*address*], *reg*$_{rd}$ |

*Description*

SWAP exchanges the less significant 32 bits of r[rd] with the contents of the word at the addressed memory location. The upper 32 bits of r[rd] are set to 0. The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA instructions addressing any or all of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

The effective address for these instructions is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1. This instruction causes a *mem_address_not_aligned* exception if the effective address is not word aligned.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120).

Accesses to a noncacheable page generates a *data_access_exception*.

**V9 Compatibility Note –** See *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, a document available from SPARC International, for information on the presence of hardware support for these instructions in the various SPARC V9 implementations.

*Exceptions*

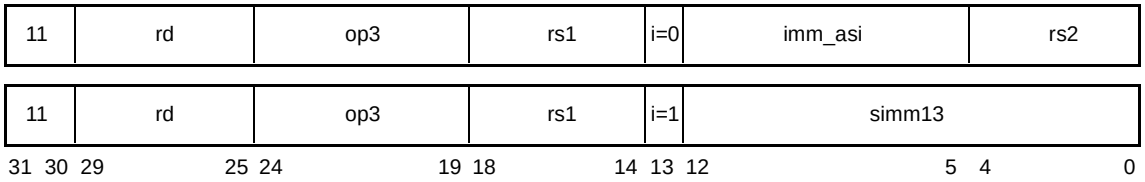
*mem_address_not_aligned*
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.71.15 Swap Register with Alternate Space Memory

The SWAPA instruction is deprecated. Use the CASXA instruction instead.

| Opcode | op3 | Operation |
|---|---|---|
| SWAPA$^{D, P_{ASI}}$ | 01 1111 | Swap register with Alternate Space Memory |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| swapa | [*regaddr*] *imm_asi*, *reg$_{rd}$* |
| swapa | [*reg_plus_imm*] %asi, *reg$_{rd}$* |

*Description*

SWAPA exchanges the less significant 32 bits of r[rd] with the contents of the word at the addressed memory location. The upper 32 bits of r[rd] are set to 0. The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA instructions addressing any or all of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

The SWAPA instruction contains the address space identifier (ASI) to be used for the load in the imm_asi field if i = 0, or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for this instruction is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

This instruction causes a *mem_address_not_aligned* exception if the effective address is not word aligned. It causes a *privileged_action* exception if PSTATE.priv = 0 and bit 7 of the ASI is 0.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep #120).

Accesses to a noncacheable page generates a *data_access_exception*.

**V9 Compatibility Note –** See *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, a document available from SPARC International, for information on the presence of hardware support for this instruction in the various SPARC V9 implementations.

*Exceptions*

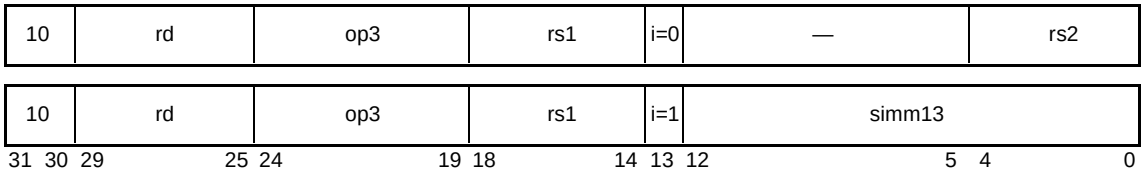
*mem_address_not_aligned*
*privileged_action*
*data_access_exception*
*data_access_error*
*fast_data_access_MMU_miss*
*fast_data_access_protection*
*PA_watchpoint*
*VA_watchpoint*

# A.71.16 Tagged Add and Trap on Overflow

The TADDccTV instruction is deprecated. Use the TADDcc followed by BPVS instruction (with instructions to save the pre-TADDcc integer condition codes if necessary).

| Opcode | op3 | Operation |
|---|---|---|
| TADDccTV[D] | 10 0010 | Tagged Add and modify cc's, or Trap on Overflow |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| taddcctv | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ |

*Description*

This instruction computes a sum that is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

TADDccTV modifies the integer condition codes if it does not trap.

A *tag_overflow* exception occurs if bit 1 or bit 0 of either operand is nonzero or if the addition generates 32-bit arithmetic overflow (that is, both operands have the same value in bit 31 and bit 31 of the sum is different).

If TADDccTV causes a tag overflow, a *tag_overflow* exception is generated and r[rd] and the integer condition codes remain unchanged. If a TADDccTV does not cause a tag overflow, the sum is written into r[rd] and the integer condition codes are updated. ccr.icc.v is set to 0 to indicate no 32-bit overflow.

In either case, the remaining integer condition codes (both the other ccr.icc bits and all the ccr.xcc bits) are also updated as they would be for a normal ADD instruction. In particular, the setting of the ccr.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). ccr.xcc.v is set, based only on the normal 64-bit arithmetic overflow condition, like a normal 64-bit add.

**JPS Compatibility Note –** `TADDccTV` traps based on the 32-bit overflow condition, just as in SPARC V8. Although the tagged add instructions set the 64-bit condition codes `ccr.xcc`, there is no form of the instruction that traps the 64-bit overflow condition.

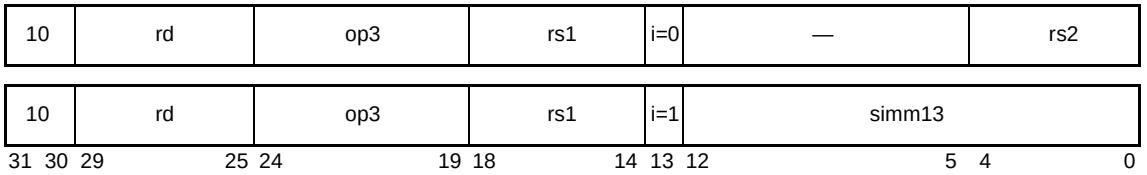
*Exceptions* *tag_overflow*

# A.71.17 Tagged Subtract and Trap on Overflow

The TSUBccTV instruction is deprecated. Use the TSUBcc instruction followed by BPVS (with instructions to save the pre-TSUBcc integer condition codes if necessary).

| Opcode | op3 | Operation |
|---|---|---|
| TSUBccTV$^{D}$ | 10 0011 | Tagged Subtract and modify cc's, or Trap on Overflow |

*Format (3)*




| Assembly Language Syntax |
|---|
| tsubcctv *reg$_{rs1}$, reg_or_imm, reg$_{rd}$* |

*Description*

This instruction computes "r[rs1] – r[rs2]" if i = 0, or "r[rs1] – sign_ext(simm13)" if i = 1.

TSUBccTV modifies the integer condition codes (icc and xcc) if it does not trap.

A tag overflow occurs if bit 1 or bit 0 of either operand is nonzero or if the subtraction generates 32-bit arithmetic overflow; that is, the operands have different values in bit 31 (the 32-bit sign bit) and the sign of the 32-bit difference in bit 31 differs from bit 31 of r[rs1].

If TSUBccTV causes a tag overflow, then a *tag_overflow* exception is generated and r[rd] and the integer condition codes remain unchanged. If a TSUBccTV does not cause a tag overflow condition, the difference is written into r[rd] and the integer condition codes are updated. ccr.icc.v is set to 0 to indicate no 32-bit overflow.

In either case, the remaining integer condition codes (both the other ccr.icc bits and all the ccr.xcc bits) are also updated as they would be for a normal subtract instruction. In particular, the setting of the ccr.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). ccr.xcc.v is set, based only on the normal 64-bit arithmetic overflow condition, like a normal 64-bit subtract.

**V9 Compatibility Note –** `TSUBccTV` traps are based on the 32-bit overflow condition, just as in SPARC V8. Although the tagged-subtract instructions set the 64-bit condition codes `ccr.xcc`, there is no form of the instruction that traps on 64-bit overflow.

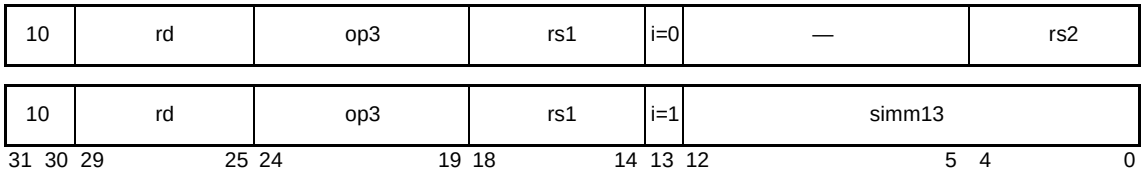
*Exceptions* *tag_overflow*

# A.71.18 Write Y Register

The WRY instruction is deprecated. It is recommended that all instructions that reference the Y register be avoided.

| Opcode | op3 | rd | Operation |
|---|---|---|---|
| WRY[D] | 11 0000 | 0 | Write Y register |
| — | 11 0000 | 1–31 | See *Write State Register* on page 380 |

*Format (3)*




| Assembly Language Syntax | |
|---|---|
| wr | *reg*$_{rs1}$, *reg_or_imm*, %y |

*Description* This instructions stores the value "r[rs1] **xor** r[rs2]" if i = 0, or "r[rs1] **xor** sign_ext(simm13)" if i = 1, to the writable fields of the Y register. **Note**: The operation is exclusive-or.

The WRY instruction is *not* a delayed-write instruction. The instruction immediately following a WRY observes the new value of the Y register.

*Exceptions* None

# IEEE Std 754-1985 Requirements for SPARC V9

The IEEE Std 754-1985 floating-point standard contains a number of implementation dependencies. This appendix specifies choices for these implementation dependencies, to ensure that SPARC V9 implementations are as consistent as possible.

The appendix contains these major sections:


- *Traps Inhibiting Results* on page 419
- *NaN Operand and Result Definitions* on page 420
- *Trapped Underflow Definition (ufm = 1)* on page 422
- *Untrapped Underflow Definition (ufm = 0)* on page 422
- *Integer Overflow Definition* on page 423
- *Floating-Point Nonstandard Mode* on page 424


Exceptions are discussed in this appendix on the assumption that instructions are implemented in hardware. If an instruction is implemented in software, it may not trigger hardware exceptions but its behavior as observed by nonprivileged software (other than timing) must be the same as if it was implemented in hardware.

## B.1 Traps Inhibiting Results

As described in *Floating-Point State Register (FSR)* on page 53 and elsewhere, when a floating-point trap occurs, the following conditions are true:

- The destination floating-point register(s) (the `F` registers) are unchanged.
- The floating-point condition codes (`fcc0`, `fcc1`, `fcc2`, and `fcc3`) are unchanged.
- The `FSR.aexc` (accrued exceptions) field is unchanged.

- The FSR.cexc (current exceptions) field is unchanged except for IEEE_754_exceptions; in that case, cexc contains a bit set to 1, corresponding to the exception that caused the trap. Only one bit shall be set in cexc.

Instructions causing an *fp_exception_other* trap because of unfinished or unimplemented FPops execute as if by hardware; that is, a trap is undetectable by user software, except that timing may be affected. A user-mode trap handler invoked for an IEEE_754_exception, whether as a direct result of a hardware *fp_exception_ieee_754* trap or as an indirect result of supervisor handling of an *fp_exception_other* trap with FSR.ftt = unfinished_FPop or FSR.ftt = unimplemented_FPop, can rely on the following behavior:

- The address of the instruction that caused the exception will be available.
- The destination floating-point register(s) are unchanged from their state prior to that instruction's execution.
- The floating-point condition codes (fcc0, fcc1, fcc2, and fcc3) are unchanged.
- The FSR aexc field is unchanged.
- The FSR cexc field contains exactly one bit set to 1, corresponding to the exception that caused the trap.
- The FSR ftt, qne, and reserved fields are zero.

# B.2 NaN Operand and Result Definitions

An untrapped floating-point result can be in a format that is either the same as, or different from, the format of the source operands. These two cases are described separately below.

## B.2.1 Untrapped Result in Different Format from Operands

- **F[sdq]TO[sdq] with a quiet NaN operand** — No exception caused; result is a quiet NaN. The operand is transformed as follows:

  **NaN transformation**: The most significant bits of the operand fraction are copied to the most significant bits of the result fraction. In conversion to a narrower format, excess low-order bits of the operand fraction are discarded. In conversion to a wider format, excess low-order bits of the result fraction are set to 0. The quiet bit (the most significant bit of the result fraction) is always set to 1, so the NaN transformation always produces a quiet NaN. The sign bit is copied from the operand to the result without modification.

- **`F[sdq]TO[sdq]` with a signalling NaN operand** — Invalid exception; result is the signalling NaN operand processed by the NaN transformation above to produce a quiet NaN.
- **`FCMPE[sdq]` with any NaN operand** — Invalid exception; the selected floating-point condition code is set to unordered.
- **`FCMP[sdq]` with any signalling NaN operand** — Invalid exception; the selected floating-point condition code is set to unordered.
- **`FCMP[sdq]` with any quiet NaN operand but no signalling NaN operand** — No exception; the selected floating-point condition code is set to unordered.

## B.2.2 Untrapped Result in Same Format as Operands

- **No NaN operand** — For an invalid operation such as **sqrt**(–1.0) or 0.0 ÷ 0.0, the result is the quiet NaN with sign = zero, exponent = all ones, and fraction = all ones. The sign is zero to distinguish such results from storage initialized to all ones.
- **One operand, a quiet NaN** — No exception; result is the quiet NaN operand.
- **One operand, a signalling NaN** — Invalid exception; result is the signalling NaN with its quiet bit (most significant bit of fraction field) set to 1.
- **Two operands, both quiet NaNs** — No exception; result is the `rs2` (second source) operand.
- **Two operands, both signalling NaNs** — Invalid exception; result is the `rs2` operand with the quiet bit set to 1.
- **Two operands, only one is a signalling NaN** — Invalid exception; result is the signalling NaN operand with the quiet bit set to 1.
- **Two operands, neither is a signalling NaN, only one is a quiet NaN** — No exception; result is the quiet NaN operand.

In TABLE B-1, NaN*n* means that the NaN is in `rs`*n*, Q means quiet, S signalling.

**TABLE B-1** Untrapped Floating-Point Results

| | | rs2 Operand | | |
|---|---|---|---|---|
| | | **Number** | **QNaN2** | **SNaN2** |
| **rs1 Operand** | **None** | IEEE 754 | QNaN2 | QSNaN2 |
| | **Number** | IEEE 754 | QNaN2 | QSNaN2 |
| | **QNaN1** | QNaN1 | QNaN2 | QSNaN2 |
| | **SNaN1** | QSNaN1 | QSNaN1 | QSNaN2 |

QSNaN*n* means a quiet NaN produced by the *NaN transformation* on a signalling NaN from `rs`*n*; the invalid exception is always indicated. The QNaN*n* results in the table never generate an exception, but IEEE 754 specifies several cases of invalid exceptions, and QNaN results from operands that are both numbers.

## B.3 Trapped Underflow Definition (`ufm` = 1)

A SPARC JPS2 virtual processor detects tininess before rounding occurs. (impl. dep. #55)

Since tininess is detected before rounding, trapped underflow occurs when the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format.

---

**Note –** The wrapped exponent results intended to be delivered on trapped underflows and overflows in IEEE 754 are irrelevant to SPARC V9 at the hardware and supervisor software levels. If they are created at all, it would be by user software in a user-mode trap handler.

---

## B.4 Untrapped Underflow Definition (`ufm` = 0)

On an implementation that detects tininess before rounding, untrapped underflow occurs when the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format *and* the correctly rounded result in the destination format is inexact.

TABLE B-2 summarizes what happens on an implementation that detects tininess before rounding, when an exact *unrounded* value *u* satisfying

$$0 \leq |u| \leq \textit{smallest normalized number}$$

would round, if no trap intervened, to a *rounded* value *r* which might be zero, subnormal, or the smallest normalized value.

In an implementation that detects tininess after rounding, TABLE B-2 applies to a narrower range of values of the exact unrounded result *u*. The precise bounds depend on the rounding direction specified in `FSR.rd`, as follows:

**TABLE B-2** Untrapped Floating-Point Underflow (Tininess Detected Before Rounding)

| | Underflow trap: Inexact trap: | ufm = 1 nxm = x | ufm = 0 nxm = 1 | ufm = 0 nxm = 0 |
|---|---|---|---|---|
| *u* = *r* | *r* is minimum normal | None | None | None |
| | *r* is subnormal | UF | None | None |
| | *r* is zero | None | None | None |
| *u* ≠ *r* | *r* is minimum normal | UF | NX | uf nx |
| | *r* is subnormal | UF | NX | uf nx |
| | *r* is zero | UF | NX | uf nx |

UF = *fp_exception_ieee_754* trap with `cexc.ufc` = 1
NX = *fp_exception_ieee_754* trap with `cexc.nxc` = 1
uf = `cexc.ufc` = 1, `aexc.ufa` = 1, no *fp_exception_ieee_754* trap
nx = `cexc.nxc` = 1, `aexc.nxa` = 1, no *fp_exception_ieee_754* trap

- Let *m* denote the smallest normalized number and *e* the absolute difference between 1 and the next larger representable number in the destination format. Then the bounds on *u* for which TABLE B-2 applies are:

**TABLE B-3** Range of Values of *u* for which TABLE B-2 Applies, if Tininess is Detected After Rounding

| FSR.RD | Round Toward | Range of Values of *u* for which TABLE B-2 applies |
|---|---|---|
| 0 | Nearest (even, if tie) | $\|u\| < m(1 - e/4)$ |
| 1 | 0 | $\|u\| < m$ |
| 2 | + ∞ | $-m < u \leq m(1 - e/2)$ |
| 3 | – ∞ | $-m(1 - e/2) \leq u < m$ |

- When *u* lies outside these ranges, underflow does not occur. However, an *fp_exception_ieee_754* exception with `cexc.nxc` = 1 still occurs when $u \neq r$ (where *r* is the rounded value).

# B.5 Integer Overflow Definition

- **`F[sdq]TOi`** — When a NaN, infinity, large positive argument ≥ 2147483648.0 or large negative argument ≤ –2147483649.0 is converted to an integer, the invalid_current (`nvc`) bit of `FSR.cexc` should be set and *fp_exception_IEEE_754* should be raised. If the floating-point invalid trap is disabled (`FSR.tem.nvm` = 0),

no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is 2147483647; if the sign bit of the operand is 1, the result is –2147483648.

- **`F[sdq]TOx`** — When a NaN, infinity, large positive argument $\geq 2^{63}$, or large negative argument $\leq -(2^{63} + 1)$ is converted to an extended integer, the invalid_current (`nvc`) bit of `FSR.cexc` should be set and *fp_exception_IEEE_754* should be raised. If the floating-point invalid trap is disabled (`FSR.tem.nvm` = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is $2^{63} - 1$; if the sign bit of the operand is 1, the result is $-2^{63}$.

## B.6 Floating-Point Nonstandard Mode

Please refer to *FSR_nonstandard_fp (ns) (Impl. Dep. #18)* on page 55 for information.

# Implementation Dependencies

This appendix summarizes implementation dependencies in the SPARC V9 standard. In SPARC V9, the notation "**IMPL. DEP. #*nn*:**" identifies the definition of an implementation dependency; the notation "(impl. dep. #*nn*)" identifies a reference to an implementation dependency. These dependencies are described by their number *nn* in TABLE C-1 on page 427.

The appendix contains these sections:


- *Definition of an Implementation Dependency* on page 426
- *Hardware Characteristics* on page 426
- *Implementation Dependency Categories* on page 427
- *List of Implementation Dependencies* on page 427


**Note –** SPARC International maintains a document, *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, that describes the implementation-dependent design features of all SPARC V9-compliant implementations. Contact SPARC International for this document at:
home page: `www.sparc.org`
email: info@sparc.org

# C.1 Definition of an Implementation Dependency

The SPARC V9 architecture is a *model* that specifies unambiguously the behavior observed by *software* on SPARC V9 systems. Therefore, it does not necessarily describe the operation of the *hardware* of any actual implementation.

An implementation is *not* required to execute every instruction in hardware. An attempt to execute a SPARC V9 instruction that is not implemented in hardware generates a trap. Whether an instruction is implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.

The two levels of SPARC V9 compliance are described in *SPARC V9 Compliance* on page 8.

Some elements of the architecture are defined to be implementation dependent. These elements include certain registers and operations that may vary from implementation to implementation; they are explicitly identified as such in this appendix.

Implementation elements (such as instructions or registers) that appear in an implementation but are not defined in this document (or its updates) are not considered to be SPARC V9 elements of that implementation.

# C.2 Hardware Characteristics

Hardware characteristics that do not affect the behavior observed by software on SPARC V9 systems are not considered architectural implementation dependencies. A hardware characteristic may be relevant to the user system design (for example, the speed of execution of an instruction) or may be transparent to the user (for example, the method used for achieving cache consistency). The SPARC International document, *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, provides a useful list of these hardware characteristics, along with the list of implementation-dependent design features of SPARC V9-compliant implementations.

In general, hardware characteristics deal with

- Instruction execution speed
- Whether instructions are implemented in hardware
- The nature and degree of concurrency of the various hardware units constituting a SPARC V9 implementation

## C.3 Implementation Dependency Categories

Many of the implementation dependencies can be grouped into four categories, abbreviated by their first letters throughout this appendix:

- **Value (v)**
  The semantics of an architectural feature are well defined, except that a value associated with the feature may differ across implementations. A typical example is the number of implemented register windows (impl. dep. #2).
- **Assigned Value (a)**
  The semantics of an architectural feature are well defined, except that a value associated with the feature may differ across implementations and the actual value is assigned by SPARC International. Typical examples are the `impl` field of Version register (`VER`) (impl. dep. #13) and the `FSR.ver` field (impl. dep. #19).
- **Functional Choice (f)**
  The SPARC V9 architecture allows implementors to choose among several possible semantics related to an architectural function. A typical example is the treatment of a catastrophic error exception, which may cause either a deferred or a disrupting trap (impl. dep. #31).
- **Total Unit (t)**
  The existence of the architectural unit or function is recognized, but details are left to each implementation. Examples include the handling of I/O registers (impl. dep. #7) and some alternate address spaces (impl. dep. #29).

## C.4 List of Implementation Dependencies

TABLE C-1 provides a complete list of the SPARC V9 implementation dependencies. The Page column lists the page for the context in which the dependency is defined.

**TABLE C-1** SPARC V9 Implementation Dependencies *(1 of 8)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| 1 | f | **Software emulation of instructions**<br>Whether an instruction is implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent. | 8, 52, 123, 124 |

**TABLE C-1** SPARC V9 Implementation Dependencies *(2 of 8)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| 2 | v | **Number of IU registers**<br>An implementation of the IU may contain from 64 to 528 general-purpose 64-bit `R` registers. This corresponds to a grouping of the registers into two sets of eight global `R` registers, plus a circular stack of from 3 to 32 sets of 16 registers each, known as register windows. Since the number of register windows present (`NWINDOWS`) is implementation dependent, the total number of registers is also implementation dependent. | 22 |
| 3 | f | **Incorrect IEEE Std 754-1985 results**<br>An implementation may indicate that a floating-point instruction did not produce a correct IEEE Std 754-1985 result by generating an *fp_exception_other* exception with `FSR.ftt` = unfinished_FPop or unimplemented FPop. In this case, privileged mode software shall emulate any functionality not present in the hardware. | 122 |
| 4-5 | | *Reserved.* | |
| 6 | f | **I/O registers privileged status**<br>Whether I/O registers can be accessed by nonprivileged code is implementation dependent. | 25 |
| 7 | t | **I/O register definitions**<br>The contents and addresses of I/O registers are implementation dependent. | 25 |
| 8 | t | **`RDASR/WRASR` target registers**<br>Software can use read/write ancillary state register instructions to read/write implementation-dependent virtual processor registers (ASRs 16–31). | 26, 343, 380 |
| 9 | f | **`RDASR/WRASR` privileged status**<br>Whether each of the implementation-dependent read/write ancillary state register instructions (for ASRs 16–31) is privileged is implementation dependent. | 26, 343, 380 |
| 10-12 | | *Reserved.* | |
| 13 | a | **`VER.impl`**<br>`VER.impl` uniquely identifies an implementation or class of software-compatible implementations of the architecture. Values $FFF0_{16}$–$FFFF_{16}$ are reserved and are not available for assignment. | 83 |
| 14-15 | | *Reserved.* | |
| 16 | | *Reserved.* | |
| 17 | | *Reserved.* | |
| 18 | f | **Nonstandard IEEE 754-1985 results**<br>When `FSR.ns` = 1, the FPU produces implementation-dependent results that may not correspond to IEEE Standard 754-1985. | |
| | | **a:** When a floating-point source operand is subnormal, it is replaced by a floating-point zero value of the same sign (instead of causing an exception). The cases in which this replacement is performed are implementation dependent. | 55 |
| | | **b:** When a floating-point operation in a SPARC JPS2 virtual processor generates a subnormal value, that value is replaced with a floating-point zero value of the same sign. The means by which that replacement occurs are implementation dependent. | 55 |

**TABLE C-1** SPARC V9 Implementation Dependencies *(3 of 8)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| 19 | a | **FPU version, `FSR.ver`**<br>Bits 19:17 of the `FSR`, `FSR.ver`, identify one or more implementations of the FPU architecture. | 55 |
| 20-21 | | *Reserved.* | |
| 22 | f | **FPU `tem`, `cexc`, and `aexc`**<br>A JPS2 implementation implements the `tem`, `cexc`, and `aexc` fields in hardware, conformant to IEEE Std 754-1985. | 62 |
| 23 | | *Reserved.* | |
| 24 | | *Reserved.* | |
| 25 | f | *Reserved.* | |
| 26-28 | | *Reserved.* | |
| 29 | t | **Address space identifier (ASI) definitions**<br>In SPARC V9, many ASIs were defined to be implementation dependent. Some of those ASIs have been allocated for standard uses in SPARC JPS2. Others remain implementation dependent in SPARC JPS2. See TABLE L-1 on page 569 and *Block Load and Store ASIs* on page 582 for details. | 110 |
| 30 | f | **ASI address decoding**<br>In SPARC V9, an implementation could choose to decode only a subset of the 8-bit ASI specifier. In SPARC JPS2 implementations, all 8 bits of each ASI specifier must be decoded. Refer to Appendix L, *Address Space Identifiers (ASIs)*, of this specification for details. | 111 |
| 31 | f | **Catastrophic error exceptions**<br>The causes and effects of catastrophic error exceptions (for example, *internal_processor_error*) are implementation dependent. They may cause precise, deferred, or disrupting traps. | 130, 165, 617 |
| 32 | t | **Deferred traps**<br>Whether any deferred traps (and associated deferred-trap queues) are present is implementation dependent. | 135 |
| 33 | f | **Trap precision**<br>Exceptions that occur as the result of program execution are precise in a JPS2 virtual processor. | 137 |
| 34 | f | **Interrupt clearing**<br>The method by which an interrupt result is removed is defined in JPS2, but how quickly a virtual processor responds to an interrupt request remains implementation dependent. | 138 |
| 35 | t | **Implementation-dependent traps**<br>Trap type (`TT`) values $060_{16}$–$07F_{16}$ are reserved for implementation-dependent exceptions. The existence of *implementation_dependent_n* traps and whether any that do exist are precise, deferred, or disrupting is implementation dependent. | 141 |

**TABLE C-1** SPARC V9 Implementation Dependencies *(4 of 8)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| 36 | f | **Trap priorities**<br>The relative priorities of traps defined in SPARC JPS2 are fixed. However, the absolute priorities of those traps are implementation dependent (because a future version of the architecture may define new traps). The priorities (both absolute and relative) of any additional traps are implementation dependent. | 146 |
| 37 | f | **Reset trap**<br>Some of a virtual processor's behavior during a reset trap is implementation dependent. | 136 |
| 38 | f | **Effect of reset trap on implementation-dependent registers**<br>Implementation-dependent registers may or may not be affected by the various reset traps. | 153 |
| 39 | f | **Entering `error_state` on implementation-dependent errors**<br>The virtual processor may enter `error_state` when an implementation-dependent error condition occurs. | 134, 134 |
| 40 | f | **`Error_state`**<br>Effects when `error_state` is entered are implementation dependent, but it is recommended that as much virtual processor state as possible be preserved upon entry to `error_state`. In addition, a SPARC JPS2 virtual processor may have other `error_state` entry traps that are implementation dependent. | 134 |
| 41 | | *Reserved.* | |
| 42 | t, f, v | **`FLUSH` instruction**<br>If `FLUSH` is not implemented in hardware, it causes an *illegal_instruction* exception, and its function is performed by system software. Whether `FLUSH` traps is implementation dependent. | 269 |
| 43 | | *Reserved.* | |
| 44 | f | **Data access FPU trap**<br>**a:** If a load floating-point instruction traps with any type of access error exception, the contents of the destination floating-point register(s) either remain unchanged or are undefined. | 275 (a), |
| | | **b:** If a load floating-point instruction traps with any type of access error, the destination floating-point register(s) remain unchanged or are undefined. | 277 (b) |
| 45 - 46 | | *Reserved.* | |
| 47 | t | **`RDASR`**<br>`RDASR` instructions with `rd` in the range 16–31 are available for implementation-dependent uses (impl. dep. #8). For an `RDASR` instruction with `rs1` in the range 16–31, the following are implementation dependent:<br>• the interpretation of bits 13:0 and 29:25 in the instruction<br>• whether the instruction is privileged (impl. dep. #9)<br>• whether it causes an *illegal_instruction* trap | 343 |

**TABLE C-1** SPARC V9 Implementation Dependencies *(5 of 8)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| 48 | t | **WRASR**<br>WRASR instructions with rd in the range 16–31 are available for implementation-dependent uses (impl. dep. #8). For a WRASR instruction with rd in the range 16–31, the following are implementation dependent:<br>• the interpretation of bits 18:0 in the instruction<br>• the operation(s) performed (for example, **xor**) to generate the value written to the ASR<br>• whether the instruction is privileged (impl. dep. #9)<br>• whether it causes an *illegal_instruction* trap | 381 |
| 49-54 | | *Reserved.* | |
| 55 | f | **Tininess detection**<br>In SPARC V9, it is implementation-dependent whether "tininess" (an IEEE 754 term) is detected before or after rounding. In all SPARC JPS2 implementations, tininess is detected before rounding. | 61, 422 |
| 56-100 | | *Reserved.* | |
| 101 | v | **Maximum trap level**<br>All SPARC JPS2 virtual processors implement MAXTL = 5 (5 trap levels beyond level 0). | 77, 82 |
| 102 | f | **Clean windows trap**<br>An implementation may choose either to implement automatic "cleaning" of register windows in hardware or to generate a *clean_window* trap, when needed, for window(s) to be cleaned by software. | 164 |
| 103 | f | **Prefetch instructions**<br>The following aspects of the PREFETCH and PREFETCHA instructions are implementation dependent:<br>**a:** whether they have an observable effect in privileged code<br>**b:** whether they can cause a *data_access_MMU_miss* exception<br>**c:** the attributes of the block of memory prefetched: its size (minimum = 64 bytes) and its alignment (minimum = 64-byte alignment)<br>**d:** whether each variant is implemented as a NOP, with its full semantics, or with common-case prefetching semantics<br>**e:** whether and how variants 16–31 are implemented<br>**f:** Whether an attempt to reference a restricted ASI ($< 80_{16}$) by a PREFETCHA instruction while in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_action* exception or executes as a NOP is implementation dependent. | 109, 335, 335, 336, 339 |
| 104 | a | **VER.manuf**<br>VER.manuf contains a 16-bit semiconductor manufacturer code. This field is optional and, if not present, reads as zero. VER.manuf may indicate the original supplier of a second-sourced processor in cases involving mask-level second-sourcing. It is intended that the contents of VER.manuf track the JEDEC semiconductor manufacturer code as closely as possible. If the manufacturer does not have a JEDEC semiconductor manufacturer code, then SPARC International will assign a VER.manuf value. | 83 |

**TABLE C-1** SPARC V9 Implementation Dependencies *(6 of 8)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| 105 | f | **TICK register**<br>The difference between the values read from the TICK register on two reads should reflect the number of processor cycles executed between the reads. If an accurate count cannot always be returned, any inaccuracy should be small, bounded, and documented. An implementation may implement fewer than 63 bits in TICK.counter; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any upper bits not implemented must read as 0. | 67 |
| 106 | f | **IMPDEP2A instructions**<br>The IMPDEP2A instructions are completely implementation dependent. Implementation-dependent aspects include their operation, the interpretation of bits 29:25 and 18:0 in their encodings, and which (if any) exceptions they may cause. | 272 |
| 107 | f | **Unimplemented LDD trap**<br>**a:** It is implementation dependent whether LDD is implemented in hardware. If not, an attempt to execute LDD will cause an *unimplemented_LDD* trap. | 395 (a) |
| | | **b:** It is implementation dependent whether LDDA is implemented in hardware. If not, an attempt to execute LDDA will cause an *unimplemented_LDD* exception. | 397 (b) |
| 108 | f | **Unimplemented STD trap**<br>**a:** It is implementation dependent whether STD is implemented in hardware. If not, an attempt to execute STD will cause an *unimplemented_STD* trap. | 406 (a) |
| | | **b:** It is implementation dependent whether STDA is implemented in hardware. If not, an attempt to execute STDA will cause an *unimplemented_STD* exception. | 409 (b) |
| 109 | f | ***LDDF_mem_address_not_aligned***<br>**a:** On a JPS2 virtual processor, if the effective address for LDDF is word aligned but not doubleword aligned, an *LDDF_mem_address_not_aligned* trap occurs. In that case, trap handler software must emulate the LDDF instruction and return. | 52, 106, 275 (a) |
| | | **b:** On a JPS2 virtual processor, if the effective address for LDDFA is word aligned but not doubleword aligned, an *LDDF_mem_address_not_aligned* trap occurs. In that case, trap handler software must emulate the LDDFA instruction and return. | 277 (b) |
| 110 | f | ***STDF_mem_address_not_aligned***<br>**a:** STDF requires only word alignment in memory. However, if the effective address is word aligned but not doubleword aligned, STDF may cause an *STDF_mem_address_not_aligned* trap. In that case, the trap handler software shall emulate the STDF instruction and return. | 106, 361 (a) |
| | | **b:** STDFA requires only word alignment in memory. If the effective address is word aligned but not doubleword aligned, STDFA may cause an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the STDFA instruction and return. | 364 (b) |
| 111 | f | ***LDQF_mem_address_not_aligned***<br>A JPS2 implementation traps and emulates LDQF and LDQFA instructions, so never generates an *LDQF_mem_address_not_aligned* exception. | 106, 275, 277 |
| 112 | f | ***STQF_mem_address_not_aligned***<br>A JPS2 implementation traps and emulates STQF and STQFA instructions, so never generates an *STQF_mem_address_not_aligned* exception. | 106, 361, 364 |

**TABLE C-1** SPARC V9 Implementation Dependencies *(7 of 8)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| 113 | f | **Implemented memory models**<br>Whether the Partial Store Order (PSO) or Relaxed Memory Order (RMO) models are supported is implementation dependent. | 75, 170 |
| 114 | f | **RED_state trap vector address (RSTVaddr)**<br>The RED_state trap vector is located at an implementation-dependent address referred to as RSTVaddr. | 132 |
| 115 | f | **RED_state**<br>SPARC JPS2 defines much of RED_state behavior (see Section 7.5, Appendix F, and Appendix O); any RED_state behavior that is not defined in JPS2 remains implementation dependent. | 132 |
| 116 | f | **SIR_enable control flag**<br>SPARC V9 states that the location of the SIR_enable control flag and the means by which it is accessed are implementation dependent. In SPARC JPS2 virtual processors, the SIR_enable control flag does not explicitly exist; the SIR instruction always behaves like a NOP in nonprivileged mode (that is, as if SIR_enable is permanently set to zero). | 359 |
| 117 | f | **MMU disabled prefetch behavior**<br>Whether PREFETCH and nonfaulting loads always succeed when the MMU is disabled is implementation dependent. | 335, 486 |
| 118 | f | **Identifying I/O locations**<br>The manner in which I/O locations are identified is implementation dependent. | 172 |
| 119 | f | **Unimplemented values for PSTATE.mm**<br>The effect of writing an unimplemented memory-mode designation into PSTATE.mm is implementation dependent. | 75, 182 |
| 120 | f | **Coherence and atomicity of memory operations**<br>The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent. | 172 |
| 121 | f | **Implementation-dependent memory model**<br>An implementation may choose to identify certain addresses and use an implementation-dependent memory model for references to them. | 172 |
| 122 | f | **FLUSH latency**<br>The latency between the execution of FLUSH on one virtual processor and the point at which the modified instructions have replaced outdated instructions in a multiprocessor is implementation dependent. | 184 |
| 123 | f | **Input/output (I/O) semantics**<br>The semantic effect of accessing I/O registers is implementation dependent. | 25 |
| 124 | v | **Implicit ASI when TL > 0**<br>In SPARC V9, when TL > 0, the implicit ASI for instruction fetches, loads, and stores is implementation dependent. In all SPARC JPS2 implementations, when TL > 0, the implicit ASI for instruction fetches is ASI_NUCLEUS; loads and stores will use ASI_NUCLEUS if PSTATE.cle = 0 or ASI_NUCLEUS_LITTLE if PSTATE.cle = 1. | 174 |

**TABLE C-1** SPARC V9 Implementation Dependencies *(8 of 8)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| 125 | f | **Address masking** When PSTATE.am = 1, the value of the high-order 32-bits of the PC transmitted to the specified destination register(s) by CALL, JMPL, RDPC, and on a trap is implementation dependent. | 76 |
| 126 | — | *Reserved.* | — |
| 127-199 | — | *Reserved.* | — |

TABLE C-2 provides a list of implementation dependencies that, in addition to those in TABLE C-1, apply to SPARC JPS2 processors. See Appendix C in the JPS2 Extensions documents for further information.

**TABLE C-2** SPARC JPS2 Implementation Dependencies *(1 of 8)*

| Nbr | Description | Page |
|---|---|---|
| 200-201 | *Reserved.* | — |
| 202 | ***fast_ECC_error* trap** Whether or not a *fast_ECC_error* trap exists is implementation dependent. If it does exist, it indicates that an ECC error was detected in an external cache and its trap type is $070_{16}$. | 167, 617 |
| 203 | **Dispatch Control Register bits 13:6 and 1** The values and semantics of bits 13:6 and bit 1 of the Dispatch Control Register are implementation dependent. | 87, 602 |
| 204 | **DCR bits 5:3 and 0** The existence, values, and semantics of DCR bits 5:3 and 0 are implementation dependent. If each is implemented, standard (recommended) semantics are as described in Section 5.2.3. If not implemented, each bit reads as 0 and writes to it are ignored. | 87, 87, 602 |
| 205 | **Instruction Trap Register** The presence of the Instruction Trap Register in a SPARC JPS2 virtual processor is implementation dependent. If implemented, the standard (recommended) implementation is described in *Instruction Trap Register* on page 96. | 96 |
| 206 | **SHUTDOWN instruction** It is implementation dependent whether SHUTDOWN acts as described in A.59 or whether in privileged mode it acts as a NOP in a given implementation. | 358 |
| 207 | **PCR register bits 47:32, 26:17, and 3** The values and semantics of bits 47:32, 26:17, and bit 3 of the PCR register are implementation dependent. | 68 |
| 208 | **Ordering of errors captured in instruction execution** The order in which errors are captured in instruction execution is implementation dependent. Ordering may be in program order or in order of detection. | 609 |

**TABLE C-2** SPARC JPS2 Implementation Dependencies *(2 of 8)*

| Nbr | Description | Page |
|---|---|---|
| 209 | **Software intervention after instruction-induced error**<br>If an instruction-induced error requires software intervention for recovery, the precision of the trap used to signal the error is implementation dependent. | 610 |
| 210 | **ERROR output signal**<br>The following aspects of the ERROR output signal are implementation dependent in SPARC JPS2:<br>• The causes of the ERROR signal<br>• Whether each of the causes of the ERROR signal, when it generates the ERROR signal, halts the virtual processor or allows the virtual processor to continue running<br>• The exact semantics of the ERROR signal | 611 |
| 211 | **Error logging registers' information**<br>The information that the error logging registers preserves beyond the reset induced by an ERROR signal is implementation dependent. | 611 |
| 212 | **Trap with fatal error**<br>Generation of a trap along with assertion of an ERROR signal upon detection of a fatal error is implementation dependent. | 612 |
| 213 | **`AFSR.priv`**<br>The existence of the `AFSR.priv` bit is implementation dependent. If `AFSR.priv` is implemented, it is implementation dependent whether the logged `AFSR.priv` indicates the privileged state upon the detection of an error or upon the execution of an instruction that induces the error. For the former implementation to be effective, operating software must provide error barriers appropriately. | 613 |
| 214 | **Enable/disable control for deferred traps**<br>Whether an implementation provides an enable/disable control feature for deferred traps is implementation dependent. | 613 |
| 215 | **Error barrier**<br>`DONE` and `RETRY` instructions may implicitly provide an error barrier function as `MEMBAR #Sync`. Whether `DONE` and `RETRY` instructions provide an error barrier is implementation dependent. | 613 |
| 216 | ***data_access_error* trap precision**<br>The precision of a *data_access_error* trap is implementation dependent. | 617 |
| 217 | ***instruction_access_error* trap precision**<br>The precision of an *instruction_access_error* trap is implementation dependent. | 617 |
| 218 | ***async_data_error***<br>Whether *async_data_error* exception is implemented is implementation dependent. If it does exist, it indicates that an error is detected in a virtual processor and its trap type is $40_{16}$. | 165, 617 |
| 219 | **Asynchronous Fault Address Register (`AFAR`)** allocation<br>Allocation of Asynchronous Fault Address Register (`AFAR`) is implementation dependent. There may be one instance or multiple instances of `AFAR`. Although the ASI for `AFAR` is defined as $4D_{16}$, the virtual address of `AFAR` if there are multiple `AFAR`s is implementation dependent. | 618 |

**TABLE C-2** SPARC JPS2 Implementation Dependencies *(3 of 8)*

| Nbr | Description | Page |
|---|---|---|
| 220 | **Addition of logging and control registers for error handling**<br>Whether the implementation supports additional logging and control registers for error handling is implementation dependent. | 618 |
| 221 | **Special/signalling ECCs**<br>The method to generate "special" or "signalling" ECCs and whether a processor ID is embedded into the data associated with special/signalling ECCs is implementation dependent. | 618 |
| 222 | **TLB organization**<br>TLB organization is implementation dependent in JPS2 processors. | 468 |
| 223 | **TLB multiple-hit detection**<br>Whether TLB multiple-hit detection is supported in JPS2 is implementation dependent. | 469 |
| 224 | **MMU physical address width**<br>Physical address width support by the MMU is implementation dependent in JPS2; minimum PA width is 43 bits. | 471 |
| 225 | **TLB locking of entries**<br>The mechanism by which entries in TLB are locked is implementation dependent in JPS2. | 472 |
| 226 | **TTE support for cv bit**<br>Whether the cv bit is supported in TTE is implementation dependent in JPS2. When the cv bit in TTE is not provided and the implementation has virtually indexed caches, the implementation should support hardware unaliasing for the caches. See also #232. | 472 |
| 227 | **TSB number of entries**<br>The maximum number of entries in a TSB is implementation dependent in JPS2. | 474 |
| 228 | **tsb_hash supplied from TSB or context-ID register**<br>Whether tsb_hash is supplied from a TSB extension register or from a context-ID register is implementation dependent in JPS2. Only for cases of direct hash with context-ID can the width of the tsb_size field be wider than 3 bits. | 476 |
| 229 | **tsb_base address generation**<br>Whether the implementation generates the tsb_base address by exclusive-ORing the TSB_Base register and a TSB register or by taking tsb_base field directly from TSB register is implementation dependent in JPS2. This implementation dependency is only to maintain compatibility with the TLB miss handling software of UltraSPARC I/II. | 476 |
| 230 | ***data_access_exception* trap**<br>The causes of a *data_access_exception* trap are implementation dependent in JPS2, but there are several mandatory causes of *data_access_exception* trap. | 480 |
| 231 | **MMU physical address variability**<br>The variability of the width of the physical address is implementation dependent in JPS2, and if variable, the initial width of the physical address after reset is also implementation dependent in JPS2. | 486 |
| 232 | **DCU Control Register bits**<br>Whether cp and cv bits exist in the DCU Control Register is implementation dependent in JPS2. See also #226. | 92, 486 |

**TABLE C-2** SPARC JPS2 Implementation Dependencies *(4 of 8)*

| Nbr | Description | Page |
|---|---|---|
| 233 | **`tsb_hash` field**<br>Whether `tsb_hash` field is implemented in I/D Primary/Secondary/Nucleus TSB Extension Register is implementation dependent in JPS2. | 489, 496 |
| 234 | **TLB replacement algorithm**<br>The replacement algorithm for a TLB entry is implementation dependent in JPS2. | 492 |
| 235 | **TLB data access address assignment**<br>The MMU TLB data access address assignment and the purpose of the address are implementation dependent in JPS2. | 493 |
| 236 | **`tsb_size` field width**<br>The width of the `tsb_size` field in the `TSB` Base Register is implementation dependent; the permitted range is from 2 to 6 bits. The least significant bit of `tsb_size` is always at bit 0 of the `TSB` register. Any bits unimplemented at the most significant end of `TSB_Size` read as 0, and writes to them are ignored. | 495, 498 |
| 237 | **`JMPL`/`RETURN` *mem_address_not_aligned***<br>Whether the fault status and/or address (`DSFSR`/`DSFAR`) are captured when a *mem_address_not_aligned* trap occurs during a `JMPL` or `RETURN` instruction is implementation dependent. | 481 |
| 238 | **TLB page offset for large page sizes**<br>When page offset bits for larger page sizes (`pa`<15:13>, `pa`<18:13>, and `pa`<21:13> for 64-Kbyte, 512-Kbyte, and 4-Mbyte pages, respectively) are stored in the TLB, it is implementation dependent whether the data returned from those fields by a Data Access read are zero or the data previously written to them. | 471 |
| 239 | **Register access by ASIs $55_{16}$ and $5D_{16}$**<br>The register(s) accessed by IMMU ASI $55_{16}$ and DMMU ASI $5D_{16}$ at virtual addresses $40000_{16}$ to $60FF8_{16}$ are implementation dependent. | 489, 573 |
| 240 | **DCU Control Register bits 47:41**<br>The presence and semantics of bits 47:41 of `DCUCR` are implementation dependent. If any of these bits is not implemented, it reads as 0 and writes to it are ignored. | 92, 602 |
| 241 | **Address masking and `DSFAR`**<br>When `PSTATE.am` = 1 and an exception occurs, the value written to the more-significant 32 bits of the Data Synchronous Fault Address Register (`DSFAR`) is implementation dependent. | 76 |
| 242 | **TLB lock bit**<br>An implementation containing multiple TLBs may implement the `L` (lock) bit in all TLBs but is only required to implement a lock bit in one TLB for each page size. If the lock bit is not implemented in a particular TLB, it reads as 0 and writes to it are ignored. | 472 |
| 243 | **Interrupt Vector Dispatch Status Register BUSY/NACK pairs**<br>The number of BUSY/NACK bit pairs implemented in the Interrupt Vector Dispatch Status Register is implementation dependent. | 593 |

**TABLE C-2** SPARC JPS2 Implementation Dependencies *(5 of 8)*

| Nbr | Description | Page |
|---|---|---|
| 244 | **Data watchpoint reliability**<br>Implementation-dependent feature(s) may be present that degrade the reliability of data watchpoints. If such features are present, it must be possible to disable them such that data watchpoints function as described in this section. Furthermore, those features should be disabled by default. | 95 |
| 245 | **Call/Branch displacement encoding in I-cache**<br>On SPARC JPS2 processors, the encoding of the least significant 11 bits of the displacement field of CALL and branch (BPcc, FBPfcc, Bicc, BPr) instructions in an instruction cache is implementation-dependent. Specifically, those bits' encoding in an instruction cache is not necessarily the same as their architectural encoding (which appears in main memory). | 97 |
| 246 | **VA<38:29> for Interrupt Vector Dispatch Register access**<br>When the Interrupt Vector Dispatch Register is written, the source module identifier (sid) is supplied in va<38:29>. Which, if any, of the 10 va<38:29> bits are interpreted by hardware is implementation dependent. | 592 |
| 247 | **Interrupt Vector Receive Register sid Fields**<br>Which, if any, of the 10 bits of the physical module ID (mid) of the interrupt source is set by hardware in the sid_u and sid_l fields of the Interrupt Vector Receive Register is implementation dependent. Also, the source of the physical module ID (mid) bits is implementation dependent. | 594 |
| 248 | **Conditions for *fp_exception_other* with unfinished_FPop**<br>The conditions under which an *fp_exception_other* exception with floating-point trap type of unfinished_FPop can occur are implementation dependent. An implementation may cause *fp_exception_other* with unfinished_FPop under a different (but specified) set of conditions. | 57 |
| 249 | **Data watchpoint for Partial Store instruction**<br>For a Partial Store instruction, the following aspects of data watchpoints are implementation dependent: (a) whether data watchpoint logic examines the byte store mask in r[rs2] or it conservatively behaves as if every Partial Store always stores all 8 bytes, and (b) whether data watchpoint logic examines individual bits in the Virtual (Physical) Data Watchpoint Mask in DCUCR to determine which bytes are being watched or (when the Watchpoint Mask is nonzero) it conservatively behaves as if all 8 bytes are being watched. | 314 |
| 250 | **PCR accessibility when PSTATE.priv = 0**<br>When the virtual processor is operating in nonprivileged mode (PSTATE.priv = 0), the accessibility of PCR as a unit and of individual fields of PCR is implementation dependent. Also, which exception is raised upon detection of an access privilege violation is implementation dependent. | 67, 68, 121, 218, 345, 382 |
| 251 | *Reserved.* | |

**TABLE C-2** SPARC JPS2 Implementation Dependencies *(6 of 8)*

| Nbr | Description | Page |
|---|---|---|
| 252 | **DCUCR.dc (Data Cache Enable)**<br>The presence of DCUCR bit 1 (DCUCR.dc, Data Cache Enable) is implementation dependent. If dc is not implemented, it reads as zero, writes to it are ignored, and software should only write zero or a value previously read from dc to dc. The remainder of this description assumes that dc is implemented.<br>The function of dc is to enable/disable operation of the data cache closest to the virtual processor (D-cache); dc = 1 enables the D-cache and dc = 0 disables it. When dc = 0, memory accesses (loads, stores, atomic load-stores) are satisfied by caches lower in the cache hierarchy. It is implementation dependent whether or not memory accesses update the D-cache while the D-cache is disabled (dc = 0). If memory accesses do not update the D-cache, then when the D-cache is reenabled (dc is set to 1) any D-cache lines still marked as "valid" may be inconsistent with the state of memory or other caches. In that case, software must handle any inconsistencies by flushing the inconsistent lines from the D-cache. | 94 |
| 253 | **DCUCR.ic (Instruction Cache Enable)**<br>The presence of DCUCR bit 0 (DCUCR.ic, Instruction Cache Enable) is implementation dependent. If ic is not implemented, it reads as zero, writes to it are ignored, and software should only write zero or a value previously read from ic to ic. The remainder of this description assumes that ic is implemented.<br>The function of ic is to enable/disable operation of the instruction cache closest to the virtual processor (I-cache); ic = 1 enables the I-cache and ic = 0 disables it. When ic = 0, instruction fetches are satisfied by caches lower in the cache hierarchy. It is implementation dependent whether or not instruction fetches update the I-cache while the I-cache is disabled (ic = 0). If instruction fetches do not update the I-cache, then when the I-cache is reenabled (ic is set to 1) any I-cache lines still marked as "valid" may be inconsistent with the state of memory or other caches. In that case, software must handle any inconsistencies by invalidating the inconsistent lines in the I-cache. | 94 |
| 254 | **Means of exiting error_state**<br>The means of exiting error_state are implementation dependent. A suggested method is for the virtual processor, upon entering error_state, to automatically generate a *watchdog_reset* (WDR). | 134, 136, 155, 136, 598, 599 |
| 255 | **LDDFA with ASI $E0_{16}$ or $E1_{16}$ and misaligned destination register number**<br>For LDDFA with ASI $E0_{16}$ or $E1_{16}$, if a destination register number rd is specified which is not a multiple of 8 ("misaligned" rd), it is implementation dependent whether the virtual processor generates a *data_access_exception* or *illegal_instruction* exception. | 582 |
| 256 | **LDDFA with ASI $E0_{16}$ or $E1_{16}$ and misaligned memory address**<br>For LDDFA with ASI $E0_{16}$ or $E1_{16}$, if a memory address is specified with less than 64-byte alignment, it is implementation dependent whether the virtual processor generates a *data_access_exception*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned* exception. | 582 |
| 257 | **LDDFA with ASI $C0_{16}$–$C5_{16}$ or $C8_{16}$–$CD_{16}$ and misaligned memory address**<br>For LDDFA with ASI $C0_{16}$–$C5_{16}$ or $C8_{16}$–$CD_{16}$, if a memory address is specified with less than 8-byte alignment, it is implementation dependent whether the virtual processor generates a *data_access_exception*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned* exception. | 583 |

**TABLE C-2** SPARC JPS2 Implementation Dependencies *(7 of 8)*

| Nbr | Description | Page |
|---|---|---|
| 258 | **ASI_SERIAL_ID**<br>The semantics and encoding of ASI_SERIAL_ID are implementation dependent. Its intended use is for a part identification number that is unique to each processor. | 573 |
| 259-299 | *Reserved.* | Reserved. |
| 300 | **Attempted access to ASI registers with LDDA**<br>If an LDDA referencing a non-memory ASI is executed, it is implementation dependent whether the LDDA executes without error or generates a *data_access_exception* exception. | 190, 397 |
| 301 | **Attempted access to ASI registers with STDA**<br>If an STDA referencing a non-memory ASI is executed, it is implementation dependent whether the STDA executes without error or generates a *data_access_exception* exception. | 190, 409 |
| 302 | **Scratchpad registers**<br>Whether Scratchpad registers are present in a SPARC JPS2 virtual processor is implementation dependent | 98 |
| 303 | **GSR.gcc bits**<br>Whether the GSR.gcc bits are implemented is implementation-dependent. If gcc is not implemented, GSR bits 11:8 are reserved. | 70 |
| 304 | **XIR**<br>Whether XIR affects only one virtual processor, or the entire system is implementation-dependent. | 598 |
| 305 | **Updating AFSR/AFAR**<br>The effects of updating AFSR or AFAR are implementation-dependent. | 579 |
| 306 | **Trap type generated upon attempted access to noncacheable page with LDDA**<br>The trap type generated when LDDA is issued to a noncacheable page is implementation dependent. | 284 |
| 307 | **Global register set enabled during normal trap @ TL = MAXTL –1**<br>**a:** When a trap due to *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, *fast_data_access_protection*, *data_access_exception*, or *instruction_access_exception* occurs and the trap is processed in RED_state (that is, either TL = MAXTL – 1 or PSTATE.red = 1), it is implementation-dependent whether PSTATE.ag or PSTATE.mg is set to 1 during the trap.<br>**b:** When an *interrupt_vector* trap occurs and the trap is processed in RED_state (that is, either TL = MAXTL – 1 or PSTATE.red = 1), an implementation may either set PSTATE.ag to 1 or set PSTATE.ig to 1. | 73, 153 |
| 308 | **Reserved Combinations of PSTATE ag, mg, and ig bits**<br>The result of an attempt to write a reserved combination of ag, ig, and mg bit values to the PSTATE register by means other than a WRPR instruction (for example, copying them from TSTATE during a DONE or RETRY instruction) is implementation dependent. | 73, **73**, 249 |
| 309 | **DCU Control Register bits 63:50**<br>The presence and semantics of bits 63:50 of DCUCR are implementation dependent. If any of these bits is not implemented, it reads as 0 and writes to it are ignored. | 92 |

**TABLE C-2** SPARC JPS2 Implementation Dependencies *(8 of 8)*

| Nbr | Description | Page |
|---|---|---|
| 310 | **Large page sizes**<br>A JPS2 implementation may support page sizes greater than 4 Mbytes. If so, a reference to such a page in a TTE or TSB entry is encoded in bits 49:48, which are reserved for that purpose. Which page size(s) over 4 Mbytes are supported and their encoding in TTE and TSB entries are implementation dependent. | 469 |
| 311–1099 | *Reserved.* | |
| 1100 | **Core Interrupt ID Register**<br>Whether any portion of the `int_id` field of the Core Interrupt ID register is read-only is implementation dependent | 193 |
| 1101 | **Interrupt Vector Dispatch Extension Register**<br>Whether a SPARC JPS2 virtual processor implements an Interrupt Vector Dispatch Extension register is implementation dependent | 193 |
| 1102 | **Power used by MTP**<br>Whether disabling a virtual processor reduces the power used by an MTP is implementation dependent. | 197 |
| 1103 | **Updating Core Enable Register**<br>Whether a restriction is provided to protect against all available virtual processors being disabled is implementation dependent. | 199 |
| 1104 | **Parking a Virtual Processor**<br>Whether parking a virtual processor reduces the power used by an MTP is implementation dependent | 200 |
| 1105 | **XIR Steering Register** | |
| | **a:** Whether `XIR_STEERING` is a read-only register or read / write register is implementation dependent. If `XIR_STEERING` is read-only, writes to it are ignored and `XIR_STEERING` is set to steer XIRs to all available virtual processors (that is, `XIR_STEERING` reads identically to `CORE_AVAILABLE`). | 206 |
| | **b:** If the XIR Steering register is read / write, it is implementation dependent whether, upon de-assertion of reset, the value of the `CORE_ENABLE` or `CORE_AVAILABLE` register is copied to `XIR_STEERING`. | 603 |
| 1106 | **Error Steering Register** | |
| | **a:** The number of implemented bits of `ERROR_STEERING.target_id` is implementation dependent, but must be sufficient to encode the highest implemented core ID. If fewer than six bits are implemented, the unused bits read as zero and writes to them are ignored. | 210 |
| | **b:** It is implementation dependent whether, upon de-assertion of reset, `ERROR_STEERING` is set to the value of the lowest-numbered virtual processor that is set to be *enabled* (as indicated by `CORE_ENABLE`) or is *running* (as indicated by `CORE_RUNNING`). | 603 |

**JPS Compatibility Note –** Implementation dependencies #1100–1106 are new MTP features for SPARC JPS2 processors.

# Formal Specification of the Memory Models

This appendix contains only material from *The SPARC Architecture Manual*, Version 9, which provides a formal description of a SPARC V9 processor's interaction with memory. The formal description is more complete and more precise than the description of Chapter 8, *Memory Models*, and therefore represents the definitive specification. Implementations must conform to this model, and programmers must use this description to resolve any ambiguity.

This formal specification is not intended to be a description of an actual implementation, only to describe in a precise and rigorous fashion the behavior that any conforming implementation must provide.

## D.1 Virtual Processor and Memory

The system model consists of a collection of virtual processors, $P_0$, $P_1$,...$P_{n-1}$. Each virtual processor executes its own instruction stream.[1] virtual processors may share address space and access to real memory and I/O locations.

To improve performance, a virtual processor may interpose a *cache* or caches in the path between the virtual processor and memory. For data and I/O references, caches are required to be transparent. The memory model specifies the functional behavior of the entire memory subsystem, which includes any form of caching. Implementations must use appropriate cache coherency mechanisms to achieve this transparency.[2]

1. Virtual processors are equivalent to their software abstraction, processes, provided that context switching is properly performed. See *Appendix J, Programming with the Memory Models*, for an example of context switch code.

2. Philip Bitar and Alvin M. Despain, "Multiprocessor Cache Synchronization: Issues, Innovations, Evolution," *Proc. 13th Annual International Symposium on Computer Architecture*, Computer Architecture News 14:2, June 1986, pp. 424-433.

The SPARC V9 memory model requires that all data references be consistent but does not require that instruction references or input/output references be maintained consistent. The `FLUSH` instruction or an appropriate operating system call may be used to ensure that instruction and data spaces are consistent. Likewise, system software is needed to manage the consistency of I/O operations.

The memory model is a local property of a virtual processor that determines the order properties of memory references. The ordering properties have global implications when memory is shared, since the memory model determines what data is visible to observing virtual processors and in what order. Moreover, the operative memory model of the observing virtual processor affects the apparent order of shared data reads and writes that it observes.

## D.2 Overview of the Memory Model Specification

The underlying goal of the memory model is to place the weakest possible constraints on the virtual processor implementations and to provide a precise specification of the possible orderings of memory operations so that shared-memory multiprocessors can be constructed.

An *execution trace* is a sequence of instructions with a specified initial instruction. An execution trace constitutes one possible execution of a program and may involve arbitrary reorderings and parallel execution of instructions. A *self-consistent* execution trace is one that generates precisely the same results as those produced by a program order execution trace.

A *program order execution trace* is an execution trace that begins with a specified initial instruction and executes one instruction at a time in such a fashion that all the semantic effects of each instruction take effect before the next instruction is begun. The execution trace this process generates is defined to be *program order.*

A *program* is defined by the collection of all possible program order execution traces.

*Dependence order* is a partial order on the instructions in an execution trace that is adequate to ensure that the execution trace is self-consistent. Dependence order can be constructed with conventional data dependence analysis techniques. Dependence order holds only between instructions in the instruction trace of a single virtual processor; instructions that are part of execution traces on different virtual processors are never dependence ordered.

*Memory order* is a total order on the memory reference instructions (loads, stores, and atomic load/stores) which satisfies the dependence order and, possibly, other order constraints such as those introduced implicitly by the choice of memory model or

explicitly by the appearance of memory barrier (MEMBAR) instructions in the execution trace. The existence of a global memory order on the performance of all stores implies that memory access is write-atomic.[1]

A *memory model* is a set of rules that constrains the order of memory references. The SPARC V9 architecture supports three memory models: Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory Order (RMO). The memory models are defined only for memory and not for I/O locations. See *Memory, Real Memory, and I/O Locations* on page 171 for more information.

The formal definition used in the SPARC V8 specification[2] remains valid for the definition of PSO and TSO, except for the FLUSH instruction, which has been modified slightly.[3] The SPARC V9 architecture introduces a new memory model, RMO, which differs from TSO and PSO in that it allows load operations to be reordered as long as single-thread programs remain self-consistent.

# D.3 Memory Transactions

## D.3.1 Memory Transactions

A memory transaction is one of the following:

- **Store** — A request by a virtual processor to replace the value of a specified memory location. The address and new value are bound to the store transaction when the virtual processor initiates the store transaction. A store is complete when the new value is visible to all virtual processors in the system.
- **Load** — A request by a virtual processor to retrieve the value of the specified memory location. The address is bound to the load transaction when the virtual processor initiates the load transaction. A load is complete when the value being returned cannot be modified by a store made by another virtual processor.

1. W. W. Collier, *Reasoning About Parallel Architectures*, Prentice-Hall, 1992, includes an excellent discussion of write-atomicity and related memory model topics.
2. Pradeep Sindhu, Jean-Marc Frailong, and Michel Ceklov. “Formal Specification of Memory Models,” Xerox Palo Alto Research Center Report CSL-91-11, December 1991**.**
3. In SPARC V8, a FLUSH instruction needs at least five instruction execution cycles before it is guaranteed to have local effects; in SPARC V9 this five-cycle requirement has been removed.

- **Atomic** — A *load/store* pair with the guarantee that no other memory transaction will alter the state of the memory between the load and the store. The SPARC V9 instruction set includes three atomic instructions: LDSTUB, SWAP, and CAS.[1] An atomic transaction is considered to be both a load and a store.[2]
- **Flush** — A request by a virtual processor to force changes in the data space aliased to the instruction space to become consistent. Flush transactions are considered to be store operations for memory model purposes.

Memory transactions are referred to by capital letters: $X_n a$, which denotes a specific memory transaction $X$ by virtual processor $n$ to memory address $a$. The virtual processor index and the address are specified only if needed. The predicate $S(X)$ is true if and only if $X$ has store semantics. The predicate $L(X)$ is true if and only if $X$ has load semantics.

MEMBAR instructions are not memory transactions; rather they convey order information above and beyond the implicit ordering implied by the memory model in use. MEMBAR instructions are applied in program order.

## D.3.2 Program Order

The *program order* is a per-virtual processor total order that denotes the sequence in which virtual processor $n$ logically executes instructions. The program order relation is denoted by $<p$ such that $X_n <p Y_n$ is true if and only if the memory transaction $X_n$ is caused by an instruction that is executed before the instruction that caused memory transaction $Y_n$.

Program order specifies a unique total order for all memory transactions initiated by one virtual processor.

Memory barrier (MEMBAR) instructions executed by the virtual processor are ordered with respect to $<p$. The predicate $M(X,Y)$ is true if and only if $X <p Y$ and there exists a MEMBAR instruction that orders $X$ and $Y$ (that is, it appears in program order between $X$ and $Y$). MEMBAR instructions can be either ordering or sequencing and may be combined into a single instruction by a bit-encoded mask.[3]

Ordering MEMBAR instructions impose constraints on the order in which memory transactions are performed.

1. There are three generic forms. CASA and CASXA reference 32-bit and 64-bit objects, respectively. Both normal and alternate ASI forms exist for LDSTUB and SWAP. CASA and CASXA only have alternate forms; however, a CASA (CASXA) with ASI = ASI_PRIMARY{_LITTLE} is equivalent to CAS (CASX). Synthetic instructions for CAS and CASX are suggested in *Synthetic Instructions* on page 514.
2. Even though the store part of a CASA is conditional, it is assumed that the store will always take place whether or not it does in a particular implementation. Since the value stored when the condition fails is the value already present and since the CASA operation is atomic, no observing virtual processor can determine whether the store occurred or not.
3. The Ordering MEMBAR instruction uses 4 bits of its argument to specify the existence of an order relation depending on whether $X$ and $Y$ have load or store semantics. The Sequencing MEMBAR uses three bits to specify completion conditions. The MEMBAR encoding is specified in A.35.

Sequencing MEMBARs introduce additional constraints that are required in cases where the memory transaction has side effects beyond storing data. Such side effects are beyond the scope of the memory model, which is limited to order and value semantics for memory.[1]

This definition of program order is equivalent to the definition given in the SPARC V8 memory model specification.

## D.3.3 Dependence Order

Dependence order is a partial order that captures the constraints that hold between instructions that access the same virtual processor register or memory location. To allow maximum concurrency in virtual processor implementations, dependence order assumes that registers are dynamically renamed to avoid false dependences arising from register reuse.

Two memory transactions *X* and *Y* are dependence ordered, denoted by $X <_d Y$, if and only if they are program ordered, $X <_p Y$, *and* at least one of the following conditions is true:

1. The execution of *Y* is conditional on *X*, and S(*Y*) is true.

2. *Y* reads a register that is written by *X*.

3. *X* and *Y* access the same memory location and *S(X)* and *L(Y)* are both true.

The dependence order also holds between the memory transactions associated with the instructions. It is important to remember that partial ordering is transitive.

Rule (1) includes all control dependences that arise from the dynamic execution of programs. In particular, a store or atomic memory transaction that is executed after a conditional branch will depend on the outcome of that branch instruction, which in turn will depend on one or more memory transactions that precede the branch instruction. Loads after an unresolved conditional branch may proceed, that is, a conditional branch does not dependence-order subsequent loads. Control dependences always order the initiation of subsequent instructions to the performance of the preceding instructions.[2]

Rule (2) captures dependences arising from register use. It is not necessary to include an ordering when *X* reads a register that is later written by *Y*, because register renaming will allow out-of-order execution in this case. Register renaming is equivalent to having an infinite pool of registers and requiring all registers to be

1. Sequencing constraints have other effects, such as controlling when a memory error is recognized or when an I/O access reaches global visibility. The need for sequencing constraints is always associated with I/O and kernel-level programming and not usually with normal, user-level application programming.

2. Self-modifying code (use of FLUSH instructions) also causes control dependences.

write-once. Observe that the condition code register is set by some arithmetic and logical instructions and used by conditional branch instructions, thus introducing a dependence order.

Rule (3) captures ordering constraints resulting from memory accesses to the same location and requires that the dependence order reflect the program order for store-load pairs, but not for load-store or store-store pairs. A load may be executed speculatively since loads are side-effect free, provided that Rule (3) is eventually satisfied.

An actual virtual processor implementation will maintain dependence order by score-boarding, hardware interlocks, data flow techniques, compiler-directed code scheduling, and so forth, or, simply, by sequential program execution. The means by which the dependence order is derived from a program is irrelevant to the memory model, which has to specify which possible memory transaction sequences are legal for a given set of data dependences. Practical implementations will not necessarily use the minimal set of constraints: adding unnecessary order relations from the program order to the dependence order only reduces the available concurrency but does not impair correctness.

## D.3.4 Memory Order

The sequence in which memory transactions are performed by the memory is called *memory order*, which is a total order on all memory transactions.

In general, the memory order cannot be known *a priori*. Instead, the memory order is specified as a set of constraints that are imposed on the memory transactions. The requirement that memory transaction *X* must be performed before memory transaction *Y* is denoted by $X <_m Y$. Any memory order that satisfies these constraints is legal. The memory subsystem may choose arbitrarily among legal memory orders; hence, multiple executions of the same programs may result in different memory orders.

# D.4 Specification of Relaxed Memory Order (RMO)

## D.4.1 Value Atomicity

Memory transactions will atomically set or retrieve the value of a memory location as long as the size of the value is less than or equal to eight bytes, the unit of coherency.

## D.4.2 Store Atomicity

All possible execution traces are consistent with the existence of a memory order that totally orders all transactions including all store operations.

This does not imply that the memory order is observable. Nor does it imply that RMO requires any central serialization mechanism.

## D.4.3 Atomic Memory Transactions

The atomic memory transactions `SWAP`, `LDSTUB`, and `CAS` are performed as one memory transaction that is both a load and a store with respect to memory order constraints. No other memory transaction can separate the load and store actions of an atomic memory transaction. The semantics of atomic instructions are defined in Appendix A, *Instruction Definitions*.

## D.4.4 Memory Order Constraints

A memory order is legal in RMO if and only if:

1. $X <_d Y \;\&\; L(X) \Rightarrow X <_m Y$
2. $M(X,Y) \Rightarrow X <_m Y$
3. $Xa <_p Ya \;\&\; S(Y) \Rightarrow X <_m Y$

Rule (1) states that the RMO model will maintain dependence when the preceding transaction is a load. Preceding stores may be delayed in the implementation, so their order may not be preserved globally.

Rule (2) states that `MEMBAR` instructions order the performance of memory transactions.

Rule (3) states that stores to the same address are performed in program order. This is necessary for virtual processor self-consistency.

## D.4.5 Value of Memory Transactions

The value of a load *Ya* is the value of the most recent store that was performed with respect to memory order or the value of the most recently initiated store by the same virtual processor. Assuming *Y* is a load to memory location *a*:

$$\text{Value}(La) = \text{Value}( \text{Max}_{<m} \{ S \mid Sa <m\ La \text{ or } Sa <p\ La \} )$$

where $Max_{<m}\{..\}$ selects the most recent element with respect to the memory order and where *Value()* yields the value of a particular memory transaction. This states that the value returned by a load is either the result of the most recent store to that address which has been performed by any virtual processor or which has been initiated by the virtual processor issuing the load. The distinction between local and remote stores permits use of store buffers, which are explicitly supported in all SPARC V9 memory models.

## D.4.6 Termination of Memory Transactions

Any memory transaction will eventually be performed. This is formalized by the requirement that only a finite number of memory ordered loads can be performed before a pending store is completed.

## D.4.7 Flush Memory Transaction

Flush instructions are treated as store memory transactions as far as the memory order is concerned. Their semantics are defined in A.21, *Flush Instruction Memory,* on page 268. Flush instructions introduce a control dependence to any subsequent (in program order) execution of the instruction that was addressed by the flush.

## D.5 Specification of Partial Store Order (PSO)

The specification of Partial Store Order (PSO) is that of Relaxed Memory Order (RMO) with the additional requirement that all memory transactions with load semantics are followed by an implied `MEMBAR #LoadLoad | #LoadStore`.

## D.6 Specification of Total Store Order (TSO)

The specification of Total Store Order (TSO) is that of Partial Store Order (PSO) with the additional requirement that all memory transactions with store semantics are followed by an implied `MEMBAR #StoreStore`.

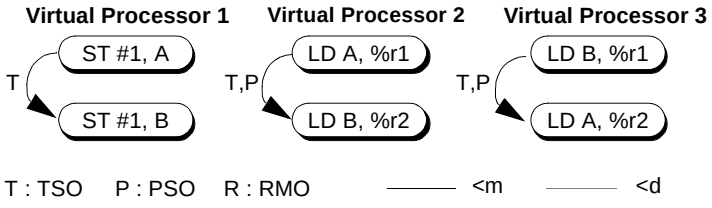
## D.7 Examples of Program Executions

This subsection lists several code sequences and an exhaustive list of all possible execution sequences under RMO, PSO, and TSO. For each example, the code is followed by the list of order relations between the corresponding memory transactions. The memory transactions are referred to by numbers. In each case, the program is executed once for each memory model.

### D.7.1 Observation of Store Atomicity

The code example in FIGURE D-1 demonstrates how store atomicity prevents multiple virtual processors from observing inconsistent sequences of events. In this case, virtual processors 2 and 3 observe changes to the shared variables *A* and *B*, which are being modified by virtual processor 1. Initially both variables are 0. The stores by virtual processor 1 do not use any form of synchronization, and they may in fact be issued by two independent virtual processors.

Should virtual processor 2 find *A* to have the new value (1) and *B* to have the old value (0), it can infer that *A* was updated before *B*. Likewise, virtual processor 3 may find *B = 1* and *A = 0*, which implies that *B* was changed before *A*. It is impossible for both to occur in all SPARC V9 memory models since there cannot exist a total order on all stores. This property of the memory models has been encoded in the assertion A1.




```
/*
 * Store atomicity
 * Note: will fail in RMO due to lack of membars between loads
 */
virtual processor 1:
        (0)             st      #1,[A]
        (1)             st      #1,[B]
virtual processor 2:
        (2)             ld      [A],%r1
        (3)             ld      [B],%r2
virtual processor 3:
        (4)             ld      [B],%r1
        (5)             ld      [A],%r2

Assertions:
A1: !((P2:%r1 == 1) && (P2:%r2 == 0)) || !((P3:%r1 == 1) && (P3:%r2  == 0))

Possible values under all memory models:
2:r1  2:r2  3:r1  3:r2    A     B   example sequence of performance in <m
  0     0     0     0     1     1   4 5 2 0 3 1
  0     0     0     1     1     1   4 2 0 5 3 1
  0     0     1     1     1     1   2 3 0 1 4 5
  0     1     0     0     1     1   4 5 2 0 1 3
  0     1     0     1     1     1   4 2 0 5 1 3
  0     1     1     1     1     1   2 0 1 3 4 5
  1     0     0     0     1     1   4 5 0 2 3 1
  1     0     0     1     1     1   4 0 5 2 3 1
  1     0     1     1     1     1   0 2 3 1 4 5
  1     1     0     0     1     1   4 5 0 2 1 3
  1     1     0     1     1     1   4 0 5 1 2 3
  1     1     1     1     1     1   0 1 4 2 5 3

Possible values under PSO & RMO, but not under TSO:
2:r1  2:r2  3:r1  3:r2    A     B   example sequence of performance in <m
  0     0     1     0     1     1   2 3 1 4 5 0
  0     1     1     0     1     1   2 1 4 3 5 0
  1     1     1     0     1     1   1 4 5 0 2 3

Possible values under RMO, but not under PSO & TSO:
2:r1  2:r2  3:r1  3:r2    A     B   example sequence of performance in <m
  1     0     1     0     1     1   5 3 0 2 1 4
```

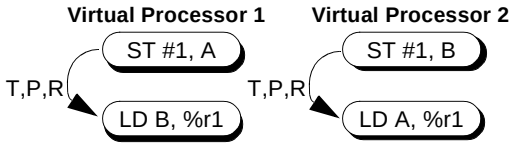
**FIGURE D-1** Store Atomicity Example

However, in RMO, the observing virtual processor must separate the load operations with MEMBAR instructions. Otherwise, the loads may be reordered and no inference on the update order can be made.

FIGURE D-1 is taken from the output of the SPARC V9 memory model simulator, which enumerates all possible outcomes of short code sequences and which can be used to prove assertions about such programs

## D.7.2 Dekker's Algorithm

The essence of Dekker's algorithm is shown in FIGURE D-2.[1]




```
/*
 * Dekker's Algorithm
 */
virtual processor 1:
        (0)             st      #1,[A]
                        membar  #StoreLoad
        (1)             ld      [B],%r1
virtual processor 2:
        (2)             st      #1,[B]
                        membar  #StoreLoad
        (3)             ld      [A],%r1

Assertions:
A1: (P1:%r1 == 1) || (P2:%r1 == 1)

Possible values under all memory models:
1:r1   2:r1   A     B   example sequence of performance in <m
  0      1     1     1   0 1 2 3
  1      0     1     1   2 3 0 1
  1      1     1     1   2 0 3 1

Possible values under PSO & RMO, but not under TSO:
                --- none ---

Possible values under RMO, but not under PSO & TSO:
                --- none ---
```

**FIGURE D-2** Dekker's Algorithm

1. See also DEC Litmus Test 8 described in the *Alpha Architecture Handbook*, Digital Equipment Corporation, 1992, p. 5-14.

To ensure mutual exclusion, each virtual processor signals its intent to enter a critical region by asserting a dedicated variable (*A* for virtual processor 1 and *B* for virtual processor 2). It then checks that the other virtual processor does not want to enter, and if it finds the other signal variable is deasserted, it enters the critical region. This code does not guarantee that any virtual processor can enter (that requires a retry mechanism, which is omitted here), but it does guarantee mutual exclusion, which means that it is impossible that each virtual processor finds the other's lock idle ( = 0) when it enters the critical section.

## D.7.3 Indirection Through Virtual Processors

Another property of the SPARC V9 memory models is that causal update relations are preserved, which is a side effect of the existence of a total memory order. In FIGURE D-3, virtual processor 3 observes updates made by virtual processor 1. Virtual processor 2 simply copies B to C, which does not impact the causal chain of events.

Again, this example intentionally exposes two potential error sources. In PSO (and RMO), the stores by virtual processor 1 are not ordered automatically and may be performed out of program order. The correct code would need to insert a `MEMBAR #StoreStore` between these stores. In RMO (but not in PSO), the observation process 3 needs to separate the two load instructions by a `MEMBAR #LoadLoad`.

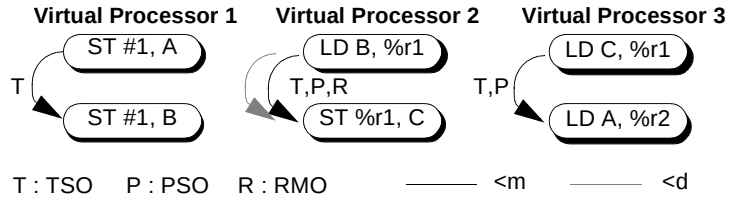
## D.7.4 PSO Behavior

The code in FIGURE D-4 shows how different results can be obtained by allowing out-of-order performance of two stores in PSO and RMO models.

A store to B is allowed to be performed before a store to A. If two loads of virtual processor 2 are performed between the two stores, then the assertion above is satisfied for the PSO and RMO models.

## D.7.5 Application to Compilers

A significant problem in a multiprocessor environment arises from the fact that normal compiler optimizations which reorder code can subvert programmer intent. The SPARC V9 memory model can be applied to the program rather than to an execution, to identify transformations that can be applied, provided that the program has a proper set of `MEMBAR`s in place. In this case, the dependence order is a program-dependence order, rather than a trace-dependence order, and must include the dependences from all possible executions.

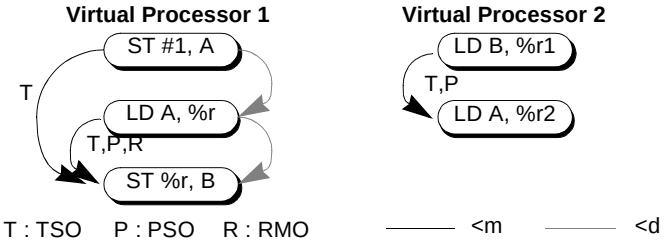



```
/*
 * Indirection through virtual processors
 * Note: Assertion will fail for PSO and RMO due to lack of
 *       membar #StoreStore after P1's first store
 */
virtual processor 1:
        (0)             st      #1,[A]
        (1)             st      #1,[B]
virtual processor 2:
        (2)             ld      [B],%r1
        (3)             st      %r1,[C]
virtual processor 3:
        (4)             ld      [C],%r1
        (5)             ld      [A],%r2

Assertions:
A1: !((P3:%r1 == 1) && (P3:%r2  == 0))

Possible values under all memory models:
2:r1  3:r1  3:r2    A     B     C   example sequence of performance in <m
  0     0     0     1     1     0   4 5 0 2 1 3
  0     0     1     1     1     0   4 2 0 5 1 3
  1     0     0     1     1     1   4 5 0 1 2 3
  1     0     1     1     1     1   4 0 5 1 2 3
  1     1     1     1     1     1   0 1 2 3 4 5

Possible values under PSO & RMO, but not under TSO:
2:r1  3:r1  3:r2    A     B     C   example sequence of performance in <m
  1     1     0     1     1     1   1 2 3 4 5 0

Possible values under RMO, but not under PSO & TSO:
                --- none ---
```

**FIGURE D-3** Indirection Through Virtual Processors

## D.7.6 Verifying Memory Models

While the SPARC V9 memory models were being defined, software tools were developed that automatically analyze and formally verify assembly-code sequences running in the models. The core of this collection of tools is the Murphi finite-state verifier developed by David Dill and his students at Stanford University.




```
/*
 * PSO behavior
 */
virtual processor 1:
 (0) st #1, [A]
 (1) ld [A], %r
 (2) st %r, [B]
virtual processor 2:
 (3) ld [B], %r1
 (4) ld [A], %r2
Assertions:
E: (P2:%r1 == 1) && (P2:%r2 == 0)


Possible values under all memory models:
 1:r  2:r1  2:r2    A     B     example sequence of performance in <m
  1     0     0     1     1     3 4 0 1 2
  1     0     1     1     1     0 3 4 1 2
  1     1     1     1     1     0 1 2 3 4
Possible values under PSO & RMO, but not under TSO:
 1:r  2:r1  2:r2    A     B     example sequence of performance in <m
  1     1     0     1     1     1 2 3 4 0
Possible values under RMO, but not under PSO & TSO:
                --- none ---
```

**FIGURE D-4** PSO Behavior

For example, these tools can be used to confirm that synchronization routines operate properly in various memory models and to generate counter example traces when they fail. The tools work by exhaustively enumerating system states in a version of the memory model, so they can only be applied to fairly small assembly code examples. We found the tools to be helpful in understanding the memory models and checking our examples.[1]

Contact SPARC International to obtain the verification tools and a set of examples.

1. For a discussion of an earlier application of similar tools to TSO and PSO, see David Dill, Seungjoon Park, and Andreas G. Nowatzyk, "Formal Specification of Abstract Memory Models" in *Research on Integrated Systems: Proceedings of the 1993 Symposium*, Ed. Gaetano Borriello and Carl Ebeling, MIT Press, 1993.

# Opcode Maps

This appendix contains the SPARC JPS2 instruction opcode maps.

In this appendix and in Appendix A, *Instruction Definitions*, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE A-1 on page 218. For deprecated opcodes, see Section A.71, *Deprecated Instructions*, starting on page 383, for preferred substitute instructions.

In the tables in this appendix, *reserved* (—) and shaded entries indicate opcodes that are not implemented in SPARC JPS2 processors. An attempt to execute a reserved opcode behaves as defined in *Reserved Opcodes and Instruction Fields* on page 123.

**TABLE E-1** op<1:0>

| op <1:0> | | | |
|---|---|---|---|
| **0** | **1** | **2** | **3** |
| Branches and SETHI *See* TABLE E-2. | CALL | Arithmetic & Miscellaneous (*See* TABLE E-3*)* | Loads / Stores (*See* TABLE E-4*)* |

**TABLE E-2** op2<2:0> (op = 0)

| op2 <2:0> | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| ILLTRAP | BPcc (*See* TABLE E-7) | Bicc[D] (*See* TABLE E-7) | BPr (bit 28=0) (*See* TABLE E-8*)* — (bit 28 = 1)[1] | SETHI NOP[2] | FBPfcc (*See* TABLE E-7) | FBfcc[D] (*See* TABLE E-7) | — |

1. See the footnote regarding bit 28 on page 237.
2. rd = 0, imm22 = 0

The ILLTRAP and *reserved* (—) encodings generate an *illegal_instruction* trap.

**TABLE E-3** op3<5:0> (op = 2)

| | | op3 <5:4> | | | |
|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** |
| **op3 <3:0>** | **0** | ADD | ADDcc | TADDcc | WRY[D] (rd = 0)<br>— (rd = 1)<br>WRCCR (rd =2)<br>WRASI (rd =3)<br>— (rd = 4, 5)<br>WRFPRS (rd =6)<br>WRASR[PASR] (*7*≤rd≤*14*)<br>SIR (rd =15, rs1 =0, i =1) |
| | **1** | AND | ANDcc | TSUBcc | SAVED[P] (fcn = 0),<br>RESTORED[P] (fcn = 1) |
| | **2** | OR | ORcc | TADDccTV[D] | WRPR[P] |
| | **3** | XOR | XORcc | TSUBccTV[D] | — |
| | **4** | SUB | SUBcc | MULScc[D] | FPop1 (*See* TABLE E-5) |
| | **5** | ANDN | ANDNcc | SLL (*x* = 0), SLLX (*x* = 1) | FPop2 (*See* TABLE E-6) |
| | **6** | ORN | ORNcc | SRL (*x* = 0), SRLX (*x* = 1) | IMPDEP1 (VIS) (*See* TABLE E-12) |
| | **7** | XNOR | XNORcc | SRA (*x* = 0), SRAX (*x* = 1) | IMPDEP2 (FMADD/SUB, etc.) (*See* Appendix C in JPS2 **SPARC64 VI** Extensions documents) |
| | **8** | ADDC | ADDCcc | RDY[D] (rs1 = 0)<br>— (rs1 = 1)<br>RDCCR (rs1 = 2)<br>RDASI (rs1 = 3)<br>RDTICK[PNPT] (rs1 = 4)<br>RDPC (rs1 = 5)<br>RDFPRS (rs1 = 6)<br>RDASR[PASR] (*7*≤rd≤*14*)<br>MEMBAR (rs1 = 15, rd = 0, i = 1)<br>STBAR[D] (rs1 = 15, rd = 0, i = 0) | JMPL |
| | **9** | MULX | — | — | RETURN |
| | **A** | UMUL[D] | UMULcc[D] | RDPR[P] | Tcc (*See* TABLE E-7*)*<br>— (bit 29 = 1) |
| | **B** | SMUL[D] | SMULcc[D] | FLUSHW | FLUSH |
| | **C** | SUBC | SUBCcc | MOVcc | SAVE |
| | **D** | UDIVX | — | SDIVX | RESTORE |
| | **E** | UDIV[D] | UDIVcc[D] | POPC (rs1 = 0)<br>— (rs1 > 0) | DONE[P] (fcn = 0)<br>RETRY[P] (fcn = 1) |
| | **F** | SDIV[D] | SDIVcc[D] | MOVr (*See* TABLE E-8) | — |

POPC and the reserved (—) opcodes cause an illegal_instruction trap.

**TABLE E-4** op3<5:0> (op = 3)

| | | op3 <5:4> | | | |
|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** |
| **op3 <3:0>** | **0** | LDUW | LDUWA$^{P_{ASI}}$ | LDF | LDFA$^{P_{ASI}}$ |
| | **1** | LDUB | LDUBA$^{P_{ASI}}$ | LDFSR$^{D}$, LDXFSR | — |
| | **2** | LDUH | LDUHA$^{P_{ASI}}$ | LDQF | LDQFA$^{P_{ASI}}$ |
| | **3** | LDD$^{D}$ | LDDA$^{D,\ P_{ASI}}$ | LDDF | LDDFA$^{P_{ASI}}$ |
| | **4** | STW | STWA$^{P_{ASI}}$ | STF | STFA$^{P_{ASI}}$ |
| | **5** | STB | STBA$^{P_{ASI}}$ | STFSR$^{D}$, STXFSR | — |
| | **6** | STH | STHA$^{P_{ASI}}$ | STQF | STQFA$^{P_{ASI}}$ |
| | **7** | STD$^{D}$ | STDA$^{P_{ASI}}$ | STDF | STDFA$^{P_{ASI}}$ |
| | **8** | LDSW | LDSWA$^{P_{ASI}}$ | — | — |
| | **9** | LDSB | LDSBA$^{P_{ASI}}$ | — | — |
| | **A** | LDSH | LDSHA$^{P_{ASI}}$ | — | — |
| | **B** | LDX | LDXA$^{P_{ASI}}$ | — | — |
| | **C** | — | — | — | CASA$^{P_{ASI}}$ |
| | **D** | LDSTUB | LDSTUBA$^{P_{ASI}}$ | PREFETCH | PREFETCHA$^{P_{ASI}}$ |
| | **E** | STX | STXA$^{P_{ASI}}$ | — | CASXA$^{P_{ASI}}$ |
| | **F** | SWAP$^{D}$ | SWAPA$^{D,\ P_{ASI}}$ | — | — |

LDQF, LDQFA, STQF, STQFA, and the *reserved* (—) opcodes cause an *illegal_instruction* trap on a SPARC JPS2 processor.

**TABLE E-5** opf<8:0> (op = 2, op3 = $34_{16}$ = fpop1)

| | opf<2:0> | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **opf<8:3>** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| $00_{16}$ | — | FMOVs | FMOVd | FMOVq | — | FNEGs | FNEGd | FNEGq |
| $01_{16}$ | — | FABSs | FABSd | FABSq | — | — | — | — |
| $02_{16}$ | — | — | — | — | — | — | — | — |
| $03_{16}$ | — | — | — | — | — | — | — | — |
| $04_{16}$ | — | — | — | — | — | — | — | — |
| $05_{16}$ | — | FSQRTs | FSQRTd | FSQRTq | — | — | — | — |
| $06_{16}$ | — | — | — | — | — | — | — | — |
| $07_{16}$ | — | — | — | — | — | — | — | — |
| $08_{16}$ | — | FADDs | FADDd | FADDq | — | FSUBs | FSUBd | FSUBq |
| $09_{16}$ | — | FMULs | FMULd | FMULq | — | FDIVs | FDIVd | FDIVq |
| $0A_{16}$ | — | — | — | — | — | — | — | — |
| $0B_{16}$ | — | — | — | — | — | — | — | — |
| $0C_{16}$ | — | — | — | — | — | — | — | — |
| $0D_{16}$ | — | FsMULd | — | — | — | — | FdMULq | — |
| $0E_{16}$ | — | — | — | — | — | — | — | — |
| $0F_{16}$ | — | — | — | — | — | — | — | — |
| $10_{16}$ | — | FsTOx | FdTOx | FqTOx | FxTOs | — | — | — |
| $11_{16}$ | FxTOd | — | — | — | FxTOq | — | — | — |
| $12_{16}$ | — | — | — | — | — | — | — | — |
| $13_{16}$ | — | — | — | — | — | — | — | — |
| $14_{16}$ | — | — | — | — | — | — | — | — |
| $15_{16}$ | — | — | — | — | — | — | — | — |
| $16_{16}$ | — | — | — | — | — | — | — | — |
| $17_{16}$ | — | — | — | — | — | — | — | — |
| $18_{16}$ | — | — | — | — | FiTOs | — | FdTOs | FqTOs |
| $19_{16}$ | FiTOd | FsTOd | — | FqTOd | FiTOq | FsTOq | FdTOq | — |
| $1A_{16}$ | — | FsTOi | FdTOi | FqTOi | — | — | — | — |
| $1B_{16}$–$3F_{16}$ | — | — | — | — | — | — | — | — |

Shaded and *reserved* (—) opcodes cause an *fp_exception_other* trap with ftt = unimplemented_FPop on a SPARC JPS2 processor.

**TABLE E-6** opf<8:0> (op = 2, op3 = $35_{16}$ = fpop2)

| | opf<3:0> | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **opf<8:4>** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8–F** |
| $00_{16}$ | — | FMOVs (fcc0) | FMOVd (fcc0) | FMOVq (fcc0) | — | † ‡ | † ‡ | † ‡ | — |
| $01_{16}$ | — | — | — | — | — | — | — | — | — |
| $02_{16}$ | — | — | — | — | — | FMOVsZ ‡ | FMOVdZ ‡ | FMOVqZ ‡ | — |
| $03_{16}$ | — | — | — | — | — | — | — | — | — |
| $04_{16}$ | — | FMOVs (fcc1) | FMOVd (fcc1) | FMOVq (fcc1) | — | FMOVsLEZ ‡ | FMOVdLEZ ‡ | FMOVqLEZ ‡ | — |
| $05_{16}$ | — | FCMPs | FCMPd | FCMPq | — | FCMPEs ‡ | FCMPEd ‡ | FCMPEq ‡ | — |
| $06_{16}$ | — | — | — | — | — | FMOVsLZ ‡ | FMOVdLZ ‡ | FMOVqLZ ‡ | — |
| $07_{16}$ | — | — | — | — | — | — | — | — | — |
| $08_{16}$ | — | FMOVs (fcc2) | FMOVd (fcc2) | FMOVq (fcc2) | — | † | † | † | — |
| $09_{16}$ | — | — | — | — | — | — | — | — | — |
| $0A_{16}$ | — | — | — | — | — | FMOVsNZ ‡ | FMOVdNZ ‡ | FMOVqNZ ‡ | — |
| $0B_{16}$ | — | — | — | — | — | — | — | — | — |
| $0C_{16}$ | — | FMOVs (fcc3) | FMOVd (fcc3) | FMOVq (fcc3) | — | FMOVsGZ ‡ | FMOVdGZ ‡ | FMOVqGZ ‡ | — |
| $0D_{16}$ | — | — | — | — | — | — | — | — | — |
| $0E_{16}$ | — | — | — | — | — | FMOVsGEZ ‡ | FMOVdGEZ ‡ | FMOVqGEZ ‡ | — |
| $0F_{16}$ | — | — | — | — | — | — | — | — | — |
| $10_{16}$ | — | FMOVs (icc) | FMOVd (icc) | FMOVq (icc) | — | — | — | — | — |
| $11_{16}$–$17_{16}$ | — | — | — | — | — | — | — | — | — |
| $18_{16}$ | — | FMOVs (xcc) | FMOVd (xcc) | FMOVq (xcc) | — | — | — | — | — |
| $19_{16}$–$1F_{16}$ | — | — | — | — | — | — | — | — | — |

† Reserved variation of FMOVr

‡ bit 13 of instruction = 0

Shaded and *reserved* (—) opcodes cause an *fp_exception_other* trap with ftt = unimplemented_FPop on a SPARC JPS2 processor.

**TABLE E-7** cond<3:0>

| | | BPcc | Bicc$^{D}$ | FBPfcc | FBfcc$^{D}$ | Tcc |
|---|---|---|---|---|---|---|
| | | op = 0<br>op2 = 1 | op = 0<br>op2 = 2 | op = 0<br>op2 = 5 | op = 0<br>op2 = 6 | op = 2<br>op3 = $3A_{16}$ |
| cond<3:0> | **0** | BPN | BN$^{D}$ | FBPN | FBN$^{D}$ | TN |
| | **1** | BPE | BE$^{D}$ | FBPNE | FBNE$^{D}$ | TE |
| | **2** | BPLE | BLE$^{D}$ | FBPLG | FBLG$^{D}$ | TLE |
| | **3** | BPL | BL$^{D}$ | FBPUL | FBUL$^{D}$ | TL |
| | **4** | BPLEU | BLEU$^{D}$ | FBPL | FBL$^{D}$ | TLEU |
| | **5** | BPCS | BCS$^{D}$ | FBPUG | FBUG$^{D}$ | TCS |
| | **6** | BPNEG | BNEG$^{D}$ | FBPG | FBG$^{D}$ | TNEG |
| | **7** | BPVS | BVS$^{D}$ | FBPU | FBU$^{D}$ | TVS |
| | **8** | BPA | BA$^{D}$ | FBPA | FBA$^{D}$ | TA |
| | **9** | BPNE | BNE$^{D}$ | FBPE | FBE$^{D}$ | TNE |
| | **A** | BPG | BG$^{D}$ | FBPUE | FBUE$^{D}$ | TG |
| | **B** | BPGE | BGE$^{D}$ | FBPGE | FBGE$^{D}$ | TGE |
| | **C** | BPGU | BGU$^{D}$ | FBPUGE | FBUGE$^{D}$ | TGU |
| | **D** | BPCC | BCC$^{D}$ | FBPLE | FBLE$^{D}$ | TCC |
| | **E** | BPPOS | BPOS$^{D}$ | FBPULE | FBULE$^{D}$ | TPOS |
| | **F** | BPVC | BVC$^{D}$ | FBPO | FBO$^{D}$ | TVC |

**TABLE E-8** Encoding of rcond<2:0> Instruction Field

| | | BPr | MOVr | FMOVr |
|---|---|---|---|---|
| | | op = 0<br>op2 = 3 | op = 2<br>op3 = $2F_{16}$ | op = 2<br>op3 = $35_{16}$ |
| rcond <2:0> | 0 | — | — | — |
| | 1 | BRZ | MOVRZ | FMOVRZ |
| | 2 | BRLEZ | MOVRLEZ | FMOVRLEZ |
| | 3 | BRLZ | MOVRLZ | FMOVRLZ |
| | 4 | — | — | — |
| | 5 | BRNZ | MOVRNZ | FMOVRNZ |
| | 6 | BRGZ | MOVRGZ | FMOVRGZ |
| | 7 | BRGEZ | MOVRGEZ | FMOVRGEZ |

**TABLE E-9** cc / opf_cc Fields (movcc and fmovcc)

| opf_cc | | | Condition Code Selected |
|---|---|---|---|
| cc2 | cc1 | cc0 | |
| 0 | 0 | 0 | fcc0 |
| 0 | 0 | 1 | fcc1 |
| 0 | 1 | 0 | fcc2 |
| 0 | 1 | 1 | fcc3 |
| 1 | 0 | 0 | icc |
| 1 | 0 | 1 | — |
| 1 | 1 | 0 | xcc |
| 1 | 1 | 1 | — |

**TABLE E-10** cc Fields (fbpfcc, fcmp, and fcmpe)

| cc1 | cc0 | Condition Code Selected |
|---|---|---|
| 0 | 0 | fcc0 |
| 0 | 1 | fcc1 |
| 1 | 0 | fcc2 |
| 1 | 1 | fcc3 |

**TABLE E-11** cc Fields (bpcc and tcc)

| cc1 | cc0 | Condition Code Selected |
|---|---|---|
| 0 | 0 | icc |
| 0 | 1 | — |
| 1 | 0 | xcc |
| 1 | 1 | — |

**TABLE E-12** IMPDEP1: opf<8:0> for VIS opcodes (op = 2, op3 = $36_{16}$)

| | | opf <8:4> | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **00** | **01** | **02** | **03** | **04** | **05** | **06** | **07** | **08** |
| opf <3:0> | **0** | EDGE8 | ARRAY8 | FCMPLE16 | — | — | FPADD16 | FZERO | FAND | SHUTDOWN |
| | **1** | EDGE8N | — | — | FMUL 8x16 | — | FPADD16S | FZEROS | FANDS | SIAM |
| | **2** | EDGE8L | ARRAY16 | FCMPNE16 | — | — | FPADD32 | FNOR | FXNOR | — |
| | **3** | EDGE8LN | — | — | FMUL 8x16AU | — | FPADD32S | FNORS | FXNORS | — |
| | **4** | EDGE16 | ARRAY32 | FCMPLE32 | — | — | FPSUB16 | FANDNOT2 | FSRC1 | — |
| | **5** | EDGE16N | — | — | FMUL 8x16AL | — | FPSUB16S | FANDNOT2S | FSRC1S | — |
| | **6** | EDGE16L | — | FCMPNE32 | FMUL 8SUx16 | — | FPSUB32 | FNOT2 | FORNOT2 | — |
| | **7** | EDGE16LN | — | — | FMUL 8ULx16 | — | FPSUB32S | FNOT2S | FORNOT2S | — |
| | **8** | EDGE32 | ALIGN ADDRESS | FCMPGT16 | FMULD 8SUx16 | FALIGNDATA | — | FANDNOT1 | FSRC2 | — |
| | **9** | EDGE32N | BMASK | — | FMULD 8ULx16 | — | — | FANDNOT1S | FSRC2S | — |
| | **A** | EDGE32L | ALIGN ADDRESS_LITTLE | FCMPEQ16 | FPACK32 | — | — | FNOT1 | FORNOT1 | — |
| | **B** | EDGE32LN | — | — | FPACK16 | FPMERGE | — | FNOT1S | FORNOR1S | — |
| | **C** | — | — | FCMPGT32 | — | BSHUFFLE | — | FXOR | FOR | — |
| | **D** | — | — | — | FPACKFIX | FEXPAND | — | FXORS | FORS | — |
| | **E** | — | — | FCMPEQ32 | PDIST | — | — | FNAND | FONE | — |
| | **F** | — | — | — | — | — | — | FNANDS | FONES | — |

# Memory Management Unit

The Memory Management Unit (MMU) conforms to the requirements set forth in the *SPARC V9 Architecture Manual*. In particular, it supports a 64-bit virtual address space, software TLB-miss processing only (no hardware page table walk), simplified protection encoding, and multiple page sizes.

This chapter describes the Memory Management Unit, as seen by the operating system software, in these sections:


- *Virtual Address Translation* on page 467
- *Translation Table Entry (TTE)* on page 470
- *Translation Storage Buffer* on page 473
- *Hardware Support for TSB Access* on page 475
- *Faults and Traps* on page 479
- *MMU Operation Summary* on page 481
- *ASI Value, Context, and Endianness Selection for Translation* on page 483
- *Reset, Disable, and RED_state Behavior* on page 485
- *SPARC V9 "MMU Requirements" Annex* on page 487
- *Internal Registers and ASI Operations* on page 487
- *MMU Bypass* on page 503
- *Translation Lookaside Buffer Hardware* on page 503


## F.1 Virtual Address Translation

The MMUs support four page sizes: 8 Kbytes, 64 Kbytes, 512 Kbytes, and 4 Mbytes. Separate Instruction and Data MMUs (IMMU and DMMU, respectively) are provided to enable concurrent virtual-to-physical address translations for instruction and data. A 64-bit virtual address (VA) space is supported, with a minimum of 43 bits of physical address (PA). In each translation, the virtual page number is replaced by a physical page number, which is concatenated with the page offset to form the full physical address, as illustrated in FIGURE F-1.

Each MMU consists of one or more Translation Lookaside Buffers (TLBs), including micro-TLB structures. The organization of TLB structures may be different between the Instruction MMU and the Data MMU.

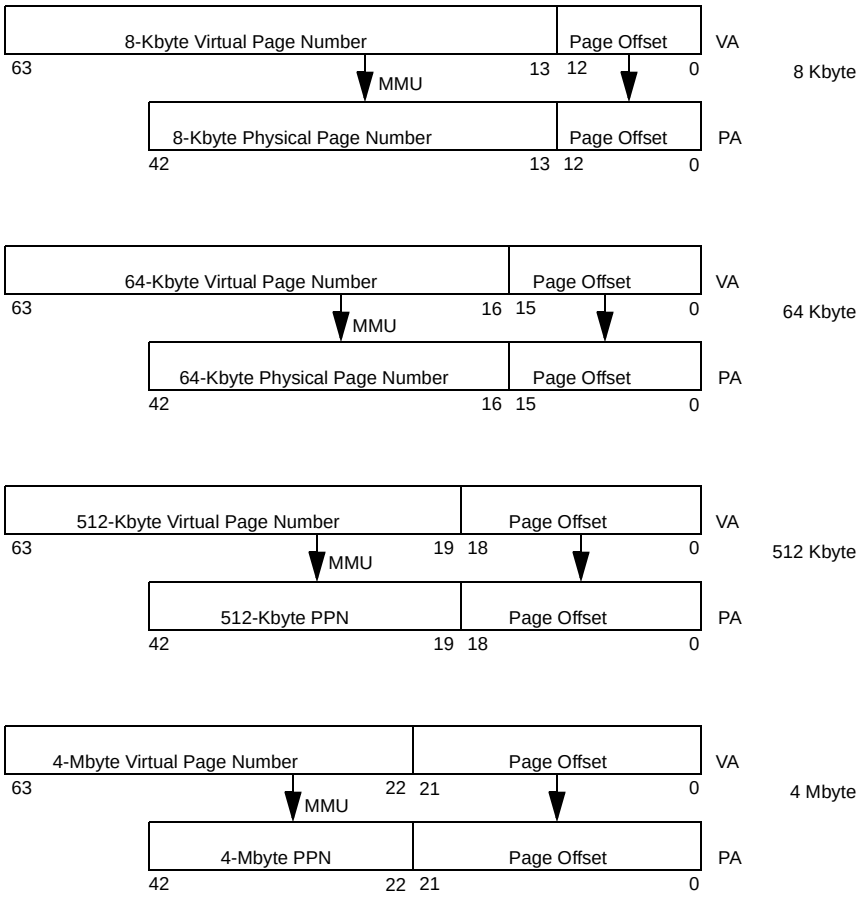



**FIGURE F-1** Virtual-to-Physical Address Translation for All Four Page Sizes

The operating system maintains translation information in an arbitrary data structure, called the *software translation table* in this appendix. The I- and D-MMU TLBs act as independent caches of the software translation table, providing appropriate concurrency for virtual-to-physical address translation.

**IMPL. DEP. #222:** TLB organization is implementation dependent in JPS2 processors.

On a TLB miss, the MMU immediately traps to software for TLB miss processing. The TLB miss handler can fill the TLB by any available means, but it is likely to take advantage of the TLB miss support features provided by the MMU, since the TLB miss handler is time-critical code. Hardware support is described in *Hardware Support for TSB Access* on page 475.

A general software view of the MMU is shown in FIGURE F-2. The TLBs, which are part of the MMU hardware, are small and fast. The software translation table is likely to be large and complex. The translation storage buffer (TSB), which acts like a direct-mapped cache, is the interface between the two. The TSB can be shared by all processes running on a processor or can be process specific. The hardware does not require any particular scheme.

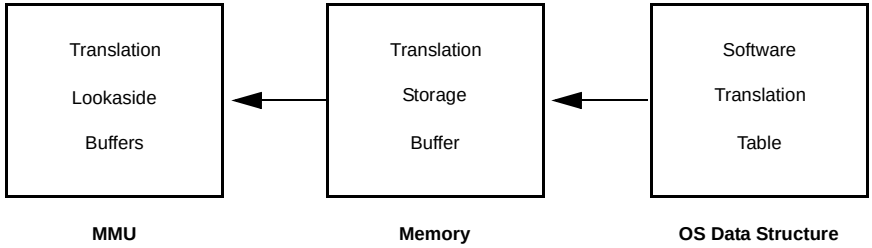



**FIGURE F-2** Software View of the MMU

Aliasing between pages of different sizes (when multiple virtual addresses map to the same physical address) can take place, as described in the SPARC V8 Reference MMU. However, the reverse case of multiple mappings from one virtual address to multiple physical addresses producing a multiple TLB match is not necessarily detected in hardware and may produce undefined results.

**IMPL. DEP. #223:** Whether TLB multiple-hit detection is supported in a JPS2 processor is implementation dependent.

---

**Note –** The hardware ensures the physical reliability of the TLB on multiple matches.

---

**IMPL. DEP. #310:** A JPS2 implementation may support page sizes greater than 4 Mbytes. If so, a reference to such a page in a TTE or TSB entry is encoded in bits 49:48, which are reserved for that purpose. Which page size(s) over 4 Mbytes are supported and their encoding in TTE and TSB entries are implementation dependent.

# F.2 Translation Table Entry (TTE)

The Translation Table Entry (TTE) is the equivalent of a SPARC V8 page table entry; it holds information for a single page mapping. The TTE is divided into two 64-bit words representing the tag and data of the translation. Just as in a hardware cache, the tag is used to determine whether there is a hit in the TSB; if there is a hit, the data are fetched by software.

The configuration of the TTE is illustrated in FIGURE F-3 and described in TABLE F-1.

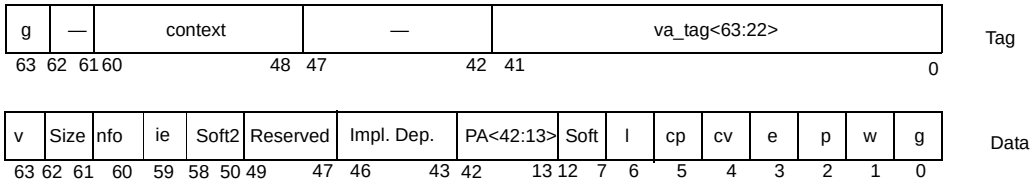



**FIGURE F-3** Translation Storage Buffer (TSB) Translation Table Entry (TTE)

**TABLE F-1** TSB and TTE Bit Description *(1 of 4)*

| Bit | Field | Description |
|---|---|---|
| Tag– 63 | g | Global. If the Global bit is set, the Context field of the TLB entry is ignored during hit detection. This behavior allows any page to be shared among all (user or supervisor) contexts running in the same virtual processor. The Global bit is duplicated in the TTE tag and data to optimize the software miss handler. |
| Tag– 60:48 | context | The 13-bit context identifier associated with the TTE. |
| Tag– 63:22 | va_tag | Virtual Address Tag. The virtual page number. Bits 21 through 13 are not maintained in the tag because these bits index the minimally sized, direct-mapped TSB of 512 entries. |
| Data – 63 | v | Valid. If the Valid bit is set, then the remaining fields of the TTE are meaningful. Note that the explicit Valid bit is redundant with the software convention of encoding an invalid TTE with an unused context. The encoding of the context field is necessary to cause a failure in the TTE tag comparison, and the explicit Valid bit in the TTE data simplifies the TLB miss handler. |
| Data – 62:61 | size | The page size of this entry, encoded as shown below.<br>**Size <1:0> Page Size**<br>00 8 Kbyte<br>01 64 Kbyte<br>10 512 Kbyte<br>11 4 Mbyte |

**TABLE F-1** TSB and TTE Bit Description *(2 of 4)*

| Bit | Field | Description |
|---|---|---|
| Data – 60 | nfo | No Fault Only. If the no-fault-only bit is set, loads with ASI_PRIMARY_NO_FAULT, ASI_SECONDARY_NO_FAULT, and their *_LITTLE variations are translated. Any other access will trap with a *data_access_exception* trap (FT = $10_{16}$). The nfo bit in the IMMU is read as 0 and ignored when written. The ITLB-miss handler should generate an error if this bit is set before the TTE is loaded into the TLB. |
| Data – 59 | ie | Invert Endianness. If this bit is set for a page, accesses to the page are processed with inverse endianness from that specified by the instruction (big for little, little for big). See page 483 for details. The ie bit in the IMMU is read as 0 and ignored when written.<br>**Note:** This bit is intended to be set primarily for noncacheable accesses. The performance of cacheable accesses will be degraded as if the access missed the D-cache. |
| Data - 58:50 | **soft2** | Software-defined field, provided for use by the operating system. The Soft2 field can be written with any value in the TSB. Hardware is not required to maintain this field in the TLB, so when it is read from the TLB, it may read as zero. |
| Data – 49:48 | *Reserved* | Reserved for future page sizes greater than 4 Mbytes. Specifically, bit 48 is reserved to contain size <2>. |
| Data – 47 | *Reserved* | |
| Data – 46:43 | Implementation dependent | This field is implementation dependent (see impl. dep. #224 below); see each Implementation Extension document for details regarding usage of this field. |
| Data – 42:13 | pa | The physical page number. Page offset bits for larger page sizes (pa<15:13>, pa<18:13>, and pa<21:13> for 64-Kbyte, 512-Kbyte, and 4-Mbyte pages, respectively) are ignored during normal translation.<br>**IMPL. DEP. #224:** Physical address width support by the MMU is implementation dependent in JPS2; minimum pa width is 43 bits.<br>**IMPL. DEP. #238:** When page offset bits for larger page sizes (pa<15:13>, pa<18:13>, and pa<21:13> for 64-Kbyte, 512-Kbyte, and 4-Mbyte pages, respectively) are stored in the TLB, it is implementation dependent whether the data returned from those fields by a Data Access read are zero or the data previously written to them. |
| Data – 12:7 | Soft | Software-defined field, provided for use by the operating system. The Soft field can be written with any value in the TSB. Hardware is not required to maintain this field in the TLB, so when it is read from the TLB, it may read as zero. |

**TABLE F-1** TSB and TTE Bit Description *(3 of 4)*

| Bit | Field | Description |
|---|---|---|
| Data – 6 | **l** | If the lock bit is set, then the TTE entry will be "locked down" when it is loaded into the TLB; that is, if this entry is valid, it will not be replaced by the automatic replacement algorithm invoked by an ASI store to the Data In Register. The lock bit has no meaning for an invalid entry. While a minimum of 14 entries shall be lockable by software in the TLB structure, software must ensure that at least one entry is not locked when replacing a TLB entry.<br>**IMPL. DEP. #225:** The mechanism by which entries in TLB are locked is implementation dependent in JPS2.<br>**IMPL. DEP. #242:** An implementation containing multiple TLBs may implement the l (lock) bit in all TLBs but is only required to implement a lock bit in one TLB for each page size. If the lock bit is not implemented in a particular TLB, it reads as 0 and writes to it are ignored. |
| Data – **5**<br>Data – 4 | **cp**,<br>**cv** | The cacheable-in-physically-indexed-cache bit and cacheable-in-virtually-indexed-cache bit determine the placement of data in the caches. Given an implementation with a physically indexed instruction cache, a virtually indexed data cache, and a physically indexed unified second-level cache, the following table illustrates how the cp and cv bits could be used. |

| Cacheable (CP, CV) | Meaning of TTE when placed in: I-TLB (Instruction Cache PA-indexed) | D-TLB (Data Cache VA-indexed) |
|---|---|---|
| 00, 01 | Noncacheable | Noncacheable |
| 10 | Cacheable E-cache, I-cache | Cacheable E-cache |
| 11 | Cacheable E-cache, I-cache | Cacheable E-cache, D-cache |

The MMU does not operate on the cacheable bits but merely passes them through to the cache subsystem. The cv bit in the IMMU is read as zero and ignored when written.

**IMPL. DEP. #226:** Whether the cv bit is supported in TTE is implementation dependent in JPS2. When the cv bit in TTE is not provided and the implementation has virtually indexed caches, the implementation should support hardware unaliasing for the caches.

| | | |
|---|---|---|
| Data – 3 | **e** | Side effect. If the side-effect bit is set, nonfaulting loads will trap for addresses within the page, noncacheable memory accesses other than block loads and stores are strongly ordered against other e-bit accesses, and noncacheable stores are not merged. This bit should be set for pages that map I/O devices having side effects. Note, however, that the e bit does not prevent normal instruction prefetching. The e bit in the IMMU is read as 0 and ignored when written.<br>**Note:** The e bit does not force a noncacheable access. It is expected, but not required, that the cp and cv bits will be set to 0 when the e bit is set. If both the cp and cv bits are set to 1 along with the e bit, the result is undefined.<br>**Note Also:** The e bit and the nfo bit are mutually exclusive; both bits should never be set in any TTE. |

**TABLE F-1** TSB and TTE Bit Description *(4 of 4)*

| Bit | Field | Description |
|---|---|---|
| Data – 2 | p | Privileged. If the p bit is set, only the supervisor can access the page mapped by the TTE. If the p bit is set and an access to the page is attempted when PSTATE.priv = 0, then the MMU signals an *instruction_access_exception* or *data_access_exception* trap (FT = $1_{16}$). |
| Data – 1 | w | Writable. If the w bit is set, the page mapped by this TTE has write permission granted. Otherwise, write permission is not granted, and the MMU causes a *fast_data_access_protection* trap if a write is attempted. The w bit in the IMMU is read as 0 and ignored when written. |
| Data – 0 | g | Global. This bit must be identical to the Global bit in the TTE tag. Like the Valid bit, the Global bit in the TTE tag is necessary for the TSB hit comparison, and the Global bit in the TTE data facilitates the loading of a TLB entry. |

**V9 Compatibility Note –** Referenced and Modified bits are maintained by software. The Global, Privileged, and Writable fields replace the 3-bit acc field of the SPARC V8 Reference MMU Page Translation Entry.

# F.3 Translation Storage Buffer

The Translation Storage Buffer (TSB) is an array of Translation Table Entries managed entirely by software. It serves as a cache of the software translation table, used to quickly reload the TLB in the event of a TLB miss. The discussion in this section assumes the use of the hardware support for TSB access described in *Hardware Support for TSB Access* on page 475, although the operating system is not required to make use of this support hardware.

Inclusion of the TLB entries in the TSB is not required; that is, translation information that is not present in the TSB can exist in the TLB.

A bit in the TSB register allows the TSB 64-Kbyte pointer to be computed for the case of common or split 8-Kbyte/64-Kbyte TSBs.

## F.3.1 TSB Indexing Support

No hardware TSB indexing support is provided for the 512-Kbyte and 4-Mbyte page TTEs. However, since the TSB is entirely software managed, the operating system may choose to place these larger page TTEs in the TSB by forming the appropriate

pointers. In addition, simple modifications to the 8-Kbyte and 64-Kbyte index pointers provided by the hardware allow formation of an M-way, set-associative TSB, multiple TSBs per page size, and multiple TSBs per process.

## F.3.2 TSB Cacheability

The TSB exists as a normal data structure in memory and therefore can be cached. Indeed, the speed of the TLB miss handler relies on the TSB accesses hitting the level-2 cache at a substantial rate. This policy may result in some conflicts with normal instruction and data accesses, but it is hoped that the dynamic sharing of the level-2 cache resource will provide a better overall solution than that provided by a fixed partitioning.

## F.3.3 TSB Organization

The TSB is arranged as a direct-mapped cache of TTEs. The MMU provides pre-computed pointers into the TSB for the 8-Kbyte and 64-Kbyte page TTEs. In each case, *N* least significant bits of the respective virtual page number are used as the offset from the TSB base address, with *N* equal to log base 2 of the number of TTEs in the TSB.

The TSB organization is illustrated in FIGURE F-4. The constant *N* is determined by the `size` field in the `TSB` Register; it can range from 512 to an implementation-dependent number.

**IMPL. DEP. #227:** The maximum number of entries in a TSB is implementation-dependent in JPS2. See impl. dep. #228 for the limitation of `tsb_size` in `TSB` registers.

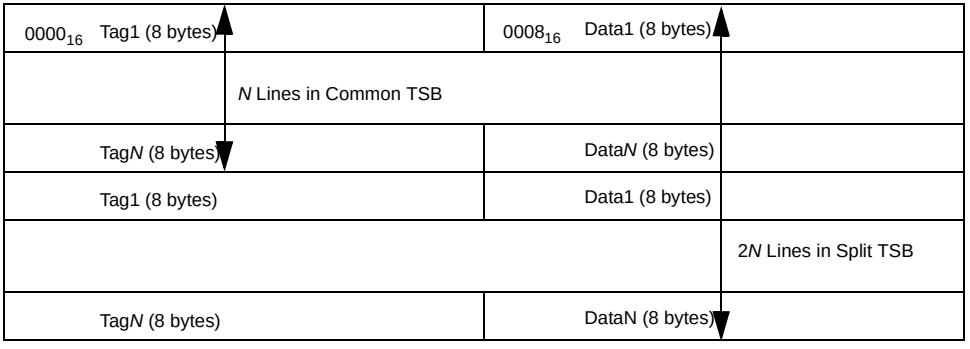



**FIGURE F-4** TSB Organization, Illustrated for Both Common and Shared Cases

# F.4 Hardware Support for TSB Access

The MMU hardware provides services to allow the TLB-miss handler to efficiently reload a missing TLB entry for an 8-Kbyte or 64-Kbyte page. These services include:

- Formation of TSB Pointers, based on the missing virtual address and address space identifier
- Formation of the TTE Tag Target used for the TSB tag comparison
- Efficient atomic write of a TLB entry with a single store ASI operation
- Alternate globals on MMU-signalled traps

## F.4.1 Typical TLB Miss / Refill Sequence

A typical TLB miss-and-refill sequence is the following:

1. A TLB miss causes either a *fast_instruction_access_MMU_miss* or a *fast_data_access_MMU_miss* exception.
2. The appropriate TLB miss handler reads the TSB Pointers and the TTE Tag Target, using ASI loads.
3. Using this information, the TLB miss handler checks to see if the desired TTE exists in the TSB. If so, the TTE data are loaded into the TLB Data In Register to initiate an atomic write of the TLB entry chosen by the replacement algorithm.
4. If the TTE does not exist in the TSB, then the TLB miss handler jumps to the more sophisticated, and slower, TSB miss handler.

The virtual address used in the formation of the pointer addresses comes from the Tag Access Register, which holds the virtual address and context of the load or store responsible for the MMU exception. See *Translation Table Entry (TTE)* on page 470.

---

**Note –** There are no separate physical registers in hardware for the pointer registers; rather, they are implemented through a dynamic reordering of the data stored in the Tag Access and the TSB registers.

---

## F.4.2 TSB Pointer Formation

Hardware provides pointers for the most common cases of 8-Kbyte and 64-Kbyte page miss processing. These pointers give the virtual addresses where the 8-Kbyte and 64-Kbyte TTEs are stored if either are present in the TSB.

### F.4.2.1 Input Values for TSB Pointer Formation

The pointer to the TTE in the TSB is generated from the following parameters as inputs:

- TSB base address (TSB_Base)
- Virtual address (VA)
- TSB_size
- TSB_split
- TSB_Hash

The TSB base address is in either of I/D Primary/Secondary (provided only for data)/Nucleus TSB Extension Registers. Depending on the context that generated the TLB miss, an appropriate TSB Extension Register is selected (which may be combined with the tsb_base field from the TSB Base Register; see Note below). Note that the context with the TLB miss is logged in I/D Synchronous Fault Status Register.

TSB_size and TSB_split are supplied also from the selected TSB Extension Register.

The virtual page number to be used for TSB pointer formation is in I/D Tag Access Register.

TSB_Hash is a representation of the context that generated a TLB miss. Depending on the implementation, the source of the TSB_Hash may vary. For details, refer to the appropriate JPS2 Extension documents.

**IMPL. DEP. #228:** Whether TSB_Hash is supplied from a TSB Extension Register or from a context-ID register is implementation dependent in JPS2. Only for cases of direct hash with context-ID can the width of the TSB_size field be wider than 3 bits.

---

**Note –** TSB_Base address may be generated by exclusive-ORing TSB_Base register and TSB Extension Register contents, for compatibility with UltraSPARC I/II TSB pointer formation. In this case, if the TSB Extension Registers hold 0 as TSB_Base, the value in TSB_Base register becomes the TSB_Base address, thereby maintaining compatibility with UltraSPARC I/II TLB miss handling software. In addition, TSB_Base may be taken directly from an appropriate TSB Extension Register. In that case, the implementation should provide the way to maintain compatibility with UltraSPARC I/II TLB miss handler software.

---

**IMPL. DEP. #229:** Whether the implementation generates the TSB_Base address by exclusive-ORing the TSB_Base register and a TSB Extension Register or by taking TSB_Base field directly from the TSB Extension Register is implementation dependent in JPS2. This implementation dependency is only to maintain compatibility with the TLB miss handling software of UltraSPARC I/II.

### F.4.2.2 TSB Pointer Formation

Hardware uses the following equations to form TSB pointers for TLB misses. In the equations, N is defined to be the `tsb_size` field of the TSB Base or TSB Extension Register; it ranges from 0 to an implementation-dependent number. Note that `tsb_size` refers to the size of each TSB when the TSB is split. The symbol ▯ designates concatenation of bit vectors.

#### *Exclusive-ORed `TSB_Base`*

For a shared TSB (TSB Register split field = 0):

```
8K_POINTER = TSB_Base[63:13+N] ⊕ TSB_Extension[63:13+N] ▯
VA[21+N:13] ▯ 0000

64K_POINTER = TSB_Base[63:13+N] ⊕ TSB_Extension[63:13+N] ▯
VA[24+N:16] ▯ 0000
```

For a split TSB (TSB Register split field = 1):

```
8K_POINTER = TSB_Base[63:14+N] ⊕ TSB_Extension[63:14+N] ▯ 0 ▯
VA[21+N:13] ▯ 0000

64K_POINTER = TSB_Base[63:14+N] ⊕ TSB_Extension[63:14+N] ▯ 1 ▯
VA[24+N:16] ▯ 0000
```

#### *`TSB_Base` from TSB Extension Registers*

For a shared TSB (TSB Register split field = 0):

```
8K_POINTER = TSB_Extension[63:13+N] ▯ (VA[21+N:13] ⊕ TSB_Hash) ▯
0000

64K_POINTER = TSB_Extension[63:13+N] ▯ (VA[24+N:16] ⊕ TSB_Hash) ▯
0000
```

For a split TSB (TSB Register split field = 1):

```
8K_POINTER = TSB_Extension[63:14+N] ▯ 0 ▯ (VA[21+N:13] ⊕ TSB_Hash)
▯ 0000

64K_POINTER = TSB_Extension[63:14+N] ▯ 1 ▯ (VA[24+N:16] ⊕
TSB_Hash) ▯ 0000
```

*Additional Information* For a more detailed description of the pointer logic with pseudocode and hardware implementation, refer to Appendix F of the JPS2 Extension documents.

The TSB Tag Target (described on page 494) is formed by alignment of the missing access `va` (from the Tag Access Register) and the current context to positions found above in the description of the `TTE` tag, allowing a simple `XOR` instruction for TSB hit detection.

## F.4.3 Required TLB Conditions

The following items must be locked in the TLB to avoid an error condition: TLB miss handler and data, TSB and linked data, asynchronous trap handlers and data.

## F.4.4 Required TSB Conditions

The following items must be locked in the TSB (not necessarily the TLB) to avoid an error condition: TSB miss handler and data, interrupt-vector handler and data.

## F.4.5 MMU Global Registers Selection

In the SPARC V9 normal trap model, the software is presented with an alternate set of global registers in the Integer Register file. A JPS2 virtual processor provides an additional feature to facilitate fast handling of TLB misses. For the following traps, the trap handler is presented with a special set of MMU globals:

- *fast_instruction_access_MMU_miss*
- *fast_data_access_MMU_miss*
- *instruction_access_exception*
- *data_access_exception*
- *fast_data_access_protection*

Trap handlers for the *privileged_action*, *mem_address_not_aligned*, and **_mem_address_not_aligned* traps use the standard alternate global registers.

---

**JPS Compatibility Note –** The MMU does not perform hardware tablewalking. JPS2 MMU hardware never directly reads or writes the TSB.

---

# F.5 Faults and Traps

The traps recorded by the MMU are listed in TABLE F-2 and described below the table, by reference number. All listed traps are precise traps.

**TABLE F-2** MMU Trap Types, Causes, and Stored State Register Update Policy

| | | | Registers Updated (Stored State in MMU) | | | | |
|---|---|---|---|---|---|---|---|
| **Ref #** | **Trap Name** | **Trap Cause** | **I-SFSR** | **I-MMU Tag Access** | **D-SFSR, SFAR** | **D-MMU Tag Access** | **Trap Type** |
| 1. | *fast_instruction_access_MMU_miss* | I-TLB miss | X | X | | | $64_{16}$–$67_{16}$ |
| 2. | *instruction_access_exception* | Several (see below) | X | X | | | $08_{16}$ |
| 3. | *fast_data_access_MMU_miss* | D-TLB miss | | | X | X | $68_{16}$–$6B_{16}$ |
| 4. | *data_access_exception* | Several (see below) | | | X | X[†] | $30_{16}$ |
| 5. | *fast_data_access_protection* | Protection violation | | | X | X | $6C_{16}$–$6F_{16}$ |
| 6. | *privileged_action* | Use of privileged ASI | | | X | | $37_{16}$ |
| 7. | watchpoint | Watchpoint hit | | | X | | $61_{16}$–$62_{16}$ |
| 8. | *mem_address_not_aligned*, **_mem_address_not_aligned* | Misaligned mem op | | | (impl. dep. #237) | | $35_{16}$, $36_{16}$, $38_{16}$, $39_{16}$ |

[†] The contents of the `context` field of the D-MMU Tag Access Register are undefined after a *data_access_exception*.

**Note –** In a SPARC JPS2 virtual processor, *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection* traps are generated instead of SPARC V9 *instruction_access_MMU_miss*, *data_access_MMU_miss*, and *data_access_protection* traps, respectively.

1. ***fast_instruction_access_MMU_miss* —** Occurs when the MMU is unable to find a translation for an instruction access; that is, when the appropriate TTE is not in the instruction TLB.

   In a SPARC JPS2 virtual processor, the *fast_instruction_access_MMU_miss* exception is generated instead of the SPARC V9 *instruction_access_MMU_miss*.

2. ***instruction_access_exception* —** Occurs when the IMMU is enabled and detects a privilege violation for an instruction fetch; that is, an attempted access to a privileged page when `PSTATE.priv` = 0.

3. ***fast_data_access_MMU_miss* —** Occurs when the MMU is unable to find a translation for a data access; that is, when the appropriate TTE is not in the data TLB.

   In a SPARC JPS2 virtual processor, the *fast_data_access_MMU_miss* exception is generated instead of the SPARC V9 *data_access_MMU_miss* trap.

4. ***data_access_exception* —** Signalled upon the detection of at least one of the following exceptional conditions:
   - The DMMU detects a privilege violation for a data access; that is, an attempted access to a privileged page when `PSTATE.priv` = 0.
   - A speculative (nonfaulting) load instruction issued to a page marked with the side effect (`e` bit) set to 1, including cases in which the DMMU is disabled.
   - An atomic instruction (including 128-bit atomic load, Load Quadword Atomic) issued to a memory address marked noncacheable in a physical cache; that is, with `cp` bit set to 0, including cases in which the DMMU is disabled.
   - An invalid `LDA/STA` ASI value, read to write-only register, or write to read-only register. Not for an attempted user access to a restricted ASI (see the *privileged_action* trap described below).
   - An access with an ASI other than "`(PRIMARY,SECONDARY)_NO_FAULT(_LITTLE)`" to a page marked with the `nfo` (no-fault-only) bit.

   The implementation may signal a *data_access_exception* if it detects any other exceptional conditions possibly caused by program errors.

   **IMPL. DEP. #230:** The causes of a *data_access_exception* trap are implementation dependent in JPS2, but there are several mandatory causes of *data_access_exception* trap.

5. ***fast_data_access_protection* —** Occurs when the MMU detects a protection violation for a data access. A protection violation is defined to be an attempted store (including atomic load-store operations) to a page that does not have write permission.

   In a SPARC JPS2 virtual processor, the *fast_data_access_protection* exception is generated instead of the SPARC V9 *data_access_protection* trap.

6. ***privileged_action* —** Occurs when an access is attempted using a *restricted* ASI while in nonprivileged mode (`PSTATE.priv` = 0).

7. **`watchpoints`** — *PA_watchpoint* and *VA_watchpoint* traps are included in this category. Watchpoint traps occur when watchpoints are enabled and the DMMU detects a load or store to the virtual or physical address specified by the watchpoint virtual or physical registers, respectively. See *Data Watchpoint Registers* on page 94. The trap is precise and is signalled before the actual event, meaning that the contents of the location are not modified when the trap is invoked.

8. ***mem_address_not_aligned* —** Occurs when a load, store, atomic load-store, JMPL, or RETURN instruction with a misaligned address is executed.

   **IMPL. DEP. #237:** Whether the fault status and/or address (DSFSR/DSFAR) are captured when a *mem_address_not_aligned* trap occurs during a JMPL or RETURN instruction is implementation dependent.

# F.6 MMU Operation Summary

The behavior of the DMMU is summarized in TABLE F-3 on page 482; the behavior of the IMMU is summarized in TABLE F-4 for normal (noninternal) ASIs. In each case and for all conditions, the behavior of each MMU is given by one of the following abbreviations:

| Abbreviation | Meaning |
|---|---|
| OK | normal translation |
| Dmiss | *fast_data_access_MMU_miss* exception |
| Dexc | *data_access_exception* exception |
| Dprot | *fast_data_access_protection* exception |
| Imiss | *fast_instruction_access_MMU_miss* exception |
| Iexc | *instruction_access_exception* exception |

The ASI is indicated by one the following abbreviations:

| Abbreviation | Meaning |
|---|---|
| NUC | ASI_NUCLEUS*. |
| PRIM | Any ASI with PRIMARY translation, except *NO_FAULT. |
| SEC | Any ASI with SECONDARY translation, except *NO_FAULT. |
| PRIM_NF | ASI_PRIMARY_NO_FAULT* |
| SEC_NF | ASI_SECONDARY_NO_FAULT* |
| U_PRIM | ASI_AS_IF_USER_PRIMARY*. |
| U_SEC | ASI_AS_IF_USER_SECONDARY*. |
| BYPASS | ASI_PHYS_* and also other ASIs that require the MMU to perform a bypass operation (such as D-cache access). |

**Note –** The *_LITTLE versions of the ASIs behave the same as the big-endian versions with regard to the MMU table of operations.

Other abbreviations include w for the writable bit, e for the side-effect bit, and p for the privileged bit.

The following cases are not covered in TABLE F-3.

- Invalid ASIs or ASIs that have no meaning for the opcodes listed; for example, ASI_PRIMARY_NOFAULT for a store or atomic load-store. Also, access to internal registers other than LDXA, LDDFA, STXA, or STDFA. See Section L.3.1, *Supported ASIs*. The MMU signals a *data_access_exception* trap (FT = $08_{16}$) for these cases.
- Attempted access using a restricted ASI in nonprivileged mode. The MMU signals a *privileged_action* exception for this case.
- An atomic instruction (including a 128-bit atomic load, Load Quadword Atomic) issued to a memory address marked noncacheable in a physical cache; that is, with the cp bit set to 0, including cases in which the DMMU is disabled. The MMU signals a *data_access_exception* trap (FT = $04_{16}$) for this case.
- A data access with an ASI other than "(PRIMARY, SECONDARY)_NO_FAULT (_LITTLE)" to a page marked with the nfo bit. The MMU signals a *data_access_exception* trap (FT = $10_{16}$) for this case.

See *ASI Assignments* on page 568 for a summary of the ASI map.

**TABLE F-3** DMMU Table of Operations for Normal ASIs

| Condition | | | | | Behavior | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | E = 0 | | E = 1 | |
| Opcode | PSTATE.priv | ASI | W | TLB Miss | P = 0 | P = 1 | P = 0 | P = 1 |
| Load | 0 | PRIM, SEC | x | Dmiss | OK | Dexc | OK | Dexc |
| | | PRIM_NF, SEC_NF | x | Dmiss | OK | Dexc | Dexc | Dexc |
| | 1 | PRIM, SEC, NUC | x | Dmiss | OK | OK | OK | OK |
| | | PRIM_NF, SEC_NF | x | Dmiss | OK | OK | Dexc | Dexc |
| | | U_PRIM, U_SEC | x | Dmiss | OK | Dexc | OK | Dexc |
| Store or Atomic Load-Store | 0 | PRIM, SEC | 0 | Dmiss | Dprot | Dexc | Dprot | Dexc |
| | | | 1 | Dmiss | OK | Dexc | OK | Dexc |
| | 1 | PRIM, SEC, NUC | 0 | Dmiss | Dprot | Dprot | Dprot | Dprot |
| | | | 1 | Dmiss | OK | OK | OK | OK |
| | | U_PRIM, U_SEC | 0 | Dmiss | Dprot | Dexc | Dprot | Dexc |
| | | | 1 | Dmiss | OK | Dexc | OK | Dexc |
| FLUSH† | 0 | | x | Dmiss | OK | Dexc | OK | Dexc |
| | 1 | | x | Dmiss | OK | OK | Dexc | Dexc |
| x | 0 | BYPASS | x | *privileged_action* | | | | |
| | 1 | BYPASS | x | Bypass | | | | |

†. The FLUSH entry in this table only applies to JPS2 implementations that translate (as opposed to ignore) the address given in FLUSH instructions.

**TABLE F-4** IMMU Table of Operations for Normal ASIs

| Condition | Behavior | | |
|---|---|---|---|
| PSTATE.priv | TLB Miss | P = 0 | P = 1 |
| 0 | Imiss | OK | Iexc |
| 1 | Imiss | OK | OK |

# F.7 ASI Value, Context, and Endianness Selection for Translation

The selection of the context for a translation is the result of a two-step process:

1. The ASI is determined (conceptually by the Integer Unit) from the instruction, ASI register, trap level, and the virtual processor endian mode (`PSTATE.cle`).
2. The Context Register is determined directly from the ASI. The context value is read by the Context Register selected by the ASI.

The ASI value and endianness (little or big) are determined for the IMMU and DMMU, respectively, according to TABLE F-5 through TABLE F-7. Note that the secondary context is never used to fetch instructions. The Instruction and Data MMUs, when using the Primary Context identifier, use the value stored in the shared Primary Context Register.

The endianness of a data access is specified by three conditions:

- The ASI specified in the opcode or ASI register
- The `PSTATE` current little-endian bit (`cle`)
- The DMMU invert endianness bit

  The DMMU invert endianness bit does not affect the ASI value recorded in the `SFSR` but does invert the endianness that is otherwise specified for the access.

---

**Note –** The DMMU invert-endianness bit inverts the endianness for all accesses, including alternate space loads, stores, and atomic load-stores that specify an ASI. For example, `LDXA [%g1]ASI_PRIMARY_LITTLE` will be `_BIG` if the `ie` bit is on.

---

**TABLE F-5** ASI Mapping for Instruction Access

| Condition for Instruction Access | Resulting Action | |
|---|---|---|
| TL | Endianness | ASI Value (in SFSR) |
| 0 | BIG | ASI_PRIMARY |
| > 0 | BIG | ASI_NUCLEUS |

**TABLE F-6** ASI Mapping for Data Accesses

| Condition for Data Access | | | | Access Processed with: | |
|---|---|---|---|---|---|
| Opcode | TL | PSTATE.cle | DMMU.ie | Endianness | ASI Value (Recorded in SFSR) |
| LD/ST/Atomic Load-Store | 0 | 0 | 0 | BIG | ASI_PRIMARY |
| | | | 1 | LITTLE | ASI_PRIMARY |
| | | 1 | 0 | LITTLE | ASI_PRIMARY_LITTLE |
| | | | 1 | BIG | ASI_PRIMARY_LITTLE |
| | > 0 | 0 | 0 | BIG | ASI_NUCLEUS |
| | | | 1 | LITTLE | ASI_NUCLEUS |
| | | 1 | 0 | LITTLE | ASI_NUCLEUS_LITTLE |
| | | | 1 | BIG | ASI_NUCLEUS_LITTLE |
| LD/ST/Atomic Load-Store Alternate with specified ASI *not* ending in _LITTLE | x | x | 0 | BIG | Specified ASI value from immediate field in opcode or ASI Register |
| | | | 1 | LITTLE | |
| LD/ST/Atomic Load-Store Alternate with specified ASI ending in _LITTLE | x | x | 0 | LITTLE | Specified ASI value from immediate field in opcode or ASI Register |
| | | | 1 | BIG | |
| FLUSH† | 0 | x | x | — | ASI_PRIMARY_* |
| | >0 | x | x | — | ASI_NUCLEUS |

†. The FLUSH entry in this table only applies to JPS2 implementations that translate (as opposed to ignore) the address given in FLUSH instructions.

The Context Register used by the data and instruction MMUs is determined according to TABLE F-7. The Context Register selection is not affected by the endianness of the access. For a comprehensive list of ASI values in the ASI map, see Appendix L, *Address Space Identifiers (ASIs)*.

**TABLE F-7** IMMU and DMMU Context Register Usage

| ASI Value | Context Register |
|---|---|
| ASI_*NUCLEUS* (any ASI name containing the string "NUCLEUS") | Nucleus ($0000_{16}$ hard-wired) |
| ASI_*PRIMARY* (any ASI name containing the string "PRIMARY") | Primary |
| ASI_*SECONDARY* (any ASI name containing the string "SECONDARY") | Secondary |
| All other ASI values | (Not applicable, no translation) |

# F.8 Reset, Disable, and RED_state Behavior

During global reset of the virtual processor, the following statements apply:

- No change occurs in any block of the DMMU.
- No change occurs in the datapath or TLB blocks of the IMMU.
- The IMMU resets its internal state machine to normal (nonsuspended) operation.
- The IMMU and DMMU Enable bits in the DCU Control Register are set to 0.

When the virtual processor enters RED_state, the following statement applies:

- The IMMU and DMMU Enable bits in the DCU Control Register are set to 0.

Either of the MMUs is defined to be disabled when its respective MMU Enable bit equals 0 or, for the IMMU only, whenever the virtual processor is in RED_state. The DMMU is enabled or disabled solely by the state of the DMMU Enable bit.

When the DMMU is disabled:

- The DMMU passes all addresses through without translation ("bypasses" them); each address is truncated to the size of a physical address on the implementation (impl. dep. #224), behaving as if the ASI_PHYS_* ASI had been used for the access.
- The virtual processor behaves as if the TTE bits were set as:
  - TTE.ie ←0
  - TTE.p ←0
  - TTE.w ←1
  - TTE.nfo ←0

- If DCUCR.cp and DCUCR.cv are implemented (impl. dep. #232):
  - TTE.cp ← DCUCR.cp
  - TTE.cv ← DCUCR.cv
  - TTE.e ← **not** DCUCR.cp
- If DCUCR.cp and DCUCR.cv are not implemented:
  - TTE.e ← **not** TTE.cp

**IMPL. DEP. #231:** The variability of the width of physical address is implementation dependent in JPS2, and if variable, the initial width of the physical address after reset is also implementation dependent in JPS2.

**IMPL. DEP. #232:** Whether cp and cv bits exist in the DCU Control Register is implementation dependent in JPS2.

However, if a bypass ASI (ASI_PHYS_*) is used while the DMMU is disabled, the bypass operation behaves as it does when the DMMU is enabled; that is, the access is processed with the e, cp, and cv bits as specified by the bypass ASI (see TABLE F-15 on page 503).

When the IMMU is disabled, it truncates all instruction accesses to the physical address size (implementation dependent) and passes the default physically cacheable bit or implementation-dependent Data Cache Unit Control Register cp bit to the cache system. The access does not generate an *instruction_access_exception* trap.

When disabled, both the IMMU and DMMU correctly perform all LDXA and STXA operations to internal registers, and traps are signalled just as if the MMU were enabled. For instance, if a nonfaulting load is issued when the DMMU is disabled and DCUCR.cp is set to 0 if the implementation has the bit, then the DMMU signals a *data_access_exception* trap (ft = $02_{16}$), since e is set to 1.

**IMPL. DEP. #117:** Whether PREFETCH and nonfaulting loads always succeed when the MMU is disabled is implementation dependent.

---

**Note –** A reset of the TLB is not automatically performed by a virtual processor reset or by entry into RED_state. Before the MMUs are enabled, the operating system software must explicitly write each entry with either a valid TLB entry or an entry with the valid bit set to 0. The operation of the IMMU or DMMU in enabled mode is undefined if the TLB valid bits have not been set explicitly beforehand.

---

# F.9 SPARC V9 “MMU Requirements” Annex

The MMU complies completely with the SPARC V9 “MMU Requirements” Annex. TABLE F-8 shows how various protection modes can be achieved, if necessary, through the presence or absence of a translation in the instruction or data MMU. Note that this behavior requires specialized TLB-miss handler code to guarantee these conditions.

**TABLE F-8** MMU SPARC V9 Annex G Protection Mode Compliance

| Condition | | | |
|---|---|---|---|
| **TTE in DMMU** | **TTE in IMMU** | **Writable Attribute Bit** | **Resultant Protection Mode** |
| Yes | No | 0 | Read-only |
| No | Yes | N/A | Execute-only |
| Yes | No | 1 | Read/Write |
| Yes | Yes | 0 | Read-only/Execute |
| Yes | Yes | 1 | Read/Write/Execute |

# F.10 Internal Registers and ASI Operations

In this section, how to access MMU registers is described, followed by descriptions of the registers themselves, as follows:

- Context Registers
- Instruction/Data MMU TLB Tag Access Registers
- I/D TLB Data In, Data Access, and Tag Read Registers
- I/D TSB Tag Target Registers
- I/D TSB Base Registers
- I/D TSB Extension Registers
- I/D TSB 8-Kbyte and 64-Kbyte Pointer and Direct Pointer Registers
- I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR)
- MMU Data Synchronous Fault Address Register

The I/D demap operation is then described.

## F.10.1 Accessing MMU Registers

All internal MMU registers can be accessed directly by the virtual processor through defined ASIs. Several of the registers have been assigned their own ASI because these registers are crucial to the speed of the TLB miss handler. Allowing the use of `%g0` for the address reduces the number of instructions required to perform the access to the alternate space (by eliminating address formation).

See Appendix L, *Address Space Identifiers (ASIs)*, Section 5.2.4, *ASI-Accessible Registers*, and Appendix P, *Error Handling*for details on the behavior of the MMU during all other internal ASI accesses. For instance, to facilitate an access to the D-cache, the MMU performs a bypass operation.

---

**Caution –** A store to an MMU register requires a `MEMBAR #Sync`, `FLUSH`, `DONE`, or `RETRY` before the point that the effect must be visible to load, store, and atomic load-store accesses. A `FLUSH`, `DONE`, or `RETRY` is needed before the point that the effect must be visible to instruction accesses, that is, `MEMBAR #Sync` is not sufficient. In either case, one of these instructions must be executed before the next noninternal store or load of any type and on or before the delay slot of a delayed-control transfer instruction of any type. This action is necessary to avoid data corruption.

---

If the low-order three bits of the VA are nonzero in an `LDXA/STXA` to or from these registers, then a *mem_address_not_aligned* trap occurs. Writes to read-only, reads to write-only, illegal ASI values, or illegal VA for a given ASI can cause a *data_access_exception* trap (`ft` = $08_{16}$). (The hardware detects VA violations in only an unspecified lower portion of the virtual address.) TABLE F-9 describes MMU registers and provides references to sections with more details.

**TABLE F-9** MMU Internal Registers and ASI Operations

| IMMU ASI | DMMU ASI | VA<63:0> | Access | Register or Operation Name | Page |
|---|---|---|---|---|---|
| $50_{16}$ | $58_{16}$ | $0_{16}$ | Read-only | I/D TSB Tag Target Registers | 494 |
| — | $58_{16}$ | $8_{16}$ | Read/Write | Primary Context Register | 489 |
| — | $58_{16}$ | $10_{16}$ | Read/Write | Secondary Context Register | 489 |
| $50_{16}$ | $58_{16}$ | $18_{16}$ | Read/Write | I/D Synchronous Fault Status Registers (`I-SFSR`, `D-SFSR`) | 497 |
| — | $58_{16}$ | $20_{16}$ | Read-only | D Synchronous Fault Address Register (`D-SFAR`) | 500 |
| $50_{16}$ | $58_{16}$ | $28_{16}$ | Read/Write | I/D TSB Base Registers | 494 |
| $50_{16}$ | $58_{16}$ | $30_{16}$ | Read/Write | I/D TLB Tag Access Registers | 490 |
| — | $58_{16}$ | $38_{16}$ | Read/Write | Virtual Watchpoint Address | 94 |

**TABLE F-9** MMU Internal Registers and ASI Operations *(Continued)*

| IMMU ASI | DMMU ASI | VA<63:0> | Access | Register or Operation Name | Page |
|---|---|---|---|---|---|
| — | $58_{16}$ | $40_{16}$ | Read / Write | Physical Watchpoint Address | 94 |
| $50_{16}$ | $58_{16}$ | $48_{16}$ | Read / Write | I / D TSB Primary Extension Registers Implementation dependent (impl. dep. #233) | 496 |
| —[†] | $58_{16}$ | $50_{16}$ | Read / Write | D TSB Secondary Extension Register Implementation dependent (impl. dep. #233) | 496 |
| $50_{16}$ | $58_{16}$ | $58_{16}$ | Read / Write | I / D TSB Nucleus Extension Registers Implementation dependent (impl. dep. #233) | 496 |
| $51_{16}$ | $59_{16}$ | $0_{16}$ | Read-only | I / D TSB 8-Kbyte Pointer Registers | 490 |
| $52_{16}$ | $5A_{16}$ | $0_{16}$ | Read-only | I / D TSB 64-Kbyte Pointer Registers | 490 |
| — | $5B_{16}$ | $0_{16}$ | Read-only | D TSB Direct Pointer Register | 490 |
| $54_{16}$ | $5C_{16}$ | $0_{16}$ | Write-only | I / D TLB Data In Registers | 491 |
| $55_{16}$ | $5D_{16}$ | $0_{16}$–$20FF8_{16}$ | Read / Write | I / D TLB Data Access Registers | 491 |
| $55_{16}$ | $5D_{16}$ | $40000_{16}$–$60FF8_{16}$ | — | Implementation dependent (impl. dep. #239) | 437 |
| $56_{16}$ | $5E_{16}$ | $0_{16}$–$20FF8_{16}$ | Read-only | I / D TLB Tag Read Registers | 491 |
| $57_{16}$ | $5E_{16}$ | See F.10.11 | Write-only | I / D MMU Demap Operations | 500 |

†. For symmetry, a "dummy" register exists at ASI $50_{16}$, VA$50_{16}$ that reads as zero and to which writes are ignored.

**IMPL. DEP. #233:** Whether `tsb_hash` field is implemented in I / D Primary / Secondary / Nucleus TSB Extension Register is implementation dependent in JPS2.

**IMPL. DEP. #239:** The register(s) accessed by IMMU ASI $55_{16}$ and DMMU ASI $5D_{16}$ at virtual addresses $40000_{16}$ to $60FF8_{16}$ are implementation dependent.

## F.10.2 Context Registers

The Primary Context Register is shared by the IMMU and the DMMU and resides in the MMU. The Primary Context Register is illustrated in FIGURE F-5,

***where:*** **`PContext`** is the context identifier for the primary address space.

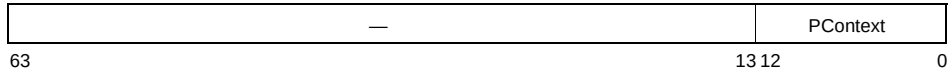



**FIGURE F-5** IMMU and DMMU Primary Context Register

The Secondary Context Register is illustrated in FIGURE F-6,

***where:*** **`SContext`** is the context identifier for the secondary address space.

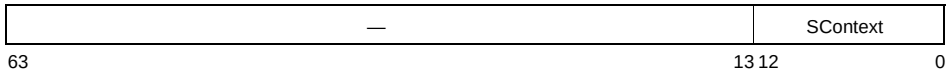



**FIGURE F-6** DMMU Secondary Context Register

The Nucleus Context Register is hardwired to zero, as illustrated in FIGURE F-7.

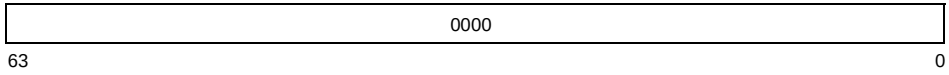



**FIGURE F-7** DMMU Nucleus Context Register

---

**V9 Compatibility Note –** The single Context Register of the SPARC V8 Reference MMU has been replaced by three separate context registers.

---

## F.10.3 Instruction / Data MMU TLB Tag Access Registers

In each MMU, the Tag Access Register is used as a temporary buffer for writing the TLB Entry tag information. The Tag Access Register holds the tag portion, and the Data In or Data Access Register holds the data being accessed.

The Tag Access Register can be updated during either of the following operations:

1. When the MMU signals a trap due to a miss, exception, or protection: The MMU hardware, with one exception, automatically writes the missing VA and the appropriate context into the Tag Access Register to facilitate formation of the TSB Tag Target Register. The exception is that after a *data_access_exception*, the contents of the `context` field of the D-MMU Tag Access Register are undefined. See TABLE F-2 on page 479 for the `SFSR` and Tag Access Register update policy.
2. An ASI write to the Tag Access Register: Before an ASI store to the TLB data access registers, the operating system must set the Tag Access Register to the values desired in the TLB Entry. Note that an ASI store to the TLB Data In Register for automatic replacement also uses the Tag Access Register, but typically the value written into the Tag Access Register by the MMU hardware is appropriate.

---

**Note –** Any update to the Tag Access Registers immediately affects the data that are returned from subsequent reads of the TSB Tag Target and TSB Pointer Registers.

---

The TLB Tag Access Register fields are defined below and illustrated in FIGURE F-8.

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:13 | VA | RW | The 51-bit virtual page number. |
| 12:0 | Context | RW | The 13-bit context identifier. This field reads 0 when there is no associated context with the access. Its contents in the D-MMU are undefined after a *data_access_exception*. |

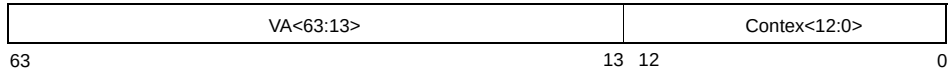



**FIGURE F-8** I/D MMU TLB Tag Access Registers

## F.10.4 I/D TLB Data In, Data Access, and Tag Read Registers

Access to the TLB is complicated because of the need to provide an atomic write of a TLB entry data item (tag and data) that is larger than 64 bits, the need to replace entries automatically through the TLB entry replacement algorithm as well as to provide direct diagnostic access, and the need for hardware assist in the TLB miss handler.

TABLE F-2 on page 479 shows when loads and stores update the Tag Access Registers.

TABLE F-10 shows how the Tag Read, Tag Access, Data In, and Data Access Registers interact to provide atomic reads and writes to the TLBs.

**TABLE F-10** MMU TLB Access Summary

| Software Operation | | Effect on MMU Physical Registers | | |
|---|---|---|---|---|
| **Load/Store** | **Register** | **TLB tag array** | **TLB data array** | **Tag Access Register** |
| Load | Tag Read | Contents returned. Entry specified by STXA's access. | No effect | No effect |
| | Tag Access | No effect | No effect | Contents returned |
| | Data In | Trap with *data_access_exception*. | | |
| | Data Access | No effect | Contents returned. Entry specified by STXA's access. | No effect |

**TABLE F-10** MMU TLB Access Summary *(Continued)*

| Software Operation | | Effect on MMU Physical Registers | | |
|---|---|---|---|---|
| **Load/Store** | **Register** | **TLB tag array** | **TLB data array** | **Tag Access Register** |
| Store | Tag Read | Trap with *data_access_exception*. | | |
| | Tag Access | No effect | No effect | Written with store data |
| | Data In | TLB entry determined by replacement policy written with contents of Tag Access Register | TLB entry determined by replacement policy written with store data | No effect |
| | Data Access | TLB entry specified by STXA address written with contents of Tag Access Register | TLB entry specified by STXA address written with store data | No effect |
| TLB miss | | No effect | No effect | Written with VA and context of access |

An ASI load from the TLB Tag Read Register initiates an internal read of the tag portion of the specified TLB entry.

### F.10.4.1 Data In and Data Access Registers

The Data In and Data Access Registers are the means of reading and writing the TLB for all operations. The TLB Data In Register is used for TLB miss handler automatic replacement writes. The TLB Data Access Register is used for operating system and diagnostic directed writes (writes to a specific TLB entry).

An ASI load from the TLB Data Access Register initiates an internal read of the data portion of the specified TLB entry.

ASI loads from the TLB Data In Register are not supported.

An ASI store to the TLB Data In Register initiates an automatic atomic replacement of the TLB Entry pointed to by an internal register that is updated by an implementation-dependent replacement algorithm. The TLB data and tag are formed as in the case of an ASI store to the TLB Data Access Register.

**IMPL. DEP. #234:** The replacement algorithm of a TLB entry is implementation dependent in JPS2.

**Caution –** Stores to the Data In Register are not guaranteed to replace the previous TLB entry, causing a fault. In particular, to change an entry's attribute bits, software must explicitly demap the old entry before writing the new entry; otherwise, a multiple match error condition can result.

Both the TLB Data In Register and the TLB Data Access Register use the TTE format shown in FIGURE F-3 on page 470. Refer to the description of the TTE data in *Translation Table Entry (TTE)* on page 470 for a complete description of the data fields. Implementations may use part of the TLB index addresses for implementation-dependent diagnostic purposes; in this case, the data format is also implementation dependent. Please refer to the JPS2 Extension documents for details.

**IMPL. DEP. #235:** The MMU TLB data access address assignment and the purpose of the address are implementation dependent in JPS2.

Writes to the TLB Data In Register require the virtual address to be set to 0. The format of the TLB Data Access Register virtual address is illustrated in FIGURE F-9, ***where:*** **`tlb index`** is the entry number to be accessed; the TLB organization (number of TLBs, size, associativity) is implementation dependent. The format of this field is implementation dependent; please refer to theJPS2 Extension documents.

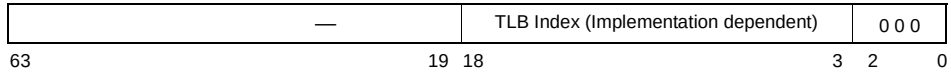



**FIGURE F-9** MMU TLB Data Access Address

## F.10.4.2 I/D MMU TLB Tag Read Register

The format for the Tag Read Register virtual address is described below and illustrated in FIGURE F-10.

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:13 | **va** | RW | The 51-bit virtual page number. In the fully associative TLB, page offset bits for larger page sizes are stored in the TLB; that is, va<15:13>, va<18:13>, and va<21:13> for 64-Kbyte, 512-Kbyte, and 4-Mbyte pages, respectively. These values are ignored during normal translation. When read, an implementation will return either 0 or the value previously written to them (impl. dep. #238). |
| 11:0 | I/D **`Context`** | RW | The 13-bit context identifier. |

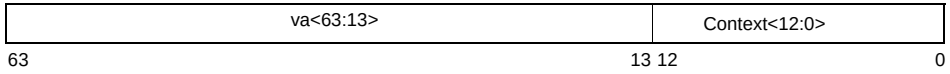



**FIGURE F-10** I/D MMU TLB Tag Read Registers

### F.10.4.3 I/D MMU TLB Tag Access Register

An ASI store to the TLB Data Access or Data In Register initiates an internal atomic write to the specified TLB Entry. The TLB entry data are obtained from the store data, and the TLB entry tag is obtained from the current contents of the TLB Tag Access Register.

## F.10.5 I/D TSB Tag Target Registers

The I and D TSB Tag Target Registers are simply bit-shifted versions of the data stored in the I and D Tag Access Registers, respectively. Since the I or D Tag Access Register is updated on an I or D TLB miss, respectively, the I and D Tag Target Registers appear to software to be updated on an I or D TLB miss. The MMU Tag Target Register is described below and illustrated in FIGURE F-11.

| Bit | Field | Type | Description |
|---|---|---|---|
| 60:48 | **Context**<12:0> | R | The context associated with the missing virtual address. |
| 41:0 | **va**<63:22> | R | The most significant bits of the missing virtual address. |

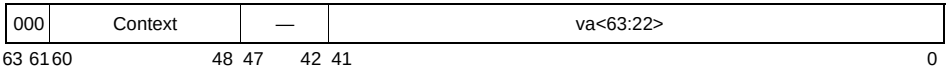



**FIGURE F-11** MMU Tag Target Registers

## F.10.6 I/D TSB Base Registers

The Translation Storage Buffer (TSB) Base registers provide information for the hardware formation of TSB pointers and tag target, to assist software in quickly handling TLB misses. If the TSB concept is not employed in the software memory management strategy and therefore the Pointer and Tag Access Registers are not used, then the TSB Base registers need not contain valid data.

The TSB Base Register is illustrated in FIGURE F-12 and described in TABLE F-11.

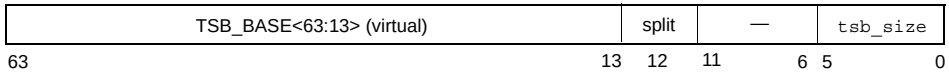



**FIGURE F-12** MMU I/D TSB Base Registers

**TABLE F-11** TSB Base Register Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:13 | I/D **tsb_base** | RW | Provides the base virtual address of the Translation Storage Buffer. Software must ensure that the TSB base address is aligned on a boundary equal to the size of the TSB (or both TSBs in the case of a split TSB). |
| 12 | **split** | RW | When split = 1, the TSB 64-Kbyte pointer address is calculated assuming separate (but abutting and equally sized) TSB regions for the 8-Kbyte and the 64-Kbyte TTEs. In this case, TSB_Size refers to the size of each TSB. The TSB 8-Kbyte pointer address calculation is not affected by the value of the Split bit. When Split = 0, the TSB 64-Kbyte pointer address is calculated assuming that the same lines in the TSB are shared by 8-Kbyte and 64-Kbyte TTEs, called a "common TSB" configuration.<br>**Caution:** In the "common TSB" configuration (TSB.split = 0), 8-Kbyte and 64-Kbyte page TTEs can conflict unless the TLB-miss handler explicitly checks the TTE for page size. Therefore, do not use the common TSB mode in an optimized handler. For example, suppose an 8-Kbyte page at VA = $2000_{16}$ and a 64-Kbyte page at VA = $10000_{16}$ both exist—a legal situation. These both map to the second TSB line (line 1) and have the same VA tag of 0. Therefore, there is no way for the miss handler to distinguish these TTEs by the TTE tag alone, and unless the miss handler checks the TTE data, it may load an incorrect TTE. |
| 5:0 | I/D **tsb_size** | RW | **IMPL. DEP. #236**: The width of the tsb_size field in the TSB Base Register is implementation dependent; the permitted range is from 2 to 6 bits. The least significant bit of tsb_size is always at bit 0 of the TSB Base Register. Any bits unimplemented at the most significant end of tsb_size read as 0, and writes to them are ignored.<br>The tsb_size field provides the size of the TSB as follows:<br>• The number of entries in the TSB (or each TSB if split) = $512 \times 2^{TSB\_Size}$.<br>• The number of entries in the TSB ranges from 512 entries at TSB_Size = 0 (8-Kbyte common TSB, 16-Kbyte split TSB), to an implementation-dependent number of entries.<br>**Note:** Any update to the TSB Base Register immediately affects the data that are returned from later reads of the Tag Target and TSB Pointer Registers. |

## F.10.7 I/D TSB Extension Registers

The TSB Extension Registers provide information for the hardware formation of TSB pointers and tag target, to assist software in handling TLB misses quickly. If the TSB concept is not employed in the software memory management strategy and therefore the pointer and Tag Access Registers are not used, then the TSB Extension Registers need not contain valid data.

The TSB Extension registers are defined as follows:

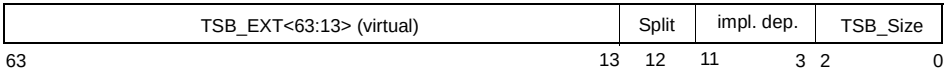



**FIGURE F-13** MMU I/D TSB Extension Registers

The register field definitions are the same as for I/D TSB Base Registers (Section F.10.6) except for an implementation-dependent field in bits 11:3. For the definition of the implementation-dependent field, refer to impl. dep. #233. The field can be used either as a `tsb_hash`, which is a representation of the context that generated the TLB miss, or as an extension to the `tsb_size` field, depending on the implementation. In the latter case, TSB pointer generation logic must incorporate the context ID into the process of TSB pointer generation, as described in *TSB Pointer Formation* on page 475. See also impl. dep. #228.

There are three TSB Extension Registers, one for each of the virtual address spaces (Primary, Secondary, Nucleus); see TABLE F-9 on page 488 for the ASI and VA of each register. Note that there is no Instruction TSB Secondary Extension Register.

When an I/D TLB miss occurs, an appropriate TSB Extension Register is selected and XORed either with the I/D TSB Register or with context ID, depending on the implementation. The result is then used to form a TSB pointer, as described in *TSB Pointer Formation* on page 475 and in each of the JPS2 Extension documents.

## F.10.8 I/D TSB 8-Kbyte and 64-Kbyte Pointer and Direct Pointer Registers

The I/D TSB 8-Kbyte and 64-Kbyte registers are provided as an aid to software in determining the location of the missing or trapping TTE in the software-maintained TSB. The TSB 8-Kbyte and 64-Kbyte Pointer Registers provide the possible locations of the 8-Kbyte and 64-Kbyte TTE, respectively.

As a fine point, the bit that controls selection of 8-Kbyte or 64-Kbyte address formation for the Direct Pointer Register is a state bit in the DMMU that is updated during a *fast_data_access_protection* exception. It records whether the page that hit in the TLB was a 64-Kbyte page or a non-64-Kbyte page, in which case, 8 Kbyte is assumed.

The registers are illustrated in FIGURE F-14,

***where:*** **VA<63:4>** is the full virtual address of the TTE in the TSB, as determined by the MMU hardware, and is described in *Hardware Support for TSB Access* on page 475.

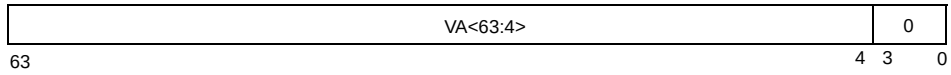



**FIGURE F-14** I/D MMU TSB 8-Kbyte/64-Kbyte Pointer and DMMU Direct Pointer Register

### F.10.8.1 TSB 8-Kbyte and 64-Kbyte Pointer Registers

The TSB Pointer Registers are implemented as a reorder of the current data stored in the Tag Access Register and the TSB Extension Register. If the Tag Access Register or TSB Extension Register is updated through a direct software write (through an `STXA` instruction), then the values in the Pointer Registers will be updated as well.

### F.10.8.2 Direct Pointer Register

The Direct Pointer Register is mapped by hardware to either the 8-Kbyte or 64-Kbyte Pointer Register in the case of a *fast_data_access_protection* exception according to the known size of the trapping TTE. In the case of a 512-Kbyte or 4-Mbyte page miss, the Direct Pointer Register returns the pointer as if the fault were from an 8-Kbyte page.

## F.10.9 I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR)

The IMMU and DMMU each maintain their own `SFSR` Register. The `SFSR` is illustrated in FIGURE F-15 and described in TABLE F-12.

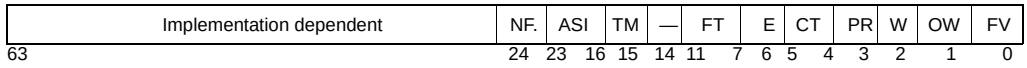



**FIGURE F-15** MMU I/D Synchronous Fault Status Registers (`I-SFSR`, `D-SFSR`)

**TABLE F-12** SFSR Bit Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:25 | — | | Implementation dependent. Refer to JPS2 Extension documents. |
| 24 | nf | RW | Set in the DMMU if the faulting instruction was a nonfaulting load (a load to ASI_NOFAULT) (IMMU = 0). NF is always 0 in I-SFSR. |
| 23:16 | asi | RW | Records the 8-bit ASI associated with the faulting instruction. This field is valid for both DMMU and IMMU SFSRs and for all traps in which the FV bit is set. A trapping alternate space load or store sets the ASI field to the ASI the instruction attempted to reference. A trapping non-alternate-space load or store sets ASI to ASI_PRIMARY if PSTATE.cle = 0 or to ASI_PRIMARY_LITTLE if PSTATE.cle = 1. A *mem_address_not_aligned* trap caused by a JMPL or RETURN either does not set DSFSR.asi or sets it as would a trapping non-alternate-space load or store. (impl. dep. #237) |
| 15 | tm | RW | I/D TLB miss. |
| 11:7 | ft | RW | Specifies the exact condition that caused the recorded fault, according to TABLE F-13 following this table. In the DMMU, the Fault Type field is valid only for *data_access_exception* faults; there is no ambiguity in all other MMU trap cases. Note that the hardware does not priority-encode the bits set in the fault type (FT) field; that is, multiple bits can be set. In particular, the following ASI stores could set both the $01_{16}$ and $08_{16}$ Fault Type bits (the page is privileged, as well as storing to a read-only ASI):<br>`stda %g0, [%g4]ASI_PRIMARY_NO_FAULT`<br>`stda %g0, [%g4]ASI_SECONDARY_NO_FAULT`<br>`stda %g0, [%g4]ASI_PRIMARY_NO_FAULT_LITTLE`<br>`stda %g0, [%g4]ASI_SECONDARY_NO_FAULT_LITTLE`<br>The FT field in the IMMU SFSR always reads 0 for *fast_instruction_access_MMU_miss* and reads $01_{16}$ for *instruction_access_exception*, as all other fault types do not apply. |
| 6 | e | RW | Side-effect bit. Associated with the faulting data access or flush instruction. Set by translating ASI accesses (see *ASI Values* on page 568) that are mapped by the TLB with the E bit set and bypass ASIs $15_{16}$ and $1D_{16}$. Other cases that update the SFSR (including bypass or internal ASI accesses) set the E bit to 0. It always reads as 0 in the IMMU. |
| 5:4 | ct | RW | Context Register selection, as described below. The context is set to $11_2$ when the access does not have a translating ASI.<br>**Context ID** / **IMMU Context**† / **DMMU Context**<br>00 / Primary / Primary<br>01 / Reserved / Secondary<br>10 / Nucleus / Nucleus<br>11 / Reserved / (None of the above)<br>†. When the ITLB is bypassed, the value of ISFSR.ct is undefined. |
| 3 | pr | RW | Privilege bit. Set if the faulting access occurred while in privileged mode. This field is valid for all traps in which the FV bit is set. |

**TABLE F-12** SFSR Bit Description *(Continued)*

| Bit | Field | Type | Description |
|---|---|---|---|
| 2 | w | RW | Write bit. Set if the faulting access indicated a data write operation (a store or atomic load-store instruction). This bit always reads as 0 in the IMMU SFSR. |
| 1 | OW | RW | Overwrite bit. When the MMU detects a fault, the Overwrite bit is set to 1 if the Fault Valid bit has not been cleared from a previous fault; otherwise, it is set to 0. |
| 0 | FV | RW | Fault Valid bit. Set when the MMU detects a fault; it is cleared only on an explicit ASI write of 0 to the SFSR. This bit is not set on an MMU miss. Therefore, overwrites of MMU misses cannot be detected.<br>When the Fault Valid bit is not set, the values of the remaining fields in the SFSR and SFAR are undefined for traps other than an MMU miss. |

TABLE F-13 describes the SFSR fault type field (FT<11:7>).

**TABLE F-13** MMU Synchronous Fault Status Register ft (Fault Type) Field

| I/D | FT[6:0] | Fault Type |
|---|---|---|
| I/D | $01_{16}$ | Privilege violation. |
| D | $02_{16}$ | Nonfaulting load instruction to page marked with E bit. This bit is 0 for internal ASI accesses. |
| D | $04_{16}$ | Atomic (including 128-bit atomic load, Load Quadword Atomic) to page marked noncacheable. |
| D | $08_{16}$ | Illegal LDA/STA ASI value, VA, RW, or size. Does not include cases where $02_{16}$ and $04_{16}$ are set. |
| D | $10_{16}$ | Access other than nonfaulting load to page marked NFO. This bit is 0 for internal ASI accesses. |

**Note –** A *fast_instruction_MMU_miss* or a *fast_data_access_MMU_miss* trap causes the SFSR and the SFAR to be overwritten without setting either the ow or the fv bits.

The SFSR and the Tag Access registers both maintain state concerning a previous translation causing an exception. The update policy for the SFSR and the Tag Access Registers is shown in TABLE F-2 on page 479.

## F.10.10 Synchronous Fault Addresses

This section describes how the IMMU and DMMU obtain a fault address.

### F.10.10.1 IMMU Fault Address

There is no IMMU Synchronous Fault Address Register. Instead, software must read the TPC register appropriately as discussed here.

For *fast_instruction_access_MMU_miss* traps, TPC contains the virtual address that was not found in the IMMU TLB.

For *instruction_access_exception* traps, "privilege violation" fault type, TPC contains the virtual address of the instruction in the privileged page that caused the exception.

### F.10.10.2 DMMU Fault Address

The Data Synchronous Fault Address Register contains the virtual memory address of the fault recorded in the DMMU Synchronous Fault Status Register. The D-SFAR can be thought of as an additional field of the D-SFSR.

The D-SFAR register is illustrated in FIGURE F-16, where **Fault Address** is the virtual address associated with the translation fault recorded in the D-SFSR; the field is set on an MMU miss fault or when the D-SFSR Fault Valid (fv) bit is set.

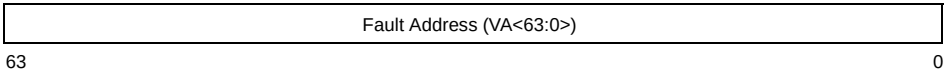



**FIGURE F-16** MMU Data Synchronous Fault Address Register (D-SFAR)

The D-SFAR register is writable, using an STXA instruction.

## F.10.11 I/D MMU Demap

Demap is an MMU operation, not an MMU register. Demap removes selected entries from the TLBs.

---

**Note –** A store to a DMMU Register requires a MEMBAR #Sync, FLUSH, DONE, or RETRY before the point that the effect must be visible to load, store, and atomic load-store accesses. A FLUSH, DONE, or RETRY is needed before the point that the effect must be visible to instruction accesses, that is, MEMBAR #Sync is not sufficient. In either case, one of these instructions must be executed before the next noninternal store or load of any type and on or before the delay slot of a delayed-control transfer instruction of any type. This action is necessary to avoid data corruption.

---

Three types of demap operations are provided:

- **Demap page —** Removes any TLB entry that matches exactly the specified virtual page and context number. It is illegal to have more than one TLB entry per page.

  Demap page may, in fact, remove more than one TLB entry in the condition of a multiple TLB match, but this is an error condition of the TLB and has undefined results.
- **Demap context —** Removes any TLB entries that match the specified context identifier.
- **Demap all —** Removes all of the TLB entries from the TLB except for locked entries.

Demap is initiated by an `STXA` with ASI $57_{16}$ for IMMU demap or $5F_{16}$ for DMMU demap. It removes TLB entries from an on-chip TLB. No bus-based demap is supported. The demap address format is illustrated in FIGURE F-17 and described in TABLE F-14.

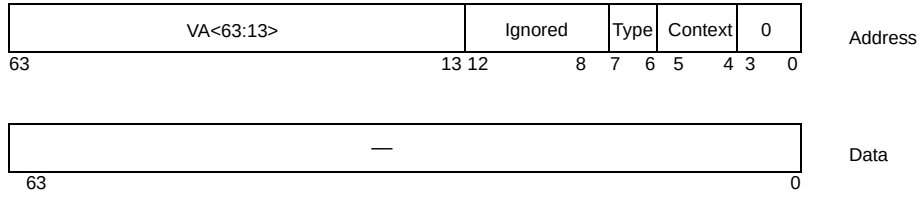



**FIGURE F-17** MMU Demap Operation Address and Data Formats

**TABLE F-14** Demap Address Format

| Field | Bit | Type | Description |
|---|---|---|---|
| 63:13 | **VA**<63:13> | RW | The virtual page number of the TTE to be removed from the TLB for Demap Page. |
| 12:8 | Ignored | | This field is ignored by hardware. |
| 7:6 | `Type` | RW | The type of demap operation, as described below:<br>**Type Field** — **Demap Operation**<br>0 — Demap page— see page 501<br>1 — Demap context—see page 501<br>2 — Demap all—see page 501<br>3 — Reserved—Ignored |
| 5:4 | **`Context` ID** | RW | Context Register selection, as described below. Use of the reserved value causes the demap to be ignored<br>**Context ID Field** — **Context Used in Demap**<br>00 — Primary<br>01 — Secondary (DMMU only)<br>10 — Nucleus<br>11 — Reserved |

A demap operation does not invalidate the TSB in memory. Software must modify the appropriate TTEs in the TSB before initiating a demap operation.

Except for Demap All, the demap operation does not depend on the value of any entry's lock bit. A demap operation demaps both locked entries and unlocked entries.

The demap operation produces no output.

Following are Instruction/Data demap page types:

- **I/D Demap Page (`Type` = 0).** Demap Page removes the TTE (from the specified TLB), matching the specified virtual page number and Context Register. The match condition with regard to the global bit is the same as a normal TLB access; that is, if the global bit is set, the contexts do not need to match.

  Virtual page offset bits 15:13, 18:13, and 21:13 for 64-Kbyte, 512-Mbyte, and 4-Mbyte page TLB entries, respectively, do not participate in the match for that entry. This is the same condition as for a translation match.

---

**Note –** Each Demap Page operation removes only one TLB entry. A demap of a 64-Kbyte, 512-Kbyte, or 4-Mbyte page does not demap any smaller page within the specified virtual address range.

---

- **I/D Demap Context (`Type` = 1).** Demap Context removes from the TLB all TTEs having the specified context. If the TTE Global bit is set, then the TTE is not removed. `va` is ignored for this operation.
- **I/D Demap All (`Type` = 2).** Demap All removes all TTEs that do not have the lock bit set. `va` and `context` are ignored for this operation.

# F.11 MMU Bypass

In a bypass access, the DMMU sets the physical address equal to the truncated virtual address; that is, the low-order bits of the virtual address are passed through without translation as the physical address (the width of which is defined in impl. dep. #224). The physical page attribute bits are set as shown in TABLE F-15.

**TABLE F-15** Bypass Attribute Bits

| | Attribute Bits | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **ASI** | **CP** | **IE** | **CV** | **E** | **P** | **W** | **NFO** | **Size** |
| ASI_PHYS_USE_EC<br>ASI_PHYS_USE_EC_LITTLE | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 8 Kbyte |
| ASI_PHYS_BYPASS_EC_WITH_EBIT<br>ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 8 Kbyte |

The IMMU can only be bypassed by being disabled. See *Reset, Disable, and RED_state Behavior* on page 485 for details on the effect of disabling the MMU.

**V9 Compatibility Note –** The virtual address is wider than the physical address; thus, there is no need to use multiple ASIs to fill in the high-order physical address bits, as is done in SPARC V8 machines.

# F.12 Translation Lookaside Buffer Hardware

This section briefly describes the TLB hardware. For more detailed information, refer to the JPS2 Extension documents or the corresponding microarchitecture specification.

## F.12.1 TLB Operations

The TLB supports exactly one of the following operations:

- **Normal translation.** The TLB receives a virtual address and a context identifier as input and produces a physical address and page attributes as output.

- **Bypass.** The TLB receives a virtual address as input and produces a physical address equal to the truncated virtual address and default page attributes as output.
- **Demap operation.** The TLB receives a virtual address, a context identifier, and type as input and sets the Valid bit to zero for any matching page entries.
- **Read operation.** The TLB reads either the Tag or Data portion of the specified entry. (Since the TLB entry is greater than 64 bits, the Tag and Data portions must be returned in separate reads. See *I/D TLB Data In, Data Access, and Tag Read Registers* on page 491.)
- **Write operation.** The TLB simultaneously writes the Tag and Data portion of the specified entry or the entry given by the replacement policy described below.
- **No operation.** The TLB performs no operation.

## F.12.2 TLB Replacement Policy

On an automatic replacement write to the TLB, the MMU picks the entry to write, based on an implement-dependent algorithm. Please refer to the Implementation Extension Supplements for details of the TLB replacement algorithm.

## F.12.3 TSB Pointer Logic Hardware Description

The algorithm used to generate the I/D TSB pointer is implementation dependent. Please refer to the JPS2 Extension documents for details of the TSB pointer generation algorithm.

# Assembly Language Syntax

This appendix supports Appendix A, *Instruction Definitions*. Each instruction description in Appendix A includes a table that describes the suggested assembly language format for that instruction. This appendix describes the notation used in those assembly language syntax descriptions and lists some synthetic instructions provided by the SPARC JPS2 assemblers for the convenience of assembly language programmers.

The appendix contains these sections:


- *Notation Used* on page 505
- *Syntax Design* on page 514
- *Synthetic Instructions* on page 514


## G.1 Notation Used

The notations defined here are also used in the assembly language syntax descriptions in Appendix A, *Instruction Definitions*.

Items in `typewriter font` are literals to be written exactly as they appear. Items in *italic font* are metasymbols that are to be replaced by numeric or symbolic values in actual SPARC V9 assembly language code. For example, "*imm_asi*" would be replaced by a number in the range 0 to 255 (the value of the *imm_asi* bits in the binary instruction) or by a symbol bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, $reg_{rs2}$ is a *reg* (register name) whose binary value will be placed in the `rs2` field of the resulting instruction.

## G.1.1 Register Names

***reg*** A *reg* is an integer register name. It can have any of the following values:[1]

`%r0`–`%r31`
`%g0`–`%g7`(*global* registers; same as `%r0`–`%r7`)
`%o0`–`%o7` (*out* registers; same as `%r8`–`%r15`)
`%l0`–`%l7`(*local* registers; same as `%r16`–`%r23`)
`%i0`–`%i7`(*in* registers; same as `%r24`–`%r31`)
`%fp`(frame pointer; conventionally same as `%i6`)
`%sp` (stack pointer; conventionally same as `%o6`)

Subscripts identify the placement of the operand in the binary instruction as one of the following:

$reg_{rs1}$(`rs1` field)
$reg_{rs2}$(`rs2` field)
$reg_{rd}$(`rd` field)

***freg*** An *freg* is a floating-point register name. It may have the following values:

`%f0`, `%f1`, `%f2`–`%f63`

See *Floating-Point `f` Registers* on page 46.

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

$freg_{rs1}$(`rs1` field)
$freg_{rs2}$(`rs2` field)
$freg_{rs3}$(`rs3` field)
$freg_{rd}$(`rd` field)

***asr_reg*** An *asr_reg* is an Ancillary State Register name. It may have one of the following values:

`%asr16`–`%asr31`

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

$asr\_reg_{rs1}$(`rs1` field)
$asr\_reg_{rd}$(`rd` field)

---

1. In actual usage, the `%sp`, `%fp`, `%g`*n*, `%o`*n*, `%l`*n*, and `%i`*n* forms are preferred over `%r`*n*.

***i_or_x_cc*** An *i_or_x_cc* specifies a set of integer condition codes**,** those based on either the 32-bit result of an operation (`icc`) or on the full 64-bit result (`xcc`). It may have either of the following values:

```
%icc
%xcc
```

***fccn*** An *fccn* specifies a set of floating-point condition codes. It can have any of the following values:

```
%fcc0
%fcc1
%fcc2
%fcc3
```

## G.1.2 Special Symbol Names

Certain special symbols appear in the syntax table in `typewriter font`. They must be written exactly as they are shown, including the leading percent sign (`%`).

The symbol names and the registers or operators to which they refer are as follows:

| | |
|---|---|
| `%asi` | Address Space Identifier Register |
| `%canrestore` | Restorable Windows Register |
| `%cansave` | Savable Windows Register |
| `%ccr` | Condition Codes Register |
| `%cleanwin` | Clean Windows Register |
| `%clear_softint` | Soft Interrupt Register (clear selected bits) |
| `%cwp` | Current Window Pointer Register |
| `%dcr` | Dispatch Control Register |
| `%fprs` | Floating-Point Registers State Register |
| `%fsr` | Floating-Point State Register |
| `%gsr` | Graphics Status Register |
| `%otherwin` | Other Windows Register |
| `%pc` | Program Counter Register |
| `%pcr` | Performance Control Register |
| `%pic` | Performance Instrumentation Counters |
| `%pil` | Processor Interrupt Level Register |
| `%pstate` | Processor State Register |
| `%set_softint` | Soft Interrupt Register (set selected bits) |
| `%softint` | Soft Interrupt Register |

| | |
|---|---|
| %sys_tick | System Timer (TICK) Register |
| %sys_tick_cmpr | System TImer (STICK) Compare Register |
| %tba | Trap Base Address Register |
| %tick | Tick (cycle count) Register |
| %tick_cmpr | Timer (TICK) Compare Register |
| %tl | Trap Level Register |
| %tnpc | Trap Next Program Counter Register |
| %tpc | Trap Program Counter Register |
| %tstate | Trap State Register |
| %tt | Trap Type Register |
| %ver | Version Register |
| %wstate | Window State Register |
| %y | Y Register |

The following special symbol names are unary operators that perform the functions described:

| | |
|---|---|
| %uhi | Extracts bits 63:42 (high 22 bits of upper word) of its operand |
| %ulo or %hm | Extracts bits 41:32 (low-order 10 bits of upper word) of its operand |
| %hi or %lm | Extracts bits 31:10 (high-order 22 bits of low-order word) of its operand |
| %lo | Extracts bits 9:0 (low-order 10 bits) of its operand |

Certain predefined value names appear in the syntax table in typewriter font. They must be written exactly as they are shown, including the leading sharp sign (#). The value names and the values to which they refer are listed in TABLE G-1.

**TABLE G-1** Value Names and Values *(1 of 4)*

| Value Name in Assembly Language | Value | Description |
|---|---|---|
| #n_reads | 0 | (for PREFETCH instruction) |
| #one_read | 1 | (for PREFETCH instruction) |
| #n_writes | 2 | (for PREFETCH instruction) |
| #one_write | 3 | (for PREFETCH instruction) |
| #page | 4 | (for PREFETCH instruction) |
| #Sync | $40_{16}$ | (for MEMBAR instruction cmask field) |
| #MemIssue | $20_{16}$ | (for MEMBAR instruction cmask field) |
| #Lookaside | $10_{16}$ | (for MEMBAR instruction cmask field) |
| #StoreStore | $08_{16}$ | (for MEMBAR instruction mmask field) |
| #LoadStore | $04_{16}$ | (for MEMBAR instruction mmask field) |

**TABLE G-1** Value Names and Values *(2 of 4)*

| Value Name in Assembly Language | Value | Description |
|---|---|---|
| #StoreLoad | $02_{16}$ | (for MEMBAR instruction mmask field) |
| #LoadLoad | $01_{16}$ | (for MEMBAR instruction mmask field) |
| #ASI_AIUP | $10_{16}$ | ASI_AS_IF_USER_PRIMARY |
| #ASI_AIUS | $11_{16}$ | ASI_AS_IF_USER_SECONDARY |
| #ASI_AIUP_L | $18_{16}$ | ASI_AS_IF_USER_PRIMARY_LITTLE |
| #ASI_AIUS_L | $19_{16}$ | ASI_AS_IF_USER_SECONDARY_LITTLE |
| #ASI_PHYS_USE_EC_L | $1C_{16}$ | ASI_PHYS_USE_EC_LITTLE |
| #ASI_PHYS_BYPASS_EC_WITH_EBIT_L | $1D_{16}$ | ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE |
| #ASI_NUCLEUS_QUAD_LDD | $24_{16}$ | ASI_NUCLEUS_QUAD_LDD |
| #ASI_NUCLEUS_QUAD_LDD_L | $2C_{16}$ | ASI_NUCLEUS_QUAD_LDD_LITTLE |
| #ASI_QUAD_LDD_PHYS, #ASI_ATOMIC_QUAD_LDD_PHYS (deprecated) | $34_{16}$ | ASI_QUAD_LDD_PHYS |
| #ASI_QUAD_LDD_PHYS_L #ASI_ATOMIC_QUAD_LDD_PHYS_L (deprecated) | $3C_{16}$ | ASI_QUAD_LDD_PHYS_LITTLE |
| #ASI_CORE_ENABLE_STATUS | $41_{16}$ | ASI_CORE_ENABLE_STATUS (with VA = $10_{16}$) |
| #ASI_CORE_ENABLE | $41_{16}$ | ASI_CORE_ENABLE (with VA = $20_{16}$) |
| #ASI_XIR_STEERING | $41_{16}$ | ASI_XIR_STEERING (with VA = $30_{16}$) |
| #ASI_ERROR_STEERING | $41_{16}$ | ASI_ERROR_STEERING (with VA = $38_{16}$) |
| #ASI_CORE_RUNNING_RW | $41_{16}$ | ASI_CORE_RUNNING_RW (with VA = $50_{16}$) |
| #ASI_CORE_RUNNING_STATUS | $41_{16}$ | ASI_CORE_RUNNING_STATUS (with VA = $58_{16}$) |
| #ASI_CORE_RUNNING_W1S | $41_{16}$ | ASI_CORE_RUNNING_W1S (with VA = $60_{16}$) |
| #ASI_CORE_RUNNING_W1C | $41_{16}$ | ASI_CORE_RUNNING_W1C (with VA = $68_{16}$) |
| #ASI_MONDO_SEND_CTRL | $48_{16}$ | ASI_INTR_DISPATCH_STATUS |
| #ASI_MONDO_RECEIVE_CTRL | $49_{16}$ | ASI_INTR_RECEIVE |
| #ASI_AFSR | $4C_{16}$ | ASI_ASYNC_FAULT_STATUS |
| #ASI_AFAR | $4D_{16}$ | ASI_ASYNC_FAULT_ADDR |
| #ASI_SCRATCHPAD_0_REG | $4F_{16}$ | ASI_SCRATCHPAD_0_REG (with VA = $00_{16}$) |
| #ASI_SCRATCHPAD_1_REG | $4F_{16}$ | ASI_SCRATCHPAD_1_REG (with VA = $08_{16}$) |
| #ASI_SCRATCHPAD_2_REG | $4F_{16}$ | ASI_SCRATCHPAD_2_REG (with VA = $10_{16}$) |
| #ASI_SCRATCHPAD_3_REG | $4F_{16}$ | ASI_SCRATCHPAD_3_REG (with VA = $18_{16}$) |
| #ASI_SCRATCHPAD_4_REG | $4F_{16}$ | ASI_SCRATCHPAD_4_REG (with VA = $20_{16}$) |

**TABLE G-1** Value Names and Values *(3 of 4)*

| Value Name in Assembly Language | Value | Description |
|---|---|---|
| #ASI_SCRATCHPAD_5_REG | $4F_{16}$ | ASI_SCRATCHPAD_5_REG (with VA = $28_{16}$) |
| #ASI_SCRATCHPAD_6_REG | $4F_{16}$ | ASI_SCRATCHPAD_6_REG (withVA = $30_{16}$) |
| #ASI_SCRATCHPAD_7_REG | $4F_{16}$ | ASI_SCRATCHPAD_7_REG (withVA = $38_{16}$) |
| #ASI_INTR_ID | $63_{16}$ | ASI_INTR_ID (withVA = $00_{16}$) |
| #ASI_CORE_ID | $63_{16}$ | ASI_CORE_ID (withVA = $10_{16}$) |
| #ASI_BLK_AIUP | $70_{16}$ | ASI_BLOCK_AS_IF_USER_PRIMARY |
| #ASI_BLK_AIUS | $71_{16}$ | ASI_BLOCK_AS_IF_USER_SECONDARY |
| #ASI_INTR_DISPATCH_EXT_W | $77_{16}$ | ASI_INTR_DISPATCH_EXT_W (withVA<13:0> = $78_{16}$) |
| #ASI_BLK_AIUPL | $78_{16}$ | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE |
| #ASI_BLK_AIUSL | $79_{16}$ | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE |
| #ASI_P | $80_{16}$ | ASI_PRIMARY |
| #ASI_S | $81_{16}$ | ASI_SECONDARY |
| #ASI_PNF | $82_{16}$ | ASI_PRIMARY_NOFAULT |
| #ASI_SNF | $83_{16}$ | ASI_SECONDARY_NOFAULT |
| #ASI_P_L | $88_{16}$ | ASI_PRIMARY_LITTLE |
| #ASI_S_L | $89_{16}$ | ASI_SECONDARY_LITTLE |
| #ASI_PNF_L | $8A_{16}$ | ASI_PRIMARY_NOFAULT_LITTLE |
| #ASI_SNF_L | $8B_{16}$ | ASI_SECONDARY_NOFAULT_LITTLE |
| #ASI_PST8_P | $C0_{16}$ | ASI_PST8_PRIMARY |
| #ASI_PST8_S | $C1_{16}$ | ASI_PST8_SECONDARY |
| #ASI_PST16_P | $C2_{16}$ | ASI_PST16_PRIMARY |
| #ASI_PST16_S | $C3_{16}$ | ASI_PST16_SECONDARY |
| #ASI_PST32_P | $C4_{16}$ | ASI_PST32_PRIMARY |
| #ASI_PST32_S | $C5_{16}$ | ASI_PST32_SECONDARY |
| #ASI_PST8_PL | $C8_{16}$ | ASI_PST8_PRIMARY_LITTLE |
| #ASI_PST8_SL | $C9_{16}$ | ASI_PST8_SECONDARY_LITTLE |
| #ASI_PST16_PL | $CA_{16}$ | ASI_PST16_PRIMARY_LITTLE |
| #ASI_PST16_SL | $CB_{16}$ | ASI_PST16_SECONDARY_LITTLE |
| #ASI_PST32_PL | $CC_{16}$ | ASI_PST32_PRIMARY_LITTLE |
| #ASI_PST32_SL | $CD_{16}$ | ASI_PST32_SECONDARY_LITTLE |

**TABLE G-1** Value Names and Values *(4 of 4)*

| Value Name in Assembly Language | Value | Description |
|---|---|---|
| #ASI_FL8_P | $D0_{16}$ | ASI_FL8_PRIMARY |
| #ASI_FL8_S | $D1_{16}$ | ASI_FL8_SECONDARY |
| #ASI_FL16_P | $D2_{16}$ | ASI_FL16_PRIMARY |
| #ASI_FL16_S | $D3_{16}$ | ASI_FL16_SECONDARY |
| #ASI_FL8_PL | $D8_{16}$ | ASI_FL8_PRIMARY_LITTLE |
| #ASI_FL8_SL | $D9_{16}$ | ASI_FL8_SECONDARY_LITTLE |
| #ASI_FL16_PL | $DA_{16}$ | ASI_FL16_PRIMARY_LITTLE |
| #ASI_FL16_SL | $DB_{16}$ | ASI_FL16_SECONDARY_LITTLE |
| #ASI_BLK_COMMIT_P | $E0_{16}$ | ASI_BLOCK_COMMIT_PRIMARY |
| #ASI_BLK_COMMIT_S | $E1_{16}$ | ASI_BLOCK_COMMIT_SECONDARY |
| #ASI_BLK_P | $F0_{16}$ | ASI_BLOCK_PRIMARY |
| #ASI_BLK_S | $F1_{16}$ | ASI_BLOCK_SECONDARY |
| #ASI_BLK_PL | $F8_{16}$ | ASI_BLOCK_PRIMARY_LITTLE |
| #ASI_BLK_SL | $F9_{16}$ | ASI_BLOCK_SECONDARY_LITTLE |

The full names of the ASIs, listed in the Description column of TABLE G-1 can also be defined.

## G.1.3 Values

Some instructions use operand values as follows:

| | |
|---|---|
| *const4* | A constant that can be represented in 4 bits |
| *const22* | A constant that can be represented in 22 bits |
| *imm_asi* | An alternate address space identifier (0–255) |
| *simm7* | A signed immediate constant that can be represented in 7 bits |
| *simm10* | A signed immediate constant that can be represented in 10 bits |
| *simm11* | A signed immediate constant that can be represented in 11 bits |
| *simm13* | A signed immediate constant that can be represented in 13 bits |
| *value* | Any 64-bit value |
| *shcnt32* | A shift count from 0–31 |
| *shcnt64* | A shift count from 0–63 |

## G.1.4 Labels

A label is a sequence of characters that comprises alphabetic letters (a–z, A–Z [with upper and lower case distinct]), underscores (_), dollar signs ($), periods (.), and decimal digits (0-9). A label may contain decimal digits, but it may not begin with one. A local label contains digits only.

## G.1.5 Other Operand Syntax

Some instructions allow several operand syntaxes, as follows:

*reg_plus_imm* Can be any of the following:

| | |
|---|---|
| $reg_{rs1}$ | (equivalent to $reg_{rs1}$ + `%g0`) |
| $reg_{rs1}$ + *simm13* | |
| $reg_{rs1}$ – *simm13* | |
| *simm13* | (equivalent to `%g0` + *simm13*) |
| *simm13* + $reg_{rs1}$ | (equivalent to $reg_{rs1}$ + *simm13*) |

*address* Can be any of the following:

| | |
|---|---|
| $reg_{rs1}$ | (equivalent to $reg_{rs1}$ + `%g0`) |
| $reg_{rs1}$ + *simm13* | |
| $reg_{rs1}$ – *simm13* | |
| *simm13* | (equivalent to `%g0` + *simm13*) |
| *simm13* + $reg_{rs1}$ | (equivalent to $reg_{rs1}$ + *simm13*) |
| $reg_{rs1}$ + $reg_{rs2}$ | |

*membar_mask* Is the following:

*const7* — A constant that can be represented in 7 bits. Typically, this is an expression involving the logical OR of some combination of `#Lookaside`, `#MemIssue`, `#Sync`, `#StoreStore`, `#LoadStore`, `#StoreLoad`, and `#LoadLoad`.

*prefetch_fcn (prefetch function)* Can be any of the following:

```
#n_reads
#one_read
#n_writes
#one_write
#page
0–31
```

*regaddr (register-only address)* Can be any of the following:

| | |
|---|---|
| $reg_{rs1}$ | (equivalent to $reg_{rs1}$ + %g0) |
| $reg_{rs1}$ + $reg_{rs2}$ | |

*reg_or_imm (register or immediate value)* Can be either of:

- $reg_{rs2}$
- *simm13*

*reg_or_imm10 (register or immediate value)* Can be either of:

- $reg_{rs2}$
- *simm10*

*reg_or_imm11 (register or immediate value)* Can be either of:

- $reg_{rs2}$
- *simm11*

*reg_or_shcnt (register or shift count value)* Can be any of:

- $reg_{rs2}$
- *shcnt32*
- *shcnt64*

*software_trap_number* Can be any of the following:

| | |
|---|---|
| $reg_{rs1}$ | (equivalent to $reg_{rs1}$ + %g0) |
| $reg_{rs1}$ + *simm7* | |
| $reg_{rs1}$ – *simm7* | |
| *simm7* | (equivalent to %g0 + *simm7*) |
| *simm7* + $reg_{rs1}$ | (equivalent to $reg_{rs1}$ + *simm7*) |
| $reg_{rs1}$ + $reg_{rs2}$ | |

The resulting operand value (software trap number) must be in the range 0–127, inclusive.

## G.1.6 Comments

Two types of comments are accepted by the SPARC V9 assembler: C-style "/*...*/" comments, which may span multiple lines, and "!..." comments, which extend from the "!" to the end of the line.

# G.2 Syntax Design

The SPARC V9 assembly language syntax is designed so that the following statements are true:

- The destination operand (if any) is consistently specified as the last (rightmost) operand in an assembly language instruction.
- A reference to the *contents* of a memory location (in a Load, Store, `CASA`, `CASXA`, `LDSTUB`(`A`), or `SWAP`(`A`) instruction) is always indicated by square brackets ([]); a reference to the *address* of a memory location (such as in a `JMPL`, `CALL`, or `SETHI`) is specified directly, without square brackets.

# G.3 Synthetic Instructions

TABLE G-2 describes the mapping of a set of synthetic (or "pseudo") instructions to actual instructions. These synthetic instructions are provided by the SPARC V9 assembler for the convenience of assembly language programmers.

**Note**: Synthetic instructions should not be confused with "pseudo ops," which typically provide information to the assembler but do not generate instructions. Synthetic instructions always generate instructions; they provide more mnemonic syntax for standard SPARC V9 instructions.

**TABLE G-2** Mapping Synthetic to SPARC V9 Instructions *(1 of 3)*

| Synthetic Instruction | | SPARC V9 Instruction(s) | | Comment |
|---|---|---|---|---|
| `cmp` | *$reg_{rs1}$*`,` *reg_or_imm* | `subcc` | *$reg_{rs1}$*`,` *reg_or_imm*`, %g0` | Compare. |
| `jmp` | *address* | `jmpl` | *address*`, %g0` | |
| `call` | *address* | `jmpl` | *address*`, %o7` | |
| `iprefetch` | *label* | `bn,a,pt` | `%xcc,` *label* | Instruction prefetch. |
| `tst` | *$reg_{rs1}$* | `orcc` | `%g0,` *$reg_{rs1}$*`, %g0` | Test. |
| `ret` | | `jmpl` | `%i7+8, %g0` | Return from subroutine. |
| `retl` | | `jmpl` | `%o7+8, %g0` | Return from leaf subroutine. |
| `restore` | | `restore` | `%g0, %g0, %g0` | Trivial `restore`. |
| `save` | | `save` | `%g0, %g0, %g0` | Trivial `save`. (Warning: trivial `save` should only be used in kernel code!) |
| `setuw` | *value*`,` *$reg_{rd}$* | `sethi` | `%hi(`*value*`),` *$reg_{rd}$* | (When ((*value*&$3FF_{16}$) == 0).) |
| | | **— or —** | | |

**TABLE G-2** Mapping Synthetic to SPARC V9 Instructions *(2 of 3)*

| Synthetic Instruction | | SPARC V9 Instruction(s) | | Comment |
|---|---|---|---|---|
| | | `or` | `%g0, value, reg_rd` | (When $0 \le value \le 4095$). |
| | | | **— or —** | |
| | | `sethi` | `%hi(value), reg_rd;` | (Otherwise) |
| | | `or` | `reg_rd, %lo(value), reg_rd` | Warning: do not use `setuw` in the delay slot of a DCTI. |
| `set` | *value*, *reg*$_{rd}$ | | | synonym for `setuw`. |
| `setsw` | *value*, *reg*$_{rd}$ | `sethi` | `%hi(value), reg_rd` | (When (value > = 0) and ((value & $3FF_{16}$) == 0).) |
| | | | **— or —** | |
| | | `or` | `%g0, value, reg_rd` | (When $-4096 \le value \le 4095$). |
| | | | **— or —** | |
| | | `sethi` | `%hi(value), reg_rd` | (Otherwise, if (*value* < 0) and ((*value* & $3FF_{16}$) = = 0)) |
| | | `sra` | `reg_rd, %g0, reg_rd` | |
| | | | **— or —** | |
| | | `sethi` | `%hi(value), reg_rd;` | (Otherwise, if *value* > = 0) |
| | | `or` | `reg_rd, %lo(value), reg_rd` | |
| | | | **— or —** | |
| | | `sethi` | `%hi(value), reg_rd;` | (Otherwise, if *value* < 0) |
| | | `or` | `reg_rd, %lo(value), reg_rd` | |
| | | `sra` | `reg_rd, %g0, reg_rd` | Warning: do not use `setsw` in the delay slot of a CTI. |
| `setx` | *value*, *reg*, *reg*$_{rd}$ | `sethi` | `%uhi(value), reg` | Create 64-bit constant. |
| | | `or` | `reg, %ulo(value), reg` | ("*reg*" is used as a temporary register.) |
| | | `sllx` | `reg,32,reg` | |
| | | `sethi` | `%hi(value), reg_rd` | Note: `setx` optimizations are possible but not enumerated here. The worst case is shown. Warning: do not use `setx` in the delay slot of a CTI. |
| | | `or` | `reg_rd, reg, reg_rd` | |
| | | `or` | `reg_rd, %lo(value), reg_rd` | |
| `signx` | *reg*$_{rs1}$, *reg*$_{rd}$ | `sra` | `reg_rs1, %g0, reg_rd` | Sign-extend 32-bit value to 64 bits. |
| `signx` | *reg*$_{rd}$ | `sra` | `reg_rd, %g0, reg_rd` | |
| `not` | *reg*$_{rs1}$, *reg*$_{rd}$ | `xnor` | `reg_rs1, %g0, reg_rd` | One's complement. |
| `not` | *reg*$_{rd}$ | `xnor` | `reg_rd, %g0, reg_rd` | One's complement. |
| `neg` | *reg*$_{rs2}$, *reg*$_{rd}$ | `sub` | `%g0, reg_rs2, reg_rd` | Two's complement. |
| `neg` | *reg*$_{rd}$ | `sub` | `%g0, reg_rd, reg_rd` | Two's complement. |
| `cas` | [*reg*$_{rs1}$], *reg*$_{rs2}$, *reg*$_{rd}$ | `casa` | `[reg_rs1]#ASI_P, reg_rs2, reg_rd` | Compare and swap. |
| `casl` | [*reg*$_{rs1}$], *reg*$_{rs2}$, *reg*$_{rd}$ | `casa` | `[reg_rs1]#ASI_P_L, reg_rs2, reg_rd` | Compare and swap, little-endian. |

**TABLE G-2** Mapping Synthetic to SPARC V9 Instructions *(3 of 3)*

| Synthetic Instruction | | SPARC V9 Instruction(s) | | Comment |
|---|---|---|---|---|
| `casx` | [*reg$_{rs1}$*], *reg$_{rs2}$*, *reg$_{rd}$* | `casxa` | [*reg$_{rs1}$*]`#ASI_P`, *reg$_{rs2}$*, *reg$_{rd}$* | Compare and swap extended. |
| `casxl` | [*reg$_{rs1}$*], *reg$_{rs2}$*, *reg$_{rd}$* | `casxa` | [*reg$_{rs1}$*]`#ASI_P_L`, *reg$_{rs2}$*, *reg$_{rd}$* | Compare and swap extended, little-endian. |
| `inc` | *reg$_{rd}$* | `add` | *reg$_{rd}$*, `1`, *reg$_{rd}$* | Increment by 1. |
| `inc` | *const13*, *reg$_{rd}$* | `add` | *reg$_{rd}$*, *const13*, *reg$_{rd}$* | Increment by *const13*. |
| `inccc` | *reg$_{rd}$* | `addcc` | *reg$_{rd}$*, `1`, *reg$_{rd}$* | Increment by 1; set `icc` & `xcc`. |
| `inccc` | *const13*, *reg$_{rd}$* | `addcc` | *reg$_{rd}$*, *const13*, *reg$_{rd}$* | Incr by *const13*; set `icc` & `xcc`. |
| `dec` | *reg$_{rd}$* | `sub` | *reg$_{rd}$*, `1`, *reg$_{rd}$* | Decrement by 1. |
| `dec` | *const13*, *reg$_{rd}$* | `sub` | *reg$_{rd}$*, *const13*, *reg$_{rd}$* | Decrement by *const13*. |
| `deccc` | *reg$_{rd}$* | `subcc` | *reg$_{rd}$*, 1, *reg$_{rd}$* | Decr by 1; set `icc` & `xcc`. |
| `deccc` | *const13*, *reg$_{rd}$* | `subcc` | *reg$_{rd}$*, *const13*, *reg$_{rd}$* | Decr by *const13*; set `icc` & `xcc`. |
| `btst` | *reg_or_imm*, *reg$_{rs1}$* | `andcc` | *reg$_{rs1}$*, *reg_or_imm*, `%g0` | Bit test. |
| `bset` | *reg_or_imm*, *reg$_{rd}$* | `or` | *reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$* | Bit set. |
| `bclr` | *reg_or_imm*, *reg$_{rd}$* | `andn` | *reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$* | Bit clear. |
| `btog` | *reg_or_imm*, *reg$_{rd}$* | `xor` | *reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$* | Bit toggle. |
| `clr` | *reg$_{rd}$* | `or` | `%g0`, `%g0`, *reg$_{rd}$* | Clear (zero) register. |
| `clrb` | [*address*] | `stb` | `%g0`, [*address*] | Clear byte. |
| `clrh` | [*address*] | `sth` | `%g0`, [*address*] | Clear half-word. |
| `clr` | [*address*] | `stw` | `%g0`, [*address*] | Clear word. |
| `clrx` | [*address*] | `stx` | `%g0`, [*address*] | Clear extended word. |
| `clruw` | *reg$_{rs1}$*, *reg$_{rd}$* | `srl` | *reg$_{rs1}$*, `%g0`, *reg$_{rd}$* | Copy and clear upper word. |
| `clruw` | *reg$_{rd}$* | `srl` | *reg$_{rd}$*, `%g0`, *reg$_{rd}$* | Clear upper word. |
| `mov` | *reg_or_imm*, *reg$_{rd}$* | `or` | `%g0`, *reg_or_imm*, *reg$_{rd}$* | |
| `mov` | `%y`, *reg$_{rd}$* | `rd` | `%y`, *reg$_{rd}$* | |
| `mov` | `%asr`***n***, *reg$_{rd}$* | `rd` | `%asr`***n***, *reg$_{rd}$* | |
| `mov` | *reg_or_imm*, `%y` | `wr` | `%g0`, *reg_or_imm*, `%y` | |
| `mov` | *reg_or_imm*, `%asr`***n*** | `wr` | `%g0`, *reg_or_imm*, `%asr`***n*** | |

# Software Considerations

This appendix contains only material from *The SPARC Architecture Manual*, Version 9, and describes how software can use the SPARC V9 architecture effectively. Examples do not necessarily conform to any specific Application Binary Interface (ABI).

This appendix is informative only. It is not part of the SPARC V9 specification.

## H.1 Nonprivileged Software

This subsection describes software conventions that have proven or may prove useful, assumptions that compilers may make about the resources available, and how compilers can use those resources. It does not discuss how supervisor software (an operating system) may use the architecture. Although a set of software conventions is described, software is free to use other conventions more appropriate for specific applications.

The following are the primary goals for many of the software conventions described in this subsection:

- Minimizing average procedure-call overhead
- Minimizing latency due to branches
- Minimizing latency due to memory access

### H.1.1 Registers

Register usage is a critical resource allocation issue for compilers. The SPARC V9 architecture provides windowed integer registers (*in*, *out*, *local*), global integer registers, and floating-point registers.

### H.1.1.1 *In* and *Out* Registers

The *in* and *out* registers are used primarily for passing parameters to and receiving results from subroutines, and for keeping track of the memory stack. When a procedure is called and executes a SAVE instruction, the caller's *out*s become the callee's *in*s.

One of a procedure's *out* registers (%o6) is used as its stack pointer, %sp. It points to an area in which the system can store %r16–%r31 (%l0–%l7 and %i0–%i7) when the register file overflows (spill trap), and is used to address most values located on the stack. A trap can occur at any time,[1] which may precipitate a subsequent spill trap. During this spill trap, the contents of the user's register window at the time of the original trap are spilled to the memory to which its %sp points.

A procedure may store temporary values in its *out* registers (except %sp) with the understanding that those values are volatile across procedure calls. %sp cannot be used for temporary values for the reasons described in *Register Windows and %sp* on page 519.

Up to six parameters[2] may be passed by placing them in *out* registers %o0–%o5; additional parameters are passed in the memory stack. The stack pointer is implicitly passed in %o6, and a CALL instruction places its own address in %o7.[3] Floating-point parameters may also be passed in floating-point registers.

After a callee is entered and its SAVE instruction has been executed, the caller's *out* registers are accessible as the callee's *in* registers.

The caller's stack pointer %sp (%o6) automatically becomes the current procedure's frame pointer %fp (%i6) when the SAVE instruction is executed.

The callee finds its first six integer parameters in %i0–%i5, and the remainder (if any) on the stack.

A function returns a scalar integer value by writing it into its *in*s (which are the caller's *out*s), starting with %i0. A scalar floating-point value is returned in the floating-point registers, starting with %f0.

A procedure's return address, normally the address of the instruction just after the CALL's delay-slot instruction, is as %i7+8.[4]

1. For example, due to an error in executing an instruction (for example, a *mem_address_not_aligned* trap), or due to any type of external interrupt.
2. Six is more than adequate, since the overwhelming majority of procedures in system code take fewer than six parameters. According to studies cited by Weicker (Weicker, R. P., "Dhrystone: A Synthetic Systems Programming Benchmark," *CACM* 27:10, October 1984), at least 97% (measured statically) take fewer than six parameters. The average number of parameters did not exceed 2.1, measured either statically or dynamically, in any of these studies.
3. If a JMPL instruction is used in place of a CALL, it should place its address in %o7 for consistency.
4. For convenience, SPARC V9 assemblers may provide a "ret" (return) synthetic instruction that generates a "jmpl %i7+8, %g0" hardware instruction. See *Synthetic Instructions* on page 514.

### H.1.1.2 Local Registers

The *local*s are used for automatic[1] variables and for most temporary values. For access efficiency, a compiler may also copy parameters (that is, those past the sixth) from the memory stack into the *local*s and use them from there.

See *Register Allocation Within a Window* on page 523 for methods of allocating more or fewer than eight registers for local values.

### H.1.1.3 Register Windows and `%sp`

Some caveats about the use of `%sp` and the `SAVE` and `RESTORE` instructions are appropriate. If the operating system and user code use register windows, it is essential that

- `%sp` *always* contains a correct value, so that when (and if) a register window spill/fill trap occurs, the register window can be correctly stored to or reloaded from memory.[2]
- Nonprivileged code uses `SAVE` and `RESTORE` instructions carefully. In particular, "walking" the call chain through the register windows using `RESTORE`s, expecting to be able to return to where one started using `SAVE`s, does not work as one might suppose. Since user code cannot disable traps, a trap (e.g., an interrupt) could write over the contents of a user register window that has "temporarily" been `RESTORE`d.[3] The safe method is to flush the register windows to user memory (the stack) with the `FLUSHW` instruction. Then, user code can safely walk the call chain through user memory, instead of through the register windows.

To avoid such problems, consider all data memory at addresses just less than `%sp` to be volatile, and the contents of all register windows "below" the current one to be volatile.

### H.1.1.4 *Global* Registers

Unlike the *in*s, *local*s, and *out*s, the *global*s are not part of any register window. The *global*s are a set of eight registers with global scope, like the register sets of more traditional processor architectures. An ABI may define conventions that the *global*s (except `%g0`) must obey. For example, if the convention assumes that *globals* are volatile across procedure calls, either the caller or the callee must take responsibility for saving and restoring their contents.

1. In the C language, an automatic variable is a local variable whose lifetime is no longer than that of its containing procedure.
2. Typically, the `SAVE` instruction is used to generate a new `%sp` value while shifting to a new register window, all in one atomic operation. When `SAVE` is used this way, synchronization of the two operations should not be a problem.
3. Another reason this might fail is that user code has no way to determine how many register windows are implemented by the hardware.

Global register %g0 has a hardwired value of zero; it always reads as zero, and writes to it have no program-visible effect.

Typically, the *global* registers other than %g0 are used for temporaries, global variables, or global pointers — either user variables, or values maintained as part of the program's execution environment. For example, one could use *global*s in the execution environment by establishing a convention that global scalars are addressed via offsets from a global base register. In the most general case, memory accessed at an arbitrary address requires six instructions; for example,

```
sethi     %hh(address),tmp
or        tmp, %hm(address), tmp
sllx      tmp, 32, tmp
sethi     %lm(address), reg
or        reg, %lo(address), reg
ld        [reg+tmp], reg
```

Use of a global base register for frequently accessed global values would provide faster (single-instruction) access to $2^{13}$ bytes of those values; for example,

```
ld        [%gn+offset], reg
```

Additional global registers could be used to provide single-instruction access to correspondingly more global values.

### H.1.1.5 Floating-Point Registers

There are 16 quad-precision floating-point registers. The registers can also be accessed as 32 double-precision registers. In addition, the first 8 quad registers can also be accessed as 32 single-precision registers. Floating-point registers are accessed with different instructions than the integer registers; their contents can be moved among themselves, and to or from memory. See *Floating-Point f Registers* on page 46 for more information about floating-point register aliasing.

Like the global registers, the floating-point registers must be managed by software. Compilers use the floating-point registers for user variables and compiler temporaries, pass floating-point parameters, and return floating-point results in them.

### H.1.1.6 The Memory Stack

A stack is maintained to hold automatic variables, temporary variables, and return information for each invocation of a procedure. When a procedure is called, a *stack frame* is allocated; it is released when the procedure returns. The use of a stack for this purpose allows simple and efficient implementation of recursive procedures.

Under certain conditions, optimization can allow a leaf procedure to use its caller's stack frame instead of one of its own. In that case, the procedure allocates no space of its own for a stack frame. See *Leaf-Procedure Optimization*, below, for more information.

The stack pointer `%sp` must always maintain the alignment required by the operating system's ABI. This is at least doubleword alignment, possibly with a constant offset to increase stack addressability using constant offset addressing.

## H.1.2 Leaf-Procedure Optimization

A *leaf procedure* is one that is a "leaf" in the program's call graph; that is, one that does not call (e.g., via `CALL` or `JMPL`) any other procedures.

Each procedure, including leaf procedures, normally uses a `SAVE` instruction to allocate a stack frame and obtain a register window for itself, and a corresponding `RESTORE` instruction to deallocate it. The time costs associated with this are

- Possible generation of register-window spill/fill traps at runtime. This only happens occasionally,[1] but when either a spill or fill trap does occur, it costs several machine cycles to process.
- The cycles expended by the `SAVE` and `RESTORE` instructions themselves.

There are also space costs associated with this convention, the cumulative cache effects of which may be nonnegligible. The space costs include

- The space occupied on the stack by the procedure's stack frame
- The two words occupied by the `SAVE` and `RESTORE` instructions

Of the above costs, the trap-processing cycles typically are the most significant.

Some leaf procedures can be made to operate *without* their own register window or stack frame, using their caller's instead. This can be done when the candidate leaf procedure meets all of the following conditions:[2]

- It contains no references to `%sp`, except in its `SAVE` instruction.
- It contains no references to `%fp`.
- It refers to (or can be made to refer to) no more than 8 of the 32 integer registers, including `%o7` (the return address).

1. The frequency of overflow and underflow traps depends on the application and on the number of register windows (`NWINDOWS`) implemented in hardware.
2. Although slightly less restrictive conditions could be used, the optimization would become more complex to perform and the incremental gain would usually be small.

If a procedure conforms to the above conditions, it can be made to operate using its caller's stack frame and registers, an optimization that saves both time and space. This optimization is called *leaf procedure optimization*. The optimized procedure may safely use only registers that its caller already assumes to be volatile across a procedure call.

The optimization can be performed at the assembly language level with the following steps:

1. Change all references to registers in the procedure to registers that the caller assumes volatile across the call.

   a. Leave references to `%o7` unchanged.

   b. Leave any references to `%g0`–`%g7` unchanged.

   c. Change `%i0`–`%i5` to `%o0`–`%o5`, respectively. If an *in* register is changed to an *out* register that was already referenced in the original unoptimized version of the procedure, all original references to that *out* register must be changed to refer to an unused *out* or *global* register.

   d. Change references to each *local* register into references to any unused register that is assumed to be volatile across a procedure call.

2. Delete the `SAVE` instruction. If it was in a delay slot, replace it with a `NOP` instruction. If its destination register was not `%g0` or `%sp`, convert the `SAVE` into the corresponding `ADD` instruction instead of deleting it.

3. If the `RESTORE`'s implicit addition operation is used for a productive purpose (such as setting the procedure's return value), convert the `RESTORE` to the corresponding `ADD` instruction. Otherwise, the `RESTORE` is only used for stack and register-window deallocation; replace it with a `NOP` instruction (it is probably in the delay slot of the `RET` and so cannot be deleted).

4. Change the `RET` (return) synthetic instruction to `RETL` (return-from-leaf-procedure synthetic instruction).

5. Perform any optimizations newly made possible, such as combining instructions or filling the delay slot of the `RETL` (or the delay slot occupied by the `SAVE`) with a productive instruction.

After the above changes, there should be no `SAVE` or `RESTORE` instructions, and no references to *in* or *local* registers in the procedure body. All original references to *in*s are now to *out*s. All other register references are to registers that are assumed to be volatile across a procedure call.

Costs of optimizing leaf procedures in this way include

- Additional intelligence in a peephole optimizer to recognize and optimize candidate leaf procedures

- Additional intelligence in debuggers to properly report the call chain and the stack traceback for optimized leaf procedures[1]

## H.1.3 Example Code for a Procedure Call

This subsection illustrates common parameter-passing conventions and gives a simple example of leaf-procedure optimization.

The code fragment in CODE EXAMPLE H-1 shows a simple procedure call with a value returned, and the procedure itself.

Since `sum3` does not call any other procedures (i.e., it is a leaf procedure), it can be optimized to become:

```
sum3:
     add       %o0, %o1, %o0
     retl                      ! (must use RETL, not RET,
     add       %o0, %o2, %o0   !  to return from leaf procedure)
```

## H.1.4 Register Allocation Within a Window

The usual SPARC V9 software convention is to allocate eight registers (`%l0`–`%l7`) for local values. A compiler could allocate more registers for local values at the expense of having fewer *out*s and *in*s available for argument passing. For example, if instead of assuming that the boundary between local values and input arguments is between `r`[23] and `r`[24] (`%l7` and `%i0`), software could, by convention, assume that the boundary is between `r`[25] and `r`[26] (`%i1` and `%i2`). This would provide 10 registers for local values and 6 *in* and *out* registers. This is shown in TABLE H-1.

**TABLE H-1** Register Allocation Within a Window

| | Standard register model | 10 local register model | Arbitrary register model |
|---|---|---|---|
| Registers for local values | 8 | 10 | *n* |
| *In* / *out* registers | | | |
| Reserved for `%sp` / `%fp` | 1 | 1 | 1 |
| Reserved for return address | 1 | 1 | 1 |
| Available for argument passing | 6 | 4 | 14 – *n* |
| Total *ins* / *outs* | 8 | 6 | 16 – *n* |

1. A debugger can recognize an optimized leaf procedure by scanning it, noting the absence of a SAVE instruction. Compilers often constrain the SAVE, if present, to appear within the first few instructions of a procedure; in such a case, only those instruction positions need be examined.

**CODE EXAMPLE H-1** Simple Procedure Call with Value Returned

```
! CALLER:
!    int i;                          /* compiler assigns "i" to register %l7 */
!    i  = sum3( 1, 2, 3 );
     ...
     mov       1, %o0                ! first arg to sum3 is 1
     mov       2, %o1                ! second arg to sum3 is 2
     call      sum3                  ! the call to sum3
     mov       3, %o2                ! last parameter to sum3 in delay slot
     mov       %o0, %l7              ! copy return value to %l7 (variable "i")
     ...

#define SA(x)  (((x)+15)&(~0x1F))  /* rounds "x" up to extended word boundary */
#define MINFRAME ((16+1+6)*8)      /* minimum size stack frame, in bytes;
                                    * 16 extended words for saving the
                                    * current register window,
                                    * 1 extended word for "hidden parameter",
                                    * and 6 extended words in which a callee
                                    * can store its arguments.
                                    */

! CALLEE:
!    int sum3( a, b, c )
!         int a, b, c;              /* args received in %i0, %i1, and %i2 */
!    {
!         return  a+b+c;
!    }
sum3:
     save      %sp,-SA(MINFRAME),%sp! set up new %sp; alloc min. stack frame
     add       %i0, %i1, %l7        ! compute sum in local %l7
     add       %l7, %i2, %l7        ! (or %i0 could have been used directly)
     ret                            ! return from sum3, and...
     restore   %l7, 0, %o0          ! move result into output reg & restore
```

## H.1.5 Other Register-Window-Usage Models

So far, this appendix has described SPARC V9 software conventions that are appropriate for use in a general-purpose multitasking computer system. However, SPARC V9 is used in many other applications, notably embedded and/or real-time systems. In such applications, other schemes for allocation of SPARC V9's register windows might be more nearly optimal than the one described above.

One possibility is to avoid using the normal register-window mechanism by not using SAVE and RESTORE instructions. Software would see 32 general-purpose registers instead of SPARC V9's usual windowed register file. In this mode, a SPARC V9 processor would operate like processors with more traditional (flat) register architectures. Procedure call times would be more determinate (due to lack of spill/fill traps), but for most types of software, average procedure call time would significantly increase, due to increased memory traffic for parameter passing and saving/restoring local variables.

Effective use of this software convention would require compilers to generate different code (direct register spills/fills to memory and no SAVE/RESTORE instructions) than for the software conventions described above.

It would be awkward, at best, to attempt to mix (link) code that uses the SAVE/RESTORE convention with code that does not use it. If both conventions *were* used in the same system, two versions of each library would be required.

It would be possible to run user code with one register-usage convention and supervisor code with another. With sufficient intelligence in supervisor software, user processes with different register conventions could be run simultaneously.[1]

## H.1.6 Self-Modifying Code

If a program includes self-modifying code, it must issue a FLUSH instruction for each modified doubleword of instructions (or a call to supervisor software having an equivalent effect).

Note that self-modifying code intended to be portable *must* use FLUSH instruction(s) (or a call to supervisor software having equivalent effect) after storing into the instruction stream.

All SPARC V9 instruction accesses are big-endian. If a program is running in little-endian mode and wishes to modify instructions, it must do one of the following:

- Use an explicit big-endian ASI to write the modified instruction to memory, or
- Reverse the byte ordering shown in the instruction formats in Appendix , *Instruction Definitions*, before doing a little-endian store, since the stored data will be reordered before the bytes are written to memory.

## H.1.7 Thread Management

SPARC V9 provides support for the efficient management of user-level threads. The cost of thread switching can be reduced by using the following features:

1. Although technically possible, this is not to suggest that there would be significant utility in mixing user processes with differing register-usage conventions.

- **User management of FPU** — The `fef` bit in the `FPRS` register allows nonprivileged code to manage the FPU. This is in addition to the management done by the supervisor code via the `pef` bit in the `PSTATE` register. A thread-management library can implement efficient switching of the FPU among threads by manipulating the `fef` bit in the `FPRS` register and by providing a user trap handler (with support from the supervisor software) for the *fp_disabled* exception. See the description of User Traps in *User Trap Handlers* on page 537.
- **`FLUSHW` instruction** — The `FLUSHW` instruction is an efficient way for a thread library to flush the register windows during a thread switch. The instruction executes as a NOP if there are no windows to flush.

## H.1.8 Minimizing Branch Latency

The SPARC V9 architecture contains several instructions that can be used to minimize branch latency. These are described below.

### H.1.8.1 Conditional Moves

The conditional move instructions for both integer and floating-point registers can be used to eliminate branches from the code generated for simple expressions or assignments. The following example illustrates this.

The C code segment

```
double    x,y;
int       i;
...
i  = (x > y) ? 1 : 2;
```

can be compiled to use a conditional move as follows:

```
fcmp      %fcc1, x, y    ! x and y are double regs
mov       1, i           ! i is int; assume x > y
movfle    %fcc1, 2, i    ! fix i if wrong
```

### H.1.8.2 Branch or Move Based on Register Contents

The use of register contents as conditions for branch and move instructions allows any integer register (other than `r0`) to hold a boolean value or the results of a comparison. This allows conditions to be used more efficiently in nested cases. It allows the generation of a condition to be moved further from its use, thereby minimizing latency. In addition, it can eliminate the need for additional arithmetic instructions to set the condition codes. This is illustrated in the following example.

The test for finding the maximum of an array of integers,

```
if (A[i] > max)
max  = A[i];
```

can be compiled as follows, allowing the condition for the loop to be set before the sequence and checked after it:

```
ldx       [addr_of_Ai], Ai
sub       Ai, max, tmp
movrgz    tmp, Ai, max
```

## H.1.9 Prefetch

The SPARC V9 architecture includes a prefetch instruction intended to help hide the latency of accessing memory.[1]

As a general rule, given a loop of the following form (using C for assembly language and assuming a cache line size of 64 bytes and that A and B are arrays of 8-byte values)

```
for (i  = 0; i < N; i++) {
     load A[i]
     load B[i]
     ...
}
```

which takes C cycles per iteration (assuming all loads hit in cache) and given L cycles of latency to memory, prefetch instructions may be inserted for data that will be needed `ceiling`(L / C') iterations in the future, where C' is number of cycles per iteration of the modified loop. Thus, the loop would be transformed into

```
K  = ceiling(L/C');
for (i  = 0; i < N; i++) {
     load A[i]
     load B[i]
     prefetch A[i+K]
     prefetch B[i+K]
     ...
}
```

This ensures that the loads will find their data in the cache and will thus complete more quickly. The first K iterations will not get any benefit from prefetching, so if the number of iterations is small (see below), then prefetching will not help.

1. Two papers describing the use of prefetch instructions are Callahan, D., K. Kennedy, A. Porterfield, "Software Prefetching," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 40-52, and Mowry, T., M. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 62-73.

Note that in cases of contiguous access (like this one), many of the prefetch instructions will in fact be unnecessary and may slow the program down. To avoid this, note that the prefetch instruction always obtains at least 64 (cache-line-aligned) bytes.

```
/* Round up access to next cache line. */
K'  = (ceiling(L/C') + 7) & ~7;
for (i  = 0; i < N; i++) {
     load A[i]
     load B[i]
     if ( ((int)(A+i) & 63)  =  = 0) {
          prefetch A[i+K']
          prefetch B[i+K']
     }
     ...
}
```

or (unrolled eight times, assuming A and B are arrays of 8-byte values)

```
/* Be sure that we access the next cache line. */
K''  = ceiling(L/C') + 7;
for (i  = 0; i < N; i++) {
     load A[i]
     load B[i]
     prefetch A[i+K'']
     prefetch B[i+K'']
     ...
     load A[i+1]
     load B[i+2]
     ... (no prefetching)
     ...
     load A[i+7]
     load B[i+7]
     ...
}
```

In the first case, the prefetching is performed exactly when needed, and thus the distance need not be adjusted. However, the prefetching may not start on the first iteration, resulting in as many as K' + 7 iterations without prefetching.

In the second case, the prefetching occurs somewhere within a cache line, and thus, it is not known exactly how long it will be until the next cache line is needed. However, by prefetching seven further ahead, we ensure that the next cache line will be prefetched soon enough. In the worst case, as many as K'' ($\leq$ K' + 7) iterations will execute without any benefit from prefetching.

TABLE H-2 illustrates the cost trade-offs between no prefetching, naive prefetching, and smart prefetching (the second choice) for a small loop (two cycles) with varying uncovered latencies to memory. Some of the latency may be overlapped with execution of surrounding instructions; that which is not is uncovered.

**TABLE H-2** Prefetch Cost Tradeoffs

| | | | | | Limit cycles/iteration | | | Smart startup costs | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | No pf | Naive | Smart | Worst | Worst |
| C | C' | L | K | K" | C+L/8 | C' | (7C+C')/8 | Misses | Breakeven |
| 2 | 4 | 8 | 4 | 11 | 3 | 4 | 2.25 | 2 | N = 21 |
| 2 | 4 | 16 | 8 | 15 | 4 | 4 | 2.25 | 2 | N = 18 |
| 2 | 4 | 32 | 16 | 23 | 6 | 4 | 2.25 | 3 | N = 26 |

Here, we treat the arrays accessed as if one were not in the cache. Thus, every eight iterations, a cache line must be fetched from memory in the no-prefetch case; and thus, the amortized cost of an iteration is C + L / 8. The cost estimate for the smart case ignores any benefits from unrolling, since it is reasonable to expect that the loop would be unrolled or pipelined in this fashion, even if prefetching were not used. The startup costs assume an alignment within the cache that maximizes the initial misses. The break-even cost was chosen by solving the following equation for N.

$$N * (C + L / 8) = WM * L + N * (7C + C') / 8 \ \{\text{e.g., } 3N = 16 + 2.25N \Rightarrow N = 21\}$$

Of course, this is a simplified model.

Another possibility to consider is the worst-case cost of prefetching. If, in the example provided, everything accessed is always cached, then the smart-prefetching loop takes 12.5% longer. For each memory latency, there is a break-even point (in terms of how often one of the array operands is cached) at which the prefetching loop begins to run faster. TABLE H-3 illustrates this.

**TABLE H-3** Cache Break-Even Points

| L | C-cached | C-missed | C-smart | Break-even % cached operands | Break-even loop cache miss rate |
|---|---|---|---|---|---|
| 8 | 2 | 3 | 2.25 | 75% | 1.56% |
| 16 | 2 | 4 | 2.25 | 88% | 0.75% |
| 32 | 2 | 6 | 2.25 | 94% | 0.375% |
| 64 | 2 | 10 | 2.25 | 97% | 0.188% |

Note that one uncached operand corresponds to one load out of sixteen missing the cache; the operand miss rate is sixteen times higher than the load miss rate. Note that this is the miss rate for this loop alone; extrapolation from whole-program miss rates is not advised.

Binaries that run efficiently across different SPARC V9 implementations can be created for cases like this (where memory accesses are regular, though not necessarily contiguous) by parameterizing the prefetch distance by machine type. In privileged code the machine type is available in the VER register; nonprivileged code should be able to obtain this information from the operating system or ABI. Based on information about known machines and estimated loop execution times, a

compiler could precalculate values for K'' (assuming smart prefetching) and store them in a table. At execution time, the proper value for K'' would be fetched from the table before entering the loop.

For regular but noncontiguous accesses, a prefetch would be issued for every load. If cache blocking is used, the prefetching strategy must be adjusted accordingly, since there is no point in prefetching data that is expected to be in the cache already.

The prefetch variant should be chosen based on what is known about the local and global use of the data prefetched. If the data is not being written locally, then variant 0 (several reads) should be used. If it is being written (and possibly also read), then variant 2 (several writes) should be used. If, in addition, it is known that this is likely to be the last use of the data for some time (for example, if the loop iteration count is one million and dependence analysis reveals no reuse of data), then it is appropriate to use either variant 1 (one read) or 3 (one write). If reuse of data is expected to occur soon, then use of variants 1 or 3 is not appropriate, because of the risk of increased bus and memory traffic on a multiprocessor.

If the hardware does not implement all variants, it is expected to provide a sensible overloading of the unimplemented variants. Thus, correct use of a specific variant need not be tied to a particular SPARC V9 implementation or multi/uniprocessor configuration.

## H.1.10 Nonfaulting Load

The SPARC V9 architecture includes a way to specify load instructions that do not generate visible faults, so that compilers can have more freedom in scheduling instructions. Note that these are not speculative loads, which may fault if their results are later used; these are normal load instructions, but tagged to indicate to the kernel and/or hardware that a fault should not be delivered to the code executing the instruction.

Five important rules govern the use of nonfaulting loads:

1. Volatile memory references in the source language should not use nonfaulting load instructions.
2. Code compiled for debugging should not use nonfaulting loads because they remove the ability to detect common errors.
3. If nonfaulting loads are used, page zero should be a page of zero values, mapped read-only. Compilers that routinely use negative offsets to register pointers should map page “–1” similarly if the operating software permits it.
4. Any use of nonfaulting loads in privileged code must be aware of how they are treated by the host SPARC V9 implementation.

5. Nonfaulting loads from unaligned addresses may be substantially more expensive than nonfaulting loads from other addresses.

Nonfaulting loads can be used to solve three scheduling problems.

- On superscalar machines, it is often desirable to obtain the right mix of instructions to avoid conflicts for any given execution unit. A nonfaulting load can be moved (backwards) past a basic block boundary to even out the instruction mix.
- On pipelined machines, there may be latency between loads and uses. A nonfaulting load can be moved past a block boundary to place more instructions between a load into a register and the next use of that register.
- Software pipelining improves the scheduling of loops, but if a loop iteration begins with a load instruction and contains an early exit, it may not be eligible for pipelining. If the load is replaced with a nonfaulting load, then the loop can be pipelined.

In the branch-laden code shown in CODE EXAMPLE H-2, nonfaulting loads could be used to separate loads from uses.

The result also has a somewhat better mix of instructions and is somewhat pipelined. The basic blocks are separated.

**CODE EXAMPLE H-2** Branch-Laden Code with Nonfaulting Loads

**Source Code**:

```
while ( x ! = 0 && x -> key ! = goal) x  = x -> next;
```

**With Normal Loads**:

```
entry:
     brnz,a    x,loop    !
     ldx       [x],t1    ! (pre)load1 (key)
loop:
     cmp       t1,goal   ! use1
     bpe       %xcc,out
     nop                 ! no filling from loop.
     ldx       [x+8],x   ! load2 (next)
     brnz,a    x,loop    ! use2
     ldx       [x],t1    ! load1
out: ...
```

**With Nonfaulting Loads**:

```
entry:
     mov       x,t2
     mov       #ASI_PNF, %asi
     ldxa      [t2]%asi,t1    ! (pre)load1 (nf-load for key)
loop:
     mov       t2,x           ! begin loop body
     brz,pn    t2,out
     ldxa      [t2+8]%asi,t2  ! load2 (nf-load for next)

     cmp       t1,goal        ! use1
     bpne      %xcc,loop
     ldxa      [t2]%asi,t1    ! use2, load1 ! nf-load for x
out: ...
```

In the loop shown in CODE EXAMPLE H-3, nonfaulting loads allow pipelining.

**CODE EXAMPLE H-3** Loop with Nonfaulting Loads

**Source Code**:

```
d_ne_index (double * d1, double * d2) {
     int i  = 0;
     while(d1[i]  =  = d2[i]) i++;
     return i;
}
```

**With Normal Loads**:

```
     mov       0,t
     mov       0,i
loop:
     lddf      [d1+t],a1
     lddf      [d2+t],a2      ! load
     add       t,8,t
     fcmpd     a1,a2          ! use
     fbe,a     loop           ! fcc use
     add       i,1,i
```

**With Nonfaulting Loads**:

```
     lddf      [d1],a1
     lddf      [d2],a2
     mov       8,t
     mov       0,i
loop:
     fcmpd     a1,a2          ! use, fcc def
     lddfa     [d1+t],%asi,a1
     lddfa     [d2+t],%asi,a2 ! load
     add       t,8,t
     fbe,a     loop           ! fcc use
     add       i,1,i
```

This loop might be improved further by using unrolling, prefetching, and multiple FCCs, but that is beyond the scope of this discussion.

## H.2 Supervisor Software

This section discusses how supervisor software can use the SPARC V9 privileged architecture. It illustrates how the architecture can be used in an efficient manner. An implementation may choose to utilize different strategies based on its requirements and implementation-specific aspects of the architecture.

## H.2.1 Trap Handling

The SPARC V9 privileged architecture provides support for efficient trap handling, especially for window traps. The following features of the SPARC V9 privileged architecture can be used to write efficient trap handlers:

- **Multiple trap levels.** The trap handlers for trap levels less than MAXTL – 1 can be written to ignore exceptional conditions and execute the common case efficiently (without checks and branches). For example, the fill/spill handlers can access pageable memory without first checking if it is resident. If the memory is not resident, the access will cause a trap that will be handled at the next trap level.
- **Vectoring of fill/spill traps.** Supervisor software can set up the vectoring of fill/spill traps prior to executing code that uses register windows and may cause spill/fill traps. This feature can be used to support SPARC V8 and SPARC V7 binaries. These binaries create stack frames with save areas for 32-bit registers. SPARC V9 binaries create stack frames with save areas for 64-bit registers. By setting up the spill/fill trap vector based on the type of binary being executed, the trap handlers can avoid checking and branching to use the appropriate load/store instructions.
- **Saved trap state.** Trap handlers need not save (restore) virtual processor state that is automatically saved (restored) on a trap (return from trap). For example, the fill/spill trap handlers can load ASI_AS_IF_USER_PRIMARY{_LITTLE} into the ASI register in order to access the user's address space without the overhead of having to save and restore the ASI register.
- **SAVED and RESTORED instructions.** The SAVED (RESTORED) instruction provides an efficient way to update the state of the register windows after successfully spilling (filling) a register window. They implement a default policy of spilling (filling) one register window at a time. If desired, the supervisor software can implement a different policy by directly updating the state registers.
- **Alternate globals.** The alternate global registers can be used to avoid saving and restoring the normal global registers. They can be used like the local registers of the trap window in SPARC V8.
- **Large trap vectors for spill/fill.** The definition of the spill and fill trap vectors with reserved space between each pair of vectors allows spill and fill trap handlers to be up to 32 instructions long, thus avoiding a branch in the handler.

## H.2.2 Example Code for Spill Handler

The code in CODE EXAMPLE H-4 shows a spill handler for a SPARC V9 user binary. The handler is located at the vector for trap type *spill_0_normal* ($080_{16}$). It is assumed that supervisor software has set the WSTATE register to 0 before executing the user binary. The handler is invoked when user code executes a SAVE instruction that results in a window overflow.

**CODE EXAMPLE H-4** Spill Handler

```
T_NORMAL_SPILL_0:
     !Set ASI to access user addr space
     wr        #ASI_AIUP, %asi
     stxa      %l0, [%sp+(8* 0)]%asi    !Store window in memory sta
     stxa      %l1, [%sp+(8* 1)]%asi
     stxa      %l2, [%sp+(8* 2)]%asi
     stxa      %l3, [%sp+(8* 3)]%asi
     stxa      %l4, [%sp+(8* 4)]%asi
     stxa      %l5, [%sp+(8* 5)]%asi
     stxa      %l6, [%sp+(8* 6)]%asi
     stxa      %l7, [%sp+(8* 7)]%asi
     stxa      %i0, [%sp+(8* 8)]%asi
     stxa      %i1, [%sp+(8* 9)]%asi
     stxa      %i2, [%sp+(8*10)]%asi
     stxa      %i3, [%sp+(8*11)]%asi
     stxa      %i4, [%sp+(8*12)]%asi
     stxa      %i5, [%sp+(8*13)]%asi
     stxa      %i6, [%sp+(8*14)]%asi
     stxa      %i7, [%sp+(8*15)]%asi
     saved                              ! Update state
     retry                              ! Retry trapped instruction
                                        ! Restores old %asi
```

## H.2.3 Client-Server Model

SPARC V9 provides mechanisms to support client-server computing efficiently. A call from a client to a server (where the client and server have separate address spaces) can be implemented efficiently through a software trap that switches the address space. This is often referred to as a *cross-domain call*. A system call in most operating systems can be viewed as a special case of a cross-domain call. The following features are useful in implementing a cross-domain call.

### H.2.3.1 Splitting the Register Windows

The register windows can be shared efficiently between multiple address spaces by using the OTHERWIN register and providing additional trap handlers to handle spill/fill traps for the other (not the current) address spaces. On a cross-domain call (a software trap), the supervisor can set the OTHERWIN register to the number of register windows used by the client (equal to CANRESTORE) and CANRESTORE to zero. At the same time the WSTATE bit vectors can be set to vector the spill/fill traps appropriately for each address space.

The sequence in CODE EXAMPLE H-5 shows a cross-domain call and return. The example assumes the simple case, where only a single client-server pair can occupy the register windows. More general schemes can be developed along the same lines.

**CODE EXAMPLE H-5** Cross-Domain Call and Return

```
cross_domain_call:
     save      ! create a new register window for the server
     ..        ! Switch to the execution environment for the server;
     ..        ! Save trap state as necessary.

     ! Set CWP for caller in TSTATE
     rdpr      %tstate, %g1
     rdpr      %cwp, %g2
     bclr      TSTATE_CWP, %g1
     wrpr      %g1, %g2, %tstate
     rdpr      %canrestore, %g1
     wrpr      %g0, 0, %canrestore
     wrpr      %g0, %g1, %otherwin
     rdpr      %wstate, %g1
     sll       %g1, 3, %g1              ! Move WSTATE_NORMAL (client
                                        ! vector) to WSTATE_OTHER
     or        %g1, WSTATE_SERVER, %g1  ! Set WSTATE_NORMAL to the
                                        !    vector for the server
     wrpr      %g0, %g1, %wstate
     ..                                 ! Load trap state for server
     done                               ! Execute server code

cross_domain_return:
     rdpr      %otherwin, %g1
     wrpr      %g0, %g1, %canrestore
     wrpr      %g0, 0, %otherwin
     rdpr      %wstate, %g1
     srl       %g1, 3, %g1
     wrpr      %g0, %g1, %wstate        ! Reset WSTATE_NORMAL to
                                        ! client's vector
     ..        ! Restore saved trap state as necessary; this includes
               ! the return PC for the caller.
     restore   ! Go back to the caller's register window.

     ! Set CWP for caller in TSTATE
     rdpr      %tstate, %g1
     rdpr      %cwp, %g2
     bclr      TSTATE_CWP, %g1
     wrpr      %g1, %g2, %tstate

     done      ! return to the caller
```

### H.2.3.2 ASI_SECONDARY{_LITTLE}

Supervisor software can use these unrestricted ASIs to support cross-address-space access between clients and nonprivileged servers. For example, some services that are currently provided as part of a large monolithic supervisor can be separated out as nonprivileged servers (potentially occupying a separate address space). This is often referred to as the *microkernel* approach.

## H.2.4 User Trap Handlers

Supervisor software can provide efficient support for user (nonprivileged) trap handlers on SPARC V9. The RETURN instruction allows nonprivileged code to retry an instruction pointed to by the previous stack frame. This provides the semantics required for returning from a user trap handler without any change in virtual processor state. Supervisor software can invoke the user trap handler by first creating a new register window (and stack frame) on its behalf and passing the necessary arguments (including the PC and nPC for the trapped instruction) in the local registers. The code in CODE EXAMPLE H-6 shows how a user trap handler may be invoked and how it returns.

**CODE EXAMPLE H-6** User Trap Handler

```
T_EXAMPLE_TRAP:     ! Supervisor trap handler for T_EXAMPLE_TRAP tra
     save           ! Create a window for the user trap handler

     !Set CWP for new window in TSTATE
     rdpr      %tstate, %l6
     rdpr      %cwp, %l5
     bclr      TSTATE_CWP, %l6
     wrpr      %l6, %l5, %tstate

     rdpr      %tpc,%l6  !Put PC for trapped instruction in local re
     rdpr      %tnpc,%l7 !Put nPC for trapped instruction in local re
     ..                  !Get the address of the user trap handler i
                         ! for example, from a supervisor data struc

     wrpr      %l5, %tnpc     ! Put PC for user trap handler in %tnp
     done                     ! Execute user trap handler.

USER_EXAMPLE_TRAP:       !User trap handler for T_EXAMPLE_TRAP trap

     ..                  !Execute trap handler logic. Local register
                         ! can be used as scratch.

     jmpl      %l6       !Return to retry the trapped instruction.
     return    %l7
```

## H.2.5 Scratchpad Register Usage

Typical uses of Scratchpad registers are listed in TABLE H-4.

**TABLE H-4** Typical Uses of Scratchpad Registers

| Scratchpad Register # | Usage |
|---|---|
| 0 | Register save |
| 1 | CPU structure pointer |
| 2 | Thread pointer |
| 3–7 | Temporary values (for example, for TSB miss processing) |

# Extending the SPARC V9 Architecture

This appendix contains only material from *The SPARC Architecture Manual*, Version 9 and describes how extensions can be effectively added to the SPARC V9 architecture. The appendix is informative only. It is not part of the SPARC V9 specification.

The appendix describes the two approved methods for adding new instructions: with read and write ancillary state register (ASR) and with implementation-dependent (`IMPDEP2A`) instructions. An implementor who wants to define and implement a new SPARC V9 instruction should, if possible, use one of those methods.

**Caution –** Programs that use SPARC V9 architectural extensions may not be portable and likely will not conform to any current or future SPARC V9 binary standards.

## I.1 Read / Write Ancillary State Registers (ASRs)

The first method of adding instructions to SPARC V9 is through the use of the implementation-dependent Write Ancillary State Register (`WRASR`) and Read Ancillary State Register (`RDASR`) instructions operating on ASRs 16–31. Through a read / write instruction pair, any instruction that requires an `rs1`, `reg_or_imm`, and `rd` field can be implemented. A `WRASR` instruction can also perform an arbitrary operation on two register sources, or on one register source and a signed immediate value, and place the result in an ASR. A subsequent `RDASR` instruction can read the result ASR and place its value in an integer destination register.

# I.2 Implementation-Dependent and Reserved Opcodes

The meaning of “reserved” for SPARC V9 opcodes differs from its meaning in SPARC V8. The SPARC V9 definition of “reserved” allows implementations to use reserved opcodes for implementation-specific purposes. While a hardware implementation that uses reserved opcodes will be SPARC V9-compliant, SPARC V9 ABI-compliant programs cannot use these reserved opcodes and remain compliant. A SPARC V9 platform that implements instructions using reserved opcodes must provide software libraries that provide the interface between SPARC V9 ABI-compliant programs and these instructions. Graphics libraries provide a good example of this. Hardware platforms have many diverse implementations of graphics acceleration hardware, but graphics application programs are insulated from this diversity through libraries.

There is no guarantee that a reserved opcode will not be used for additional instructions in a future version of the SPARC architecture. Implementors who use reserved opcodes should keep this fact in mind.

In some cases, forward compatibility may not be an issue; for example, in an embedded application, binary compatibility may not be an issue. These implementations can use any reserved opcodes for extensions.

Even when forward compatibility is an issue, future SPARC revisions are likely to contain few changes to opcode assignments, given that backward compatibility with previous versions must be maintained. It is recommended that implementations wishing to remain forward-compatible use the new `IMPDEP2A` reserved opcodes with $op3<5:0> = 11\ 0111_2$.

It is further recommended that SPARC International be notified of any use of `IMPDEP2A` or other reserved opcodes. When and if future revisions to SPARC are contemplated and if any SPARC V9 implementations have made use of reserved opcodes, SPARC International will make every effort not to use those opcodes. Coordinating through SPARC International provides feedback and cooperation in the choice of opcodes and maximizes the probability of forward compatibility. Given the historically small number of implementation-specific changes, coordinating through SPARC International should be sufficient to ensure future compatibility.

Note that SPARC JPS2 has already made use of many `IMPDEP1` opcodes (see TABLE E-12 on page 465). Specific JPS2 implementations have used additional opcodes from `IMPDEP1` and/or `IMPDEP2`; see individual JPS2 Extension documents for details.

# Programming with the Memory Models

This appendix contains only material from *The SPARC Architecture Manual*, Version 9, and describes how to program with the SPARC V9 memory models. An intuitive description of the models is provided in Chapter 8, *Memory Models*. A complete formal specification appears in Appendix D, *Formal Specification of the Memory Models*. In this appendix, general programming guidelines are given first, followed by specific examples showing how low-level synchronization can be implemented in TSO, PSO, and RMO.

Note that code written for a weaker memory model will execute correctly in any of the stronger memory models. Furthermore, the only possible difference between code written for a weaker memory model and the corresponding code for a stronger memory model is the presence of memory ordering instructions (`MEMBAR`s) that are not needed for the stronger memory model. Hence, transforming code from/to a stronger memory model to/from a weaker memory model means adding/removing certain memory ordering instructions.[1] The required memory ordering directives are monotonically ordered with respect to the strength of the memory model, with the weakest memory model requiring the strongest memory ordering instructions.

The code examples given below are written to run correctly using the RMO memory model. The comments on the `MEMBAR` instructions indicate which ordering constraints (if any) are required for the PSO and TSO memory models.

1. `MEMBAR` instructions specify seven independent ordering constraints; thus, there are cases where the transition to a stronger memory model allows the use of a less restrictive `MEMBAR` instruction but still requires a `MEMBAR` instruction. To demonstrate this property, the code examples given in this appendix use multiple `MEMBAR` instructions if some of the ordering constraints are needed in some but not all memory models. Multiple, adjacent `MEMBAR` instructions can always be replaced with a single `MEMBAR` instruction by ORing the arguments.

# J.1 Memory Operations

Programs access memory via five types of operations, namely, load, store, LDSTUB, SWAP, and compare-and-swap. Load copies a value from memory or an I/O location to a register. Store copies a value from a register into memory or an I/O location. LDSTUB, SWAP, and compare-and-swap are atomic load-store instructions that store a value into memory or an I/O location and return the old value in a register. The value written by the atomic instructions depends on the instruction. LDSTUB stores all ones in the accessed byte, SWAP stores the supplied value, and compare-and-swap stores the supplied value only if the old value equals the second supplied value.

Memory order and consistency are controlled by MEMBAR instructions. For example, a MEMBAR #StoreStore (equivalent to a STBAR in SPARC V8) ensures that all previous stores have been performed before subsequent stores and atomic load-stores are executed by memory. This particular memory order is guaranteed implicitly in TSO, but PSO and RMO require this instruction if the correctness of a program depends on the order in which two store instructions can be observed by another virtual processor.

---

**Note –** Memory order is of concern only to programs containing multiple threads that share writable memory and that may run on multiple virtual processors, and to those programs that reference I/O locations. Note that from the virtual processor's point of view, I/O devices behave like other virtual processors.

---

FLUSH is not a memory operation, but it is relevant here in the context of synchronizing stores with instruction execution. When a virtual processor modifies an instruction at address *A*, it does a store to *A* followed by a FLUSH *A*. The FLUSH ensures that the change made by the store will become visible to the instruction fetch units of all virtual processors in the system.

# J.2 Memory Model Selection

Given that all SPARC V9 systems are required to support TSO, programs written for any memory model will be able to run on any SPARC V9 system. However, a system running with the TSO model generally will offer lower performance than PSO or RMO because less concurrency is exposed to the virtual processor and the memory system. The motivation for weakening the memory model is to allow the virtual processor to issue multiple, concurrent memory references in order to hide memory

latency and increase access bandwidth. For example, PSO and RMO allow the virtual processor to initiate new store operations before an outstanding store has completed.

Using a weaker memory model for an MP (multiprocessor) application that exhibits a high degree of read-write memory sharing with fine granularity and a high frequency of synchronization operations may result in frequent `MEMBAR` instructions.

In general, it is expected that the weaker memory models offer a performance advantage for multiprocessor SPARC V9 implementations.

## J.3 Virtual Processor and Processes

In the SPARC V9 memory models, the term "virtual processor" may be replaced systematically by the term "process" or "thread," as long as the code for switching processes or threads is written properly. The correct process-switch sequence is given in *Process Switch Sequence* on page 549. If an operating system implements this process-switch sequence, application programmers may completely ignore the difference between a process/thread and a virtual processor.

## J.4 Higher-Level Programming Languages and Memory Models

The SPARC V9 memory models are defined at the machine instruction level. Special attention is required to write the critical parts of MP/MT (multithreaded) applications in a higher-level language. Current higher-level languages do not support memory ordering instructions and atomic operations. As a result, MP/MT applications that are written in a higher-level language generally will rely on a library of MP/MT support functions, for example, the *parmacs* library from Argonne National Laboratory.[1] The details of constructing and using such libraries are beyond the scope of this document.

Compiler optimizations such as code motion and instruction scheduling generally do not preserve the order in which memory is accessed, but they do preserve the data dependencies of a single thread. Compilers do not, in general, deal with the additional dependency requirements to support sharing read-write data among multiple concurrent threads. Hence, the memory semantics of a SPARC V9 system in

1. Lusk, E. L., R .A. Overbeek, "Use of Monitors in Fortran: A Tutorial on the Barrier, Self-scheduling Do-Loop, and Askfor Monitors," TR# ANL-84-51, Argonne National Laboratory, June 1987.

general are not preserved by optimizing compilers. For this reason, and because memory ordering directives are not available from higher-level languages, the examples presented in this appendix use assembly language.

Future compilers may have the ability to present the programmer with a sequentially consistent memory model despite the underlying hardware's providing a weaker memory model.[1]

## J.5 Portability and Recommended Programming Style

Whether a program is portable across various memory models depends on how it synchronizes access to shared read-write data. Two aspects of a program's style are relevant to portability:

- **Good semantics** refers to whether the synchronization primitives chosen and the way in which they are used are such that changing the memory model does not involve making any changes to the code that uses the primitives.
- **Good structure** refers to whether the code for synchronization is encapsulated through the use of primitives such that when the memory model is changed, required changes to the code are confined to the primitives.

Good semantics are a prerequisite for portability, and good structure makes porting easier.

Programs that use single-writer / multiple-reader locks to protect all access to shared read-write data are portable across all memory models. The code that implements the lock primitives themselves is portable across all models only if it is written to run correctly on RMO. If the lock primitives are collected into a library, then, at worst, only the library routines must be changed. Note that mutual exclusion (mutex) locks are a degenerate type of single-writer / multiple-readers lock.

Programs that use write locks to protect write accesses but read without locking are portable across all memory models only if writes to shared data are separated by `MEMBAR #StoreStore` instructions and if reading the lock is followed by a `MEMBAR #LoadLoad` instruction. If the `MEMBAR` instructions are omitted, the code is portable only across TSO and Strong Consistency,[2] but generally it will not work with PSO and RMO. The code that implements the lock primitives is portable across all models

1. See Gharachorloo, K., S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill, "Programming for Different Memory Consistency Models," *Journal of Parallel and Distributed Systems*, 15:4, August 1992.

2. Programs that assume a sequentially consistent memory are not guaranteed to run correctly on any SPARC V9-compliant system, since TSO is the strongest memory model required to be supported. However, sequential consistency is the most natural and intuitive programming model. This motivates the development of compiler techniques that allow programs written for sequential consistency to be translated into code that runs correctly (and efficiently) on systems with weaker memory models.

only if it is written to run correctly on RMO. If the lock routines are collected into a library, the only possible changes not confined to the library routines are the MEMBAR instructions.

Programs that do synchronization without using single-writer / multiple-reader locks, write locks, or their equivalent are, in general, not portable across different memory models. More precisely, the memory models are ordered from RMO (which is the weakest, least constrained, and most concurrent), PSO, TSO, to sequentially consistent (which is the strongest, most constrained, and least concurrent). A program written to run correctly for any particular memory model will also run correctly in any of the stronger memory models, but not vice versa. Thus, programs written for RMO are the most portable, those written for TSO are less so, and those written for strong consistency are the least portable. This general relationship between the memory models is shown graphically in FIGURE J-1.

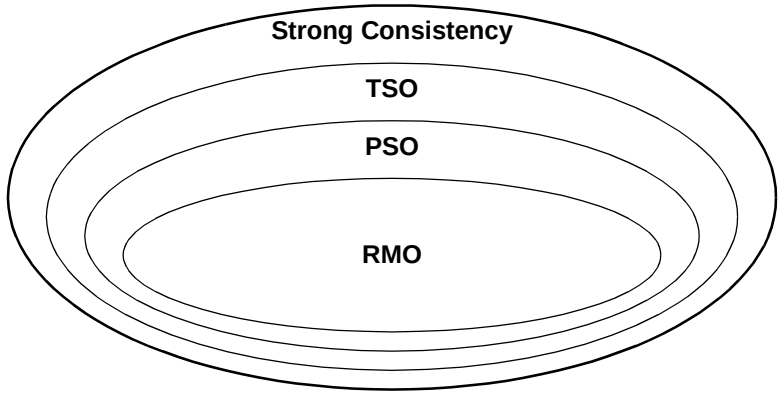



**FIGURE J-1** Portability Relations Among Memory Models

The style recommendations may be summarized as follows: Programs should use single-writer / multiple-reader locks, or their equivalent, when possible. Other lower-level forms of synchronization (such as Dekker's algorithm for locking) should be avoided when possible. When use of such low-level primitives is unavoidable, it is recommended that the code be written to work on the RMO model to ensure portability. Additionally, lock primitives should be collected together into a library and written for RMO to ensure portability.

Appendix D, *Formal Specification of the Memory Models*, describes a tool and method that allows short code sequences to be formally verified for correctness.

# J.6 Spin Locks

A spin lock is a lock for which the “lock held” condition is handled by busy waiting. The code in CODE EXAMPLE J-1 shows how spin locks can be implemented with LDSTUB. A nonzero value for the lock represents the locked condition, and a zero value means that the lock is free. Note that the code busy-waits by doing loads to avoid generating expensive stores to a potentially shared location. The MEMBAR #StoreStore in UnLockWithLDSTUB ensures that pending stores are completed before the store that frees the lock.

**CODE EXAMPLE J-1** Lock and Unlock with LDSTUB

```
LockWithLDSTUB(lock)

retry:
     ldstub    [lock],%l0
     tst       %l0
     be        out
     nop
loop:
     ldub      [lock],%l0
     tst       %l0
     bne       loop
     nop
     ba,a      retry
out:
     membar    #LoadLoad | #LoadStore

UnLockWithLDSTUB(lock)
     membar    #StoreStore         !RMO and PSO only
     membar    #LoadStore          !RMO only
     stub      %g0,[lock]
```

The code in CODE EXAMPLE J-2 shows how spin locks can be implemented with CASA. Again, a nonzero value for the lock represents the locked condition; a zero value means the lock is free. The nonzero lock value (ID) is supplied by the caller and may be used to identify the current owner of the lock. This value is available while spinning and could be used to maintain a timeout or to verify that the thread holding the lock is still running. As in the previous case, the code busy-waits by doing loads, not stores.

**CODE EXAMPLE J-2** Lock and Unlock with CAS

```
LockWithCAS(lock, ID)
retry:
     mov       [ID],%l0
     cas       [lock],%g0,%l0
     tst       %l0
     be        out
     nop
loop:
     ld        [lock],%l0
     tst       %l0
     bne       loop
     nop
     ba,a      retry

out:
     membar    #LoadLoad | #LoadStore  !See CODE EXAMPLE J-1

UnLockWithCAS(lock)
     membar    #StoreStore             !RMO and PSO only
     membar    #LoadStore              !RMO only
     st        %g0,[lock]
```

# J.7 Producer-Consumer Relationship

In a producer-consumer relationship, the producer process generates data and puts it into a buffer, while the consumer process takes data from the buffer and uses it. If the buffer is full, the producer process stalls when trying to put data into the buffer. If the buffer is empty, the consumer process stalls when trying to remove data.

FIGURE J-2 shows the buffer data structure and register usage. CODE EXAMPLE J-3 shows the producer and consumer code. The code assumes the existence of two procedures, `IncrHead` and `IncrTail`, which increment the head and tail pointers of the buffer in a wraparound manner and return the incremented value, but do not modify the pointers in the buffer.

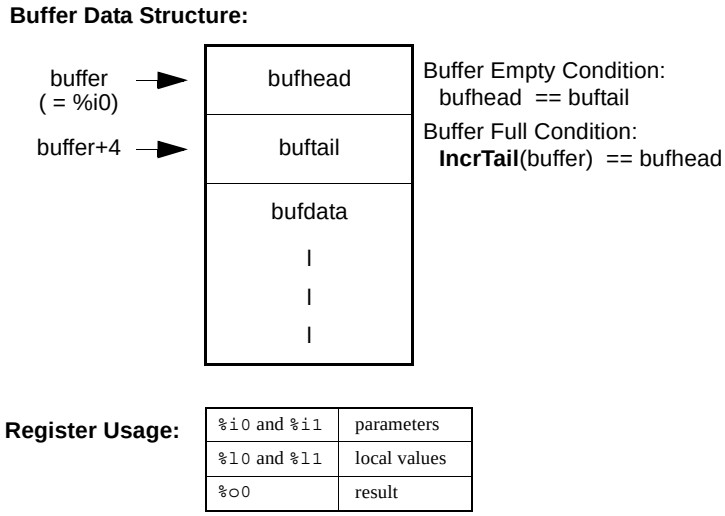



**FIGURE J-2** Data Structures for Producer-Consumer Code

**CODE EXAMPLE J-3** Producer and Consumer Code

```
Produce(buffer, data)
     call      IncrTail
full:
     ld        [%i0],%l0
     cmp       %l0,%o0
     be        full
     ld        [%i0+4],%l0
     st        %i1,[%l0]
     membar    #StoreStore         !RMO and PSO only
     st        %o0,[%i0+4]

Consume(buffer)
     ld        [%i0],%l0
empty:
     ld        [%i0+4],%l1
     cmp       %l0,%l1
     be        empty
     call      IncrHead
     ld        [%l0],%l0
     membar    #LoadStore          !RMO only
     st        %o0,[%i0]
     mov       %l0,%o0
```

# J.8 Process Switch Sequence

This section provides code that must be used during process or thread switching to ensure that the memory model seen by a process or thread is the one seen by a virtual processor. The `HeadSequence` must be inserted at the beginning of a process or thread when it starts executing on a virtual processor. The `TailSequence` must be inserted at the end of a process or thread when it relinquishes a virtual processor.

CODE EXAMPLE J-4 shows the head and tail sequences. The two sequences refer to a per-process variable `tailDone`. The value 0 for `tailDone` means that the process is running; the value –1 (all ones) means that the process has completed its tail sequence and may be migrated to another virtual processor if the process is runnable. When a new process is created, `tailDone` is initialized to –1.

**CODE EXAMPLE J-4** Process or Thread Switch Sequence

```
HeadSequence(tailDone)
nrdy:
     ld        [tailDone],%l0
     cmp       %l0,-1
     bne       nrdy
     st        %g0, [tailDone]
     membar    #StoreLoad

TailSequence(tailDone)
     mov       -1,%l0
     membar    #StoreStore         !RMO and PSO only
     membar    #LoadStore          !RMO only (combine with above)
     st        %l0,[tailDone]
```

The `MEMBAR` in `HeadSequence` is required to be able to provide a switching sequence that ensures that the state observed by a process in its source virtual processor will also be seen by the process in its destination virtual processor. Since flushes and stores are totally ordered, the head sequence need not do anything special to ensure that flushes performed prior to the switch are visible by the new virtual processor.

**Programming Note –** A conservative implementation may simply use a `MEMBAR` with all barriers set.

## J.9 Dekker's Algorithm

Dekker's algorithm is the classical sequence for synchronizing entry into a critical section, using loads and stores only. We show this example here to illustrate how one may ensure that a store followed by a load in issuing order will be executed by the memory system in that order. Dekker's algorithm is *not* a valid synchronization primitive for SPARC V9, because it requires a sequentially consistent (SC) memory model in order to work. Dekker's algorithm (and similar synchronization sequences) can be coded on RMO, PSO, and TSO by adding appropriate `MEMBAR` instructions. This example also illustrates how future compilers can provide the equivalent of sequential consistency on systems with weaker memory models.

CODE EXAMPLE J-5 shows the entry and exit sequences for Dekker's algorithm.

The locations *A* and *B* are used for synchronization. *A* = 0 means that process P1 is outside its critical section, and any other value means that P1 is inside it; similarly, *B* = 0 means that P2 is outside its critical section, and any other value means that P2 is inside it.

Dekker's algorithm guarantees mutual exclusion, but it does not guarantee freedom from deadlock. In this case, it is possible that both virtual processors end up trying to enter the critical region without success. The code above tries to address this problem by briefly releasing the lock in each retry loop. However, both stores are likely to be combined in a store buffer, so the release has no chance of success. A more realistic implementation would use a probabilistic back-off strategy that increases the released period exponentially while waiting. If any randomization is used, such an algorithm will avoid deadlock with arbitrarily high probability.

## J.10 Code Patching

The code patching example illustrates how to modify code that is potentially being executed at the time of modification. Two common uses of code patching are in debuggers and dynamic linking.

**CODE EXAMPLE J-5** Dekker's Algorithm

```
P1Entry( )
     mov       -1,%l0
busy:
     st        %l0,[A]
     membar    #StoreLoad
     ld        [B],%l1
     tst       %l1
     bne,a     busy
     st        %g0,[A]

P1Exit( )
     membar    #StoreStore         !RMO and PSO only
     membar    #LoadStore          !RMO only
     st        %g0,[A]

P2Entry( )
     mov       -1,%l0
busy:
     st        %l0,[B]
     membar    #StoreLoad
     ld        [A],%l1
     tst       %l1
     bne,a     busy
     st        %g0,[B]

P2Exit( )
     membar    #StoreStore         !RMO and PSO only
     membar    #LoadStore          !RMO only
     st        %g0,[B]
```

Code patching involves a modifying process, *Pm*, and one or more target processes *Pt*. For simplicity, assume that the sequence to be modified is four instructions long: the old sequence is (*Old*1, *Old*2, *Old*3, *Old*4) and the new sequence is (*New*1, *New*2, *New*3, *New*4). There are two examples: *noncooperative* modification, in which the changes are made without cooperation from *Pt*; and *cooperative* modification, in which the changes require explicit cooperation from *Pt*.

In noncooperative modification, illustrated in CODE EXAMPLE J-6, changes are made in reverse execution order.

The three partially modified sequences (*Old*1, *Old*2, *Old*3, *New*4), (*Old*1, *Old*2, *New*3, *New*4), and (*Old*1, *New*2, *New*3, *New*4) must be legal sequences for *Pt*, in that *Pt* must do the right thing if it executes any of them. Additionally, none of the locations to be modified, except the first, may be the target of a branch. The code assumes that `%i0` contains the starting address of the area to be patched and `%i1`, `%i2`, `%i3`, and `%i4` contain *New*1, *New*2, *New*3, and *New*4.

**CODE EXAMPLE J-6** Noncooperative Code Patching

```
NonCoopPatch(addr, instructions...)
     st        %i4,[%i0+12]
     flush     %i0+12
     membar    #StoreStore             !RMO and PSO only
     st        %i3,[%i0+8]
     flush     %i0+8
     membar    #StoreStore             !RMO and PSO only
     st        %i2,[%i0+4]
     flush     %i0+4
     membar    #StoreStore             !RMO and PSO only
     st        %i1,[%i0]
     flush     %i0
```

The constraint that all partially modified sequences must be legal is quite restrictive. When this constraint cannot be satisfied, noncooperative code patching may require the target virtual processor to execute `FLUSH` instructions. One method of triggering such a non-local `FLUSH` would be to send an interrupt to the target virtual processor.

In cooperative code patching, illustrated in CODE EXAMPLE J-7, changes to instructions can be made in any order.

When *Pm* is finished with the changes, it writes into the shared variable *done* to notify *Pt*. *Pt* waits for *done* to change from 0 to some other value as a signal that the changes have been completed. The code assumes that `%i0` contains the starting address of the area to be patched, `%i1`, `%i2`, `%i3`, and `%i4` contain *New*1, *New*2, *New*3, and *New*4, and `%g1` contains the address of *done*. The `FLUSH` instructions in *Pt* ensure that the instruction buffer of *Pt*'s virtual processor is flushed so that the old instructions are not executed.

## J.11 Fetch_and_Add

`Fetch_and_Add` performs the sequence $a = a + b$ atomically with respect to other `Fetch_and_Add`s to location *a*. Two versions of `Fetch_and_Add` are shown. The first (CODE EXAMPLE J-8) uses the routine `LockWithLDSTUB` described above. This approach uses a lock to guard the value. Since the memory model dependency is embodied in the lock access routines, the code does not depend on the memory model.[1]

1. Inlining of the lock-access functions with subsequent optimization may break this code.

**CODE EXAMPLE J-7** Cooperative Code Patching

```
CoopPatch(addr, instructions...) !%i0 = addr, %i1..%i4 = instruction
     st        %i1,[%i0]
     st        %i2,[%i0+4]
     st        %i3,[%i0+8]
     st        %i4,[%i0+12]
     mov       -1,%l0
     membar    #StoreStore        !RMO and PSO only
     st        %l0,[%g1]

TargetCode( )
wait:
     ld        [%g1],%l0
     cmp       %l0,0
     be        wait
     flush     A
     flush     A+4
     flush     A+8
     flush     A+12
A:
     Old1
     Old2
     Old3
     Old4
```

**CODE EXAMPLE J-8** Fetch and Add with LDSTUB

```
/*Fetch and Add using LDSTUB*/
int Fetch_And_Add(Index, Increment, Lock)
     int *Index;
     int Increment;
     int *Lock;
     {
          int old_value;
          LockWithLDSTUB(Lock);
          old_value  = *Index;
          *Index  = old_value + Increment;
          UnlockWithLDSTUB(Lock);
          return(old_value);
     }
```

Fetch_and_Add originally was invented to avoid lock contention and to provide an efficient means to maintain queues and buffers without cumbersome locks. Hence, using a lock is inefficient and contrary to the intentions of the Fetch_and_Add. The CAS synthetic instruction allows a more efficient version, as shown in CODE EXAMPLE J-9.

**CODE EXAMPLE J-9** Fetch and Add with CAS

```
FetchAndAddCAS(address, increment)!%i0 = address, %i1 = increment
retry:
     ld        [%i0],%l0
     add       %l0,%i1,%l1
     cas       [%i0],%l0,%l1
     cmp       %l0,%l1
     bne       retry
     mov       %l1,%o0             !return old value
```

# J.12 Barrier Synchronization

Barrier synchronization ensures that each of *N* processes is blocked until all of them reach a given state. The point in the flow of control at which this state is reached is called the barrier; hence the name. The code uses the variable `Count` initialized to *N*. As each process reaches its desired state, it decrements `Count` and waits for `Count` to reach zero before proceeding further.

Similar to the fetch and add operation, barrier synchronization is easily implemented using a lock to guard the counter variable, as shown in CODE EXAMPLE J-10.

**CODE EXAMPLE J-10** Barrier Synchronization with LDSTUB

```
/*Barrier Synchronization using LDSTUB*/
Barrier(Count,Lock)
int *Count;
int *Lock;
{
     LockWithLdstUB(Lock);
     *Count  = *Count - 1;
     UnlockWithLdstUB(Lock);
     while(*Count > 0)  { ; /*busy-wait*/ }
}
```

The CAS implementation of barrier synchronization, shown in CODE EXAMPLE J-11, avoids the extra lock access.

A practical barrier synchronization must be reusable because it is typically used once per iteration in applications that require many iterations. Barriers that are based on counters must have means to reset the counter. One solution to this

**CODE EXAMPLE J-11** Barrier Synchronization with CAS

```
BarrierCAS(Count)       !%i0 = address of counter variable
retry:
     ld        [%i0],%l0
     add       %l0,-1,%l1
     cas       [%i0],%l0,%l1
     cmp       %l0,%l1
     bne       retry
     nop
wait:
     ld        [%i0],%l0
     tst       %l0
     bne       wait
     nop
```

problem is to alternate between two complementary versions of the barrier: one that counts down to 0 and the other that counts up to *N*. In this case, passing one barrier also initializes the counter for the next barrier.

Passing a barrier can also signal that the results of one iteration are ready for processing by the next iteration. In this case, RMO and PSO require a `MEMBAR #StoreStore` instruction prior to the barrier code to ensure that all local results become globally visible prior to passing the barrier.

Barrier synchronization among a large number of virtual processors will lead to access contention on the counter variable, which may degrade performance. This problem can be solved by use of multiple counters. The butterfly barrier uses a divide-and-conquer technique to avoid any contention and can be implemented without atomic operations.[1]

## J.13 Linked List Insertion and Deletion

Linked lists are dynamic data structures that might be used by a multithreaded application. As in the previous examples, a lock can be used to guard access to the entire data structure. However, single locks guarding large and frequently shared data structures can be inefficient.

1. Brooks, E. D., "The Butterfly Barrier," *International Journal of Parallel Programming* 15(4), pp. 295-307, 1986.

In CODE EXAMPLE J-12, the CAS synthetic instruction is used to operate on a linked list without explicit locking. Each list element starts with an address field that contains either the address of the next list element or zero. The head of the list is the address of a variable that holds the address of the first list element. The head is zero for empty lists.

**CODE EXAMPLE J-12** List Insertion and Removal

```
ListInsert(Head, Element)    !%i0 = Head addr, %i1 = Element addr
retry:
     ld        [%i0],%l0
     st        %l0, [%i1]
     mov       %i1, %l1
     cas       [%i0],%l0,%l1
     cmp       %l0,%l1
     bne       retry
     nop

ListRemove(Head)             !%i0  = Head addr
retry:
     ld        [%i0],%o0
     tst       %o0
     be        empty
     nop
     ld        [%o0],%l0
     cas       [%i0],%o0,%l0
     cmp       %o0,%l0
     bne       retry

empty:
     nop
```

In the example, there is little difference in performance between the CAS and lock approaches; however, more complex data structures can allow more concurrency. For example, a binary tree allows the concurrent insertion and removal of nodes in different branches.

## J.14 Communicating with I/O Devices

I/O accesses may be reordered just as other memory reference are reordered. Because of this, the programmer must take special care to meet the constraint requirements of physical devices, in both the uniprocessor and multiprocessor cases.

Accesses to I/O locations require sequencing MEMBARs under certain circumstances to properly manage the order of accesses arriving at the device and the order of device accesses with respect to memory accesses. The following rules describe the use of MEMBARs in these situations. Maintaining the order of accesses to multiple devices will require higher-level software constructs, which are beyond the scope of this discussion.

1. **Accesses to the same I/O location address:**
   - A store followed by a store is ordered in all memory models.
   - A load followed by a load requires a MEMBAR #LoadLoad in RMO only.

---

**V9 Compatibility Note –** This MEMBAR is not needed in implementations that provide SPARC V8 compatibility for I/O accesses in RMO.

---

   - A load followed by a store is ordered in all memory models.
   - A store followed by a load requires MEMBAR #Lookaside between the accesses for all memory models; however, implementations that provide SPARC V8 compatibility for I/O accesses in any of TSO, PSO, and RMO do not need the MEMBAR in any model that provides this compatibility.

2. **Accesses to different I/O location addresses:**
   - The appropriate ordering MEMBAR is required to guarantee order within a range of addresses assigned to a device.
   - Device-specific synchronization of completion, such as reading back from an address after a store, may be required to coordinate accesses to multiple devices. This is beyond the scope of this discussion.

3. **Accesses to an I/O location address and a memory address.**
   - A MEMBAR #MemIssue is required between an I/O access and a memory access if it is required that the I/O access reaches global visibility before the memory access reaches global visibility. For example, if the memory location is a lock that controls access to an I/O address, then MEMBAR #MemIssue is required between the last access to the I/O location and the store that clears the lock.

4. **Accesses to different I/O location addresses within an implementation-dependent range of addresses are strongly ordered once they reach global visibility.** Beyond the point of global visibility there is no guarantee of global order of accesses arriving at different devices having disjoint implementation-dependent address ranges defining the device. Programmers can rely on this behavior from implementations.

5. **Accesses to I/O locations protected by a lock in shared memory that is subsequently released, with attention to the above barrier rules, are strongly ordered with respect to any subsequent accesses to those locations that respect the lock.**

## J.14.1 I/O Registers with Side Effects

I/O registers with side effects are commonly used in hardware devices such as UARTs. One register is used to address an internal register of the I/O device, and a second register is used to transfer data to or from the selected internal register.

In CODE EXAMPLE J-13 and CODE EXAMPLE J-14, let X be the address of a device with two such registers; X.P is a port register, and X.D is a data register. The address of an internal register is stored into X.P; that internal register can then be read or written by loading into or storing from X.D.

**CODE EXAMPLE J-13** I/O Registers with Side-Effects: Store Followed by Store

```
st        %i1, [X+P]
membar    #StoreStore                  ! PSO and RMO only
st        %i2, [X+D]
```

**CODE EXAMPLE J-14** I/O Registers with Side Effects: Store Followed by Load

```
st        %i1, [X+P]
membar    #StoreLoad |#MemIssue ! RMO only
ld        [X+D], %i2
```

Access to these registers, of course, must be protected by a mutual-exclusion lock to ensure that multiple threads accessing the registers do not interfere. The sequencing MEMBAR is required to ensure that the store actually completes before the load is issued.

## J.14.2 Control and Status Registers

A control and status register is an I/O location that is updated by an I/O device independent of access by the virtual processor. For example, such a register might contain the current sector under the head of a disk drive.

In CODE EXAMPLE J-15, let Y be the address of a control and status register that is read to obtain status and written to assert control. Bits read differ from the last data that was stored to them.

**CODE EXAMPLE J-15** Accessing a Control/Status Register

```
ld        [Y], %i1       ! obtain status
st        %i2, [Y]       ! write a command
membar    #Lookaside     ! make sure we really read the register
ld        [Y], %i3       ! obtain new status
```

Access to these registers, of course, must be protected by a mutual-exclusion lock to ensure that multiple threads accessing the registers do not interfere. The sequencing MEMBAR is needed to ensure the value produced by the load comes from the register and not from the write buffer, since the write has side-effects. No MEMBAR is needed between the load and the store because of the anti-dependency on the memory address.

## J.14.3 The Descriptor

In CODE EXAMPLE J-16, let A be the address of a descriptor in memory. After the descriptor is initialized with information, the address of the descriptor is stored into device register D or made available to some other portion of the program that will make decisions based upon the value(s) in the descriptor. It is important to ensure that the stores of the data have completed before making the address (and hence the data in the descriptor) visible to the device or program component.

**CODE EXAMPLE J-16** Accessing a Memory Descriptor

```
st        %i1, [A]
st        %i2, [A+4]
...                          ! more stores
membar    #StoreStore        ! PSO and RMO only
st        A, [D]
```

Access must be protected by a mutual-exclusion lock to ensure that multiple threads accessing the registers do not interfere. In addition, the agent reading the descriptor must use a load-barrier MEMBAR after reading D to ensure that the most recent values are read.

## J.14.4 Lock-Controlled Access to a Device Register

Let A be a lock in memory that is used to control access to a device register D. The code that accesses the device might look like that shown in CODE EXAMPLE J-17.

The sequencing MEMBAR is needed to ensure that another core which grabs the lock and loads from the device register will actually see any changes in the device induced by the store. The ordering MEMBARs in the lock and unlock code (see *Spin Locks* on page 546), while ensuring correctness when protecting ordinary memory, are insufficient for this purpose when accessing device registers. Compare with *I/O Registers with Side Effects* on page 558.

**CODE EXAMPLE J-17** Accessing a Device Register

```
set A, %l1          ! address of lock
set D, %l2          ! address of device register
call lock           ! lock(A);
mov %l1, %o0
ld [%l2], %i1       ! read the register

...                 ! do some computation

st %i2, [%l2]       ! write the register
membar #MemIssue    ! all memory models
call unlock         ! unlock(A);
mov %l1, %o0
```

# Changes from SPARC V8 to SPARC V9

This appendix contains only material from *The SPARC Architecture Manual*, Version 9, The appendix is informative only. It is not part of the SPARC V9 specification.

SPARC V9 is complementary to the SPARC V8 architecture; it does not replace it. SPARC V9 was designed to be a higher-performance peer to SPARC V8.

Application software for the 32-bit SPARC V8 (Version 8) microprocessor architecture can execute, unchanged, on SPARC V9 systems. SPARC V8 software executes natively on SPARC V9-conformant processors; no special compatibility mode is required.

Changes to the SPARC V9 architecture since SPARC V8 are in seven main areas: the trap model, data formats, endianness, the registers, alternate address space access, the instruction set, and the memory model.

## K.1 Trap Model

The trap model, visible only to privileged software, has changed substantially.

- Instead of one level of traps, four or more levels are now supported. This allows first-level trap handlers, notably register window spill and fill (formerly called overflow and underflow) traps, to execute much faster. Such trap handlers can now execute without costly run-time checks for lower-level trap conditions, such as page faults or a misaligned stack pointer. Also, multiple trap levels support more robust fault-tolerance mechanisms.
- Most traps no longer change the `CWP`. Instead, the trap state (including the `CWP` register) is saved in register stacks called `TSTATE`, `TT`, `TPC`, and `TNPC`.

- New instructions (DONE and RETRY) are used instead of RETT to return from a trap handler.
- A new instruction (RETURN) is provided for returning from a trap handler running in nonprivileged mode, providing support for user trap handlers.
- Terminology about privileged-mode execution has changed: from “supervisor / user” to “privileged / nonprivileged.”
- A new processor state, RED_state, has been added to facilitate processing resets and nested traps that would exceed MAXTL.

## K.2 Data Formats

Data formats for extended (64-bit) integers have been added.

## K.3 Little-Endian Support

Data accesses can be either big-endian or little-endian. Bits in the PSTATE register control the implicit endianness of data accesses. Special ASI values are provided to allow specific data accesses to be in a specific endianness.

## K.4 Little-Endian Byte Order

In SPARC V8, all instruction and data accesses were performed in big-endian byte order. SPARC V9 supports both big- and little-endian byte orders for data accesses only; instruction accesses in SPARC V9 are always performed using big-endian order.

## K.5 Registers

These privileged SPARC V8 registers have been deleted:

- **PSR:** Processor State Register
- **TBR:** Trap Base Register
- **WIM**: Window Invalid Mask

These registers have been widened from 32 to 64 bits:

- All integer registers
- **All state registers:** FSR, PC, nPC, Y

The contents of the following register has changed:

- **FSR:** Floating-Point State Register: fcc1, fcc2, and fcc3 (additional floating-point condition code) bits have been added and the register widened to 64 bits.

These SPARC V9 registers are fields within a register in SPARC V8:

- **PIL:** Processor Interrupt Level register
- **CWP:** Current Window Pointer register
- **TT[MAXTL]:** Trap Type register
- **TBA:** Trap Base Address register
- **VER:** Version register
- **CCR:** Condition Codes Register

These registers have been added:

- **Sixteen additional double-precision floating-point registers, F[32]–F[62]**, which are aliased with and overlap eight additional quad-precision floating-point registers, F[32]–F[60]
- **FPRS:** Floating-Point Register State register
- **ASI:** ASI register
- **PSTATE:** Processor State register
- **TL:** Trap Level register
- **TPC[MAXTL]:** Trap Program Counter register
- **TNPC[MAXTL]:** Trap Next Program Counter register
- **TSTATE[MAXTL]:** Trap State register
- **TICK:** Hardware clock-tick counter
- **CANSAVE:** Savable windows register
- **CANRESTORE:** Restorable windows register
- **OTHERWIN:** Other Windows register
- **CLEANWIN:** Clean Windows register
- **WSTATE:** Window State register

The SPARC V9 CWP register is incremented during a SAVE instruction and decremented during a RESTORE instruction. Although this is the opposite of PSR.CWP's behavior in SPARC V8, the only software it should affect is a few trap handlers that operate in privileged mode, and those must be rewritten for SPARC V9 anyway. This change will have no effect on nonprivileged software.

# K.6 Alternate Space Access

In SPARC V8, access to all alternate address spaces is privileged. In SPARC V9, loads and stores to ASIs $00_{16}$–$7F_{16}$ are privileged; those to ASIs $80_{16}$–$FF_{16}$ are nonprivileged. That is, load- and store-alternate instructions to one-half of the alternate spaces can now be used in user code.

# K.7 Instruction Set

All changes to the instruction set were made such that application software written for SPARC V8 can run unchanged on a SPARC V9 processor. Application software written for SPARC V8 should not even be able to detect that its instructions now process 64-bit values.

The definitions of the following instructions were extended or modified to work with the 64-bit model:

- **FCMP, FCMPE:** Floating-Point Compare: Can set any of the four floating-point condition codes
- **LDUW, LDUWA** (same as "LD, LDA" in SPARC V8)
- **LDFSR, STFSR:** Load / Store FSR: Only affect low-order 32 bits of FSR
- **RDASR/WRASR:** Read / Write State Registers: Access additional registers
- SAVE / RESTORE
- SETHI
- **SRA, SRL, SLL:** Shifts: Split into 32-bit and 64-bit versions
- **Tcc:** (was Ticc): Operates with either the 32-bit integer condition codes (icc), or the 64-bit integer condition codes (xcc)
- **All other arithmetic operations now operate on 64-bit operands and produce 64-bit results.** Application software written for SPARC V8 cannot detect that arithmetic operations are now 64 bits wide. This is due to retention of the 32-bit integer condition codes (icc), addition of 64-bit integer condition codes (xcc), and the carry-propagation rules of two's-complement arithmetic.

The following instructions have been added to provide support for 64-bit operations and / or addressing:

- **F[sdq]TOx:** Convert floating point to 64-bit word
- **FxTO[sdq]:** Convert 64-bit word to floating point
- **FMOV[dq]:** Floating-point Move, double and quad
- **FNEG[dq]:** Floating-point Negate, double and quad

- **FABS[dq]:** Floating-point Absolute Value, double and quad
- **LDDFA, STDFA, LDFA, STFA:** Alternate address space forms of LDDF, STDF, LDF, and STF
- **LDSW:** Load a signed word
- **LDSWA:** Load a signed word from an alternate space
- **LDX:** Load an extended word
- **LDXA:** Load an extended word from an alternate space
- **LDXFSR:** Load all 64 bits of the FSR register
- **STX:** Store an extended word
- **STXA:** Store an extended word into an alternate space
- **STXFSR:** Store all 64 bits of the FSR register

The following instructions have been added to support the new trap model:

- **DONE:** Return from trap and skip instruction that trapped
- **RDPR and WRPR:** Read and Write privileged registers
- **RESTORED:** Adjust state of register windows after RESTORE
- **RETRY:** Return from trap and reexecute instruction that trapped
- **RETURN:** Return
- **SAVED:** Adjust state of register windows after SAVE
- **SIR:** Signal Monitor (generate software-initiated reset)

The following instructions have been added to support implementation of higher-performance systems:

- **BPcc:** Branch on integer condition code with prediction
- **BPr:** Branch on integer register contents with prediction
- **CASA, CASXA:** Compare and Swap from an alternate space
- **FBPfcc:** Branch on floating-point condition code with prediction
- **FLUSHW:** Flush windows
- **FMOVcc:** Move floating-point register if condition code is satisfied
- **FMOVr:** Move floating-point register if integer register contents satisfy condition
- **LDQF(A), STQF(A):** Load / Store Quad Floating-point (in an alternate space)
- **MOVcc:** Move integer register if condition code is satisfied
- **MOVr:** Move integer register if register contents satisfy condition
- **MULX:** Generic 64-bit multiply
- **POPC:** Population Count
- **PREFETCH, PREFETCHA:** Prefetch Data
- **SDIVX, UDIVX**: Signed and Unsigned 64-bit divide

The definitions of the following instructions have changed:

- **IMPDEP*n*:** SPARC V8 CPop instructions have been replaced by VIS, IMPDEP2A, and IMPDEP2B instructions.

The following instruction was added to support memory synchronization:

- **MEMBAR:** Memory barrier

The following instructions have been deleted:

- Coprocessor loads and stores
- **RDTBR and WRTBR:** TBR no longer exists. It has been replaced by TBA, which can be read/written with RDPR/WRPR instructions.
- **RDWIM and WRWIM**: WIM no longer exists. WIM has been subsumed by several register-window state registers.
- **RDPSR and WRPSR:** PSR no longer exists. It has been replaced by several separate registers which are read/written with other instructions.
- **RETT:** Return from trap (replaced by DONE/RETRY).
- **STDFQ:** Store Double from Floating-point Queue (replaced by the RDPR FQ instruction).

## K.8 Memory Model

SPARC V9 defines a new memory model called Relaxed Memory Order (RMO). This very weak model allows the core hardware to schedule memory accesses such as loads and stores in nearly any order, as long as the program computes the correct answer. Hence, the hardware can instantaneously adjust to resource contentions and schedule accesses in the most efficient order, leading to much faster memory operations and better performance.

# Address Space Identifiers (ASIs)

This appendix describes address space identifiers (ASIs) in the following sections:


- *Address Space Identifiers and Address Spaces* on page 567
- *ASI Values* on page 568
- *ASI Assignments* on page 568


## L.1 Address Space Identifiers and Address Spaces

A SPARC V9 processor provides an address space identifier (ASI) with every address sent to memory. The ASI does the following:

- Distinguishes between different address spaces
- Provides an attribute that is unique to an address space
- Maps internal control and diagnostics registers within a virtual processor

The memory management hardware uses a 64-bit virtual address and an 8-bit ASI to generate a physical address. This physical address space can be accessed through virtual-to-physical address mapping or through the MMU bypass mode.

SPARC V9 also extended the limit of virtual addresses from 32 bits to 64 bits for each address space. SPARC V9 supports 32-bit addressing through masking of the upper 32 bits to 0 when the address mask (`am`) bit in the `PSTATE` register is set.

# L.2 ASI Values

The SPARC V9 address space identifier (ASI) is evenly divided into restricted and unrestricted halves. ASIs in the range $00_{16}$–$7F_{16}$ are restricted. ASIs in the range $80_{16}$–$FF_{16}$ are unrestricted. An attempt by nonprivileged software to access a restricted ASI causes a *privileged_action* trap.

*Normal* or *translating* ASIs are translated by the MMU.

*Nontranslating* ASIs are not translated by the MMU and instead pass through their virtual addresses as physical addresses.

*Bypass* ASIs, like nontranslating ASIs, are not translated by the MMU and instead pass through their virtual addresses as physical addresses. However, unlike a nontranslating ASI, access to a bypass ASI can cause a *PA_watchpoint* trap.

A table of Translating, Nontranslating, and Bypass ASIs is provided in TABLE 5-17 on page 94.

Implementation-dependent ASIs may or may not be translated by the MMU. See Appendix L in the JPS2 Extension documents for a given implementation for detailed information about specific implementation-dependent ASIs.

# L.3 ASI Assignments

Every load or store address in a SPARC V9 processor has an 8-bit Address Space Identifier (ASI) appended to the virtual address (VA). The VA plus the ASI fully specify the address. For instruction fetches and for data loads or stores that do not use the load or store alternate instructions, the ASI is an implicit ASI generated by the hardware. If a load alternate or store alternate instruction is used, the value of the ASI can be specified in the `%asi` register or as an immediate value in the instruction. In practice, ASIs are not only used to differentiate address spaces but are also used for other functions like referencing registers in the MMU unit.

## L.3.1 Supported ASIs

TABLE L-1 lists both the SPARC V9 architecture-defined ASIs and ASIs that were not defined in SPARC V9 but are required for JPS2 virtual processor.

ASIs marked with a closed bullet (●) are SPARC V9 architecture-defined ASIs. All operand sizes are supported when accessing one of these ASIs.

ASIs marked with an open bullet (❍) were not defined in SPARC V9 but were defined in JPS1 and are required to be implemented in all JPS1 and JPS2 processors. ASIs marked with an open square bullet (❑) were not required by SPARC V9 or JPS1 but are required to be implemented in all JPS2 processors. These ASIs can be used only with LDXA, STXA, LDDFA, or STDFA instructions, unless otherwise noted. An attempt to access any of these ASIs with other load or store alternate instructions (for example, using a byte-, halfword-, or word-length access) causes a *data_access_exception* trap, unless otherwise noted. Whether access with LDDA/STDA causes a *data_access_exception* trap is implementation dependent (impl. dep. # 300, 301).

The word "decoded" in the Virtual Address column of TABLE L-1 indicates that the the supplied virtual address is decoded by the virtual processor.

Attempting to access an address space described as "Implementation dependent" in TABLE L-1 produces implementation-dependent results.

**TABLE L-1** JPS2 ASIs *(1 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $00_{16}$–$03_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $04_{16}$ | ● | ASI_NUCLEUS (ASI_N) | RW | (decoded) | T | — | Implicit address space, nucleus privilege, TL > 0 | 8.3 |
| $05_{16}$–$0B_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $0C_{16}$ | ● | ASI_NUCLEUS_LITTLE (ASI_NL) | RW | (decoded) | T | — | Implicit address space, nucleus privilege, TL > 0, little-endian | 8.3 |
| $0D_{16}$–$0F_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $10_{16}$ | ● | ASI_AS_IF_USER_PRIMARY (ASI_AIUP) | RW[2] | (decoded) | T | — | Primary address space, user privilege | 8.3 |
| $11_{16}$ | ● | ASI_AS_IF_USER_SECONDARY (ASI_AIUS) | RW[2] | (decoded) | T | — | Secondary address space, user privilege | 8.3 |
| $12_{16}$–$13_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $14_{16}$ | ❍ | ASI_PHYS_USE_EC | RW[3,4] | (decoded) | B | — | Physical address external cacheable only | L.3.2 |
| $15_{16}$ | ❍ | ASI_PHYS_BYPASS_EC_WITH_EBIT | RW[3] | (decoded) | B | — | Physical address, noncacheable, with side effect | L.3.2 |

**TABLE L-1** JPS2 ASIs *(2 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $16_{16}$–$17_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $18_{16}$ | ● | ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL) | RW[2] | (decoded) | T | — | Primary address space, user privilege, little-endian | 8.3 |
| $19_{16}$ | ● | ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL) | RW[2] | (decoded) | T | — | Secondary address space, user privilege, little-endian | 8.3 |
| $1A_{16}$–$1B_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $1C_{16}$ | ❍ | ASI_PHYS_USE_EC_LITTLE (ASI_PHYS_USE_EC_L) | RW [3,4] | (decoded) | B | — | Physical address, external cacheable only, little-endian | L.3.2 |
| $1D_{16}$ | ❍ | ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE (ASI_PHYS_BYPASS_EC_WITH_EBIT_L) | RW [3] | (decoded) | B | — | Physical address, noncacheable, with side effect, little-endian | L.3.2 |
| $1E_{16}$–$23_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $24_{16}$ | ❍ | ASI_NUCLEUS_QUAD_LDD | R[5,9] | (decoded) | T | — | Cacheable, 128-bit (quad) atomic load | L.3.2 |
| $25_{16}$–$2B_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $2C_{16}$ | ❍ | ASI_NUCLEUS_QUAD_LDD_LITTLE, (ASI_NUCLEUS_QUAD_LDD_L) | R[5,9] | (decoded) | T | — | Cacheable, 128-bit (quad) atomic load, little-endian | L.3.2 |
| $2D_{16}$–$33_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $34_{16}$ | ❑ | ASI_QUAD_LDD_PHYS (ASI_ATOMIC_QUAD_LDD_PHYS†) | R[5,9] | (decoded) | B | — | 128-bit atomic load | L.3.2 |
| $35_{16}$–$3B_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $3C_{16}$ | ❑ | ASI_QUAD_LDD_PHYS_LITTLE (ASI_QUAD_LDD_PHYS_L, ASI_ATOMIC_QUAD_LDD_PHYS_LITTLE†, ASI_ATOMIC_QUAD_LDD_PHYS_L†) | R[5,9] | (decoded) | B | — | 128-bit atomic load, little-endian | L.3.2 |

**TABLE L-1** JPS2 ASIs *(3 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $3D_{16}$–$40_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $41_{16}$ | ❑ | MTP control/status (shared) | | | | | | |
| $41_{16}$ | ❑ | ASI_CORE_AVAILABLE | R | $00_{16}$ | N | shared | Core Available Register | 9.4.1 |
| $41_{16}$ | ❑ | ASI_CORE_ENABLE_STATUS | R | $10_{16}$ | N | shared | Core Enabled Register | 9.4.2 |
| $41_{16}$ | ❑ | ASI_CORE_ENABLE | RW | $20_{16}$ | N | shared | Core Enable Register | 9.4.2 |
| $41_{16}$ | ❑ | ASI_XIR_STEERING | R/RW | $30_{16}$ | N | shared | XIR Steering Register<br>Implementation dependent<br>(impl. dep. #1105) | 9.5.3 |
| $41_{16}$ | ❑ | ASI_ERROR_STEERING | RW | $40_{16}$ | N | shared | MTP Error Steering Register | 9.6.2 |
| $41_{16}$ | ❑ | ASI_CORE_RUNNING_RW | RW | $50_{16}$ | N | shared | Core Running Register, general access. | 9.4.3 |
| $41_{16}$ | ❑ | ASI_CORE_RUNNING_STATUS | R | $58_{16}$ | N | shared | Core Running Status Register | 9.4.3 |
| $41_{16}$ | ❑ | ASI_CORE_RUNNING_W1S | W | $60_{16}$ | N | shared | Core Running Register, general access. Write one to set bits | 9.4.3 |
| | ❑ | ASI_CORE_RUNNING_W1C | W | $68_{16}$ | N | shared | Core Running Register, general access. Write one to set bits | 9.4.3 |
| $42_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $45_{16}$ | ❍ | ASI_DCU_CONTROL_REGISTER (ASI_DCUCR) | RW | $0_{16}$ | N | /core | Data Cache Unit Control Register | 5.2.4 |
| $46_{16}$–$47_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $48_{16}$ | ❍ | ASI_INTR_DISPATCH_STATUS (ASI_MONDO_SEND_CTRL) | R[5] | $0_{16}$ | N | /core | Interrupt vector dispatch status | N.2, N.5.4 |
| $49_{16}$ | ❍ | ASI_INTR_RECEIVE (ASI_MONDO_RECEIVE_CTRL) | RW | $0_{16}$ | N | /core | Interrupt vector receive status | N.5.6 |
| $4A_{16}$ | ❍ | (reserved for interconnect configuration) | — | — | — | — | Implementation dependent[1] | — |
| $4B_{16}$ | ● | — | RW | $0_{16}$ | — | — | Implementation dependent[1] | — |
| $4C_{16}$ | ❍ | ASI_ASYNC_FAULT_STATUS (ASI_AFSR) | RW[14] | $0_{16}$ | N | /core | Async fault status register | P.4 |

**TABLE L-1** JPS2 ASIs *(4 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $4C_{16}$ | ❍ | — | — | $08_{16}$ | — | — | Implementation dependent[1] | — |
| $4C_{16}$ | ❍ | — | — | $10_{16}$ | — | — | Implementation dependent[1] | — |
| $4C_{16}$ | ❍ | — | — | $18_{16}$ | — | — | Implementation dependent[1] | — |
| $4D_{16}$ | ❍ | ASI_ASYNC_FAULT_ADDR (ASI_AFAR) | RW[14] | $0_{16}$ | N | /core | Async fault address register | P.4 |
| $4D_{16}$ | ❍ | — | — | $08_{16}$ | — | — | Implementation dependent[1] | |
| $4E_{16}$ | ❍ | — | — | — | — | — | Implementation dependent[1] | — |
| $4F_{16}$ | ❑ | Scratchpad registers | | | | | | |
| | ❑ | ASI_SCRATCHPAD_0_REG | RW | $0_{16}$ | N | /core | Implementation dependent[13] | p. 98 |
| | ❑ | ASI_SCRATCHPAD_1_REG | RW | $8_{16}$ | N | /core | Implementation dependent[13] | p. 98 |
| | ❑ | ASI_SCRATCHPAD_2_REG | RW | $10_{16}$ | N | /core | Implementation dependent[13] | p. 98 |
| | ❑ | ASI_SCRATCHPAD_3_REG | RW | $18_{16}$ | N | /core | Implementation dependent[13] | p. 98 |
| | ❑ | ASI_SCRATCHPAD_4_REG | RW | $20_{16}$ | N | /core | Implementation dependent[13] | p. 98 |
| | ❑ | ASI_SCRATCHPAD_5_REG | RW | $28_{16}$ | N | /core | Implementation dependent[13] | p. 98 |
| | ❑ | ASI_SCRATCHPAD_6_REG | RW | $30_{16}$ | N | /core | Implementation dependent[13] | p. 98 |
| | ❑ | ASI_SCRATCHPAD_7_REG | RW | $38_{16}$ | N | /core | Implementation dependent[13] | p. 98 |
| $50_{16}$ | ❍ | ASI_IMMU_... | | | | | | |
| | ❍ | ASI_IMMU_TAG_TARGET_REG | R[5] | $0_{16}$ | N | /core | IMMU tag target register | F.10.5 |
| | ❍ | ASI_IMMU_SFSR | RW | $18_{16}$ | N | /core | IMMU sync fault status register | F.10.9 |
| | ❍ | ASI_IMMU_TSB_BASE | RW | $28_{16}$ | N | /core | IMMU TSB base register | F.10.6 |
| | ❍ | ASI_IMMU_TAG_ACCESS | RW | $30_{16}$ | N | /core | IMMU TLB tag access register | F.10.4.3 |
| | ❍ | ASI_IMMU_TSB_PEXT_REG | RW | $48_{16}$ | N | /core | IMMU TSB primary extension register | F.10.7 |

**TABLE L-1** JPS2 ASIs *(5 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| | ❍ | ASI_IMMU_TSB_NEXT_REG | RW | $58_{16}$ | N | /core | IMMU TSB nucleus extension register | F.10.7 |
| $51_{16}$ | ❍ | ASI_IMMU_TSB_8KB_PTR_REG | R[5] | $0_{16}$ | N | /core | IMMU TSB 8-Kbyte pointer register | F.10.8 |
| $52_{16}$ | ❍ | ASI_IMMU_TSB_64KB_PTR_REG | R[5] | $0_{16}$ | N | /core | IMMU TSB 64-Kbyte pointer register | F.10.8 |
| $53_{16}$ | ❍ | ASI_SERIAL_ID‡ (*semantics and encoding are implementation dependent)* | R[5] | $0_{16}$[12] | N | /core | *Implementation dependent* (impl. dep. #258) [7] | — |
| $54_{16}$ | ❍ | ASI_ITLB_DATA_IN_REG | W[10] | $0_{16}$ | N | /core | IMMU TLB data in register | F.10.4.1 |
| $55_{16}$ | ❍ | ASI_ITLB_DATA_ACCESS_REG | RW | $0_{16}$–$20FF8_{16}$ | N | /core | IMMU TLB data access register | F.10.4.1 |
| $55_{16}$ | ● | *Implementation dependent* (impl. dep. #239) | | $40000_{16}$–$60FF8_{16}$ | N | — | Implementation dependent [1] | — |
| $56_{16}$ | ❍ | ASI_ITLB_TAG_READ_REG | R[5] | <17:0> | N | /core | IMMU TLB tag read register | F.10.4.2 |
| $57_{16}$ | ❍ | ASI_IMMU_DEMAP | W[10] | (decoded; see F.10.11) | N | | IMMU TLB demap | F.10.11 |
| $58_{16}$ | ❍ | ASI_DMMU_. . . | | | | | | |
| | ❍ | ASI_DMMU_TAG_TARGET_REG | R[5] | $0_{16}$ | N | /core | DMMU tag target register | F.10.5 |
| | ❍ | ASI_PRIMARY_CONTEXT_REG | RW | $8_{16}$ | N | /core | I/D MMU primary context register | F.10.2 |
| | ❍ | ASI_SECONDARY_CONTEXT_REG | RW | $10_{16}$ | N | /core | DMMU secondary context register | F.10.2 |
| | ❍ | ASI_DMMU_SFSR | RW | $18_{16}$ | N | /core | DMMU sync fault status register | F.10.9 |
| | ❍ | ASI_DMMU_SFAR | RW | $20_{16}$ | N | /core | DMMU sync fault address register | F.10.10.2 |
| | ❍ | ASI_DMMU_TSB_BASE | RW | $28_{16}$ | N | /core | DMMU TSB register | F.10.6 |
| | ❍ | ASI_DMMU_TAG_ACCESS | RW | $30_{16}$ | N | /core | DMMU TLB tag access register | F.10.4.3 |
| | ❍ | ASI_DMMU_VA_WATCHPOINT_REG | RW | $38_{16}$ | N | /core | DMMU VA data watchpoint register | p. 95 |
| | ❍ | ASI_DMMU_PA_WATCHPOINT_REG | RW | $40_{16}$ | N | /core | DMMU PA data watchpoint register | p. 95 |
| | ❍ | ASI_DMMU_TSB_PEXT_REG | RW | $48_{16}$ | N | /core | DMMU TSB primary ext register | F.10.7 |

**TABLE L-1** JPS2 ASIs *(6 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| | ❍ | ASI_DMMU_TSB_SEXT_REG | RW | $50_{16}$ | N | /core | DMMU TSB secondary ext register | F.10.7 |
| | ❍ | ASI_DMMU_TSB_NEXT_REG | RW | $58_{16}$ | N | /core | DMMU TSB nucleus ext register | F.10.7 |
| $59_{16}$ | ❍ | ASI_DMMU_TSB_8KB_PTR_REG | R[5] | $0_{16}$ | N | /core | DMMU TSB 8-K pointer register | F.10.8 |
| $5A_{16}$ | ❍ | ASI_DMMU_TSB_64KB_PTR_REG | R[5] | $0_{16}$ | N | /core | DMMU TSB 64-K pointer register | F.10.8 |
| $5B_{16}$ | ❍ | ASI_DMMU_TSB_DIRECT_PTR_REG | R[5] | $0_{16}$ | N | /core | DMMU TSB direct pointer register | F.10.8 |
| $5C_{16}$ | ❍ | ASI_DTLB_DATA_IN_REG | W[10] | $0_{16}$ | N | /core | DMMU TLB data in register | F.10.4.1 |
| $5D_{16}$ | ❍ | ASI_DTLB_DATA_ACCESS_REG | RW | VA<18>=0 | N | /core | DMMU TLB data access register | F.10.4.1 |
| $5D_{16}$ | ● | *Implementation dependent* (impl. dep. #239) | | $40000_{16}$–$60FF8_{16}$ | — | — | Implementation dependent [1] | — |
| $5E_{16}$ | ❍ | ASI_DTLB_TAG_READ_REG | R[5] | <17:0> | N | /core | DMMU TLB tag read register | F.10.4.2 |
| $5F_{16}$ | ❍ | ASI_DMMU_DEMAP | W[10] | (decoded; see F.10.11) | N | /core | DMMU TLB demap | F.10.11 |
| $60_{16}$ | ❍ | ASI_IIU_INST_TRAP | RW | $0_{16}$ | N | /core | Instruction breakpoint register | 5.2.4.3 |
| $61_{16}$–$62_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $63_{16}$ | ❑ | MTP control/status (per-core) | | | | | | |
| $63_{16}$ | ❑ | ASI_INTR_ID | RW | $00_{16}$ | N | /core | Core Interrupt ID | 9.3.3 |
| $63_{16}$ | ❑ | ASI_CORE_ID | R | $10_{16}$ | N | /core | Core ID | 9.3.1 |
| | ❑ | *Implementation dependent* | | $\geq 40_{16}$ | N | /core | Virtual addresses 0x40 and above are reserved for Implementation specific registers [1] | — |
| $64_{16}$–$6E_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $6F_{16}$ | ❍ | (reserved for ASI_BARRIER_SYNCH_P) | — | — | — | — | Implementation dependent[1] | — |
| $70_{16}$ | ❍ | ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP) | RW[2,8] | (decoded) | T | /core | Primary address space, block load/store, user privilege | L.3.2 |

**TABLE L-1** JPS2 ASIs *(7 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $71_{16}$ | ❍ | ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS) | RW[2,8] | (decoded) | T | /core | Secondary address space, block load/store, user priv | L.3.2 |
| $72_{16}$–$76_{16}$ | ● | – | | — | — | — | Implementation dependent[1] | — |
| $77_{16}$ | ❍ | ASI_INTR_DATA0_W | W[10] | $40_{16}$ | N | /core | Outgoing interrupt vector data Register 0 H | N.5.1 |
| $77_{16}$ | ❍ | ASI_INTR_DATA1_W | W[10] | $48_{16}$ | N | /core | Outgoing interrupt vector data Register 0 L | N.5.1 |
| $77_{16}$ | ❍ | ASI_INTR_DATA2_W | W[10] | $50_{16}$ | N | /core | Outgoing interrupt vector data Register 1 H | N.5.1 |
| $77_{16}$ | ❍ | ASI_INTR_DATA3_W | W[10] | $58_{16}$ | N | /core | Outgoing interrupt vector data Register 1 L | N.5.1 |
| $77_{16}$ | ❍ | ASI_INTR_DATA4_W | W[10] | $60_{16}$ | N | /core | Outgoing interrupt vector data Register 2 H | N.5.1 |
| $77_{16}$ | ❍ | ASI_INTR_DATA5_W | W[10] | $68_{16}$ | N | /core | Outgoing interrupt vector data Register 2 L | N.5.1 |
| $77_{16}$ | ❍ | ASI_INTR_DISPATCH_W | W[10] | $70_{16}$ | N | /core | Interrupt vector dispatch | N.5.2 |
| $77_{16}$ | ❑ | ASI_INTR_DISPATCH_EXT_W | W[10] | $78_{16}$ | N | /core | Interrupt vector dispatch extension register. VA<13:0>=0x78<br>VA<63:14>= Target and source information.<br>Implementation Dependent<br>(impl. dep. #1101) | N.5.3 |
| $77_{16}$ | ❍ | ASI_INTR_DATA6_W | W[10] | $80_{16}$ | N | /core | Outgoing interrupt vector data Register 3 H | N.5.1 |
| $77_{16}$ | ❍ | ASI_INTR_DATA7_W | W[10] | $88_{16}$ | N | /core | Outgoing interrupt vector data Register 3 L | N.5.1 |
| $77_{16}$ | ❍ | ASI_INTR_DISPATCH_W | W[10] | $0100000070_{16}$–$8FFFFFC070_{16}$ | N | /core | Interrupt vector dispatch | N.5.2 |
| $78_{16}$ | ❍ | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUPL) | RW [2,8] | $0_{16}$ | T | /core | Primary address space, block load/store, user privilege, little-endian | L.3.2 |
| $79_{16}$ | ❍ | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL) | RW[2,8] | $0_{16}$ | T | /core | Secondary address space, block load/store, user privilege, little-endian | L.3.2 |

**TABLE L-1** JPS2 ASIs *(8 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $7A_{16}$-$7E_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $7F_{16}$ | ❍ | ASI_INTR_DATA0_R | R[5] | $40_{16}$ | N | /core | Incoming interrupt vector data Register 0 H | N.5.3 |
| $7F_{16}$ | ❍ | ASI_INTR_DATA1_R | R[5] | $48_{16}$ | N | /core | Incoming interrupt vector data Register 0 L | N.5.3 |
| $7F_{16}$ | ❍ | ASI_INTR_DATA2_R | R[5] | $50_{16}$ | N | /core | Incoming interrupt vector data Register 1 H | N.5.5 |
| $7F_{16}$ | ❍ | ASI_INTR_DATA3_R | R[5] | $58_{16}$ | N | /core | Incoming interrupt vector data Register 1 L | N.5.5 |
| $7F_{16}$ | ❍ | ASI_INTR_DATA4_R | R[5] | $60_{16}$ | N | /core | Incoming interrupt vector data Register 2 H | N.5.5 |
| $7F_{16}$ | ❍ | ASI_INTR_DATA5_R | R[5] | $68_{16}$ | N | /core | Incoming interrupt vector data Register 2 L | N.5.5 |
| $7F_{16}$ | ❍ | ASI_INTR_DATA6_R | R[5] | $80_{16}$ | N | /core | Incoming interrupt vector data Register 3 H | N.5.5 |
| $7F_{16}$ | ❍ | ASI_INTR_DATA7_R | R[5] | $88_{16}$ | N | /core | Incoming interrupt vector data Register 3 L | N.5.5 |
| $80_{16}$ | ● | ASI_PRIMARY (ASI_P) | RW | (decoded) | T | — | Implicit primary address space | 8.3 |
| $81_{16}$ | ● | ASI_SECONDARY (ASI_S) | RW | (decoded) | T | — | Secondary address space | 8.3 |
| $82_{16}$ | ● | ASI_PRIMARY_NO_FAULT (ASI_PNF) | R[5] | (decoded) | T | — | Primary address space, no fault | 8.3 |
| $83_{16}$ | ● | ASI_SECONDARY_NO_FAULT (ASI_SNF) | R[5] | (decoded) | T | — | Secondary address space, no fault | 8.3 |
| $84_{16}$–$87_{16}$ | ● | — | — | — | — | — | Reserved | 8.3 |
| $88_{16}$ | ● | ASI_PRIMARY_LITTLE (ASI_PL) | RW | (decoded) | T | — | Implicit primary address space, little-endian | 8.3 |
| $89_{16}$ | ● | ASI_SECONDARY_LITTLE (ASI_SL) | RW | (decoded) | T | — | Secondary address space, little-endian | 8.3 |
| $8A_{16}$ | ● | ASI_PRIMARY_NO_FAULT _LITTLE (ASI_PNFL) | R[5] | (decoded) | T | — | Primary address space, no fault, little-endian | 8.3 |
| $8B_{16}$ | ● | ASI_SECONDARY_NO_FAULT _LITTLE (ASI_SNFL) | R[5] | (decoded) | T | — | Physical address, noncacheable, with side effect, little-endian | 8.3 |
| $8C_{16}$–$BF_{16}$ | ● | — | — | — | — | — | *Reserved* | — |

**TABLE L-1** JPS2 ASIs *(9 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $C0_{16}$ | ❍ | ASI_PST8_PRIMARY (ASI_PST8_P) | W[11] | (decoded) | T | — | Primary address space, 8×8-bit partial store | L.3.2 |
| $C1_{16}$ | ❍ | ASI_PST8_SECONDARY (ASI_PST8_S) | W[11] | (decoded) | T | — | Secondary address space, 8x8-bit partial store | L.3.2 |
| $C2_{16}$ | ❍ | ASI_PST16_PRIMARY (ASI_PST16_P) | W[11] | (decoded) | T | — | Primary address space, 4×16-bit partial store | L.3.2 |
| $C3_{16}$ | ❍ | ASI_PST16_SECONDARY (ASI_PST16_S) | W[11] | (decoded) | T | — | Secondary address space, 4×16-bit partial store | L.3.2 |
| $C4_{16}$ | ❍ | ASI_PST32_PRIMARY (ASI_PST32_P) | W[11] | (decoded) | T | — | Primary address space, 2x32-bit partial store | L.3.2 |
| $C5_{16}$ | ❍ | ASI_PST32_SECONDARY (ASI_PST32_S) | W[11] | (decoded) | T | — | Secondary address space, 2×32-bit partial store | L.3.2 |
| $C6_{16}$–$C7_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $C8_{16}$ | ❍ | ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL) | W[11] | (decoded) | T | — | Primary address space, 8x8-bit partial store, little-endian | L.3.2 |
| $C9_{16}$ | ❍ | ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL) | W[11] | (decoded) | T | — | Secondary address space, 8×8-bit partial store, little-endian | L.3.2 |
| $CA_{16}$ | ❍ | ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL) | W[11] | (decoded) | T | — | Primary address space, 4x16-bit partial store, little-endian | L.3.2 |
| $CB_{16}$ | ❍ | ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL) | W[11] | (decoded) | T | — | Secondary address space, 4×16-bit partial store, little-endian | L.3.2 |
| $CC_{16}$ | ❍ | ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL) | W[11] | (decoded) | T | — | Primary address space, 2×32-bit partial store, little-endian | L.3.2 |
| $CD_{16}$ | ❍ | ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL) | W[11] | (decoded) | T | — | Second address space, 2×32-bit partial store, little-endian | L.3.2 |
| $CE_{16}$–$CF_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $D0_{16}$ | ❍ | ASI_FL8_PRIMARY (ASI_FL8_P) | RW[8] | (decoded) | T | — | Primary address space, one 8-bit floating-point load / store | L.3.2 |

**TABLE L-1** JPS2 ASIs *(10 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $D1_{16}$ | ❍ | ASI_FL8_SECONDARY (ASI_FL8_S) | $RW^{8}$ | (decoded) | T | — | Second address space, one 8-bit floating-point load / store | L.3.2 |
| $D2_{16}$ | ❍ | ASI_FL16_PRIMARY (ASI_FL16_P) | $RW^{8}$ | (decoded) | T | — | Primary address space, one 16-bit floating-point load / store | L.3.2 |
| $D3_{16}$ | ❍ | ASI_FL16_SECONDARY (ASI_FL16_S) | $RW^{8}$ | (decoded) | T | — | Second address space, one 16-bit floating-point load / store | L.3.2 |
| $D4_{16}$–$D7_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $D8_{16}$ | ❍ | ASI_FL8_PRIMARY_LITTLE (ASI_FL8_PL) | $RW^{8}$ | (decoded) | T | — | Primary address space, one 8-bit floating point load / store, little-endian | L.3.2 |
| $D9_{16}$ | ❍ | ASI_FL8_SECONDARY_ LITTLE (ASI_FL8_SL) | $RW^{8}$ | (decoded) | T | — | Second address space, one 8-bit floating point load / store, little-endian | L.3.2 |
| $DA_{16}$ | ❍ | ASI_FL16_PRIMARY_ LITTLE (ASI_FL16_PL) | $RW^{8}$ | (decoded) | T | — | Primary address space, one 16-bit floating-point load / store, little-endian | L.3.2 |
| $DB_{16}$ | ❍ | ASI_FL16_SECONDARY_ LITTLE (ASI_FL16_SL) | $RW^{8}$ | (decoded) | T | — | Second address space, one 16-bit floating point load / store, little-endian | L.3.2 |
| $DC_{16}$–$DF_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $E0_{16}$ | ❍ | ASI_BLOCK_COMMIT_ PRIMARY (ASI_BLK_COMMIT_P) | $W^{6,11}$ | (decoded) | T | — | Primary address space, 8x8- byte block store commit operation | L.3.2 |
| $E1_{16}$ | ❍ | ASI_BLOCK_COMMIT_ SECONDARY (ASI_BLK_COMMIT_S) | $W^{6,11}$ | (decoded) | T | — | Secondary address space, 8x8-byte block store commit operation | L.3.2 |
| $E2_{16}$–$EE_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |
| $EF_{16}$ | ❍ | (reserved for ASI_BARRIER_SYNCH) | — | — | — | — | Implementation dependent[1] | — |
| $F0_{16}$ | ❍ | ASI_BLOCK_PRIMARY (ASI_BLK_P) | $RW^{8}$ | (decoded) | T | — | Primary address space, 8x8- byte block load / store | A.4 |
| $F1_{16}$ | ❍ | ASI_BLOCK_SECONDARY (ASI_BLK_S) | $RW^{8}$ | (decoded) | T | — | Secondary address space, block load / store | A.4 |
| $F2_{16}$–$F7_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |

**TABLE L-1** JPS2 ASIs *(11 of 11)*

| ASI Value | V9 (●); JPS1 (❍); JPS2 (❑) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | T/ Non-T/ Bypass | Shared /Per-Core | Description | Sec'n or Pg |
|---|---|---|---|---|---|---|---|---|
| $F8_{16}$ | ❍ | ASI_BLOCK_PRIMARY_ LITTLE (ASI_BLK_PL) | RW[8] | (decoded) | T | — | Primary address space, block load / store, little endian | L.3.2 |
| $F9_{16}$ | ❍ | ASI_BLOCK_SECONDARY_ LITTLE (ASI_BLK_SL) | RW[8] | (decoded) | T | — | Secondary address space, block load / store, little endian | L.3.2 |
| $FA_{16}$–$FF_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] | — |

† This ASI name has been changed, for consistency; although use of this name is deprecated and software should use the new name, the old name is listed here for compatibility

‡ This ASI was named ASI_DEVICE_ID+SERIAL_ID in older documents

1 Implementation dependent ASI (impl. dep. #29); available for use by implementors. Software that references this ASI may not be portable.

2 Causes a *data_access_exception* trap if the page being accessed is privileged.

3 8-bit, 16-bit, 32-bit, and 64-bit accesses are allowed.

4 Can be used with LDSTUBA, SWAPA, CAS(X)A.

5 Read (load)-only ASI; an attempted store or atomic load-store to this ASI causes a *data_access_exception* exception.

6 May only be used in an STDDA instruction. Use of this ASI in any other store instruction causes a *data_access_exception*.

7 Implementation dependent ASI (impl. dep. #258), intended for use for a part identification number that is unique to each virtual processor. Software that references this ASI may not be portable.

8 May only be used in a LDDFA or STDFA instruction. Use in any other load or store instruction causes a *data_access_exception* exception.

9 May only be used in an LDDA instruction. Use of this ASI in any other load instruction causes a *data_access_exception*.

10 Write(store)-only ASI; a load from this ASI causes a *data_access_exception*.

11 Write(store)-only ASI; a load from this ASI causes an exception (see section L.3.2 for details)

12 An implementation may or may not decode the virtual address when this ASI is accessed, but for future compatibility software should always supply VA = 0.

13 Implementation-dependent ASI (impl. dep. #302). The ASI is reserved exclusively for this use, but may not be present in all implementations. Software that references this ASI may not be portable.

14 **IMPL. DEP. #305:** The effect of writing to AFSR and AFAR is implementation dependent.

## L.3.2 Special Memory Access ASIs

This section describes special memory access ASIs that are not specified in SPARC V9 and are not described in other sections.

### L.3.2.1 ASI $14_{16}$ (ASI_PHYS_USE_EC)

When this ASI is specified in any memory access instructions, hardware does the following:

- VA is passed through to PA, and context values are disregarded.
- Address masking is ignored (PSTATE.am; see *PSTATE_address_mask (AM)* on page 76); the va is used. The va passed through pa is implementation dependent (impl. dep. #224).
- Memory access behaves as if its byte order is big-endian.

Even if data address translation is disabled, the access with this ASI is still a cacheable access.

### L.3.2.2 ASI $15_{16}$ (ASI_PHYS_BYPASS_EC_WITH_EBIT)

Accesses with this ASI bypass the external cache and behave as if the side effect bit (e bit) is set. When this ASI is specified in any memory access instructions, hardware does the following:

- va is passed through to pa, and context values are disregarded.
- Address masking is ignored (PSTATE.am; see *PSTATE_address_mask (AM)* on page 76); the va is used. The va passed through pa is implementation dependent (impl. dep. #224).
- Memory access behaves as if its byte order is big-endian.

### L.3.2.3 ASI $1C_{16}$ (ASI_PHYS_USE_EC_LITTLE)

Accesses with this ASI are cacheable. This ASI is a little-endian version of ASI $14_{16}$. When this ASI is specified in any memory access instructions, hardware does the following:

- va is passed through to pa, and context values are disregarded.
- Address masking is ignored (PSTATE.am; see *PSTATE_address_mask (AM)* on page 76); the va is used. The va passed through pa is implementation dependent (impl. dep. #224).
- Memory access behaves as if its byte order is little-endian.

### L.3.2.4 ASI $1D_{16}$ (ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE)

Accesses with this ASI bypass the external cache and behave as if the side effect bit (e bit) is set. This ASI is a little-endian version of ASI $15_{16}$. When this ASI is specified in any memory access instructions, hardware does the following:

- va is passed through to pa, and context values are disregarded.
- Address masking is ignored (PSTATE.am; see *PSTATE_address_mask (AM)* on page 76); the va is used. The va passed through pa is implementation dependent (impl. dep. #224).
- Memory access behaves as if its byte order is little-endian.

### L.3.2.5 ASIs $24_{16}$, $2C_{16}$, $34_{16}$, and $3C_{16}$ (Load Quadword Atomic)

ASIs $24_{16}$ (ASI_NUCLEUS_QUAD_LDD), $2C_{16}$ (ASI_NUCLEUS_QUAD_LDD_LITTLE), $34_{16}$ (ASI_QUAD_LDD_PHYS), and $3C_{16}$ (ASI_QUAD_LDD_PHYS_LITTLE) exist for use with the LDDA instruction as Load Quadword Atomic operations (see A.30 on page 283). These four ASIs are distinguished by the type of addressing used (virtual or physical) and whether the access is performed using implicit-endian or little-endian byte ordering (see *PSTATE_current_little_endian (CLE)* on page 74), as illustrated in the following table:

**TABLE L-2** Load Quadword Atomic ASIs

| | ASI | |
|---|---|---|
| | **Virtual Addressing** | **Physical Addressing** |
| **Implicit Endianness (Big when PSTATE.cle = 0, Little when PSTATE.cle = 1)** | $24_{16}$ | $34_{16}$ |
| **Explicit Little Endian access** | $2C_{16}$ | $3C_{16}$ |

When these ASIs are used with LDDA for Load Quadword Atomic, a *mem_address_not_aligned* exception is generated if the operand address is not 16-byte aligned.

If these ASIs are used with any other Load Alternate instruction or any Store Alternate instruction, a *data_access_exception* is always generated and *mem_address_not_aligned* is not generated.

### L.3.2.6 ASI $60_{16}$ (ASI_IIU_INST_TRAP)

See *Instruction Trap Register* on page 96 for a description of ASI $60_{16}$ and its uses.

### L.3.2.7 Block Load and Store ASIs

ASIs $70_{16}$, $71_{16}$, $78_{16}$, $79_{16}$, $E0_{16}$, $E1_{16}$, $F0_{16}$, $F1_{16}$, $F8_{16}$, and $F9_{16}$ exist for use with LDDFA and STDFA instructions as Block Load and Block Store operations (see A.4 on page 231).

When these ASIs are used with LDDFA (STDFA) for Block Load (Store), a *mem_address_not_aligned* exception is generated if the operand address is not 64-byte aligned.

ASIs $E0_{16}$ and $E1_{16}$ are only defined for use in Block Store with Commit operations (using the STDFA opcode). Neither ASI $E0_{16}$ nor $E1_{16}$ should be used with LDDFA; however, if it is, the following behavior occurs:

1. **IMPL. DEP. #255:** For LDDFA with ASI $E0_{16}$ or $E1_{16}$, if a destination register number rd is specified which is not a multiple of 8 ("misaligned" rd), it is implementation dependent whether the virtual processor generates a *data_access_exception* or *illegal_instruction* exception.
2. **IMPL. DEP. #256:** For LDDFA with ASI $E0_{16}$ or $E1_{16}$, if a memory address is specified with less than 64-byte alignment, it is implementation dependent whether the virtual processor generates a *data_access_exception*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned* exception.
3. If both rd and the memory address are correctly aligned, a *data_access_exception* occurs.

If a Block Load or Block Store ASI is used with any other Load Alternate or Store Alternate instruction, a *data_access_exception* exception is always generated and *mem_address_not_aligned* is not generated.

### L.3.2.8 Partial Store ASIs

ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–$CD_{16}$ exist for use with the STDFA instruction as Partial Store operations (see A.42 on page 313).

When these ASIs are used with STDFA for Partial Store, a *mem_address_not_aligned* exception is generated if the operand address is not 8-byte aligned and an *illegal_instruction* exception is generated if i = 1 in the instruction.

If one of these ASIs is used with a Store Alternate instruction other than STDFA or with a Load Alternate instruction other than LDDFA, a *data_access_exception* exception is generated and *mem_address_not_aligned*, *LDDF_mem_address_not_aligned*, and *illegal_instruction* (for i = 1) are not generated.

None of these Partial Store ASIs should be used with LDDFA; however, if any of ASIs $C0_{16}$–$C5_{16}$ or $C8_{16}$–$CD_{16}$ is used with LDDFA, the LDDFA behaves as follows:

1. **IMPL. DEP. #257:** For LDDFA with ASI $C0_{16}$–$C5_{16}$ or $C8_{16}$–$CD_{16}$, if a memory address is specified with less than 8-byte alignment, it is implementation dependent whether the virtual processor generates a *data_access_exception*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned* exception.

2. If the memory address is correctly aligned, the virtual processor generates a *data_access_exception*.

### L.3.2.9 Short Floating-Point Load and Store ASIs

ASIs $D0_{16}$–$D3_{16}$ and $D8_{16}$–$DB_{16}$ exist for use with the LDDFA and STDFA instructions as Short Floating-point Load and Store operations (see A.58 on page 356).

When ASI $D2_{16}$, $D3_{16}$, $DA_{16}$, or $DB_{16}$ is used with LDDFA (STDFA) for a 16-bit Short Floating-point Load (Store), a *mem_address_not_aligned* exception is generated if the operand address is not halfword-aligned.

If any of these ASIs are used with any other Load or Store Alternate instruction, a *data_access_exception* is always generated and *mem_address_not_aligned* is not generated.

# Caches and Cache Coherency

For implementation-dependent caches and cache coherency information, please refer to Appendix M in specific SPARC JPS2 Extension documents.

# Interrupt Handling

Virtual processors and I/O devices can interrupt a selected virtual processor by assembling and sending an interrupt packet consisting of eight 64-bit words of interrupt vector data. The contents of these data are defined by software convention. Thus, hardware interrupts and cross-calls can have the same hardware mechanism for interrupt delivery and can share a common software interface for processing.

The interrupt requesting/receiving mechanism is a two-step process: the sending of an interrupt request on an interrupt vector data register to the target virtual processor and the scheduling of the received interrupt request on the target virtual processor upon receipt.

An interrupt request packet is sent by the interrupter through the interrupt vector dispatch mechanism and is received by the specified target through the interrupt vector receive mechanism. Upon receipt of an interrupt request packet, a special trap is invoked on the target virtual processor. The trap handler software invoked in the target virtual processor then schedules the interrupt request to itself by posting the interrupt into `SOFTINT` register at the desired interrupt level.

Note that a virtual processor may not send an interrupt request packet to itself through the interrupt dispatch mechanism. However, a virtual processor may send an interrupt to another virtual processor in the same processor. Separate sets of dispatch (outgoing) and receive (incoming) interrupt data registers allow simultaneous interrupt dispatching and receiving.

In the following sections, we describe these aspects of interrupt handling:


- *Interrupt Vector Dispatch* on page 588
- *Interrupt Vector Receive* on page 589
- *Interrupt Global Registers* on page 590
- *Interrupt ASI Registers* on page 590
- *Software Interrupt Register (SOFTINT)* on page 594

# N.1 Core Interrupt ID Register (ASI_INTR_ID)

This per-core register allows the software to assign a 16-bit interrupt ID to a virtual processor that is unique within the system. This is important to enable virtual processors to receive interrupts. See *Register Specification* on page 191 for details.

# N.2 Interrupt Vector Dispatch

To dispatch an interrupt or cross-call, a core or I/O device first writes to the Outgoing Interrupt Vector Data registers according to an established software convention, described below. A subsequent write to the Interrupt Vector Dispatch register (see *Interrupt Vector Dispatch Register* on page 591) triggers the interrupt delivery. The status of the interrupt dispatch can be read by polling the ASI_INTR_DISPATCH_STATUS's busy and nack bits. A MEMBAR #Sync should be used before polling begins to ensure that earlier stores are completed. If both NACK and BUSY are cleared, the interrupt has been successfully delivered to the target core. With the nack bit cleared and busy bit set, the interrupt delivery is pending. Finally, if the delivery cannot be completed (if it is rejected by the target core), the nack bit is set. The pseudocode sequence in CODE EXAMPLE N-1 sends an interrupt.

The ASI_INTR_DISPATCH_STATUS register contains 32 pairs of busy/nack bit pairs, enabling interrupts to be pipelined. Specifying a unique pair of busy/nack bits to be used for each interrupt when writing the Interrupt Dispatch register enables up to 32 interrupts to be outstanding at one time.

---

**Note –** A virtual processor may not send an interrupt vector to itself through outgoing interrupt vector data registers. Doing so causes undefined interrupt vector data to be returned.

---

**CODE EXAMPLE N-1** Code Sequence for Interrupt Dispatch

```
Read state of ASI_INTR_DISPATCH_STATUS; Error if BUSY
<no pending interrupt dispatch packet>
Repeat
    Begin atomic sequence(PSTATE.IE ← 0)
    Store to IV data reg 0 at ASI_INTR_W, VA=0x40 (optional)
```

**CODE EXAMPLE N-1** Code Sequence for Interrupt Dispatch *(Continued)*

```
Store to IV data reg 1 at ASI_INTR_W, VA=0x48 (optional)
Store to IV data reg 2 at ASI_INTR_W, VA=0x50 (optional)
Store to IV data reg 3 at ASI_INTR_W, VA=0x58 (optional)
Store to IV data reg 4 at ASI_INTR_W, VA=0x60 (optional)
Store to IV data reg 5 at ASI_INTR_W, VA=0x68 (optional)
Store to IV data reg 6 at ASI_INTR_W, VA=0x80 (optional)
Store to IV data reg 7 at ASI_INTR_W, VA=0x88 (optional)
Store to IV dispatch at ASI_INTR_W, VA<63:29>=0,
    VA<28:24>=BUSY/NACK bit #,VA<23:14>=ITID,
    VA<13:0>=0x70[1] initiates interrupt delivery
Membar #Sync (wait for stores to finish)
Poll state of ASI_INTR_DISPATCH_STATUS (BUSY, NACK)
    Loop if BUSY
End atomic sequence(PSTATE.IE ← 1)
```

1. If a JPS2 processor implements the dispatch extension register (ASI_INTR_DISPACH_EXT_W), using VA = $78_{16}$ is recommended; using VA = $70_{16}$ is also valid.

**Note –** To avoid deadlocks, enable interrupts for some period before retrying the atomic sequence. Alternatively, implement the atomic sequence with locks without disabling interrupts.

# N.3 Interrupt Vector Receive

When an interrupt is received, all eight Interrupt Data registers are updated, regardless of which are being used by software. This update is done in conjunction with the setting of the BUSY bit in the ASI_INTR_RECEIVE register. At this point, the virtual processor inhibits further interrupt packets from the system bus. If interrupts are enabled (PSTATE.IE = 1), then an *interrupt_vector* trap (trap type $60_{16}$) is generated. Software reads the ASI_INTR_RECEIVE register and Incoming Interrupt Data registers to determine the entry point of the appropriate trap handler. All of the external interrupt packets are processed at the highest interrupt priority level and are then reprioritized as lower-priority interrupts in the software handler. CODE EXAMPLE N-2 illustrates interrupt receive handling.

**CODE EXAMPLE N-2** Code Sequence for an Interrupt Receive

```
Read state of ASI_INTR_RECEIVE; Error if !BUSY
Read from IV data reg 0 at ASI_SDB_INTR_R, VA=0x40 (optional)
Read from IV data reg 1 at ASI_SDB_INTR_R, VA=0x48 (optional)
Read from IV data reg 2 at ASI_SDB_INTR_R, VA=0x50 (optional)
```

**CODE EXAMPLE N-2** Code Sequence for an Interrupt Receive *(Continued)*

```
Read from IV data reg 3 at ASI_SDB_INTR_R, VA=0x58 (optional)
Read from IV data reg 4 at ASI_SDB_INTR_R, VA=0x60 (optional)
Read from IV data reg 5 at ASI_SDB_INTR_R, VA=0x68 (optional)
Read from IV data reg 6 at ASI_SDB_INTR_R, VA=0x80 (optional)
Read from IV data reg 7 at ASI_SDB_INTR_R, VA=0x88 (optional)
Determine the appropriate handler
Handle interrupt or reprioritize this trap and
    set the SOFTINT register
```

# N.4 Interrupt Global Registers

A separate set of global registers is implemented to expedite interrupt processing. As described in *Interrupt Vector Receive*, above, a virtual processor takes an implementation-dependent interrupt trap after receiving an interrupt packet. Software uses a number of scratch registers while determining the appropriate handler and constructing the interrupt state.

A separate set of eight Interrupt Global registers replaces the eight programmer-visible global registers during interrupt processing. After an interrupt trap is dispatched, the hardware selects the interrupt global registers by setting the `PSTATE.ig` field to 1 (and zeroing both `PSTATE.ag` and `PSTATE.mg`). The `PSTATE` extension is described in *Processor State (PSTATE) Register* on page 72. The previous value of `PSTATE` is restored from the trap stack by a `DONE` or `RETRY` instruction on exit from the interrupt handler.

# N.5 Interrupt ASI Registers

`MEMBAR #Sync` is generally needed after stores to interrupt ASI registers to avoid unnecessary effects caused by possible prefetches to the locations with side effect. For examples, see Section 8.9.1, *Instruction Prefetch to Side-Effect Locations*, in the JPS2 Extension documents to this specification.

## N.5.1 Outgoing Interrupt Vector Data<7:0> Register

`ASI_INTR_W` (data 0): ASI = $77_{16}$, VA<63:0> = $40_{16}$
`ASI_INTR_W` (data 1): ASI = $77_{16}$, VA<63:0> = $48_{16}$
`ASI_INTR_W` (data 2): ASI = $77_{16}$, VA<63:0> = $50_{16}$

ASI_INTR_W (data 3): ASI = $77_{16}$, VA<63:0> = $58_{16}$
ASI_INTR_W (data 4): ASI = $77_{16}$, VA<63:0> = $60_{16}$
ASI_INTR_W (data 5): ASI = $77_{16}$, VA<63:0> = $68_{16}$
ASI_INTR_W (data 6): ASI = $77_{16}$, VA<63:0> = $80_{16}$
ASI_INTR_W (data 7): ASI = $77_{16}$, VA<63:0> = $88_{16}$

Name: ASI_INTR_W: Outgoing Interrupt Vector Data registers (privileged, write-only).

TABLE N-1 describes the register field of the eight Outgoing Interrupt Vector Data registers.

**TABLE N-1** Outgoing Interrupt Vector Data Register Format

| Bits | Field | Type | Description |
|---|---|---|---|
| 63:0 | **data** | W | Interrupt data. |

Nonprivileged access to this register causes a *privileged_action* trap. An attempt to read this register causes any reasonable trap.

## N.5.2 Interrupt Vector Dispatch Register

ASI $77_{16}$

VA<63:39> = 0
VA<38:29> = sid<9:0> (see impl. dep. #246)
VA<28:24> = BUSY / NACK bit pair # (bn),
VA<23:14> = interrupt target identifier (itid),
VA<13:0> = $70_{16}$

Name: ASI_INTR_DISPATCH_W (interrupt dispatch) (Privileged, write-only)

TABLE N-2 describes the components of the virtual addresses for accessing the Interrupt Vector Dispatch register.

**TABLE N-2** Interrupt Vector Dispatch Register Address Field

| Bits | Field | Type | Description |
|---|---|---|---|
| VA<28:24> | **bn** | W | Specifies which of the BUSY / NACK bit pairs to use for the interrupt. $0_{16}$ in this field (which current software is using) selects BUSY / NACK bits ASI_INTR_DISPATCH_STATUS<1:0> for backward compatibility. $1_{16}$ in this field selects BUSY / NACK bits ASI_INTR_DISPATCH_STATUS<3:2>. |
| VA<23:14> | **itid** | W | Interrupt Target ID. Specifies the interrupt target core using the BUSY / NACK bit pair bn, along with the contents of the eight Interrupt Vector Data registers. |

The Interrupt Vector Dispatch register conveys information about sending an interrupt. The data written is ignored.

A write to this ASI triggers an interrupt vector dispatch to the target core identified with ITID (Interrupt Target ID), using BUSY/NACK bit pair bn along with the contents of the eight Interrupt Vector Data registers.

**IMPL. DEP. #246:** When the Interrupt Vector Dispatch register is written, the source module identifier (SID) is supplied in va<38:29>. Which, if any, of the ten va<38:29> bits are interpreted by hardware is implementation dependent.

A read from the Interrupt Vector Dispatch register causes a *data_access_exception* trap. Nonprivileged access to this register causes a *privileged_action* trap.

## N.5.3 Interrupt Vector Dispatch Extension Register

The Interrupt Vector Dispatch Extension register is an implementation dependent alternative register to the Interrupt Vector Dispatch register. The reason for this new register is to increase the Interrupt ID from ten bits in legacy (e.g., SPARC JPS1) implementations to 16 bits in MTP virtual processors. The existing Interrupt Vector Dispatch register could also be preserved, for software compatibility. See *Interrupt Vector Dispatch Extension Register (ASI_INTR_DISPATCH_EXT_W)* on page 194 for details.

## N.5.4 Interrupt Vector Dispatch Status Register

ASI $48_{16}$

VA<63:0> = 0

Name: ASI_INTR_DISPATCH_STATUS (Privileged, read-only)

TABLE N-3 describes the fields of the Interrupt Vector Dispatch Status register.

**TABLE N-3** Interrupt Dispatch Status Register Format

| Bits | Field | Type | Description |
|---|---|---|---|
| odd | nack | R | Set if interrupt dispatch has failed. Cleared (set to 0) at the start of every interrupt dispatch to the specified bit number; set when a dispatch has failed. |
| even | busy | R | Set when there is an outstanding dispatch. Cleared (set to 0) when the corresponding interrupt is either successfully delivered or rejected with a NACK. |

**IMPL. DEP. #243**: The number of busy/nack bit pairs implemented in the Interrupt Vector Dispatch Status register is implementation dependent.

The status of up to 32 outgoing interrupts can be read from ASI_INTR_DISPATCH_STATUS BUSY/NACK bits. This register contains up to 32 pairs of BUSY/NACK bit pairs: the pair at <1:0> is referred to as pair 0, <3:2> as pair 1, and so on up to pair 31 at bits <63:62>. The va<28:24> field of the Interrupt Dispatch register specifies which BUSY/NACK bit pair will be used for the interrupt.

Writes to this ASI cause a *data_access_exception* trap. Nonprivileged access to this register causes a *privileged_action* trap.

## N.5.5 Incoming Interrupt Vector Data<7:0>

ASI_INTR_R (data 0): ASI = $7F_{16}$, VA<63:0> = $40_{16}$
ASI_INTR_R (data 1): ASI = $7F_{16}$, VA<63:0> = $48_{16}$
ASI_INTR_R (data 2): ASI = $7F_{16}$, VA<63:0> = $50_{16}$
ASI_INTR_R (data 3): ASI = $7F_{16}$, VA<63:0> = $58_{16}$
ASI_INTR_R (data 4): ASI = $7F_{16}$, VA<63:0> = $60_{16}$
ASI_INTR_R (data 5): ASI = $7F_{16}$, VA<63:0> = $68_{16}$
ASI_INTR_R (data 6): ASI = $7F_{16}$, VA<63:0> = $80_{16}$
ASI_INTR_R (data 7): ASI = $7F_{16}$, VA<63:0> = $88_{16}$

Name: ASI_INTR_R

TABLE N-4 describes the register field of the eight Incoming Interrupt Vector Data registers.

**TABLE N-4** Incoming Interrupt Vector Data Register Format

| Bits | Field | Type | Use — Description |
|---|---|---|---|
| 63:0 | Data | R | Interrupt data. |

A read from these registers returns incoming interrupt information from the incoming Interrupt Receive Data registers.

Nonprivileged access to this register causes a *privileged_action* trap.

## N.5.6 Interrupt Vector Receive Register

ASI $49_{16}$

VA<63:0> = 0

Name: ASI_INTR_RECEIVE (Privileged)

TABLE N-5 describes the fields of the Interrupt Receive register.

**TABLE N-5** Interrupt Receive Register Format

| Bits | Field | Type | Description |
|---|---|---|---|
| 63:11 | | R | Reserved. |
| 16:6 | **sid_u** | R | Most significant (Upper) 11 bits of the physical module ID (mid) of the interrupter. Source ID bits <15:5> of interrupter. An implementation may use a subset of these bits (impl. dep. #247). |
| 5 | **busy** | RW | Set when an interrupt vector is received. The BUSY bit must be cleared by software writing 0. |
| 4:0 | **sid_l** | R | Least significant (Lower) 5 bits of the physical module ID (mid) of the interrupter. |

The status of an incoming interrupt can be read from ASI_INTR_RECEIVE. The busy bit is cleared by writing 0 to this register.

**IMPL. DEP. #247:** Which, if any, of the physical module ID (mid) bits of the interrupt source is set by hardware in the sid_u and sid_l fields of the Interrupt Vector Receive register is implementation dependent. Also, the source of the physical module ID (mid) bits is implementation dependent.

Nonprivileged access to the Interrupt Vector Receive register causes a *privileged_action* trap.

# N.6 Software Interrupt Register (SOFTINT)

To schedule interrupt vectors for processing at a later time, each virtual processor can send itself signals by setting bits in the SOFTINT register.

TABLE 5-15 on page 89 describes the fields of the SOFTINT register.

The SOFTINT register (ASR $16_{16}$) is used for communication from nucleus (TL > 0) code to kernel (TL = 0) code. Interrupt packets and other service requests can be scheduled in queues or mailboxes in memory by the nucleus, which then sets SOFTINT<*n*> to cause an interrupt at level *n*.

Nonprivileged access to this register causes a *privileged_opcode* trap.

## N.6.1 Setting the Software Interrupt Register

Setting SOFTINT<*n*> is done by a write to the SET_SOFTINT register (ASR $14_{16}$), with bit *n* corresponding to the interrupt level set. The value written to the SET_SOFTINT register is effectively ORed into the SOFTINT register. This approach allows the interrupt handler to set one or more bits in the SOFTINT register with a single instruction.

Read accesses to the SET_SOFTINT register cause an *illegal_instruction* trap. Nonprivileged accesses to this register cause a *privileged_opcode* trap.

When the nucleus returns, if (PSTATE.ie = 1) and (*n* > PIL), then the virtual processor will receive the highest-priority interrupt of the asserted bits in SOFTINT<16:0>. The virtual processor then takes a trap for the interrupt request, and the nucleus sets the return state to the interrupt handler at that PIL and returns to TL = 0. In this manner, the nucleus can schedule services at various priorities and process them according to their priority.

## N.6.2 Clearing the Software Interrupt Register

When all interrupts scheduled for service at level *n* have been serviced, the kernel writes a 1 to bit *n* of the CLEAR_SOFTINT pseudo-register (ASR $15_{16}$) to clear that interrupt. The complement of the value written to the CLEAR_SOFTINT register is effectively ANDed with the SOFTINT register. This approach allows the interrupt handler to clear one or more bits in the SOFTINT register with a single instruction.

An attempt to read the CLEAR_SOFTINT register causes an *illegal_instruction* trap. Nonprivileged write accesses to this register cause a *privileged_opcode* trap.

The timer interrupt TICK_INT and system timer interrupt STICK_INT are equivalent to SOFTINT<14> and have the same effect.

**Programming Note –** To avoid a race condition between the kernel clearing an interrupt and the nucleus setting it, the kernel should examine the queue for any valid entries again after clearing the interrupt bit.

TABLE N-6 summarizes the SOFTINT ASRs.

**TABLE N-6** SOFTINT ASRs

| ASR Value | ASR Name | Type | Description |
|---|---|---|---|
| $14_{16}$ | SET_SOFTINT | W | Set bit(s) in Soft Interrupt register. |
| $15_{16}$ | CLEAR_SOFTINT | W | Clear bit(s) in Soft Interrupt register. |
| $16_{16}$ | SOFTINT | RW | Per-core Soft Interrupt register. |

# Reset, RED_state, and error_state

RED_state (Reset, Error, and Debug state) is a restricted execution state reserved for processing traps that occur when TL = MAXTL – 1 and for processing hardware- and software-initiated resets.

This chapter examines RED_state in the following sections:


- *RED_state Characteristics* on page 597
- *Resets* on page 598
- *RED_state Trap Vector* on page 599
- *Machine States* on page 600


## O.1 RED_state Characteristics

A reset or trap that sets PSTATE.red (including a trap in RED_state) will clear the DCU Control Register, including the enable bits for I-cache, D-cache, IMMU, DMMU, and virtual and physical watchpoints. The characteristics of RED_state include the following:

- The default access in RED_state is noncacheable, so there must be noncacheable scratch memory somewhere in the system.
- The D-cache, watchpoints, and DMMU can be enabled by software in RED_state, but any trap will disable them again.
- The IMMU and consequently the I-cache are always disabled in RED_state. Disabling overrides the enable bits in the DCU control register.
- When PSTATE.red is explicitly set by a software write, there are no side effects other than disabling the IMMU. Software must create the appropriate state itself.

- A trap when TL = MAXTL – 1 immediately brings the virtual processor into RED_state. In addition, trap when TL = MAXTL immediately brings the virtual processor into error_state. Recovery from error_state, regardless of the means (impl. dep. #254), returns the virtual processor to RED_state.
- Any trap when TL = MAXTL will cause the virtual processor to enter error_state.
- A SIR instruction generates an SIR trap on the virtual processor.
- A trap to software-initiated reset causes an SIR trap on the virtual processor and brings the virtual processor into RED_state.
- The External Reset pin generates an XIR trap, which is used for system debug.
- The caches continue to snoop and maintain coherence if DMA or other virtual processors are still issuing cacheable accesses.

---

**Note –** Exiting RED_state by writing 0 to PSTATE.red in the delay slot of a JMPL is not recommended. A noncacheable instruction prefetch can be made to the JMPL target, which may be in a cacheable memory area. This condition could result in a bus error on some systems and cause an instruction access error trap. Exiting RED_state with DONE or RETRY avoids the problem.

---

# O.2 Resets

Reset priorities from highest to lowest are hard power-on resets, system reset, externally initiated reset (XIR), watchdog reset (WDR), and software-initiated reset (SIR).

Please refer to *Reset Traps* on page 136 and *RED_state Trap Processing* on page 153. See also Section O.2 in JPS2 Extension documents for implementation-specific details.

## O.2.1 Externally Initiated Reset (XIR)

An externally initiated reset (XIR) is sent to the virtual processor through an external hardware pin. It causes a SPARC V9 XIR, which has a trap type $3_{16}$ at physical address offset $60_{16}$. XIR has higher priority than all other resets except hard POR and soft POR.

**IMPL. DEP. #304:** Whether XIR affects only one virtual processor or the entire system is implementation-dependent.

Memory state, cache state, and most Control Status Register state (see TABLE O-1) are unchanged. System coherency is *not* guaranteed to be maintained through an XIR reset. The saved PC and nPC will only be approximate because the trap is not precise with respect to pipeline state. An XIR will reset internal pipeline state machines to free a hardware pipeline hang condition.

### O.2.2 error_state and Watchdog Reset (WDR)

A SPARC JPS2 virtual processor enters error_state when a trap occurs at TL = MAXTL.

If the virtual processor automatically exits error_state using WDR (impl. dep. #254), the virtual processor signals itself internally to take a watchdog reset (WDR) and sets TT = 2. The WDR traps at physical address offset $40_{16}$.

WDR affects only one virtual processor, rather than the entire system. CWP updates due to window traps that cause watchdog traps are the same as the no watchdog trap case.

### O.2.3 Software-Initiated Reset (SIR)

A software-initiated reset is initiated by a SIR instruction within any virtual processor. This per-virtual processor reset has a trap type 4 at physical address offset $80_{16}$. SIR affects only one virtual processor, rather than the entire system.

## O.3 RED_state Trap Vector

When a SPARC V9 processor processes a reset or trap that enters RED_state, it takes a trap at an offset relative to the RED_state_trap_vector base address (RSTVaddr).

- RSTVaddr is fixed at virtual address FFFF FFFF F000 $0000_{16}$.
- RSTVaddr passes through to an implementation-dependent physical address (impl. dep. #114).

# O.4 Machine States

TABLE O-1 shows the machine states created as a result of any reset or when RED_state is entered.

**TABLE O-1** Machine State After Reset and When Entering RED_state *(1 of 4)*

| Name | Fields | Hard_POR | System Reset | WDR | XIR | SIR | RED_state‡ |
|---|---|---|---|---|---|---|---|
| Integer registers | | Undefined | Unchanged | Unchanged | | | |
| Floating-point registers | | Undefined | Unchanged | Unchanged | | | |
| RSTVaddr | | VA = FFFF FFFF F000 $0000_{16}$<br>PA = *implementation dependent* (impl. dep. #114) | | | | | |
| PC<br>nPC | | RSTV \| $20_{16}$<br>RSTV \| $24_{16}$ | RSTV \| $20_{16}$<br>RSTV \| $24_{16}$ | RSTV \| $40_{16}$<br>RSTV \| $44_{16}$ | RSTV \| $60_{16}$<br>RSTV \| $64_{16}$ | RSTV \| $80_{16}$<br>RSTV \| $84_{16}$ | RSTV \| $A0_{16}$<br>RSTV \| $A4_{16}$ |
| PSTATE | mm | 0 (TSO) | 0 (TSO) | 0 (TSO) | | | |
| | red | 1 (RED_state) | 1 (RED_state) | 1 (RED_state) | | | |
| | pef | 1 (FPU on) | 1 (FPU on) | 1 (FPU on) | | | |
| | am | 0 (Full 64-bit address) | 0 (Full 64-bit address) | 0 (Full 64-bit address) | | | |
| | priv | 1 (Privileged mode) | 1 (Privileged mode) | 1 (Privileged mode) | | | |
| | ie | 0 (Disable interrupts) | 0 (Disable interrupts) | 0 (Disable interrupts) | | | |
| | ag | 1 (Alternate globals selected) | 1 (Alternate globals selected) | 1 (Alternate globals selected) | | | |
| | cle | 0 (Current little-endian) | 0 (Current little-endian) | PSTATE.tle | | | |
| | tle | 0 (Trap little-endian) | 0 (Trap little-endian) | Undefined | | | |
| | ig | 0 (Interrupt globals not selected) | 0 (Interrupt globals not selected) | 0 (Interrupt globals not selected) | | | |
| | mg | 0 (MMU globals not selected) | 0 (MMU globals not selected) | 0 (MMU globals not selected) | | | |
| TBA<63:15> | | Undefined | Unchanged | Unchanged | | | |

**TABLE O-1** Machine State After Reset and When Entering RED_state *(2 of 4)*

| Name | Fields | Hard_POR | System Reset | WDR | XIR | SIR | RED_state[‡] |
|---|---|---|---|---|---|---|---|
| Y | | Undefined | Unchanged | Unchanged | | | |
| PIL | | Undefined | Unchanged | Unchanged | | | |
| CWP | | Undefined | Unchanged | Unchanged except for register window traps | | | |
| TT[TL] | | 1 | 1 | trap type or 2[†] | 3 | 4 | trap type |
| CCR | | Undefined | Unchanged | Unchanged | | | |
| ASI | | Undefined | Unchanged | Unchanged | | | |
| TL | | MAXTL | MAXTL | Min(TL+1, MAXTL) | | | |
| TPC[TL] | | Undefined | PC | PC | Impl. dep. | PC | |
| TNPC[TL] | | Undefined | nPC | nPC | Impl. dep | nPC | |
| TSTATE [TL] | ccr | Undefined | Undefined | CCR | | | |
| | asi | Undefined | Undefined | ASI | | | |
| | pstate | Undefined | Undefined | PSTATE | | | |
| | cwp | Undefined | Undefined | CWP | | | |
| TICK | npt | 1 | 1 | Unchanged | | | |
| | counter | Restart at 0 | Restart at 0 | Count | Restart at 0 | Count | |
| CANSAVE | | Undefined | Unchanged | Unchanged | | | |
| CANRESTORE | | Undefined | Unchanged | Unchanged | | | |
| OTHERWIN | | Undefined | Unchanged | Unchanged | | | |
| CLEANWIN | | Undefined | Unchanged | Unchanged | | | |
| WSTATE | other | Undefined | Unchanged | Unchanged | | | |
| | normal | Undefined | Unchanged | Unchanged | | | |
| VER | manuf | *Implementation dependent* (impl. dep. #104) | | | | | |
| | impl | *Implementation dependent* (impl. dep. # 13) | | | | | |
| | mask | Mask dependent | | | | | |
| | maxtl | 5 | | | | | |
| | maxwin | 7 | | | | | |
| FSR | all | 0 | *Impl. dep.* | Unchanged | | | |
| FPRS | all | Undefined | Unchanged | Unchanged | | | |
| Non-SPARC V9 ASRs | | | | | | | |
| SOFTINT | | Undefined | Unchanged | Unchanged | | | |
| TICK_COMPARE | int_dis | 1 (off) | 1 (off) | Unchanged | | | |
| | tick_cmpr | 0 | 0 | Unchanged | | | |
| STICK | npt | 1 | 1 | Unchanged | | | |
| | counter | 0 | 0 | Count | | | |

**TABLE O-1** Machine State After Reset and When Entering `RED_state` *(3 of 4)*

| Name | Fields | Hard_POR | System Reset | WDR | XIR | SIR | RED_state‡ |
|---|---|---|---|---|---|---|---|
| STICK_COMPARE | int_dis | 1 (off) | 1 (off) | Undefined | | | |
| | tick_cmpr | 0 | 0 | Undefined | | | |
| PCR | Implementation dependent | | | | | | |
| PIC | Implementation dependent | | | | | | |
| GSR | im | 0 | *Impl. dep.* | Unchanged | | | |
| | others | Undefined | Unchanged | Unchanged | | | |
| DCR | ms | 0 (impl. dep. # 204) | 0 (impl. dep. # 204) | Undefined (impl. dep. # 204) | | | |
| | si | 0 (impl. dep. # 204) | 0 (impl. dep. # 204) | | | | |
| | rpe | 0 (impl. dep. # 204) | 0 (impl. dep. # 204) | | | | |
| | bpe | 0 (impl. dep. # 204) | 0 (impl. dep. # 204) | | | | |
| | bits 13:6 | (impl. dep. #203) | (impl. dep. #203) | | | | |
| | bit 1 | (impl. dep. #203) | (impl. dep. #203) | | | | |
| **Non-SPARC V9 ASIs** | | | | | | | |
| DCUCR | bits 47:41 | (impl. dep. #240) | | | | | |
| | all others | 0 (off) | 0 (off) | 0 (off) | | | |
| ASI_IIU_INST_TRAP | all | 0 (off) | *Impl. dep.* | Unchanged | | | |
| VA_DATA_WATCHPOINT | | Undefined | Undefined | Undefined | | | |
| PA_DATA_WATCHPOINT | | Undefined | Undefined | Undefined | | | |
| I_SFSR, D_SFSR | fv (SFSR valid) | *Impl. dep.* | *Impl. dep.* | Unchanged | | | |
| | All others | Undefined | Unchanged | Unchanged | | | |
| D_SFAR | | Undefined | Unchanged | Unchanged | | | |
| Interrupt Vector Dispatch Status register (ASI_INTR_DISPATCH_STATUS) | all | 0 | Undefined | Unchanged | | | |
| Interrupt Vector Receive register (ASI_INTR_RECEIVE) | busy | 0 | Unchanged | Unchanged | | | |
| | mid | Undefined | *Impl. dep.* | Unchanged | | | |

**TABLE O-1** Machine State After Reset and When Entering RED_state *(4 of 4)*

| Name | Fields | Hard_POR | System Reset | WDR | XIR | SIR | RED_state‡ |
|---|---|---|---|---|---|---|---|
| AFAR | pa | Undefined | Undefined | Undefined | | | |
| AFSR | all | *Impl. dep.* | Unchanged | Unchanged | | | |

*This register is read-only from the system.

‡ Virtual processor states are only updated according to this table if RED_state is entered because of a reset or a trap. If RED_state is entered because the PSTATE.RED bit was explicitly set to 1, then software must create the appropriate states itself.

UPDATED - this register field is updated from its shadow register.

† If WDR occurs in error_state, then it preserves the trap type of the trap that caused entry into error_state. If WDR occurs outside of error_state, it sets TT[TL] to 2.

## O.4.1 Machines States for MTP

**TABLE O-2** Machine State After Reset and in RED_state for Shared Registers

| Name | Field | Hard_POR | System Reset | WDR | XIR | SIR | RED_state |
|---|---|---|---|---|---|---|---|
| ASI_CORE_AVAILABLE | | Predefined value set by hardware ('1' for each implemented virtual processor, '0' for other bits) | | | | | |
| ASI_CORE_ENABLE_STATUS | | ASI_CORE_ENABLE_STATUS gets the value of ASI_CORE_ENABLE at the time of deassertion of reset. | | Unchanged | | | |
| ASI_CORE_ENABLE | | Hardware initializes value at assertion of reset to the value of ASI_CORE_AVAILABLE | Unchanged (value can be overwritten by service processor during reset) | Unchanged | | | |
| ASI_XIR_STEERING | | XIR_STEERING is set at the time of de-assertion of reset. **IMPL. DEP. #1105b**: If the XIR Steering register is read/write, it is implementation dependent whether, upon de-assertion of reset, the value of the CORE_ENABLE or CORE_AVAILABLE register is copied to XIR_STEERING. | | Unchanged | | | |
| ASI_ERROR_STEERING | | **IMPL. DEP. #1106b**: It is implementation dependent whether, upon de-assertion of reset, ERROR_STEERING is set to the value of the lowest-numbered virtual processor that is set to be *enabled* (as indicated by CORE_ENABLE) or is *running* (as indicated by CORE_RUNNING). | | Unchanged | | | |

**TABLE O-2** Machine State After Reset and in RED_state for Shared Registers *(Continued)*

| Name | Field | Hard_POR | System Reset | WDR | XIR | SIR | RED_state |
|---|---|---|---|---|---|---|---|
| ASI_CORE_RUNNING_RW | | Hardware initializes value at assertion of reset. '1' in bit position of lowest-numbered available virtual processor, and '0' in all other bit positions.<br><br>NOTE: if the service processor changes ASI_CORE_ENABLE during the reset and disables the lowest-numbered virtual processor it must update this register. If CORE_RUNNING has a value inconsistent with the current value of ASI_CORE_ENABLE at any time during the reset, either a '1' in a bit position that ASI_CORE_ENABLE has '0' in, or '0's in all bit positions that ASI_CORE_ENABLE has '1's in, hardware may overwrite the value with an implementation dependent value. | Hardware initializes value at assertion of reset. The value is '1' in bit position of lowest-numbered virtual processor that is expected to be enabled as specified by ASI_CORE_ENABLE at the time of assertion of reset, and '0' in all other bit positions.<br><br>NOTE: If an external agent changes ASI_CORE_ENABLE during the reset and disables the lowest-numbered virtual processor it must update this register. If Core Running has a value inconsistent with the current value of ASI_CORE_ENABLE at any time during the reset, either a '1' in a bit position that ASI_CORE_ENABLE has '0' in, or '0's in all bit positions that ASI_CORE_ENABLE has '1's in, hardware may overwrite the value with an implementation dependent value. | Unchanged | | | |
| ASI_CORE_RUNNING_STATUS | | Equal to the ASI_CORE_RUNNING register at deassertion of reset. | Equal to the ASI_CORE_RUNNING register at deassertion of reset. | Not affected | | | |

**TABLE O-3** Machine State After Reset and in RED_state for PER-CORE Register

| Name | Fields | Hard_POR | System Reset | WDR | XIR | SIR | RED_state |
|---|---|---|---|---|---|---|---|
| ASI_CORE_ID | max_core_id | Predefined value set by hardware | | | | | |
| | Core ID | Predefined value set by hardware | | | | | |
| ASI_INTR_ID | | Unknown | Unchanged | | | | |
| ASI_SCRATCHPAD_*n*_REG | | Unknown | Unchanged | | | | |

# Error Handling

This appendix describes processor behavior to operating system and OpenBoot PROM programmers writing error diagnosis and recovery code for SPARC JPS2 processors. The information provides them with the basics in the intent and assumptions in defining error handling.

A traditional approach to handling errors in SPARC is to share the process between hardware and software. This approach gives us much more information about the error for later analysis and maintains more flexibility in the process of error handling than do the alternatives. As the latest SPARC V9 processors, JPS2 processors follow tradition and takes a hardware-software combined approach in handling errors.

Errors are detected by hardware and are signalled through a trap to the recovery software, which is usually operating system software. The most basic part of the information is recorded in hardware and is saved by the recovery software for later analysis. The major part of the error information (for example, contents of critical part in main storage) is also saved by the software for later analysis. In many respects, maintaining data integrity is a key objective in error handling.

Errors are categorized by severity into few classes. Depending on the severity of error, the way to signal an error varies. Traps range from a request just to save the error information recorded by hardware to a request for immediate processing of an error with software.

Some errors provide sufficient cause to halt the entire system immediately, because of possible loss of system consistency. Such cases can signal system hardware directly without software intervention, requesting error handling from system hardware or a service processor for appropriate recovery.

Error handling is described in the following sections:


- *Error Classes and Signalling* on page 608
- *Corrective Actions* on page 609
- *Related Traps* on page 616
- *Related Registers/Error Logging* on page 617
- *Signalling/Special ECC* on page 618

# P.1 Error Classes and Signalling

An error is categorized according to its severity and its characteristics with respect to instruction execution.

## P.1.1 Error Classes in Severity

The classes of error in order of severity are as follows:

1. **Hardware-corrected errors.** Hardware tries to correct the error automatically. A trap is optionally generated to log the error conditions when the error is corrected to enable the actions for preventive maintenance. Upon failure to correct the error, the virtual processor could invoke a trap requesting software recovery.
2. **Software-correctable errors.** Hardware does not correct the error automatically. Instead, it invokes a trap requesting the recovery software to correct the error. Corrective actions are expected from the recovery software. If recovery is successful, the system should continue the operation.
3. **Uncorrectable errors.** The error is by its nature uncorrectable, and hardware invokes a trap to signal the occurrence of the error to appropriate recovery software. Depending on the condition under which the error occurs, the system may be able to recover from the error and continue operation. If not, it may be able to isolate the error to a particular process and terminate it. Otherwise, the software should shut down the system gracefully.
4. **Fatal errors.** By its nature, the error indicates either loss of system consistency or a system interconnect protocol error. It is dangerous to continue operation in this situation because of the impending threat of a failure to maintain data integrity. Therefore, upon the detection of the error, the processor generates an ERROR signal to its interconnect, expecting to be halted/reset by the system. System actions induced by the ERROR signal are system-implementation-dependent.

## P.1.2 Errors Asynchronous to Instruction Execution

Some errors can be detected asynchronously to instruction execution; other errors are detected in the course of an instruction execution.

An error asynchronous to instruction execution is signalled either through a disrupting trap to the virtual processor or through an ERROR signal to system hardware to induce a system reset, depending on the severity of the error.

The errors signalled through a disrupting trap do not directly affect the result of programs, and it is difficult to locate programs affected by the error without any visible effect. Therefore, the actions in response to the errors asynchronous to instruction execution are to save the error information for preventive maintenance.

Errors signalled with an ERROR are meant either to be loss of system consistency or a protocol error on system interconnect.

On the other hand, an error detected in the course of an instruction execution is signalled through an error trap to the instruction, with additional information recorded in hardware. The trap is either precise or deferred. The program (process) affected by the error should be given a corrected response, or if the error is uncorrectable, the process should be terminated appropriately.

**IMPL. DEP. #208:** The order in which errors are captured in instruction execution is implementation dependent in SPARC JPS2. Ordering could be in program order or in the order of detection.

---

**Note –** Both hardware and software must take special care in handling a deferred trap invoked with an error. Hardware must record the state information of the privileged_mode bit (`PSTATE.priv`) either upon detection of the error or upon execution of the instruction that encounters the error. Software must insert an error barrier at the environmental boundary to make valid the privileged / nonprivileged status information recorded by hardware.

---

# P.2 Corrective Actions

Errors are handled by invocation of one of the following actions:

- **Reset-inducing ERROR signal.** A fatal error generates an ERROR signal to induce a system reset. Both an error detected in the course of instruction execution and an error asynchronous to instruction execution may generate an ERROR signal (impl. dep. #212).
- **Precise traps.** Either a software-correctable error or an uncorrectable error generates a precise trap to request software to intervene before normal virtual processor execution continues. An error detected in the course of an instruction execution generates this type of trap.
- **Deferred traps.** An uncorrectable error requiring immediate attention generates a deferred trap to request software intervention. The recovery software examines the recorded error information to determine the extent of the damage caused by the error. Depending on the observed effect, the system may need to be brought

down, or it may continue to run when the effect is within the user program. In any event, the error does not require immediate reset for the system. An error detected in the course of an instruction execution generates this type of trap.

- **Disrupting traps.** Either an instruction-induced error or an error asynchronous to instruction execution generates this type of trap to request logging and clearing. The error does not otherwise affect virtual processor execution.

Although traps have some implementation-dependent characteristics for signalling errors, the following subsections describe trap types and trap names corresponding to the actions described above. Relation between errors and actions is depicted in FIGURE P-1.

**IMPL. DEP. #209:** If an instruction-induced error requires software intervention for recovery, the precision of the trap used to signal the error is implementation dependent.

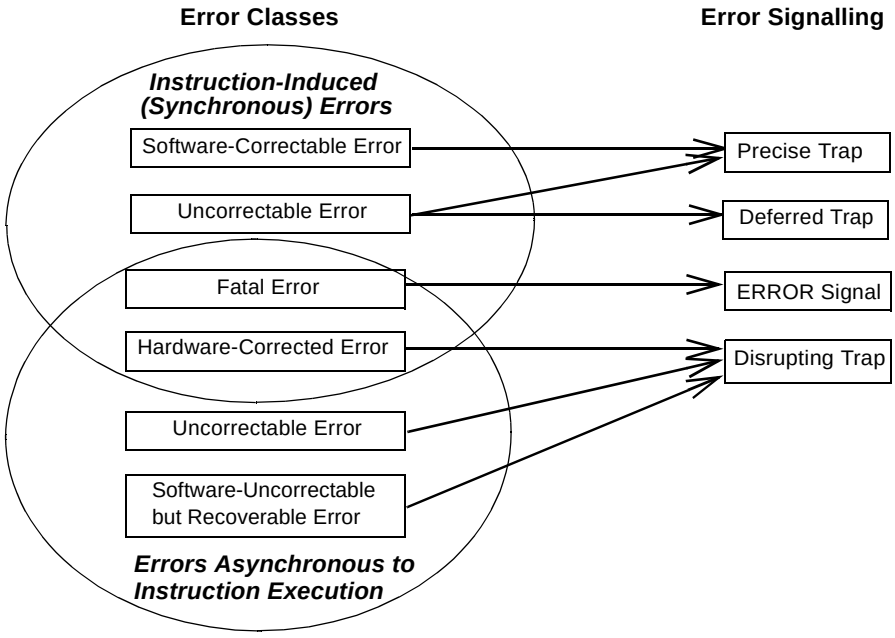



**FIGURE P-1** Error Classes and Corrective Actions

## P.2.1 Reset-Inducing ERROR Signal

It is usually impossible to recover a system or a coherence domain that suffers loss of system or domain coherency because of various error conditions in caches or in the system interconnect that carries coherency transactions. The characteristic examples of such error conditions are an interconnect address error and an error in E-cache tag status information.

Upon the detection of an error condition that indicates loss of system or domain coherency, the system or processor tries to stop everything to maintain data integrity by shutting down the system or domain as soon as possible. The error class that indicates loss of system / domain coherence is called *fatal error*, as defined in the preceding section. When a fatal error is detected within a processor, the processor asserts its ERROR output signal to system hardware. Although the processor expects the system to generate a system reset for entire system or coherence domain in response to the ERROR output signal, the actual response of the system when it receives an ERROR signal depends on the system design.

**IMPL. DEP. #210**: The following aspects of the ERROR output signal are implementation dependent in SPARC JPS2:

- The causes of the ERROR signal
- Whether each of the causes of the ERROR signal, when it generates the ERROR signal, halts the processor or allows the processor to continue running
- The exact semantics of the ERROR signal

For exact definitions of the causes and semantics of the ERROR output signal, please refer to individual JPS2 Extension documents.

To provide basic information for fault analysis on a fatal error, the processor preserves some part of the contents of error logging registers beyond the system / domain reset induced by the ERROR signal. For the definitions of machine states after reset, please refer to Appendix O, *Reset, RED_state, and Error_state*, of JPS2 Extension documents.

The expected scenario is as follows:

After the system / domain reset in response to an ERROR signal, system / domain initialization software takes charge of system recovery. During the system initialization process, the software examines error logging registers to locate the source of the reset and the cause of the fatal error. The software further saves the information, including the contents of error logging registers, to provide later fault analysis with as much information as possible.

**IMPL. DEP. #211:** The information that the error logging registers preserves beyond the reset induced by an ERROR signal is implementation dependent in SPARC JPS2.

Although most fatal errors that lead to an assertion of ERROR signal do not cause any special processor behavior, in some cases, depending on the implementation, there are a few fatal errors for which the processor asserts an ERROR signal and begins a trap execution.

**IMPL. DEP. #212:** Generation of a trap along with assertion of an ERROR signal upon detection of a fatal error is implementation dependent in SPARC JPS2.

## P.2.2 Precise Traps

A precise trap occurs before any program-visible state has been modified by the instruction to which the `TPC` points. When a precise trap occurs, several conditions are true:

1. The `PC` saved in `TPC[TL]` points to the instruction that induced the trap and the `nPC` saved in `TNPC[TL]` points to the instruction that was to be executed next.
2. All instructions issued before the instruction pointed to by the `TPC` have completed execution.
3. Any instructions issued after the instruction pointed to by the `TPC` remain unexecuted.

A precise trap is invoked when a software-correctable error or an uncorrectable error is detected. By its definition, a precise trap is only generated with an error detected in the course of an instruction execution.

A precise trap is signalled when an error that needs software intervention for recovery is detected. Depending on the condition under which the error occurs, the system may be able to recover from the error and continue operation. If not, it may be able to isolate the error to a particular process and terminate it. Otherwise, the software should shut down the system gracefully. State information saved in the trap stack can help the software error handler identify the privileged state under which the error is detected.

## P.2.3 Deferred Traps

Depending on the implementation of instruction execution control, there may be cases in which completion of instruction execution is out of its program order. If an error is detected in the course of such instruction, the error is signalled with a deferred trap. Deferred traps may corrupt the state of the virtual processor. Such traps lead to termination of the currently executing process or result in a system shutdown if the system state has been corrupted. Error logging information allows software to determine if system state has been corrupted.

In SPARC JPS2 virtual processors, the privileged state information related to the error is logged into one of the error status registers. A bit in the error status register indicates the privileged state of the virtual processor, either when it detects the error or when it executes the instruction that encounters the error. For details, refer to the JPS2 Extension documents.

---

**Note –** In SPARC JPS2, Asynchronous Fault Status Register (AFSR) may contain a bit to indicate the privileged state bit (AFSR.priv) associated with the error.

---

**IMPL. DEP. #213:** The existence of the AFSR.priv bit is implementation dependent. If AFSR.priv is implemented, it is implementation dependent whether the logged AFSR.priv indicates the privileged state upon the detection of an error or upon the execution of an instruction that induces the error. For the former implementation to be effective, operating software must provide error barriers appropriately.

See *Error Barriers*, below, for more information.

**IMPL. DEP. #214:** Whether an implementation provides an enable / disable control feature for deferred traps is implementation dependent in SPARC JPS2.

### P.2.3.1 Error Barriers

A MEMBAR #Sync instruction provides an error barrier for deferred traps. It ensures that deferred traps from earlier memory references are not reported after the MEMBAR. To provide error isolation between processes, use a MEMBAR #Sync when context switching or whenever the error logging information that identifies AFSR.priv bit is changed. Note that traps do not provide the same function as MEMBAR #Sync.

**IMPL. DEP. #215:** DONE and RETRY instructions may implicitly provide an error barrier function as MEMBAR #Sync. Whether DONE and RETRY instructions provide an error barrier is implementation dependent in SPARC JPS2.

### P.2.3.2 TPC, TNPC, and Deferred Traps

After a deferred trap, the contents of TPC[TL] and TNPC[TL] are undefined. They do *not* generally contain the oldest nonexecuted instruction and its next PC. Because of this, execution cannot normally be resumed from the point that the trap is taken.

### P.2.3.3 Deferred Trap Handler Functionality

The following is a possible sequence to handle an error signalled through a deferred trap. In this sequence, a pair of error logging registers—an error status register and an error address register—is assumed as described above.

1. Log the error(s).
2. Reset the error logging bits in the error status register. Perform a `MEMBAR #Sync` to complete internal state changes.
3. Panic if the error occurs under privileged state not performing an intentional peek/poke (See *Special Access Sequence for Recovering Deferred Traps* on page 614); otherwise, try to continue.
4. Validate the faulty location, if required, based on the information logged.
5. Abort the current process.
6. For user-process uncorrectable errors in a conventional UNIX system, once all processes using the physical page in error have been signalled and terminated, as part of the normal page recycling mechanism, clear the uncorrectable error from main memory by zeroing the page, using block store instructions.
7. Resume execution.

### P.2.3.4 Special Access Sequence for Recovering Deferred Traps

A special access sequence is required for intentional peeks and pokes to determine device presence and correctness, and for I/O accesses from hardened drivers that must survive faults in an I/O device. This special access sequence allows the error trap handler to recover predictably, even though the trap is deferred. One possible sequence is described here.

The procedure is for an error signalled for data reference, meaning that the sequence is executed in a *data_access_error* trap handler.

```
The_peeker:
       <set a flag indicating special peek sequence is about to
       occur. This flag includes specifying the handler as a
       Special_peek_handler if a deferred TO/BERR does occur>

       MEMBAR#Sync /* error barrier for deferred traps, [1] see
                   explanation below*/
       <call routine to do the peek>
       <reset the peek_sequence>
          <check success/failure indication from peek>

Do_the_peek_routine:
       <perform load. If deferred trap occurs, execution will never
       resume here>
       MEMBAR#Sync /* error barrier; make sure load takes */
       <indicate peek success>
       <return to peeker>
```

```
Special_peek_handler:
       <indicate peek failure>
       <return to peeker as if returning from Do_the_peek_routine>

Deferred_trap_handler: (TL = 1)
       <If the deferred trap handler sees a UE or TO or BERR and
       the peek_sequence_flag is set, it resumes execution at the
       Special_peek_handler (by setting TPC and TNPC)>
```

Other than the load (or store, in the case of poke), `Do_the_peek_routine` should not have any other side effect since the deferred trap means that the code is not restartable. Execution after the trap is resumed in `Special_peek_handler`.

The code in `Deferred_trap_handler` must be able to recognize any deferred traps that happen as a result of hitting the error barrier in `The_peeker` as not being from the peek operation. This will typically be part of setting the `peek_sequence_flag`.

A `MEMBAR #Sync` is required as the first instruction in the trap table entry for `Deferred_trap_handler` to collect all potential trapping stores together to avoid a `RED_state` exception (see *Error Barriers* on page 613).

`TPC` or `AFAR` can be used to identify whether a deferred trap came from a peek or poke sequence. If `TPC` is used, the locality of the trap to `Do_the_peek_routine` must be ensured by use of an error barrier, as in the example above. If `AFAR` is used, the presence of orphaned errors, resulting from the asynchronous activity of the instruction fetcher, must be considered. If an orphaned error occurs, then the source of the `TO` or `BERR` report cannot be determined from the `AFAR`. Given the error barrier sequence above, it is reasonable to expect that the `TO` or `BERR` resulted from the peek or poke and to proceed accordingly. To reduce the likelihood of this event, orphaned errors could be cleaned at point [1] above. The source of the `TO` or `BERR` could be confirmed by retrying the peek or poke: If the `TO` or `BERR` happens again, the system can continue with the normal peek or poke failure case. If the `TO` or `BERR` does not happen, the system must panic.

The peek access should be preceded and followed by `MEMBAR #Sync` instructions. The destination register of the access may be destroyed, but no other state will be corrupted. If `TPC` points to the `MEMBAR #Sync` following the access, then the trap handler knows that a recoverable error has occurred and resumes execution after setting a status flag. The trap handler must set `TNPC` to `TPC` + 4 before resuming because the contents of `TNPC` are otherwise undefined.

## P.2.4 Disrupting Traps

Disrupting traps, like deferred traps, may have changed their program-visible state since the instruction that caused them. The following are true for a disrupting trap:

1. The PC saved in TPC[TL] points to a valid instruction that will be executed by the program, and the nPC saved in TNPC[TL] points to the instruction that will be executed after that one.

2. All instructions issued before the one pointed to by the TPC have completed execution.

3. Any instructions issued after the one pointed to by the TPC remain unexecuted.

Errors that lead to disrupting traps are hardware-corrected errors and uncorrectable errors. A hardware-corrected error in the course of an instruction execution, an uncorrectable error, or a hardware-corrected error triggered with an asynchronous event may cause a disrupting trap.

The disrupting trap handler should save the information on the error logged in error status registers. No special operations such as cache flushing are required for correctness after a disrupting trap. However, for many errors, it is appropriate to correct the data that produced the original error so that later references to the same faulty data do not produce the same trap again. For uncorrectable errors, software must determine the recovery mechanism with the minimum system impact.

For hardware-corrected errors, SPARC JPS2 implementations should provide a mechanism to enable and disable traps for software error handling. In some cases, software disables these disrupting traps and only reads the logging information periodically to gather error statistics for later preventive maintenance.

---

**Note –** To prevent multiple traps from the same error, software should not reenable interrupts until after the disrupting error status bit in AFSR is cleared.

---

# P.3 Related Traps

SPARC JPS2 virtual processors use the following traps for error signalling:

- ***data_access_error*** [TT = $32_{16}$]: An error, either precise or deferred, detected during execution of data reference. The error is detected in the course of a memory reference instruction execution.
- ***instruction_access_error*** [TT = $0A_{16}$]: An error, either precise or deferred, detected during instruction fetch reference. The error is detected in the course of an instruction fetch reference.
- ***ECC_error*** [TT = $63_{16}$]: A disrupting trap. Either an error corrected automatically by hardware or an uncorrectable error. Both an instruction-inducing error and an error asynchronous to instruction execution are possible.

**IMPL. DEP. #216:** The precision of a *data_access_error* trap is implementation dependent in SPARC JPS2.

**IMPL. DEP. #217:** The precision of an *instruction_access_error* trap is implementation dependent in SPARC JPS2.

The followings traps are used in SPARC JPS2 virtual processors as JPS2 implementation-dependent traps. See the JPS2 Extension documents for details.

- ***fast_ECC_error*** [TT = $70_{16}$]: A precise trap for the system to be able to continue operation. A single-bit or multiple-bit ECC error is detected (impl. dep. #202).
- ***async_data_error*** [TT = $40_{16}$]: An implementation-dependent trap (impl. dep. #31, #218) that signifies an urgent error, to be processed as soon as possible. Precise, deferred, or disrupting. When *async_data_error* is not precise, the TPC and TNPC stacked by the trap are not necessarily related to the instruction or data access that caused the error.

  **IMPL. DEP. #218:** Whether *async_data_error* trap is implemented is implementation dependent. If it does exist, it indicates that an error is detected in a virtual processor and its trap type is $40_{16}$.

# P.4 Related Registers / Error Logging

Although a SPARC JPS2 virtual processor contains some error logging registers, information about an error is always recorded in a pair of error logging registers: an error status register to identify causes of error, and an error address register to get address information for an error. Therefore, the amount of information for an error does not exceed two 64-bit words. Note that an error status register and an error address register do not mean a single entity each. Rather, they mean one of multiple status registers and one of multiple address registers paired to the status register. In each error condition, the valid pair of error logging register varies, for convenience in software error handling.

In addition, for instruction-related errors signalled as an *instruction_access_error* trap, address information is provided in TPC[TL], not in an explicit logging register.

The following registers are provided in a SPARC JPS2 virtual processor for error handling:

- Instruction Synchronous Fault Status Register (ISFSR: ASI = $50_{16}$, VA = $18_{16}$)
- Data Synchronous Fault Status Register (DSFSR: ASI = $58_{16}$, VA = $18_{16}$)
- Data Synchronous Fault Address Register (DSFAR: ASI = $58_{16}$, VA = $20_{16}$)
- Asynchronous Fault Status Register (AFSR: ASI = $4C_{16}$, VA = $0_{16}$)

- Asynchronous Fault Address Register (AFAR: ASI = $4D_{16}$, VA = implementation dependent)

**IMPL. DEP. #219:** Allocation of Asynchronous Fault Address Register (AFAR) is implementation dependent in SPARC JPS2. There may be one instance or multiple instances of AFAR. Although the ASI for AFAR is defined as $4D_{16}$, the virtual address of AFAR if there are multiple AFARs is implementation dependent in SPARC JPS2.

**IMPL. DEP. #220:** Whether the implementation supports additional logging/control registers for error handling is implementation dependent in SPARC JPS2.

Note that an error signalled through a precise trap may be logged in AFSR/AFAR pair, whereas an error signalled through a deferred trap is never logged in a synchronous status/address register pair. For details about error logging mechanisms, refer to the appropriate JPS2 Extension document.

These error status and error address registers may be overwritten with subsequent error(s) depending on the implementation. For overwriting policies, refer to the appropriate JPS2 Extension document.

# P.5 Signalling/Special ECC

SPARC JPS2 virtual processors provide a special feature for memory-related errors. The feature is called as *signalling ECC* or *special ECC*, depending on the implementation. This feature aids in survival and diagnosis of an error in a coherent domain; it avoids misprocessing of bad memory data.

If bad data is replaced with a good data upon detection of an error, the other processes that share the data could continue processing without knowledge of erroneous data even if they receive replaced data. To prevent the use of “fixed” faulty data, a SPARC JPS2 virtual processor replaces the data that has a bad ECC code with an uncorrectable ECC error generated in a predetermined method. In addition to preventing the use of faulty data, the signalling/special ECC may be used for locating the faulty component in a system by embedding a processor module ID code into the replaced data.

For details, refer to the appropriate JPS2 Extension document.

**IMPL. DEP. #221:** The method to generate “special” or “signalling” ECCs and whether processor module ID is embedded into the data associated with special/signalling ECCs is implementation dependent in SPARC JPS2.

# P.6 Error Handling in MTPs

Errors in a structure private to a virtual processor are reported to that virtual processor, using the error reporting mechanism of that virtual processor. These errors are considered core-specific. An error in a shared structure is, whenever possible, reported to the virtual processor initiating the request that caused or detected the error. These errors are considered core specific. Some errors in a shared structure cannot be attributed to a virtual processor and are therefore non-core-specific. See *Error Handling in MTPs* on page 207 for details.

# Performance Instrumentation

For implementation-dependent performance instrumentation information, please refer to Appendix Q in the specific SPARC JPS2 Extension document.

# Bibliography

## General References


Boney, Joel. “SPARC Version 9 Points the Way to the Next Generation RISC,” *SunWorld*, October 1992, pp. 100-105.

Cohen, D., “On Holy Wars and a Plea for Peace.” *Computer* 14:10, October 1981, pp. 48-54.

Comer, Douglas. “The Ubiquitous B-Tree.” *ACM Computing Surveys*, Vol. 11, No. 2, June 1979.

*Implementation Characteristics of Current SPARC V9-based Products, Revision 9.x*, SPARC International, Inc.

Knuth, Donald. *The Art of Computer Programming, Volume 3, Searching and Sorting. Addison-Wesley, 1974.*

Weaver, David L., editor. *The SPARC Architecture Manual, Version 8*, Prentice-Hall, Inc., 1992.

Weaver, David L., and Tom Germond, eds. *The SPARC Architecture Manual-Version 9*, Prentice-Hall, Inc., 1994.

# Index


## A

a field of instructions, **102**, 238, 241, 242, 246, 385, 388
accrued exception (aexc) field of FSR register, 57, **58**, 139, 419, 420
ADD instruction, **224**, 516
ADDC instruction, **224**
ADDcc instruction, **224**, 355, 516
ADDCcc instruction, **224**
address
  64-bit virtual data watchpoint, 95
  aliased, 171
  aliasing, 469
  memory, 171
  operand syntax, 512
  physical, 171
  physical address data watchpoint, 95
  space identifier (ASI), 171, 567
  virtual, 171
  virtual address
    data watchpoint, 95
    watchpoint priority, 95
  virtual passed to physical, 93
address mask (am) field of PSTATE register, **76**
address space, 5
address space identifier (ASI), 9
  accessing MMU registers, 488
  affected by watchpoint traps, 94
  alternate address spaces, privileged / nonprivileged, 564
  appended to memory address, 23, 100
  architecturally specified, **174**
  bit 7 setting for *privileged_action* exception, 364
  bypass, **9**, 94, 568
  definition, 9
  encoding address space information, 105
  explicit values, 110
  explicitly specified in instruction, 110
  imm_asi instruction field, 103
  implicit values, 110
  load floating-point instructions, 275
  load from TLB Data Access register, 492
  load from TLB Data In register, 492
  load from TLB Tag Read register, 492
  load integer instructions, 280
  nontranslating, **14**, 94
  operations, 482, 487
  with prefetch instructions, 336
  restricted, 174, 482, 568
  restriction indicator, 66
  SPARC V9 address, 173
  translating ASIs, 94, 568
  unrestricted, **174**, 537, 568
address space identifier (ASI) register
  for load / store alternate instructions, 66
  bypass ASIs, 97
  and LDDA instruction, 396
  and LDSTUBA instruction, 286
  load floating-point from alternate space instructions, 277
  load integer doubleword instructions, 394
  load integer from alternate space instructions, 282
  nontranslating, 397, 409
  with prefetch instructions, 336
  for register-immediate addressing, 174
  reserved ranges, 196

restoring saved state, 249
saved trap state, 534
saving state, 129
state after reset, 601
and STDA instruction, 408
store floating-point into alternate space instructions, 364
store integer to alternate space instructions, 369
and SWAPA instruction, 412
translating, 97
after trap, 27
and TSTATE Register, 80
and write state register instructions, 381
addressing modes, 5
ADDX instruction (SPARC V8), 225
ADDXcc instruction (SPARC V8), 225
AFAR, *See* Asynchronous Fault Address Register (AFAR)
AFSR, *See* Asynchronous Fault Status Register (AFSR)
alias
address, 171
floating-point registers, 46
aliased, 9
ALIGNADDRESS instruction, 226
ALIGNADDRESS_LITTLE instruction, 226
alignment
data (load/store), 106, 173
doubleword, 106, 173
extended-word, 106
halfword, 106, 173
instructions, 106, 173
integer registers, 394, 396, 582
maintaining, 521
memory, 163, 173
quadword, 106, 173
stack pointer, 521
word, 106, 173
alternate address space, 336
alternate global registers, 22, 41, 42, 534
alternate globals enable (ag) field of PSTATE register, 42, **77**
alternate space instructions, 24, 66, 564
ancillary state registers (ASRs), 62 to 91
adding instructions to SPARC V9, 539
assembly language syntax, 506
I/O register access, 24
number, 62
possible registers included, 345, 382
privileged, 428
reading/writing implementation-dependent processor registers, 428
writing to, 381
AND instruction, **291**
ANDcc instruction, **291**, 516
ANDN instruction, **291**, 516
ANDNcc instruction, **291**
annul bit
in branch instructions, 238
in conditional branches, 386
in control transfer instruction, 62
annulled branches, 238
application program, **9**, 62, 562, 564
architectural extensions, 539
architecture, meaning for SPARC V9, 1
arguments to a subroutine, 518
arithmetic overflow, 64
ARRAY16 instruction, 228
ARRAY32 instruction, 228
ARRAY8 instruction, 228
ASI, *See* address space identifier (ASI)
ASI_AFSR, 571, 572
ASI_AIUP, 569
ASI_AIUPL, 570
ASI_AIUS, 569
ASI_AIUSL, 570
ASI_AS_IF_USER_PRIMARY, 174, 534, 569
ASI_AS_IF_USER_PRIMARY_LITTLE, 174, 534, 570
ASI_AS_IF_USER_SECONDARY, 174, 569
ASI_AS_IF_USER_SECONDARY_LITTLE, 174, 570
ASI_ASYNC_FAULT_STATUS, 571, 572
ASI_ATOMIC_QUAD_LDD_PHYS, 570
ASI_ATOMIC_QUAD_LDD_PHYS_L, 570
ASI_ATOMIC_QUAD_LDD_PHYS_LITTLE, 570
ASI_BARRIER_SYNCH, 578
ASI_BARRIER_SYNCH_P, 574
ASI_BLK_AIUP, 574
ASI_BLK_AIUPL, 575
ASI_BLK_AIUS, 575
ASI_BLK_AIUSL, 575
ASI_BLK_COMMIT_P, 578
ASI_BLK_COMMIT_S, 578
ASI_BLK_P, 578
ASI_BLK_PL, 579
ASI_BLK_S, 578
ASI_BLK_SL, 579
ASI_BLOCK_AS_IF_USER_PRIMARY, 574

ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE, 575
ASI_BLOCK_AS_IF_USER_SECONDARY, 575
ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE, 575
ASI_BLOCK_COMMIT_PRIMARY, 578
ASI_BLOCK_COMMIT_SECONDARY, 578
ASI_BLOCK_PRIMARY, 578
ASI_BLOCK_PRIMARY_LITTLE, 579
ASI_BLOCK_SECONDARY, 578
ASI_BLOCK_SECONDARY_LITTLE, 579
ASI_CORE_AVAILABLE, 197, 212, 213
ASI_CORE_ENABLE, 198, 212, 213
ASI_CORE_ENABLE_STATUS, 198, 212, 213
ASI_CORE_ID, 191, 212, 213
ASI_CORE_RUNNING, 200
ASI_CORE_RUNNING_RW, 200, 201, 212, 213
ASI_CORE_RUNNING_STATUS, 203, 212, 213
ASI_CORE_RUNNING_W1C, 201, 212, 213
ASI_CORE_RUNNING_W1S, 200, 212, 213
ASI_DCU_CONTROL_REGISTER, 91, 571
ASI_DCUCR, 571
ASI_DEVICE_ID+SERIAL_ID, 579
ASI_DMMU, 573
ASI_DMMU_DEMAP, 574
ASI_DMMU_PA_WATCHPOINT_REG, 573
ASI_DMMU_SFAR, 573
ASI_DMMU_SFSR, 573
ASI_DMMU_TAG_ACCESS, 573
ASI_DMMU_TAG_TARGET_REG, 573
ASI_DMMU_TSB_64KB_PTR_REG, 574
ASI_DMMU_TSB_8KB_PTR_REG, 574
ASI_DMMU_TSB_BASE, 573
ASI_DMMU_TSB_DIRECT_PTR_REG, 574
ASI_DMMU_TSB_NEXT_REG, 574
ASI_DMMU_TSB_PEXT_REG, 573
ASI_DMMU_TSB_SEXT_REG, 574
ASI_DMMU_VA_WATCHPOINT_REG, 573
ASI_DTLB_DATA_ACCESS_REG, 574
ASI_DTLB_DATA_IN_REG, 574
ASI_DTLB_TAG_READ_REG, 574
ASI_ERROR_STEERING, 208, 212, 213
ASI_FL16_P, 578
ASI_FL16_PL, 578
ASI_FL16_PRIMARY, 578
ASI_FL16_PRIMARY_LITTLE, 578
ASI_FL16_S, 578
ASI_FL16_SECONDARY, 578
ASI_FL16_SECONDARY_LITTLE, 578
ASI_FL16_SL, 578
ASI_FL8_P, 577
ASI_FL8_PL, 578
ASI_FL8_PRIMARY, 577
ASI_FL8_PRIMARY_LITTLE, 578
ASI_FL8_S, 578
ASI_FL8_SECONDARY, 578
ASI_FL8_SECONDARY_LITTLE, 578
ASI_FL8_SL, 578
ASI_IIU_INST_TRAP, 574, 581
ASI_IIU_INST_TRAP register, 602
ASI_IMMU, 572
ASI_IMMU_DEMAP, 573
ASI_IMMU_SFSR, 572
ASI_IMMU_TAG_TARGET, 572
ASI_IMMU_TSB_64KB_PTR_REG, 573
ASI_INTR_DATA0_R, 576
ASI_INTR_DATA0_W, 575
ASI_INTR_DATA1_R, 576
ASI_INTR_DATA1_W, 575
ASI_INTR_DATA2_R, 576
ASI_INTR_DATA2_W, 575
ASI_INTR_DATA3_R, 576
ASI_INTR_DATA3_W, 575
ASI_INTR_DATA4_R, 576
ASI_INTR_DATA4_W, 575
ASI_INTR_DATA5_R, 576
ASI_INTR_DATA5_W, 575
ASI_INTR_DATA6_R, 576
ASI_INTR_DATA6_W, 575
ASI_INTR_DATA7_R, 576
ASI_INTR_DATA7_W, 575
ASI_INTR_DISPATCH_EXT_W, 194, 214
ASI_INTR_DISPATCH_STATUS, 588, 592, 593, 602
  busy bit, 588, 591
  nack bit, 588, 591
ASI_INTR_DISPATCH_W, 575, 591
ASI_INTR_ID, 192, 212, 213
ASI_INTR_RECEIVE, 195, 571, 589, 593, 602
ASI_INTR_W, 588, 591
ASI_ITLB_DATA_ACCESS_REG, 573
ASI_ITLB_DATA_IN_REG, 573
ASI_ITLB_TAG_READ_REG, 573
ASI_MONDO_RECEIVE_CTRL, 571
ASI_MONDO_SEND_CTRL, 571
ASI_N, 569
ASI_NL, 569
ASI_NUCLEUS, 110, 484, 569
ASI_NUCLEUS_LITTLE, 110, 484, 569

ASI_NUCLEUS_QUAD_LDD, 570, 581
ASI_NUCLEUS_QUAD_LDD_L, 570
ASI_NUCLEUS_QUAD_LDD_LITTLE, 570, 581
ASI_P, 576
ASI_PHYS_BYPASS_EC_WITH_EBIT, 486, 569, 580
ASI_PHYS_BYPASS_EC_WITH_EBIT_L, 570
ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE, 570, 581
ASI_PHYS_USE_EC, 569, 580
ASI_PHYS_USE_EC_L, 570
ASI_PHYS_USE_EC_LITTLE, 570, 580
ASI_PL, 576
ASI_PNF, 576
ASI_PNFL, 576
ASI_PRIMARY, 110, 174, 484, 576
ASI_PRIMARY_*, 484
ASI_PRIMARY_CONTEXT_REG, 573
ASI_PRIMARY_LITTLE, 110, 174, 483, 484, 576
ASI_PRIMARY_NO_FAULT, 498, 576
ASI_PRIMARY_NO_FAULT_LITTLE, 498, 576
ASI_PRIMARY_NOFAULT, 174, 175
ASI_PST16_P, 313, 577
ASI_PST16_PL, 313, 577
ASI_PST16_PRIMARY, 577
ASI_PST16_PRIMARY_LITTLE, 577
ASI_PST16_S, 313, 577
ASI_PST16_SECONDARY, 577
ASI_PST16_SECONDARY_LITTLE, 577
ASI_PST16_SL, 313
ASI_PST32_P, 313, 577
ASI_PST32_PL, 313, 577
ASI_PST32_PRIMARY, 577
ASI_PST32_PRIMARY_LITTLE, 577
ASI_PST32_S, 313, 577
ASI_PST32_SECONDARY, 577
ASI_PST32_SECONDARY_LITTLE, 577
ASI_PST32_SL, 313, 577
ASI_PST8_P, 313, 577
ASI_PST8_PL, 313, 577
ASI_PST8_PRIMARY, 577
ASI_PST8_PRIMARY_LITTLE, 577
ASI_PST8_S, 313, 577
ASI_PST8_SECONDARY, 577
ASI_PST8_SECONDARY_LITTLE, 577
ASI_PST8_SL, 313, 577
ASI_QUAD_LDD, 283
ASI_QUAD_LDD_L, 283
ASI_QUAD_LDD_PHYS, 283, 570, 581
ASI_QUAD_LDD_PHYS_L, 283, 570
ASI_QUAD_LDD_PHYS_LITTLE, 570, 581
ASI_S, 576
ASI_SCRATCHPAD_n_REG, 212
ASI_SDB_INTR, 590, 593
ASI_SDB_INTR_R, 589
ASI_SECONDARY, 174, 537, 576
ASI_SECONDARY_CONTEXT_REG, 573
ASI_SECONDARY_LITTLE, 537, 576
ASI_SECONDARY_NO_FAULT, 498, 576
ASI_SECONDARY_NO_FAULT_LITTLE, 498, 576
ASI_SECONDARY_NOFAULT, 174, 175
ASI_SERIAL_ID, 573
ASI_SL, 576
ASI_SNF, 576
ASI_SNFL, 576
ASI_XIR_STEERING, 206, 212, 213
asr_reg, **506**
ASRs, *See* ancillary state registers (ASRs)
assembler, synthetic instructions, 514
*async_data_error* exception, 165, 275, 282, 285, 397, 617
Asynchronous Fault Address Register (AFAR), 603, 615, 618
Asynchronous Fault Status Register (AFSR), 603, 613, 617
atomic
  load quadword, 283
  memory operations, 179, **182**, 283
  store doubleword instruction, 406, 408
  store instructions, 366, 369
atomic load-store instructions, 105, 247
  compare and swap, **246**
  load-store unsigned byte, **285**, 410, 412
  load-store unsigned byte to alternate space, **286**
  swap r register with alternate space memory, **412**
  swap r register with memory, 247, **410**
atomicity, 172, 433
automatic variables, 519

## B

BA instruction, 387, 388, 462
BCC instruction, 387, 462
BCLR synthetic instruction, **516**
BCS instruction, 387, 462
BE instruction, 387, 462
Berkeley RISCs, 6
BG instruction, 387, 462

BGE instruction, 387, 462
BGU instruction, 387, 462
Bicc instructions, 65, **387**, 457, 462
big-endian byte order, 23, 74, 106
binary compatibility, 7
bit vector concatenation, 3
BL instruction, 462
BLD, *See* block load instructions
BLE instruction, 387, 462
BLEU instruction, 387, 462
block
  load instructions, 47, 231
  store instructions, 47, 231
    with commit, 232, 582
block load instructions, 582
block store instructions, 582
BMASK instruction, 235
BN instruction, 340, 387, 388, 462, 514
BNE instruction, 387, 462
BNEG instruction, 387, 462
BP instructions, 463
BPA instruction, 242, 462
BPCC instruction, 242, 462
BPcc instructions, 65, 102, 103, 104, **242**, 340, 464
BPCS instruction, 242, 462
BPE instruction, 242, 462
BPG instruction, 242, 462
BPGE instruction, 242, 462
BPGU instruction, 242, 462
BPL instruction, 242, 462
BPLE instruction, 242, 462
BPLEU instruction, 242, 462
BPN instruction, 242, 462
BPNE instruction, 242, 462
BPNEG instruction, 242, 462
BPOS instruction, 387, 462
BPPOS instruction, 242, 462
BPr instructions, 103, 104, **237**, 462
BPVC instruction, 242, 462
BPVS instruction, 242, 462
branch
  annulled, 238
  delayed, 100
  elimination, 117, 118
  fcc-conditional, 241, 386
  icc-conditional, 389
  prediction bit, 238
  unconditional, 241, 243, 385, 388
  with prediction, 5
branch if contents of integer register match condition instructions, **237**
branch on floating-point condition codes instructions, **384**
branch on floating-point condition codes with prediction instructions, **239**
branch on integer condition codes instructions, *See* Bicc instructions
branch on integer condition codes with prediction (BPcc) instructions, 242
breakpoint
  data, *See* watchpoints
  instruction, *See* Instruction Trap Register
BRGEZ instruction, 237
BRGZ instruction, 237
BRLEZ instruction, 237
BRLZ instruction, 237
BRNZ instruction, 237
BRZ instruction, 237
BSET synthetic instruction, **516**
BSHUFFLE instruction, 235
BST, *See* block store instructions
BTOG synthetic instruction, **516**
BTST synthetic instruction, **516**
BVC instruction, 387, 462
BVS instruction, 387, 462
bypass ASI, **9**, 94, 97, 568
byte, 9
  addressing, 108
  data format, **29**
  order, 23
  order, big-endian, 23, 74
  order, implicit, 74
  order, little-endian, 23, 74

## C

cache
  data, 177
  instruction, 177
  miss, 340
  nonconsistent instruction cache, 177
  system, 6
caching, TSB, 474
call chain, walking, 519
CALL instruction
  address in out register, 518
  description, 245
  determining a procedure's return address, 518

displacement, 26
does not change CWP, 45
and JMPL instruction, 273
leaf procedure, 521
writing address into r[15], 45
CALL synthetic instruction, **514**
CANRESTORE register, 85, 601
CANSAVE register, 85, 601
carry (c) bit of condition fields of CCR, **64**
CAS synthetic instruction, 179, **515**
CASA instruction, 24, 105, 182, **246**, 285, 286, 410, 412, 515
CASX synthetic instruction, 179, 182, **516**
CASXA instruction, 24, 105, 182, **246**, 285, 286, 410, 412, 516
*catastrophic_error* exception, **130**
cc0 field of instructions, **102**, 241, 242, 255, 306, **463**
cc1 field of instructions, **102**, 241, 242, 255, 306, 463
cc2 field of instructions, **102**, 306, 463
CCR, *See* condition codes (CCR) register
certificate of compliance, 8
clean register window, 86, 118, 125, 126, 128, 164, 348
clean window, 9
clean windows (CLEANWIN) register, 83, **86**, 118, 125, 126, 128, 341, 377, 601
*clean_window* exception, 86, 118, 127, 137, 140, **164**, 349, 350, 431
CLEAR_SOFTINT pseudo-register, 89, 121, 595
CLEAR_SOFTINT register, 88
clipping values, *See* FPACK instructions
clock cycle, 66
clock-tick register (STICK), **70**
clock-tick register (TICK), **66**, 163, 341, 377, 432
CLR synthetic instruction, **516**
CMP
synthetic instruction, 371
cmp synthetic instruction, **514**
code
kernel, 594
nucleus, 594
coherence, 10
and address remapping, 171
between processors, 433
data cache, 177
memory, 172
unit, memory, 173
compare and swap instructions, **246**
comparison instruction, 112, 371
compatibility with SPARC V8, *See* SPARC V8 compatibility
completed, 10
compliance
certificate of, 8
certification process, 8
claim, 8
Level I, 8
Level II, 8
SPARC V9, 487
compliant SPARC V9 implementation, **7**
concatenation of bit vectors, 3
cond field of instructions, **103**, 241, 242, 298, 306, 385, 388
condition codes, 247
adding, 372
extended integer (xcc), 64
floating-point, 385
icc field, 64
integer, 64
results of integer operation (icc), 65
subtracting, 370, 373
trapping on, 375
xcc field, 64
condition codes (CCR) register, 27, 64, 80, 121, 129, 224, 249, 381, 400, 601
conditional branches, 241, 386, 389
conditional move instructions, 27
conforming SPARC V9 implementation, **7**
const22 field of instructions, 271
constants, generating, 353
context
during TLB miss, 475, 476
selection for translation, 483
unused, 470
used to form TSB Tag Target, 477
context field of Tag Access Register, *See* Tag Access Register
context register
determination of, 483
Nucleus, 490
Primary, 489
Secondary, 489
context-ID register, 476
control-transfer instruction, *See* CTI
control-transfer instructions (CTIs), 25, 249
conventions
font, 3
notational, 3
software, 517

conversion
between floating-point formats instructions, 259, 420
floating-point to integer instructions, 257, 423
integer to floating-point instructions, 261
pixel to fixed, 38
planar to packed, 331
Core Available register, 197
Core Enable register, 198
state after reset, 199
updating, 199
Core Enable Status register, 198
state after reset, 198
Core ID register, 191
Core Interrupt ID register, 192
Core Running register, 200
accessing, 200
restrictions on updating, 201
state after reset, 202
updating, 201
Core Running Status register, 203
counter field of STICK register, **70**
counter field of TICK register, **66**
CPI (cycles per instruction), 10
cross-call, 10
cross-domain call, **535**
CTI
delayed, 99
immediate, 99
CTI, *See* control transfer instructions
current exception (cexc) field of FSR register, 54, 57, 58, **59**, 59, 60, 122, 164, 420
current window, 10
current window pointer (CWP) register
and CALL / JMPL instructions, 45
and clean windows, 86, 126
definition, 10
and FLUSHW instruction, 270
function, 84
incremented / decremented, 44, 349
and overlapping windows, 44
range of values, 83
reading CWP with RDPR instruction, 341
and RESTORE instruction, 118, 348
restored during DONE or RETRY, 249
and SAVE instruction, 118, 348
saved during a trap, 129
after spill trap, 127
state after reset, 601
after trap, 28
and TSTATE Register, 80
on window trap, 127
writing CWP with WRPR instruction, 377
current_little_endian (cle) field of PSTATE register, **74**, 74, 174

## D

D superscript on instruction name, **218**
d16hi field of instructions, **103**, 238
d16lo field of instructions, **103**, 238
data
aggregate
argument passed by value, 518
examples of, 518
breakpoint, *See* watchpoints
cache
coherence, 177
and RED_state, 597
fixed-to-pixel conversion, 38
flow order constraints
memory reference instructions, **176**
register reference instructions, **176**
formats
byte, **29**
doubleword, **29**
extended word, **29**
halfword, **29**
quadword, **29**
tagged word, **29**
word, **29**
memory, 183
MMU, *See* D-MMU
pixel-to-fixed conversion, 38
types
floating-point, 29
signed integer, 29
unsigned integer, 29
width, 29
watchpoint
behavior, 95
exception, 314
physical address, 96
register format, 95
virtual address, 95
Data Cache Enable bit, 94
Data Cache Unit Control Register, *See* DCUCR
Data Synchronous Fault Address Register, *See* D-SFAR

Data Synchronous Fault Status Register, *See* D-SFSR
*data_access_error* exception, 234, 248, 275, 280, 284, 285, 287, 314, 357, 361, 365, 367, 369, 395, 405, 407, 409, 614, 616
*data_access_exception* exception, 74, **161**, 166, 248, 275, 278, 285, 287, 361, 364, 367, 369, 407, 409, 411, 413, 473, 478, 479, 480, 486, 488, 498, 581, 582, 583, 592, 593
*data_access_MMU_miss* exception, 431, 479, 480
*data_access_protection* exception, 234, 280, 282, 284, 314, 357, 395, 397, 479, 480, 497
db_pa field of PA Data Watchpoint register, 96
DCR
branch and return control, 87
fields
BPE (branch prediction enable), 87
MS (multiscalar dispatch enable), 88
RPE (return address prediction enable), 87
SI (single issue disable), 87
instruction dispatch control, 87
layout, 87
RDASR/WRASR support, 121
state after reset and in RED_state, 602
DCTI, **10**, 63
DCUCR
access data format, 92
clearing, 597
cp (cacheability) field, 92
cv (cacheability) field, 92
dc (data cache enable) field, 94
dm (DMMU enable) field, 93
ic (instruction cache enable) field, 94
im (IMMU enable) field, 93
overriding enable bits, 597
pm (PA data watchpoint mask) field, 92
pr/pw (PA watchpoint enable) fields, 93
RED_state, 485
after reset, 602
vm (VA data watchpoint mask) field, 93
vr/vw (VA data watchpoint enable) fields, 93
watchpoint byte masks/enable bits, 95
DEC synthetic instruction, **516**
DECcc synthetic instruction, **516**
deferred trap
catastrophic error exception, 429
error barrier, 613
floating-point, 342
handling of, 613
Impl. Dep., 135
occurrence, 135
processor state corruption, 612
queue, floating-point (FQ), 341
software actions, 135
TPC/TNPC, 613
vs. disrupting trap, 135
Dekker's algorithm, 545
delay instruction, 26, 62, 238, 241, 244, 249, 346, 385, 518, 522
delayed branch, 100
delayed control transfer, 62, 238
delayed CTI, 99
demap, 10
demap operation and output, 502
denormal, **10**
deprecated, 10
deprecated instructions
BA, 387
BCC, 387
BCS, 387
BE, 387
BG, 387
BGE, 387
BGU, 387
Bicc, 387
BLE, 387
BLEU, 387
BN, 387
BNE, 387
BNEG, 387
BPOS, 387
BVC, 387
BVS, 387
FBA, 384
FBE, 384
FBG, 384
FBGE, 384
FBL, 384
FBLE, 384
FBLG, 384
FBN, 384
FBNE, 384
FBO, 384
FBU, 384
FBUE, 384
FBUGE, 384
FBUL, 384
FBULE, 384
LDD, 394

LDDA, 396
LDFSR, 393
MULScc, 63, 400
RDY, 63, 343, 402
SDIV, 63, 390
SDIVcc, 63, 390
SMUL, 63, 398
SMULcc, 63, 398
STD, 406
STDA, 408
STFSR, 404
SWAP, 410
SWAPA, 412
TSUBccTV, 414, 416
UDIV, 63, 390
UDIVcc, 63, 390
UMUL, 63, 398
UMULcc, 63, 398
WRY, 63, 380, 418
Direct Pointer Register, 497
disable (core), 10
disabled (core), 10
disp19 field of instructions, **103**, 241, 242
disp22 field of instructions, **103**, 385, 388
disp30 field of instructions, **103**, 245
dispatch, 10
Dispatch Control Register, *See* DCR
disrupting traps, 135, 136, 429
divide instructions, 25, 311, **390**
divide-by-zero mask (dzm) bit of tem field of FSR register, **60**
*division_by_zero* exception, 112, **161**, 311
division-by-zero accrued (dza) bit of aexc field of FSR register, **61**
division-by-zero current (dzc) bit of cexc field of FSR register, **61**
D-MMU
and RED_state, 597
context register usage, 485
determining ASI value and endianness, 483
Direct Pointer register, 497
disabled, 486
Enable bit, 485
enable bits, 485
memory operation summary, 481
Nucleus Context Register, 490
Registers:Primary, Secondary, Nucleus, 489
Secondary Context Register, 490
D-MMU Tag Access Register
context field after *data_access_exception*, 479
DONE instruction, 65, 73, 80, 81, 129, 131, 162, **249**, 440
modifying condition codes, 65
restoring ag, ig, mg bits, 73
target address, 26
doubleword
addressing, 108
alignment, 106, 173
data format, **29**
definition, 11
in memory, **46**
D-SFAR
defined, 500
description, 488
error logging, 617
state after reset, 602
D-SFSR
and ASI operations, 488
bit description, 497
error logging, 617
ftype field upon data_access_exception, 161
state after reset, 602

## E

*ECC_error* exception, 166, 616
EDGE16 instruction, 250
EDGE16L instruction, 250
EDGE16LN instruction, 250
EDGE16N instruction, 250
EDGE32 instruction, 250
EDGE32L instruction, 250
EDGE32LN instruction, 250
EDGE32N instruction, 250
EDGE8 instruction, 250
EDGE8L instruction, 250
EDGE8LN instruction, 250
EDGE8N instruction, 250
emulating multiple unsigned condition codes, 118
enable (core), 11
enable floating-point (fef) field of FPRS register, **65**, 76, 122, 138, 161, 241, 275, 277, 361, 364, 386
enable floating-point (pef) field of PSTATE register, 65, **76**, 122, 138, 161, 241, 275, 277, 361, 364, 386, 526
enable RED_state field (red) of PSTATE register, 131

enabled (core), 11
Error Steering register, 209
error_state, 130, 131, 133, 134, 146, 147, 148, 150, 151, 159, 430
  and watchdog reset, 599
errors
  deferred, 612, 613
  disrupting, 615
  hardware-corrected, 616
  logging, 616
  recoverable ECC errors, 616
  uncorrectable, 616
exceptions, 11
  *See also* trap *and* traps
  *async_data_error*, 275, 282, 285, 397
  *catastrophic_error*, 130
  causing traps, 129
  *clean_window*, 86, 118, 127, 137, 140, **164**, 349, 350, 431
  *data_access_error*, 234, 248, 275, 280, 284, 285, 287, 314, 357, 361, 365, 367, 369, 395, 404, 407, 409
  *data_access_exception*, **161**, 248, 275, 278, 285, 287, 361, 364, 367, 369, 407, 409, 411, 413, 582, 583
  *data_access_MMU_miss*, 431
  *data_access_protection*, 234, 280, 282, 284, 314, 357, 395, 397
  definition, 129
  *division_by_zero*, 112, **161**, 311
  *fill_n_normal*, 137, 347, 350
  *fill_n_other*, 137, 347, 350
  *fp_disabled*, 65, 122, 137, **161**, 241, 254, 258, 260, 262, 264, 266, 275, 277, 301, 303, 308, 361, 364, 386, 404, 526
  *fp_exception_ieee_754*, 54, 58, 59, 60, 139, 164, 254, 258, 260, 262, 266, 420
  *fp_exception_other*, 52, 123, 164, 254, 256, 258, 260, 262, 264, 266, 267, 303
  *illegal_instruction*, 46, 78, 80, 124, **161**, 238, 244, 249, 271, 275, 308, 310, 333, 342, 344, 351, 361, 376, 379, 394, 395, 396, 397, 404, 406, 407, 408, 409, 430, 582
  *implementation_dependent_n*, 141, 429
  *instruction_access_error*, 137
  *instruction_access_exception*, 137, **162**
  *internal_processor_error*, 165
  *LDDF_mem_address_not_aligned*, 106, 137, **164**, 277, 432, 582, 583
  *mem_address_not_aligned*, 106, **163**, 248, 273, 275, 277, 280, 282, 314, 346, 347, 361, 364, 367, 369, 393, 395, 397, 404, 407, 409, 411, 413, 582, 583
  pending, 28
  *privileged_action*, 66, 71, 110, 137, **163**, 248, 277, 282, 286, 287, 344, 345, 364, 369, 397, 408, 409, 413
  privileged_action, 110
  *privileged_instruction* (SPARC V8), 163
  *privileged_opcode*, 137, **163**, 249, 342, 351, 379
  *spill_n_normal*, 137, **163**, 270, 350
  *spill_n_other*, 137, **163**, 270, 350
  *STDF_mem_address_not_aligned*, 106, 137, **164**, **166**, 361, 364, 432
  *tag_overflow*, 112, **163**, 372, 373, 414, 415, 417
  *trap_instruction*, 137, **163**, 375, 376
  *unimplemented_LDD*, 395, 432
  *unimplemented_STD*, 407, 432
  *window_fill*, 85, 118, 346, 521
  *window_spill*, 85, 521
execute unit, 175
execute_state, 130, 146, 147, 148, 150, 151
extended word, 11
extended word addressing, 108
extended word data format, **29**
extensions, architectural, 539
External Reset pin, 598
  *externally_initiated_reset* (XIR), 131, 132, 133, 137, 153, 156, 164, 598

## F

f registers, 11
f registers, 22, 139, 419, 430
FABSd instruction, **263**, 460, 461
FABSq instruction, **263**, 460, 461
FABSs instruction, **263**
FADDd instruction, **253**
FADDq instruction, **253**
FADDs instruction, **253**
FALIGNDATA instruction, 226
FAND instruction, 288
FANDNOT1 instruction, 288
FANDNOT1S instruction, 288
FANDNOT2 instruction, 289
FANDNOT2S instruction, 289
FANDS instruction, 288
  *fast_data_access_MMU_miss* exception, 73, 74, 475, 478, 479, 480, 499

*fast_data_access_protection* exception, 73, 74, 473, 478, 479, 480
*fast_ECC_error* exception, 167, 617
*fast_instruction_access_MMU_miss* exception, 73, 74, 475, 478, 479, 498, 500
*fast_instruction_MMU_miss* exception, 499
FBA instruction, 384, 385, 462
FBE instruction, 384, 462
FBfcc instructions, 53, 122, 161, **384**, 386, 457, 462
FBG instruction, 384, 462
FBGE instruction, 384, 462
FBL instruction, 384, 462
FBLE instruction, 384, 462
FBLG instruction, 384, 462
FBN instruction, 384, 385, 462
FBNE instruction, 384, 462
FBO instruction, 384, 462
FBPA instruction, 239, 241, 462
FBPcc instructions, 103
FBPE instruction, 239, 462
FBPfcc instructions, 53, 102, 104, 122, **239**, 386, 457, 462, 464
FBPG instruction, 239, 462
FBPGE instruction, 239, 462
FBPL instruction, 239, 462
FBPLE instruction, 239, 462
FBPLG instruction, 239, 462
FBPN instruction, 239, 241, 462
FBPNE instruction, 239, 462
FBPO instruction, 239, 462
FBPU instruction, 239, 462
FBPUE instruction, 239, 462
FBPUG instruction, 239, 462
FBPUGE instruction, 239, 462
FBPUL instruction, 239, 462
FBPULE instruction, 239, 462
FBU instruction, 384, 462
FBUE instruction, 384, 462
FBUG instruction, 384, 462
FBUGE instruction, 384, 462
FBUL instruction, 384, 462
FBULE instruction, 384, 462
fcc-conditional branches, 241, 386
fccN, 11
FCMP instructions, 464
FCMP* instructions, 53, 54, 255
FCMPd instruction, **255**, 421, 461
FCMPE instructions, 464
FCMPE* instructions, 53, 54, 255
FCMPEd instruction, **255**, 421, 461
FCMPEQ instruction, 324
FCMPEq instruction, **255**, 421, 461
FCMPEQ16 instruction, 323
FCMPEQ32 instruction, 323
FCMPEs instruction, **255**, 421, 461
FCMPG instruction, 324
FCMPGT16 instruction, 323
FCMPGT32 instruction, 323
FCMPL instruction, 324
FCMPLE16 instruction, 323
FCMPLE32 instruction, 323
FCMPNE instruction, 324
FCMPNE16 instruction, 323
FCMPNE32 instruction, 323
FCMPq instruction, **255**, 421, 461
FCMPs instruction, **255**, 421, 461
fcn field of instructions, 249, 334
FDIVd instruction, **265**
FDIVq instruction, **265**
FDIVs instruction, **265**
FdMULq instruction, **265**
FdTOi instruction, **257**, 423
FdTOq instruction, **259**, 420
FdTOs instruction, **259**, 420
FdTOx instruction, **257**, 460, 461
FEXPAND instruction, 326, 330
FEXPAND operation, 330
fill register window, 45, 118, 120, 125, 126, 127, 128, 348, 349, 351, 534
*fill_n_normal* exception, 137, 161, 347, 350
*fill_n_other* exception, 137, 161, 347, 350
FiTOd instruction, **261**
FiTOq instruction, **261**
FiTOs instruction, **261**
fixed-point scaling, 318
floating point complex calculations, 37
floating-point add and subtract instructions, **253**
floating-point compare instructions, 53, 54, **255**, 255, 421
floating-point condition code bits, 385
floating-point condition codes (fcc) fields of FSR register, **53**, 53, 57, 139, 241, 256, 386, 420, **507**
floating-point data type, 29
floating-point deferred-trap queue (FQ), 341, 342
floating-point enable (fef) field of FPRS register, 526
floating-point exception, 11, 56
floating-point move instructions, **263**
floating-point multiply and divide instructions, **265**

floating-point operate (FPop) instructions, 27, 56, 59, 103, 122, 161, 162, 164, 393
floating-point registers, 52, 419, 430, 520
floating-point registers state (FPRS) register, **65**, 121, 344, 381, 601
floating-point square root instructions, **267**
floating-point state (FSR) register, **53**, 58, 59, 62, 360, 393, 404, 419, 601
floating-point trap type (ftt) field of FSR register, 59
floating-point trap type (ftt) field of FSR register, 53, **56**, 58, 122, 164, 360, 404, 420
floating-point trap types, 11
  fp_disabled, 76
  FPop_unfinished, 123
  FPop_unimplemented, 123
  IEEE_754_exception, 57, **57**, 58, 59, 62, 420
  *IEEE_754_exception*, 139, 164
  invalid_fp_register, 52
  numeric values, **56**
  unfinished_FPop, 57, **57**, 62, 266, 420
  unimplemented_FPop, 57, 62, 254, 256, 260, 262, 266, 301, 303, 420
  *unimplemented_FPop*, 258
floating-point traps
  deferred, 342
  precise, 342
floating-point unit, 11
floating-point unit (FPU), **22**
FLUSH instruction, 183, 201, **268**, 433, 525, 542
flush instruction memory, *See* FLUSH instruction
FLUSH latency, 433
flush register windows instruction, **270**
FLUSHW instruction, 27, 120, 126, 127, 163, **270**, 519
FMOVA instruction, 296
FMOVcc instruction, 296
FMOVcc instructions, 53, 65, 102, 103, 117, 122, **296**, 300, 301, 308, 463
FMOVccd instruction, 461
FMOVccq instruction, 461
FMOVccs instruction, 461
FMOVCS instruction, 296
FMOVd instruction, **263**, 460, 461
FMOVDcc instruction, 298
FMOVE instruction, 296
FMOVFA instruction, 297
FMOVFE instruction, 297
FMOVFG instruction, 297
FMOVFGE instruction, 297
FMOVFL instruction, 297
FMOVFLE instruction, 297
FMOVFLG instruction, 297
FMOVFN instruction, 297
FMOVFNE instruction, 297
FMOVFO instruction, 297
FMOVFU instruction, 297
FMOVFUE instruction, 297
FMOVFUG instruction, 297
FMOVFUGE instruction, 297
FMOVFUL instruction, 297
FMOVFULE instruction, 297
FMOVG instruction, 296
FMOVGE instruction, 296
FMOVGU instruction, 296
FMOVL instruction, 296
FMOVLE instruction, 296
FMOVLEU instruction, 296
FMOVN instruction, 296
FMOVNE instruction, 296
FMOVNEG instruction, 296
FMOVPOS instruction, 296
FMOVq instruction, **263**, 460, 461
FMOVQcc instruction, 298
FMOVr instructions, 103, 104, 122, **302**, 463
FMOVRGEZ instruction, **302**
FMOVRGZ instruction, **302**
FMOVRLEZ instruction, **302**
FMOVRLZ instruction, **302**
FMOVRNZ instruction, **302**
FMOVRZ instruction, **302**
FMOVs instruction, **263**
FMOVScc instruction, 298
FMOVVC instruction, 296
FMOVVS instruction, 296
FMUL8SUx16 instruction, 317, 320
FMUL8ULx16 instruction, 317, 320
FMUL8x16 instruction, 317, 318
FMUL8x16AL instruction, 317, 319
FMUL8x16AU instruction, 317, 319
FMULd instruction, **265**
FMULD8SUx16 instruction, 317, 321
FMULD8ULx16 instruction, 317, 322
FMULq instruction, **265**
FMULs instruction, **265**
FNAND instruction, 288
FNANDS instruction, 288
FNEGd instruction, **263**, 460, 461
FNEGq instruction, **263**, 460, 461

FNEGs instruction, **263**
FNOR instruction, 288
FNORS instruction, 288
FNOT1 instruction, 288
FNOT1S instruction, 288
FNOT2 instruction, 288
FNOT2S instruction, 288
FONE instruction, 288
FONES instruction, 288
FOR instruction, 288
formats, instruction, 100
FORNOT1 instruction, 288
FORNOT1S instruction, 288
FORNOT2 instruction, 288
FORNOT2S instruction, 288
FORS instruction, 288
*fp_disabled* exception, 65, 76, 122, 137, **161**, 241, 254, 258, 260, 262, 264, 266, 275, 277, 301, 303, 308, 357, 361, 364, 386, 404, 526
*fp_exception* exception, 58
*fp_exception_ieee_754* "invalid" exception, 258
*fp_exception_ieee_754* exception, 54, 58, 59, 60, 139, **164**, 254, 258, 260, 262, 266, 420
*fp_exception_other* exception, 52, 57, 122, 123, 162, **164**, 218, 254, 256, 258, 260, 262, 264, 266, 267, 303, 420, 428
FPACK instruction, 70
FPACK instructions, 38, 326 to 330
FPACK16 instruction, 326, 327
FPACK16 operation, 327
FPACK32 instruction, 326, 328
FPACK32 operation, 328
FPACKFIX instruction, 326, 329
FPACKFIX instruction, conversion, 38
FPACKFIX operation, 330
FPADD16 instruction, 315
FPADD16S instruction, 315
FPADD32 instruction, 315
FPADD32S instruction, 315
FPMERGE instruction, 326, 331
FPop, 11
FPop, *See* floating-point operate (FPop) instructions
FPRS register
  *See also* floating-point registers state (FPRS) register
  description, 65
  fef field, 381
FPSUB16 instruction, 315
FPSUB16S instruction, 315
FPSUB32 instruction, 315
FPSUB32S instruction, 315
FqTOd instruction, **259**, 420
FqTOi instruction, **257**, 423
FqTOs instruction, **259**, 420
FqTOx instruction, **257**, 460, 461
frame pointer register, 518
freg, **506**
FsMULd instruction, **265**
FSQRTd instruction, **267**
FSQRTq instruction, **267**
FSQRTs instruction, **267**
FSR, *See* floating-point state (FSR) register
FSRC1 instruction, 288
FSRC1S instruction, 288
FSRC2 instruction, 288
FSRC2S instruction, 288
FsTOd instruction, **259**, 420
FsTOi instruction, **257**, 423
FsTOq instruction, **259**, 420
FsTOx instruction, **257**, 460, 461
FSUBd instruction, **253**
FSUBq instruction, **253**
FSUBs instruction, **253**
function return value, 518
functional choice, implementation-dependent, 427
FXNOR instruction, 288
FXNORS instruction, 288
FXOR instruction, 288
FXORS instruction, 288
FxTOd instruction, **261**, 460, 461
FxTOq instruction, **261**, 460, 461
FxTOs instruction, **261**, 460, 461
FZERO instruction, 288
FZEROS instruction, 288

## G

generating constants, 353
global registers, 5, 22, 41, **42**, 42, 519
graphics data format
  8-bit, 37
  fixed 16-bit, 37
Graphics Status Register, *See* GSR
GSR
  fields
    align, 70
    gcc, 70
    im (interval mode) field, 70

irnd (rounding), 70
mask, 70
scale, 70
RDASR / WRASR support, 121
state after reset, 602

## H

halfword, 11
addressing, 108
alignment, 106, 173
data format, **29**
halt, 146
hardware
dependency, **426**
table walking, 478
TLB, 503
traps, 141
hardware-corrected errors, 616

## I

i field of instructions, **103**, 224, 268, 270, 273, 274, 276, 279, 285, 286, 292, 306, 309, 311, 332, 334, 344, 346, 390, 393, 394, 396, 398, 400, 402
I / D
MMU Demap Operation, 489
MMU TLB Tag Access Registers, 491
MMU TSB Pointer register, 497
I / D TSB Tag Target registers, 488
icc field of CCR register, 64, **65**, 224, 244, 292, 308, 370, 372, 375, 389, 391, 392, 399, 400, 401
icc-conditional branches, 389
IE, Invert Endianness bit, 471
IEEE Std 1596.5-1992, 21
IEEE Std 754-1985, 12, 21, 54, 57, 60, 62, 122, 419, 428
IEEE_754_exception, 57
IEEE_754_exception floating-point trap type, 57, 58, 62, 420
*IEEE_754_exception* floating-point trap type, 12, 139, 164
IEEE_754_exception> floating-point trap type, **57**
IER register (SPARC V8), 382
*illegal_instruction* exception, 46, 78, 80, 123, **161**, 218, 238, 244, 249, 271, 275, 308, 310, 333, 340, 342, 344, 351, 361, 376, 379, 394, 395, 396, 397, 404, 406, 407, 408, 409, 430, 582
ILLTRAP instruction, 161, **271**
images
band interleaved, 37
band sequential, 37
imm_asi field of instructions, **103**, 246, 274, 276, 279, 285, 286, 334, 393, 394, 396
imm22 field of instructions, **103**
immediate CTI, 99
I-MMU
context register usage, 485
determining ASI value and endianness, 483
disabled, 486
Enable bit, 93, 485
enable bits, 485
memory operation summary, 481
Registers: Primary, Secondary, Nucleus, 489
IMPDEP1 instructions, 272, 465
IMPDEP2A instructions, 123, 162, 272, 432, 539
IMPDEP2B instructions, 123, 162, 272
impl field of VER register, 56
implementation, 12
implementation dependency, **425**
implementation dependent, 12
implementation note, **4**
implementation number (impl) field of VER register, **83**
*implementation_dependent_n* exception, 141, 429
implementation-dependent functional choice, 427
implementation-dependent instructions, *See* IMPDEP2A instructions
implicit
byte order, 74
implicit ASI, 12
in registers, 44, 348, 518
INC synthetic instruction, **516**
INCcc synthetic instruction, **516**
inexact accrued (nxa) bit of aexc field of FSR register, **61**, 423
inexact current (nxc) bit of cexc field of FSR register, **61**, 423
inexact mask (nxm) bit of tem field of FSR register, **60**
inexact quotient, 391, 392
infinity, 423, 424
informative appendix, 12
initiated, **12**
input / output (I / O) locations
access by nonprivileged code, 428
addressing by primitives, 182
behavior, 172

contents and addresses, 428
identifying, 172, 433
order, 172
semantics, 433
value semantics, 172
input/output (I/O) register access, 24
instruction breakpoint, *See* Instruction Trap Register
instruction cache
disabled in RED_state, 597
Instruction Cache Enable bit, 94
instruction fields, 12
a, **102**, 238, 242, 246, 385, 388
cc0, **102**, 241, 242, 255, 306
cc1, **102**, 241, 242, 255, 306
cc2, **102**, 306
cond, **103**, 241, 242, 298, 306, 385, 388
const22, 271
d16hi, **103**, 238
d16lo, **103**, 238
disp19, **103**, 241, 242
disp22, **103**, 385, 388
disp30, **103**, 245
fcn, 249, 334
i, **103**, 224, 268, 270, 273, 274, 276, 279, 285, 286, 292, 306, 309, 311, 332, 334, 344, 346, 390, 393, 394, 396, 398, 400, 402
imm_asi, **103**, 246, 274, 276, 279, 334, 393, 394, 396
imm22, **103**
mmask, **103**, 403
op3, **103**, 224, 246, 249, 268, 270, 273, 274, 276, 279, 285, 286, 292, 311, 334, 341, 344, 346, 390, 393, 394, 396, 398, 400, 402
opf, **103**, 253, 255, 257, 259, 261, 263, 265, 267
opf_cc, **103**, 298
opf_low, **103**, 298, 302
p, **104**, 238, 241, 242
rcond, **104**, 238, 302, 309
rd, **104**, 224, 246, 253, 257, 259, 261, 263, 265, 267, 273, 274, 276, 279, 285, 286, 292, 298, 302, 306, 309, 311, 332, 341, 344, 390, 393, 394, 396, 398, 400, 402, 539
reg_or_imm, 539
reserved, 217
rs1, **104**, 224, 238, 246, 253, 255, 265, 268, 273, 274, 276, 279, 285, 286, 292, 302, 309, 311, 334, 341, 344, 346, 390, 393, 394, 396, 398, 400, 402, 539
rs2, **104**, 224, 246, 253, 255, 257, 259, 261, 263, 265, 267, 268, 273, 274, 276, 279, 285, 286, 292, 298, 302, 306, 309, 311, 332, 334, 346, 390, 393, 394, 396, 398, 400
shcnt32, **104**
shcnt64, **104**
simm10, **104**, 309
simm11, **104**, 306
simm13, **104**, 224, 268, 273, 274, 276, 279, 281, 285, 286, 292, 311, 332, 334, 346, 390, 393, 394, 396, 398, 400
sw_trap#, **104**
x, **104**
instruction group, 12
instruction MMU, *See* I-MMU
instruction set architecture (ISA), xv, 6, 12, 12
Instruction Synchronous Fault Status Register, *See* I-SFSR
Instruction Trap Register
exception, 96, 97
Mask field, 96
match field, 96
store, 97
*instruction_access_error* (ISA) exception, 616, 617
*instruction_access_error* exception, 137
*instruction_access_exception* (ISA) exception, 137
*instruction_access_exception* exception, 74, **162**, 473, 478, 479, 486, 498, 500
*instruction_access_MMU_miss* exception, 479
instructions, 62
alignment, 173, 226
alignment, 106
array addressing, 228
atomic, 247
atomic load-store, 105, 246, 247, 285, 286, 410, 412
block load and store, 232
branch if contents of integer register match condition, **237**
branch on floating-point condition codes, **384**
branch on floating-point condition codes with prediction, **239**
branch on integer condition codes, **387**
branch on integer condition codes with prediction, **242**
breakpoint trap priority, 96
cache, 177
cache consistency, 177
causing illegal instruction, 271
compare and swap, **246**
comparison, 112, 371
conditional move, 27
control-transfer (CTIs), 25, 249

convert between floating-point formats, **259**, 420
convert floating-point to integer, **257**, 423
convert integer to floating-point, **261**
count of number of bits, 332
divide, 25, 311, **390**
DONE, 73, **249**
edge handling, 251
fetches, 106
floating-point add and subtract, **253**
floating-point compare, 53, 54, **255**, 255, 421
floating-point move, **263**
floating-point multiply and divide, **265**
floating-point operate (FPop), 27, 56, 59, 393
floating-point square root, **267**
flush instruction memory, **268**, 542
flush register windows, **270**
formats, 5, **100**
generate software-initiated reset, 359
implementation-dependent, *See* IMPDEP2A instructions
jump and link, 26, **273**
load, 542
load floating-point, 105, **274**, 393
load floating-point from alternate space, **276**
load integer, 105, **279**, **394**
load integer from alternate space, **281**, **396**
load quadword atomic, 283, 397, 581
load-store unsigned byte, 247, **285**, 410, 412
load-store unsigned byte to alternate space, **286**
logical, **291**
logical operate, 290
memory, 183
move floating-point register if condition is true, **296**
move floating-point register if contents of integer register satisfy condition, **302**
move integer register if condition is satisfied, **304**
move integer register if contents of integer register satisfies condition, **309**
move on condition, 5
multiply, 25, 311, **398**, 398
ordering MEMBAR, 111
partial store, 314
partitioned add/subtract, 316
partitioned multiply, 317
permuting bytes specified by GSR.MASK, 235
pixel compare, 323
pixel component distance, 325
pixel formatting (PACK), 326
prefetch data, **334**
read privileged register, **341**
read state register, 26, 62, 86, **343**, **402**
register window management, 27
reordering, 175
reserved, 123
reserved fields, 217
RETRY, 73, **249**
RETURN vs. RESTORE, 346
sequencing MEMBAR, 111
set high bits of low word, 353
set interval arithmetic mode, 352
setting GSR.mask field, 235
shift, 25, **354**
shift count, 354
short floating-point load/store, 357
shut down to enter power-down mode, 358
software-initiated reset, 359
store, 366, 542
store floating point, 105, 360
store floating-point into alternate space, **363**, 363
store integer, 105, 366
store integer into alternate space, 368
subtract, **370**, 370
swap r register with alternate space memory, **412**
swap r register with memory, **410**
synthetic, 514
tagged addition, 372
tagged arithmetic, 25
tagged subtraction, 373
test-and-set, 183
timing, 218
trap on condition codes, 375
trap on integer condition codes, **374**
trap register, 96, 97
unimplemented, 124
write privileged register, **377**
write state register, 86, 381
writing privileged register, 378
WRPR, 440
integer unit (IU), 12
condition codes, 64
description, 22
interconnect configuration ASI ($4A_{16}$), 571
*internal_processor_error* exception, 130, 165
interrupt
enable (ie) field of PSTATE register, **77**, 136, 138, 162

level, 78
request, 12, 28, 129
trap, 589
vector dispatch, 588
vector dispatch register, 591
vector dispatch status register, 592, 602
vector receive, 589
vector receive register, 593, 602
interrupt target identifier (ID), *See* itid field of Interrupt Vector Dispatch register
Interrupt Vector Dispatch Extension register, 194
*interrupt_vector* exception, 73, 167
INTR_DISPATCH, *See* Interrupt Vector Dispatch Status register
INTR_RECEIVE, *See* Interrupt Vector Receive register
invalid accrued (nva) bit of aexc field of FSR register, **61**
invalid current (nvc) bit of cexc field of FSR register, **61**
invalid current (nvc) bit of cexc field of FSR register, 423, 424
invalid mask (nvm) bit of tem field of FSR register, **60**
*invalid_exception* exception, 258
invalid_fp_register floating-point trap type, 52
IPREFETCH synthetic instruction, **514**
ISA, *See* instruction set architecture
I-SFSR
and ASI operations, 488
bit description, 497
error logging, 617
nf field always 0, 498
state after reset, 602
issue unit, 175, **175**
issued, 12
italic font, in assembly language syntax, 505
itid field of Interrupt Vector Dispatch register, 589, 591

## J

JMP synthetic instruction, **514**
JMPL instruction
computing target address, 26
description, 273
does not change CWP, 45
leaf procedure, 521
mapping to synthetic instructions, 514
*mem_address_not_aligned* exception, 163
reexecuting trapped instruction, 346
jump and link (JMPL) instruction, 26, **273**

## K

kernel code, 594

## L

LD instruction (SPARC V8), 280
LDD instruction, 46, **394**, 432
LDDA instruction, 46, 283, **396**, 581
LDDF instruction, 52, 106, 164, **274**
*LDDF_mem_address_not_aligned* exception, 106, 137, **164**, 277, 432, 582, 583
LDDFA instruction, 52, 106, 190, 231, **276**, 314, 356, 582, 583
LDF instruction, **274**
LDFA instruction, **276**
LDFSR instruction, 53, 56, 162, **393**
LDQF instruction, 52, 124, **274**
*LDQF_mem_address_not_aligned* exception, 166
LDQFA instruction, 52, **276**
LDSB instruction, **279**, **394**
LDSBA instruction, **281**
LDSH instruction, **279**, **394**
LDSHA instruction, **281**
LDSTUB instruction, 105, 179, 183, **285**, 286, 546
LDSTUBA instruction, 285, **286**
LDSW instruction, **279**, **394**
LDSWA instruction, **281**
LDUB instruction, **279**, **394**
LDUBA instruction, **281**
LDUH instruction, **279**, **394**
LDUHA instruction, **281**
LDUW instruction, **279**, **394**
LDUWA instruction, **281**
LDX instruction, **279**, **394**
LDXA instruction, 190, **281**
LDXFSR instruction, 53, 56, 162, **274**, 393
leaf procedure, 13
description, 521
modifying windowed registers, 119
optimization, 521, 523
space allocation, 521
Level I compliance (SPARC V9), 8
Level II compliance (SPARC V9), 8
little-endian byte order, **13**, 23, 74

load
block, *See* block load instructions
floating-point from alternate space instructions, **276**
floating-point instructions, **274**, 393
instructions, 542
integer from alternate space instructions, **281**, **396**
integer instructions, **279**, **394**
quadword atomic, 283, 397, 581
short floating-point, *See* short floating-point load instructions
load instructions, 13, 105
LoadLoad MEMBAR relationship, 179, 294
LoadLoad predefined constant, 512
loads
from alternate space, 24, 66, 110, 564
nonfaulting, 174, **175**
load-store alignment, 106, 173
load-store instructions, 23
compare and swap, **246**
definition, 13
and *fast_data_access_protection* exception, 166
load-store unsigned byte, 247, **285**, 410, 412
load-store unsigned byte to alternate space, **286**
swap r register with alternate space memory, **412**
swap r register with memory, 247, **410**
LoadStore MEMBAR relationship, 179, 294
LoadStore predefined constant, 512
local registers, 44, 348, 519, 523
logical instructions, **291**
Lookaside MEMBAR relationship, 294
Lookaside predefined constant, 512
lower registers dirty (dl) field of FPRS register, **66**

## M

machine state
after reset, 600
in RED_state, 600
manufacturer (manuf) field of VER register, 83
manufacturer (manuf) field of VER register, 431
mask number (mask) field of VER register, 83
MAXTL, 77, 131, 133, 148, 150, 151, 359
for SPARC JPS2, 77
may (keyword), 13
*mem_address_not_aligned* exception, 106, **163**, 248, 273, 275, 277, 280, 282, 314, 346, 347, 357, 361, 364, 367, 369, 393, 395, 397, 404, 407, 409, 411, 413, 478, 479, 481, 488, 498, 581, 582, 583
MEMBAR
#LoadLoad, 179, 294, 512
#LoadStore, 179, 294, 512
#StoreLoad, 179, 294, 512
#StoreStore, 179, 294, 512
#Sync, 488, 500, 614
error isolation, 613
instruction, 103, 111, 172, 177, 178 to 180, 181, 183, 268, **293**, 344, 403, 542, 590
membar_mask, **512**
MemIssue MEMBAR relationship, 294
MemIssue predefined constant, 512
memory
access instructions, 23, 105
alignment, 173
atomic operations, 182
atomicity, 433
coherence, 171, 172, 433
coherency unit, 173
data, 183
instruction, 183
models, **169**
ordering unit, 173
real, 172
reference instructions, data flow order constraints, 176
stack layout, 520
Memory Management Unit (MMU), 6, 13, 467
memory model, 180 to 184
barrier synchronization, 554
Dekker's algorithm, 545
issuing order, 550
mode control, 181
mutex (mutual exclusion) locks, 544
operations, 541
partial store order (PSO), **170**, **181**, 433, 541
portability and recommended programming style, 543
processors and processes, 542
relaxed memory order (RMO), **170**, **180**, 433, 541
sequential consistency, **171**
SPARC V9, 180
spin lock, 545
strong, 171
strong consistency, 171, 544, 550
total store order (TSO), **170**, **181**, 182, 541
weak, 170
memory order
pending transactions, 178

program order, 175
SPARC V9, 6
memory_model (mm) field of PSTATE register, **74**, 177, 181, 182, 433
microkernel, 537
mmask field of instructions, **103**, 403
MMU
accessing registers, 488
behavior during reset, 485
bypass, 503
bypass mode, 567
D Synchronous Fault Address Register, 488
D TSB Secondary Extension Registers, 489
demap, 500
all, 501
context, 501, 502
operation syntax, 501
page, 501, 502
disable, 485
global registers, 478
I/D Synchronous Fault Status Registers, 488, 497
I/D TLB Data Access Registers, 489
I/D TLB Data In Registers, 489
I/D TLB Tag Access register, 488
I/D TLB Tag Read Register, 489
I/D TSB 64K Pointer Registers, 489
I/D TSB 8K Pointer Registers, 489
I/D TSB Direct Pointer Register, 489
I/D TSB Extension Registers, 496
I/D TSB Nucleus Extension Register, 489
I/D TSB Primary Extension Register, 489
I/D TSB register, 488
I/D TSB Registers, 495
page sizes, 467
Physical Watchpoint Address, 489
Primary Context Register, 488
Secondary Context Register, 488
SPARC V9 compliance, 487
Synchronous Fault Address Registers, 500
Synchronous Fault Status Register
fault types, 499
Tag Target Registers, 494
Virtual Watchpoint Address, 488
mode
nonprivileged, 7, 22
privileged, 22, 71, 174
user, 40, 66, 519
MOV synthetic instruction, **516**
MOVA instruction, 304
MOVCC instruction, 304
MOVcc instructions, 53, 65, 102, 104, 117, 300, 301, **304**, 308, 462, 463
MOVCS instruction, 304
move floating-point register if condition is true, **296**
move floating-point register if contents of integer register satisfy condition, **302**
MOVE instruction, 304
move integer register if condition is satisfied instructions, **304**
move integer register if contents of integer register satisfies condition instructions, **309**
move on condition instructions, 5
MOVFA instruction, 305
MOVFE instruction, 305
MOVFG instruction, 305
MOVFGE instruction, 305
MOVFL instruction, 305
MOVFLE instruction, 305
MOVFLG instruction, 305
MOVFN instruction, 305
MOVFNE instruction, 305
MOVFO instruction, 305
MOVFU instruction, 305
MOVFUE instruction, 305
MOVFUG instruction, 305
MOVFUGE instruction, 305
MOVFUL instruction, 305
MOVFULE instruction, 305
MOVG instruction, 304
MOVGE instruction, 304
MOVGU instruction, 304
MOVL instruction, 304
MOVLE instruction, 304
MOVLEU instruction, 304
MOVN instruction, 304
MOVNE instruction, 304
MOVNEG instruction, 304
MOVPOS instruction, 304
MOVr instructions, 104, 117, **309**, 463
MOVRGEZ instruction, **309**
MOVRGZ instruction, **309**
MOVRLEZ instruction, **309**
MOVRLZ instruction, **309**
MOVRNZ instruction, **309**
MOVRZ instruction, **309**
MOVVC instruction, 304
MOVVS instruction, 304
MTP, **185**

definition, 187 to 188
performance implications, 188
Programming Model, 185, 189, 197
registers
access, 189
characteristics, 190
Core Available, 197
Core Enable, 198
Core Enable Status, 198
Core ID, 191
Core Interrupt ID, 192
Core Running, 200
Core Running Status, 203
Error Steering, 209
fields, 191
XIR Steering, 206
shared structures, 207
terminology, 186 to 187
multiple unsigned condition codes, emulating, 118
multiply instructions, 25, 311, **398**, 398
multiprocessor synchronization instructions, 246, 410, 412
multiprocessor system, 13, 177, 268, 337, 410, 412, 433
multithreaded processor, *See* MTP
MULX instruction, **311**
must (keyword), 13
mutex (mutual exclusion) locks, 544
M-way set-associative TSB, 474


## N


NaN (not-a-number)
conversion to integer, 423
converting floating-point to integer, 258
quiet, 256, 421
signalling, 54, 256, 260, 421
transformation, 420
NEG synthetic instruction, **515**
negative (n) bit of condition fields of CCR, **64**
negative infinity, 423, 424
nested traps, 6
next program counter (nPC), 13, 27, **62**, 79, 99, 249, 312, 537, 600, 616
nonfaulting load, 14, 174, **175**, 472, 486, 499
nonleaf routine, 273
nonprivileged, 14
mode, 7, 9, 14, 22, 56
registers, **40**
software, 65
nonprivileged trap (npt) field of STICK register, **70**
nonprivileged trap (npt) field of TICK register, **66**, 344
nonstandard floating-point, *See* floating-point status register (FSR) ns field
nontranslating ASI, **14**, 94, 397, 409
nonvirtual memory, 338
NOP instruction, 241, **312**, 335, 375, 385, 388
normal trap, 14
normal traps, **130**, 140, 148, **148**, **150**, 150, **151**, 153
normative appendix, 14
NOT synthetic instruction, **515**
note
implementation, **4**
programming, **4**
nPC register, *See* next program counter (nPC)
Nucleus code, 594
Nucleus Context Register, 490
NUMA, 188
number of windows (maxwin) field of VER register, 83
number of windows (maxwin) field of VER register, 127
NWINDOWS, 14, 22, **44**, 44, 83, 348, 349, 428


## O


op3 field of instructions, **103**, 224, 246, 249, 268, 270, 273, 274, 276, 279, 285, 286, 292, 311, 334, 341, 344, 346, 390, 393, 394, 396, 398, 400, 402
op4 field of instructions, **103**
opcode, 14
Mask, 96
Match, 96
reserved, 540
opf field of instructions, **103**, 253, 255, 257, 259, 261, 263, 265, 267
opf_cc field of instructions, **103**, 298, 463
opf_low field of instructions, **103**, 298, 302
optional, 14
OR instruction, **291**, 516
ORcc instruction, **291**, 514
ordering MEMBAR instructions, 111
ordering unit, memory, 173
ORN instruction, **291**
ORNcc instruction, **291**
other windows (OTHERWIN) register, 83, 85, 119, 120, 125, 127, 270, 341, 349, 377, 535, 601

out register #7, 45
out registers, 44, 348, 518
overflow
  causing spill trap, 126
  window, 534
overflow (V) bit of condition fields of CCR, 112
overflow (v) bit of condition fields of CCR, **64**
overflow accrued (ofa) bit of aexc field of FSR register, **61**
overflow current (ofc) bit of cexc field of FSR register, **61**
overflow mask (ofm) bit of tem field of FSR register, **60**

## P

p field of instructions, **104**, 238, 241, 242
P superscript on instruction name, **218**
PA Data Watchpoint Register
  db_pa field, 95
  format, 96
  state after reset, 602
*PA_watchpoint* exception, 95, 167, 568
packed-to-planar conversion, 331
packing instructions, *See* FPACK instructions
page fault, 338
page table entry (PTE), *See* translation table entry (TTE)
parameters to a subroutine, 518
park, 14
parked, 15
partial store instructions, 313, 582
partial store order (PSO) memory model, 75, **170**, 170, **181**, 433, 541
$P_{ASI}$ superscript on instruction name, **218**
$P_{ASR}$ superscript on instruction name, **218**
PC register, *See* program counter (PC)
PCR
  fields
    priv, 68, 163
    sl (select lower bits of PIC) field, 68
    st (system trace enable) field, 68
    su (select upper bits of PIC) field, 68
    ut (user trace enable) field, 68
  RDASR/WRASR support, 121
  state after reset, 602
PDIST instruction, 325
Performance Control Register, *See* PCR
Performance Instrumentation Counter, *See* PIC register
physical address, 15
  data watchpoint, 96
  memory address, 171
physical core, 15, 186
PIC register
  and PCR, 67
  picl field, 69
  picu field, 69
  RDASR/WRASR support, 121
  state after reset, 602
PIL, *See* processor interrupt level (PIL) register
pixel instructions
  comparison, 323
  component distance, 325
  formatting, 326
pixel multiplication, 38
planar-to-packed conversion, 331
$P_{NPT}$ superscript on instruction name, **218**
POPC instruction, 124, **332**
POR, *See power_on_reset* (POR)
positive infinity, 423, 424
power failure, 137, 156
*power_on_reset* (POR), 79, 80, 81, 82, 131, 132, 153
power-on reset (POR), 67, 71
$P_{PCR}$ superscript on instruction name, **218**
$P_{PIC}$ superscript on instruction name, **218**
precise floating-point traps, 342
precise trap
  catastrophic error exception, 429
  conditions, 612
  conditions for, 134
  software actions, 134
  vs. disrupting trap, 135
predefined constants
  LoadLoad, 512
  lookaside, 512
  MemIssue, 512
  StoreLoad, 512
  StoreStore, 512
  Sync, 512
predict bit, 238
prefetch
  for one read, 337
  for one write, 337
  for several reads, 337
  for several writes, 337
  instruction, 340
  page, 338
prefetch data instruction, **334**

PREFETCH instruction, 105, **334**, 335, 431, 486
prefetch_fcn, **512**
PREFETCHA instruction, **334**, 431
prefetchable, 15
Primary Context Register, 489
priority
  traps, 138, 143, 145
  VA vs. PA_watchpoint, 95
privileged
  mode, 15, 22, 71, 174
  registers, **71**
  software, 7, 45, 56, 76, 110, 140, 270, 431
privileged (priv) field of PCR register, 163
privileged (priv) field of PSTATE register, **77**, 163, 174, 247, 277, 286, 340, 344, 364, 369, 408, 412
privileged mode (priv) field of PSTATE register, **77**
*privileged_action* exception, 66, 71, 110, 137, **163**, 248, 277, 282, 286, 287, 344, 345, 364, 369, 397, 408, 409, 413, 478, 479, 480, 482, 568, 591, 592, 593, 594
  PIC access, 69
privileged_action exception, 110
*privileged_instruction* exception (SPARC V8), 163
*privileged_opcode* exception, 137, **163**, 249, 342, 351, 379, 594
processor, 15
  execute unit, 175
  halt, 146
  issue unit, 175, **175**
  reorder unit, 175
  self-consistency, **176**
processor interrupt level (PIL) register, **78**, 136, 138, 162, 341, 377, 595, 601
processor state (PSTATE) register, 27, 44, 74, 80, 129, 131, 249, 341, 377, 600
processor states
  error_state, 131, 134, 146, 147, 148, 150, 151, 159, 430
  execute_state, 146, 147, 148, 150, 151
  RED_state, 131, 132, 133, 141, 146, 148, 150, 151, 153, 154, 159, 182, 433
program counter (PC), 16, 27, **62**, 78, 99, 121, 129, 245, 249, 273, 312, 537, 600, 616
program order, **175**, 176
programming note, **4**
PSO, *See* partial store order (PSO) memory model
PSR register (SPARC V8), 382
PSTATE
  am field, 567
  cle field, and implicit ASIs, 110
  global register selection encodings, 72
  ie field, 595
  ig field, 72, 590
  *illegal_instruction* exception, 162
  mg field, 72
  pef field, 381
  PRIV field, 14, 15
  priv field, 473, 480
  red field, 88, 597
PTE (page table entry), *See* translation table entry (TTE)

## Q

Quad FPop instructions, 124
quadword, 16
  addressing, 108
  alignment, 106, 173
  data format, **29**
queue not empty (qne) field of FSR register, **58**, 420
quiet NaN (not-a-number), 54, 256, 420, 421

## R

r register, 16
r register
  #15, 45
  number in IU, 428
  special-purpose, 45
  alignment, 394, 396
rational quotient, 391
R-A-W, *See* read-after-write memory hazard
rcond field of instructions, **104**, 238, 302, 309, 463
rd field of instructions, **104**, 224, 246, 253, 257, 259, 261, 263, 265, 267, 273, 274, 276, 279, 285, 286, 292, 298, 302, 306, 309, 311, 332, 341, 344, 390, 393, 394, 396, 398, 400, 402, 539
RDASI instruction, **343**, 343, **402**
RDASR instruction, 24, 62, 162, **343**, 343, **402**, 403, 430, 516, 539
RDCCR instruction, **343**, 343, **402**
RDDCR instruction, 343
RDFPRS instruction, **343**, 343, **402**
RDGSR instruction, 69, 343
RDPC instruction, 63, **343**, 343, **402**
RDPCR instruction, 68, 343
RDPIC instruction, 163, 343
RDPR FQ instruction, 124

RDPR instruction, 72, 79, 80, 81, 82, 83, 125, 162, **341**, 345
RDSOFTINT instruction, 89, 343
RDSTICK instruction, 71, 343
RDSTICK_CMPR instruction, 343
RDTICK instruction, **343**, 343, 345, **402**
RDTICK_CMPR instruction, 343
RDY instruction, 63, 516
read privileged register (RDPR) instruction, **341**
read state register instructions, 26, 62, 86, **343**, 402
read-after-write memory hazard, 176
real memory, 172
real-time software, 524
RED_state, 16, 130, 131, 132, 133, 141, 146, 148, 150, 151, 153, 154, 159, 182, 485, 486, **597**
  MMU behavior, 485
  restricted environment, 132
  trap vector, 131, 132, 433, 599
RED_state (red) field of PSTATE register, **74**, 131
RED_state trap, 16
RED_state trap table, 141
RED_state trap vector address (RSTVaddr), 433
reference MMU, 7, 505
reg, **506**
reg_or_imm field of instructions, **513**, 539
reg_plus_imm, **512**
regaddr, **513**
register reference instructions, data flow order constraints, 176
register window management instructions, 27
register windows, 6, 44, 518, 519
  clean, 86, 118, 125, 126, 128, 164
  fill, 45, 118, 120, 125, 126, 127, 128, 349, 351
  spill, 45, 118, 119, 120, 125, 126, 127, 128, 163, 349, 351
registers, 602
  accessing MMU registers, 488
  address space identifier (ASI), 129, 174, 249, 277, 282, 286, 336, 364, 369, 381, 396, 408, 412, 534
  allocation within a window, 523
  alternate global, 22
  alternate global, 41, **42**, 534
  ancillary state registers (ASRs), 24, **62**, 539
  ASI, **66**, 80, 601
  CANRESTORE, 85, 601
  CANSAVE, 85, 601
  clean windows (CLEANWIN), 83, **86**, 118, 125, 126, 128, 341, 377
  CLEAR_SOFTINT, 88, 595
clock-tick (TICK), 163
condition codes register (CCR), 80, 129, 224, 249, 381, 400
Core Available, 197
Core Enable, 198
Core Enable Status, 198
Core ID, 191
Core Interrupt ID, 192
Core Running, 200
Core Running Status, 203
current window pointer (CWP), 44, 80, 83, **84**, 86, 126, 127, 129, 249, 270, 341, 348, 349, 377
Data Cache Unit Control (DCUCR), 92
dispatch control register (DCR), 86
Error Steering, 209
f (floating point), 139, 419, 430
floating-point, 22, 52, 430, 520
floating-point deferred-trap queue (FQ), 342
floating-point registers state (FPRS), **65**, 344, 381
floating-point state (FSR), **53**, 58, 59, 62, 393, 404, 419
frame pointer, 518
global, 5, 22, 41, **42**, 42, 519, 523
graphics status (GSR), 69
IER (SPARC V8), 382
in, 44, 348, 518
Instruction Trap, 96
Interrupt Vector Dispatch Extension, 194
Interrupt Vector Dispatch register, 591
Interrupt Vector Dispatch Status register, 592, 602
Interrupt Vector Receive register, 593, 602
local, 44, 348
MMU Tag Target, 494
nonprivileged, **40**
other windows (OTHERWIN), 83, 85, 119, 120, 125, 127, 270, 341, 349, 377, 535, 601
out, 44, 348, 518
out #7, 45
PA_WATCHPOINT, 602
PC, 62
performance control (PCR), 67
performance instrumentation counter (PIC), 69
privileged, **71**
processor interrupt level (PIL), **78**, 341, 377
processor state (PSTATE), 44, 74, 80, 129, 131, 249, 341, 377
PSR (SPARC V8), 382
r register #15, 45

r register, general-purpose, 428
renaming mechanism, 176
restorable windows (CANRESTORE), 45, 83, 86, 118, 119, 120, 125, 126, 341, 349, 351, 377, 535
savable windows (CANSAVE), 45, 83, **85**, 118, 119, 120, 126, 127, 270, 341, 349, 351, 377
scratchpad, 98
SET_SOFTINT, 88, 595
SOFTINT, 88, **594**
stack pointer, 518, 519
STICK, 70
STICK_COMPARE, 602
TBA, 600
TBR (SPARC V8), 382
TICK, **66**, **70**, 341, 377, 601
TICK_COMPARE, 90, 601
trap base address (TBA), **82**, 129, 139, 341, 377
trap level (TL), **77**, 77, 78, 79, 80, 81, 82, 83, 85, 129, 249, 341, 342, 351, 359, 377, 378
trap next program counter (TNPC), **79**, 341, 377
Trap Program Counter, 500
trap program counter (TPC), **78**, 341, 342, 377
trap state (TSTATE), **73**, **80**, 249, 341, 377
trap type (TT), **81**, 82, 85, 140, 146, 157, 341, 376, 377, 429, 601
update, 499
version register (VER), **82**, 341
WIM (SPARC V8), 382
window state (WSTATE), **83**, **85**, 127, 270, 341, 349, 377, 534, 535
window usage models, 524
WSTATE, 601
XIR Steering, 206
Y, 63, **63**, 390, 398, 400, 418, 601
relaxed memory order (RMO) memory model, 75, **170**, **180**, 433, 541
renaming mechanism, register, 176
reorder unit, 175
reordering instruction, 175
reserved, 16
fields in instructions, 217
opcodes, 540
reset
*externally_initiated_reset* (XIR), 13.51, 132, 133, 137, 153, 156, **164**
global, 485
*power_on_reset* (POR), 131, 132, 153, 163, **163**
power-on, 67, 71
processing, 131
PSTATE.RED, 597
request, 131, 163
reset trap, 16, 67, 71, 81, 135, 136
*software_initiated_reset* (SIR), 131, 133, 136, 137, 146, 147, 158, **163**, 598
trap, 430
trap vector address, *See* RSTVaddr
*watchdog_reset* (WDR), 153, 156, **164**
Reset, Error, and Debug state, *See* RED_state
restorable windows (CANRESTORE) register, 45, 83, 86, 118, 119, 120, 125, 126, 341, 349, 351, 377, 535
RESTORE instruction, 348 to 350
actions, 118
avoiding normal register-window mechanism, 525
and current window, 46
decrementing CWP register, 44
fill trap, 126
followed by SAVE instruction, 45
leaf-procedure optimization, 521, 522
managing register windows, 27
operation, 348
performance trade-off, 349
relationship to %sp, 519
and restorable windows (CANRESTORE) register, 85
restoring register window, 349
role in register state partitioning, 125
SPARC V9 vs. SPARC V8, 84
RESTORE synthetic instruction, **514**
RESTORED instruction, 120, 128, 350, **351**, 351, 534
use by privileged software, 27
restricted, 16
restricted address space identifier, 110
restricted ASI, 482, 568
RET synthetic instruction, **514**, 522
RETL synthetic instruction, **514**, 522
RETRY instruction, 26, 65, 73, 79, 80, 81, 128, 129, 131, 136, 162, **249**, 440
restoring ag, ig, mg bits, 73
return address, 518, 521
RETURN instruction, 346 to 347
computing target address, 26
*mem_address_not_aligned* exception, 163
operation, 346
reexecuting trapped instruction, 346
support for nonprivileged trap handlers, 537
RMO, *See* relaxed memory order (RMO) memory model

rounding
behavior in GSR, 70
for floating-point results, 54
image computations, 38
in signed division, 392
rounding direction (rd) field of FSR register, 254, 257, 259, 261, 266, 267
rounding direction rd field of FSR register, **54**
routine, nonleaf, 273
rs1 field of instructions, **104**, 224, 238, 246, 253, 255, 265, 268, 273, 274, 276, 279, 285, 286, 292, 302, 309, 311, 334, 341, 344, 346, 390, 393, 394, 396, 398, 400, 402, 539
rs2 field of instructions, **104**, 224, 246, 253, 255, 257, 259, 261, 263, 265, 267, 268, 273, 274, 276, 279, 292, 298, 302, 306, 309, 311, 332, 334, 390, 393, 394, 396, 398, 400
RSTVaddr, 141, 433, 599
running, 17

## S

savable windows (CANSAVE) register, 45, 83, **85**, 118, 119, 120, 126, 127, 270, 341, 349, 351, 377
SAVE instruction, 348 to 350
actions, 118
after a callee is entered, 518
after RESTORE instruction, 346
avoiding normal register-window mechanism, 525
*clean_window* exception, 126, 164
and current window, 46
decrementing CWP register, 44
in leaf procedure optimization, 522
leaf procedure, 273, 521
and local/out registers of register window, 45
managing register windows, 27
no clean window available, 86
number of usable windows, 86
operation, 348
performance trade-off, 349
relationship to %sp, 519
role in register state partitioning, 125
and savable windows (CANSAVE) register, 85
SPARC V9 vs. SPARC V8, 84
spill trap, 126, 127, 163
SAVE synthetic instruction, **514**
SAVED instruction, 27, 119, 128, 350, **351**, 351, 534
scaling of the coefficient, 318
scratchpad registers, 98
SDIV instruction, 63, **390**
SDIVcc instruction, 63, **390**
SDIVX instruction, **311**
Secondary Context Register, 489
self-consistency, processor, 176
self-modifying code, 269, 525
*sequence_error* floating-point trap type, 164
sequencing MEMBAR instructions, 111
sequential consistency memory model, **171**
SET synthetic instruction, **514**, **515**
SET_SOFTINT pseudo-register, 88, 121, 595
SET_SOFTINT register, 88
SETHI instruction, 25, 103, 112, 312, **353**, 353, 514, 515, 520
SFAR Fault Address field, 500
SFSR
bit description, 497
update policy, 499
shall (keyword), 17
shared memory, 169, 544, 546, 552
shcnt32 field of instructions, **104**
shcnt64 field of instructions, **104**
shift count encodings, 355
shift instructions, 25, 112, **354**
short floating-point load and store instructions, 356, 583
should (keyword), 17
SHUTDOWN instruction, 358
SIAM instruction, 352
side effects, 17, 172
signalling ECC, 618
signalling NaN (not-a-number), 54, 256, 260, 421
signed integer data type, 29
sign-extended 64-bit constant, 104
sign-extension, 515
SIGNX synthetic instruction, **515**
SIMD, 17
SIMD instructions, 37
simm10 field of instructions, **104**, 309
simm11 field of instructions, **104**, 306
simm13 field of instructions, **104**, 224, 268, 273, 274, 276, 279, 285, 286, 292, 311, 332, 334, 346, 390, 393, 394, 396, 398, 400
Single Instruction/Multiple Data, *See* SIMD
single-issue mode, *See* DCUCR si (single-issue disable) field
SIR instruction, 136, 158, 163, 359, 381, 599
SIR, *See software_initiated_reset (SIR)*

SLL instruction, **354**, 354
SLLX instruction, **354**, 354, 515
SMUL instruction, 63, **398**
SMULcc instruction, 63, **398**
SOFTINT register, 88, 89, 121, 594
software conventions, 517
software interrupt (SOFTINT) register
  after reset & in RED_state, 601
  clearing, 595
  in code sequence for Interrupt Receive, 590
  scheduling interrupt vectors, 594
  setting, 595
software translation table, 468
software trap, 140, 141, **141**, 375
*software_initiated_reset* (SIR), 131, 133, 136, 137, 146, 147, 153, 158, 163, 359, 598, 599
software_trap_number, **513**
SPARC V8 compatibility
  ADDC / ADDCcc renamed, 225
  current window pointer (CWP) register differences, 84
  delay instruction, 26
  delay instruction fetch, 115
  executing delayed conditional branch, 115
  existing nonprivileged SPARC V8 software, 44
  instruction between FBfcc / FBPfcc, 241
  LD, LDUW instructions, 280
  level 15 interrupt, 78
  operations to I / O locations, 172
  read state register instructions, 345
  STA instruction renamed, 369
  STBAR instruction, 295, 403
  STD instruction, 407
  STDA instruction, 409
  STFSR instruction, 404
  tagged add instructions, 415
  tagged subtract instructions, 417
  Ticc instruction, 376
  UNIMP instruction renamed, 271
  *window_overflow* exception superseded, 163
  *window_underflow* exception superseded, 161
  write state register instructions, 382
SPARC V9
  compliance, 14, 487
  features, 5
  memory models, 180
SPARC V9 Application Binary Interface (ABI), 7, 8
special trap, name change, 130
special traps, **130**, 141
speculative load, 17
spill register window, 45, 118, 119, 120, 125, 126, 127, 128, 163, 349, 351, 534
spill windows, 348
*spill_n_normal* exception, **163**, 270, 350
*spill_n_other* exception, **163**, 270, 350
spin lock, 545
SRA instruction, **354**, 354, 515
SRAX instruction, **354**, 354
SRL instruction, **354**, 354
SRLX instruction, **354**, 354
ST instruction, 516
stack frame, 349
stack pointer alignment, **521**
stack pointer register, 518, 519
STB instruction, 366, 516
STBA instruction, 368
STBAR instruction, 177, 179, 295, 344
STD instruction, 46, 432
STDA instruction, 46
STDF instruction, 106, 164, 166, 360
*STDF_mem_address_not_aligned* exception, 106, 137, **164**, **166**, 361, 364, 432
STDFA instruction, 106, 231, 313, 356, **363**, 363, 582, 583
STF instruction, 360
STFA instruction, 363
STFSR instruction, 53, 56, 162
STH instruction, 366, 516
STHA instruction, 368
STICK register, 70, 121, 343, 601
STICK_COMPARE register, 90, 121, 343, 602
STICK_INT, 595
store
  block, *See* block store instructions
  partial, *See* partial store instructions
  short floating-point, *See* short floating-point store instructions
store floating-point into alternate space instructions, **363**
store instructions, 17, 105, 166, 542
StoreLoad MEMBAR relationship, 179, 294
StoreLoad predefined constant, 512
stores to alternate space, 24, 66, 110, 564
StoreStore MEMBAR relationship, 179, 294
StoreStore predefined constant, 512
STQF instruction, 124, 360
*STQF_mem_address_not_aligned* exception, 166
STQFA instruction, **363**, 363

strand, 18, 186
strong consistency memory model, 171, 544, 550
strong ordering, 171
STW instruction, 366
STWA instruction, 368
STX instruction, 366
STXA instruction, 190, 368, 409, 500
STXFSR instruction, 53, 56, 162, 360
SUB instruction, **370**, 370, 515, 516
SUBC instruction, **370**, 370
SUBcc instruction, 112, **370**, 370, 514
SUBCcc instruction, **370**, 370
subnormal numbers, 18
subtract instructions, **370**
superscalar, 18
supervisor software, 18, 24, 44, 57, 129, 147, 158, 428, 517, 533, 534, 537
supervisor-mode trap handler, 140
sw_trap# field of instructions, **104**
SWAP instruction, 105, 179, 183, 285, 286, **410**, 546
swap r register with alternate space memory instructions, **412**
swap r register with memory instructions, 247, **410**
SWAPA instruction, 285, 286, **412**
Sync MEMBAR relationship, 294
Sync predefined constant, 512
Synchronous Fault Address Register (SFAR), 500
Synchronous Fault Status Register (SFSR)
  fault types, 499
  register bits, 497
synthetic instructions
  BCLR, **516**
  BSET, **516**
  BTOG, **516**
  BTST, **516**
  CALL, **514**
  CAS, **515**
  CASX, **516**
  CLR, **516**
  cmp, 371, **514**
  DEC, **516**
  DECcc, **516**
  INC, **516**
  INCcc, **516**
  IPREFETCH, **514**
  JMP, **514**
  MOV, **516**
  NEG, **515**
  NOT, **515**
  RESTORE, **514**
  RET, **514**, 522
  RETL, **514**, 522
  SAVE, **514**
  SET, **514**, **515**
  SIGNX, **515**
  TST, **514**
synthetic instructions in assembler, 514
system call, 535
system clock-tick register (STICK), **70**
system software, 163, 174, 184, 269, 430, 519, 521, 525, 526, 534, 535, 537
System Tick Compare Register, *See* STICK_COMPARE register
System Tick Register, *See* STICK register
system timer interrupt, STICK_INT, 595

## T

TA instruction, 374, 462
TADDcc instruction, 112, 372
TADDccTV instruction, 112, 163
Tag Access Register, 475, 490, 491
tag overflow, 112
*tag_overflow* exception, 112, **163**, 372, 373, 414, 415, 417
tagged arithmetic, 112
tagged arithmetic instructions, 25
tagged word data format, **29**
tagged words, 29
TBA register, 600
TBR register (SPARC V8), 382
TCC instruction, 374
Tcc instructions, 27, 65, 102, 129, 140, 141, 162, 163, **374**, 458, 462, 464
TCS instruction, 374, 462
TE instruction, 374, 462
test-and-set instruction, 183
TG instruction, 374, 462
TGE instruction, 374, 462
TGU instruction, 374, 462
thread, 18, 186
Ticc instruction (SPARC V8), 376
TICK register, 66, 121, 601
TICK_COMPARE register, 90, 121, 601
TICK_INT, 595
timer interrupt, TICK_INT, 595
timing of instructions, 218
tininess (floating-point), 61

TL instruction, 374, 462
TL register, 378
and implicit ASIs, 110
TL = MAXTL, 598
TLB
and RED_state, 485
bypass operation, 504
Data Access register, 492
data In register, 475
data in register, 492, 493
demap operation, 504
hardware, 503
hit, 18
instruction, 479
miss, 18
fast handling, 478
handler, 469, 475
MMU behavior, 469
reloading TLB, 473
miss / refill sequence, 475
missing entry, 475
operations, 503
read operation, 504
replacement policy, 504
required conditions, 478
specialized miss handler code, 487
Tag Access Registers, 490
translation operation, 503
write operation, 504
TLE instruction, 374, 462
TLEU instruction, 374, 462
TN instruction, 374, 462
TNE instruction, 374, 462
TNEG instruction, 374, 462
TNPC register, 78, 134, 616
total order, **178**
total store order (TSO) memory model, 75, **170**, 170, **181**, 182, 541
TPC register, 78, 615, 616
TPOS instruction, 374, 462
translating ASI, 97
translating ASIs, 94, 568
Translation
Storage Buffer (TSB), 18, 473, 495, 496
Table Entry (TTE), 470, 479
Translation Lookaside Buffer (TLB), 18
Translation Table Entry, *See* TTE
trap, 18
*See also* exceptions *and* traps
definition, 129
*ECC_error*, 166
*fast_data_access_MMU_miss*, 73
*fast_data_access_protection*, 73
*fast_instruction_access_MMU_miss*, 73
fp_disabled
GSR access, 381
level, 77
model, **137**
normal, 14
out register overflow, 518
priority, 138, 143, 145
processing, 146
RED_state, 16
stack, 6, 72, 149, 150, 152
*VA_ / PA_watchpoint*, 95
vector, RED_state, 131
trap base address (TBA) register, **82**, 129, 139, 341, 377
trap categories
deferred, 135
disrupting, **135**, 136
precise, 135
reset, **136**
trap enable mask (tem) field of FSR register, **54**, 58, 60, 138, 139, 164
trap handler, 249
supervisor-mode, 140
user, 57, 422, 537
trap level (TL) register, **77**, 77, 78, 79, 80, 81, 82, 83, 85, 129, 249, 341, 342, 351, 359, 377, 378, 601
trap next program counter (TNPC) register, **79**, 341, 377, 601
trap on integer condition codes instructions, **374**
trap program counter (TPC) register, **78**, 341, 342, 377, 500, 601
trap state (TSTATE) register, **73**, **80**, 249, 341, 377, 601
trap type (TT) register, **81**, 82, 85, 140, 146, 157, 341, 376, 377, 429, 601
*trap_instruction* (ISA) exception, 137, **163**, 375, 376
trap_little_endian (tle) field of PSTATE register, **74**, 74
traps
*See also* exceptions *and* trap
causes, **28**, 28
deferred, 134, 429
definition, 27
disrupting, 134, 429

hardware, 141
nested, 6
normal, **130**, 140, 148, **148**, **150**, 150, **151**, 153
precise, 134, 429
reset, 81, 134, 135, 136, 147, 430
software, 141, 375
*software_initiated_reset* (SIR), 153
special, **130**, 141
window fill, 140
window spill, 140
TSB
cacheability, 474
caching, 474
demap operation, 502
Direct Pointer registers, 496
Extension Register, 476, 477, 489, 496
I/D Translation Storage Buffer Register, **494**
indexing support, 473
miss handler, 475
organization, 474
pointer logic hardware, 504
Pointer register, 497
register, computing 64-Kbyte pointer, 473
required conditions, 478
split, 477
split field, 495
Tag Target register, 477, 494
tsb_base field, 495
tsb_size field, 495
TSO, *See* total store order (TSO) memory model
TST synthetic instruction, **514**
TSTATE, *See* trap state (TSTATE) register
TSUBcc instruction, 112, 373
TSUBccTV instruction, 112, 163
TTE, 18
cp field, 472
cv field, 472
e field, 472
g field, 470, 473
l field, 472
nfo field, 471
p field, 473
pa field, 471
size field, 470
soft2 field, 471
v field, 470
va_tag field, 470
w field, 473
TVC instruction, 374, 462
TVS instruction, 374, 462
typewriter font, in assembly language syntax, 505

## U

UDIV instruction, 63, **390**
UDIVcc instruction, 63, **390**
UDIVX instruction, **311**
UMUL instruction, 63, **398**
UMULcc instruction, 63, **398**
unassigned, 18
unconditional branches, 241, 243, 385, 388
uncorrectable errors, 616
undefined, 19
underflow, 126
underflow accrued (ufa) bit of aexc field of FSR register, **61**, 423
underflow current (ufc) bit of cexc field of FSR register, **61**, 423
underflow mask (ufm) bit of tem field of FSR register, **60**
underflow mask (ufm) bit of tem field of FSR register, 61, 422
unfinished_FPop floating-point trap type, 57, **57**, 62, 123, 266, 420
UNIMP instruction (SPARC V8), 271
unimplemented, 19
unimplemented instructions, 124
unimplemented_FPop floating-point trap type, 57, **58**, 62, 123, 254, 256, 258, 260, 262, 266, 301, 303, 420
*unimplemented_LDD* exception, 165, 432
*unimplemented_STD* exception, 165, 432
uniprocessor system, 19
unpark, 19
unrestricted, 19
unrestricted address space identifier, 537
unsigned integer data type, 29
user
mode, 40, 66, 519
software, 525
trap handler, 57, 422, 537

## V

VA Data Watchpoint Register
db_va field, 95
state after reset, 602
*VA_watchpoint* exception, 95, 167

value clipping, *See* FPACK instructions
value semantics of input/output (I/O) locations, 172
variables, automatic, 519
version (ver) field of FSR register, **55**
version register (VER), **82**, 341, 601
virtual address, 19
  data watchpoint, 95
  memory address, 171
virtual memory, 338
virtual processor
  behavior, 188
  characteristics, 188
  identification, 191
  interrupt ID, 193
  numbering, 192
  register access, 189
  shared register, 190
  updating Core Running register, 202
VIS instructions, 69
  encoding, 465
Visual Instruction Set, *See* VIS instructions

## W

walking the call chain, 519
W-A-R, *See* write-after-read memory hazard
*watchdog_reset* (WDR), 134, 153, 156, 164, 439, 599
watchpoints
  data registers, 94
  PA/VA watchpoint traps, 480
  and RED_state, 597
  trap, 479
W-A-W, *See* write-after-write memory hazard
WDR, *See watchdog_reset* (WDR)
WIM register (SPARC V8), 382
window fill exception, *See also fill_n_normal* exception
window fill trap handler, 27
window overflow, 45, 126, 534
window spill exception, *See also spill_n_normal* exception
window spill trap handler, 27
window state (WSTATE) register
  description, 85
  and fill/spill exceptions, 127
  normal field, 127
  other field, 127
  overview, 83
  reading WSTATE with RDPR instruction, 341
  and spill/fill traps, 535
  spill exception, 270
  spill handler example, 534
  spill trap, 349
  writing WSTATE with WRPR instruction, 377
window underflow, 45, 126
window, clean, 348
*window_fill* exception, 85, 118, 140, 346, 521
*window_spill* exception, 85, 140
windows, register, 519
word, 19
  addressing, 108
  alignment, 106, 173
  data format, **29**
WRASI instruction, 380
WRASR instruction, 24, 62, 162, 380, 431, 516, 539
  WRDCR instruction, 380
  WRGSR instruction, 380
  WRPCR instruction, 380
  WRPIC instruction, 380
  WRSOFTINT instruction, 380
  WRSOFTINT_CLR instruction, 380
  WRSOFTINT_SET instruction, 380
  WRSTICK instruction, 380
  WRSTICK_CMPR instruction, 380
  WRTICK_CMP instruction, 380
WRCCR instruction, 65, 380
WRFPRS instruction, 380
WRGSR instruction, 69
WRIER instruction (SPARC V8), 382
write privileged register instruction, **377**
write state register, 62
write state register instructions, 62, 86
write-after-read memory hazard, 176
write-after-write memory hazard, 176
WRPCR instruction, 68
WRPIC instruction, 163
WRPR instruction, 67, 72, 79, 80, 81, 82, 83, 84, 125, 131, 162, **377**, 377, 440
WRPSR instruction (SPARC V8), 382
WRSOFTINT instruction, 89
WRSOFTINT_CLR instruction, 88
WRSOFTINT_SET instruction, 88
WRTBR instruction (SPARC V8), 382
WRWIM instruction (SPARC V8), 382
WRY instruction, 63, 380, 516
WSTATE register, 601

## X

x field of instructions, **104**
xcc field of CCR register, **64**
xcc field of CCR register, 224, 244, 292, 308, 370, 372, 391, 392, 399, 401
XIR Steering register, 206
XIR, *See externally_initiated_reset* (XIR)
XNOR instruction, **291**, 515
XNORcc instruction, **291**
XOR instruction, **291**, 516
XORcc instruction, **291**

## Y

Y register, 63, **63**, 121, 390, 398, 400, 418, 601

## Z

zero (z) bit of condition fields of CCR, **64**