# UltraSPARC Architecture 2007

*One Architecture*
*... Multiple Innovative Implementations*

*Draft D0.9.2, 19 Jun 2008*

*Privilege Levels:* *Hyperprivileged,*
*Privileged,*
*and Nonprivileged*

*Distribution:* *Public*

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A. 650-960-1300

Part No: 950-5553-12
Revision: Draft D0.9.2, 19 Jun 2008

Copyright 2002-2005 Sun Microsystems, Inc., 4150 Network Circle • Santa Clara, CA 950540 USA. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd. For Netscape Communicator™, the following notice applies: Copyright 1995 Netscape Communications Corporation. All rights reserved.

Sun, Sun Microsystems, the Sun logo, Solaris, UltraSPARC, and VIS are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

**RESTRICTED RIGHTS**: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID

.

---

Copyright 2002–2005 Sun Microsystems, Inc., 4150 Network Circle • Santa Clara, CA 950540 Etats-Unis. Tous droits réservés.

Des parties de ce document est protégé par un copyright© 1994 SPARC International, Inc.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Solaris, UltraSPARC et VIS sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

.

---

Comments and "bug reports" regarding this document are welcome; they should be submitted to email address: `UA-editor@sun.com`

# Contents

**Preface** . . . . . **i**

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

1.1 Navigating *UltraSPARC Architecture 2007* . . . . . 1
1.2 Fonts and Notational Conventions. . . . . 2
1.2.1 Implementation Dependencies . . . . . 4
1.2.2 Notation for Numbers . . . . . 4
1.2.3 Informational Notes . . . . . 5
1.3 Reporting Errors in this Specification . . . . . 5

**2 Definitions** . . . . . **7**

**3 Architecture Overview** . . . . . **21**

3.1 The UltraSPARC Architecture 2007 . . . . . 22
3.1.1 Features . . . . . 22
3.1.2 Attributes. . . . . 23
3.1.2.1 Design Goals . . . . . 23
3.1.2.2 Register Windows. . . . . 24
3.1.3 System Components. . . . . 24
3.1.3.1 Binary Compatibility . . . . . 24
3.1.3.2 UltraSPARC Architecture 2007 MMU . . . . . 24
3.1.3.3 Privileged Software . . . . . 25
3.1.4 Architectural Definition. . . . . 25
3.1.5 UltraSPARC Architecture 2007 Compliance with SPARC V9 Architecture 25
3.1.6 Implementation Compliance with UltraSPARC Architecture 2007 25
3.2 Processor Architecture . . . . . 26
3.2.1 Integer Unit (IU) . . . . . 26
3.2.2 Floating-Point Unit (FPU) . . . . . 26
3.3 Instructions . . . . . 27

3.3.1 Memory Access . . . . . 27
3.3.1.1 Memory Alignment Restrictions . . . . . 28
3.3.1.2 Addressing Conventions . . . . . 28
3.3.1.3 Addressing Range . . . . . 28
3.3.1.4 Load/Store Alternate . . . . . 28
3.3.1.5 Separate Instruction and Data Memories . . . . . 29
3.3.1.6 Input/Output (I/O) . . . . . 29
3.3.1.7 Memory Synchronization . . . . . 30
3.3.2 Integer Arithmetic / Logical / Shift Instructions . . . . . 30
3.3.3 Control Transfer . . . . . 30
3.3.4 State Register Access . . . . . 31
3.3.4.1 Ancillary State Registers . . . . . 31
3.3.4.2 PR State Registers . . . . . 31
3.3.4.3 HPR State Registers . . . . . 31
3.3.5 Floating-Point Operate . . . . . 32
3.3.6 Conditional Move . . . . . 32
3.3.7 Register Window Management . . . . . 32
3.3.8 SIMD . . . . . 32
3.4 Traps . . . . . 32
3.5 Chip-Level Multithreading (CMT) . . . . . 33

**4 Data Formats . . . . . 35**

4.1 Integer Data Formats . . . . . 36
4.1.1 Signed Integer Data Types . . . . . 37
4.1.1.1 Signed Integer Byte, Halfword, and Word . . . . . 37
4.1.1.2 Signed Integer Doubleword (64 bits) . . . . . 37
4.1.1.3 Signed Integer Extended-Word (64 bits) . . . . . 38
4.1.2 Unsigned Integer Data Types . . . . . 38
4.1.2.1 Unsigned Integer Byte, Halfword, and Word . . . . . 38
4.1.2.2 Unsigned Integer Doubleword (64 bits) . . . . . 39
4.1.2.3 Unsigned Extended Integer (64 bits) . . . . . 39
4.1.3 Tagged Word (32 bits) . . . . . 39
4.2 Floating-Point Data Formats . . . . . 40
4.2.1 Floating Point, Single Precision (32 bits) . . . . . 40
4.2.2 Floating Point, Double Precision (64 bits) . . . . . 41
4.2.3 Floating Point, Quad Precision (128 bits) . . . . . 42
4.2.4 Floating-Point Data Alignment in Memory and Registers . . . . 43
4.3 SIMD Data Formats . . . . . 43
4.3.1 Uint8 SIMD Data Format . . . . . 44
4.3.2 Int16 SIMD Data Formats . . . . . 44
4.3.3 Int32 SIMD Data Format . . . . . 44

**5 Registers . . . . . 47**

5.1 Reserved Register Fields . . . . . 48
5.2 General-Purpose R Registers . . . . . 48
5.2.1 Global R Registers . . . . . 50

5.2.2 Windowed R Registers . . . . . 50
5.2.3 Special R Registers . . . . . 54
5.3 Floating-Point Registers . . . . . 54
5.3.1 Floating-Point Register Number Encoding . . . . . 57
5.3.2 Double and Quad Floating-Point Operands . . . . . 58
5.4 Floating-Point State Register (FSR) . . . . . 60
5.4.1 Floating-Point Condition Codes (fcc0, fcc1, fcc2, fcc3) . . . . . 60
5.4.2 Rounding Direction (rd) . . . . . 61
5.4.3 Trap Enable Mask (tem) . . . . . 61
5.4.4 Nonstandard Floating-Point (ns) . . . . . 62
5.4.5 FPU Version (ver) . . . . . 62
5.4.6 Floating-Point Trap Type (ftt) . . . . . 62
5.4.7 FQ Not Empty (qne) . . . . . 65
5.4.8 Accrued Exceptions (aexc) . . . . . 65
5.4.9 Current Exception (cexc) . . . . . 66
5.4.10 Floating-Point Exception Fields . . . . . 67
5.4.11 FSR Conformance . . . . . 69
5.5 Ancillary State Registers . . . . . 69
5.5.1 32-bit Multiply/Divide Register (Y) (ASR 0) . . . . . 70
5.5.2 Integer Condition Codes Register (CCR) (ASR 2) . . . . . 71
5.5.2.1 Condition Codes (CCR.xcc and CCR.icc) . . . . . 71
5.5.3 Address Space Identifier (ASI) Register (ASR 3) . . . . . 73
5.5.4 Tick (TICK) Register (ASR 4) . . . . . 73
5.5.5 Program Counters (PC, NPC) (ASR 5) . . . . . 75
5.5.6 Floating-Point Registers State (FPRS) Register (ASR 6) . . . . . 75
5.5.7 General Status Register (GSR) (ASR 19) . . . . . 76
5.5.8 SOFTINT$^{P}$ Register (ASRs 20, 21, 22) . . . . . 77
5.5.8.1 SOFTINT_SET$^{P}$ Pseudo-Register (ASR 20) . . . . . 78
5.5.8.2 SOFTINT_CLR$^{P}$ Pseudo-Register (ASR 21) . . . . . 79
5.5.9 Tick Compare (TICK_CMPR$^{P}$) Register (ASR 23) . . . . . 79
5.5.10 System Tick (STICK) Register (ASR 24) . . . . . 80
5.5.11 System Tick Compare (STICK_CMPR$^{P}$) Register (ASR 25) . . 81
5.6 Register-Window PR State Registers . . . . . 82
5.6.1 Current Window Pointer (CWP$^{P}$) Register (PR 9) . . . . . 83
5.6.2 Savable Windows (CANSAVE$^{P}$) Register (PR 10) . . . . . 83
5.6.3 Restorable Windows (CANRESTORE$^{P}$) Register (PR 11) . . . . . 84
5.6.4 Clean Windows (CLEANWIN$^{P}$) Register (PR 12) . . . . . 84
5.6.5 Other Windows (OTHERWIN$^{P}$) Register (PR 13) . . . . . 84
5.6.6 Window State (WSTATE$^{P}$) Register (PR 14) . . . . . 85
5.6.7 Register Window Management . . . . . 85
5.6.7.1 Register Window State Definition . . . . . 85
5.6.7.2 Register Window Traps . . . . . 86
5.7 Non-Register-Window PR State Registers . . . . . 87
5.7.1 Trap Program Counter (TPC$^{P}$) Register (PR 0) . . . . . 87
5.7.2 Trap Next PC (TNPC$^{P}$) Register (PR 1) . . . . . 88
5.7.3 Trap State (TSTATE$^{P}$) Register (PR 2) . . . . . 89

5.7.4 Trap Type ($TT^P$) Register (PR 3) . . . . . 90
5.7.5 Trap Base Address ($TBA^P$) Register (PR 5) . . . . . 91
5.7.6 Processor State ($PSTATE^P$) Register (PR 6) . . . . . 91
5.7.7 Trap Level Register ($TL^P$) (PR 7) . . . . . 97
5.7.8 Processor Interrupt Level ($PIL^P$) Register (PR 8) . . . . . 99
5.7.9 Global Level Register ($GL^P$) (PR 16) . . . . . 99
5.8 HPR State Registers . . . . . 101
5.8.1 Hyperprivileged State ($HPSTATE^H$) Register (HPR 0) . . . . . 102
5.8.2 Hyperprivileged Trap State ($HTSTATE^H$) Register (HPR 1) . 103
5.8.3 Hyperprivileged Interrupt Pending ($HINTP^H$) Register (HPR 3) . 104
5.8.4 Hyperprivileged Trap Base Address ($HTBA^H$) Register (HPR 5) . 105
5.8.5 Hyperprivileged Implementation Version ($HVER^H$) Register (HPR 6) 106
5.8.6 Hyperprivileged System Tick Compare ($HSTICK_CMPR^H$) Register (HPR 31) 107

**6 Instruction Set Overview . . . . . 109**

6.1 Instruction Execution . . . . . 109
6.2 Instruction Formats . . . . . 110
6.3 Instruction Categories . . . . . 111
6.3.1 Memory Access Instructions . . . . . 111
6.3.1.1 Memory Alignment Restrictions . . . . . 112
6.3.1.2 Addressing Conventions . . . . . 113
6.3.1.3 Address Space Identifiers (ASIs). . . . . 117
6.3.1.4 Separate Instruction Memory . . . . . 119
6.3.2 Memory Synchronization Instructions. . . . . 120
6.3.3 Integer Arithmetic and Logical Instructions . . . . . 120
6.3.3.1 Setting Condition Codes . . . . . 120
6.3.3.2 Shift Instructions. . . . . 120
6.3.3.3 Set High 22 Bits of Low Word . . . . . 121
6.3.3.4 Integer Multiply/Divide . . . . . 121
6.3.3.5 Tagged Add/Subtract . . . . . 121
6.3.4 Control-Transfer Instructions (CTIs) . . . . . 121
6.3.4.1 Conditional Branches . . . . . 123
6.3.4.2 Unconditional Branches. . . . . 124
6.3.4.3 CALL and JMPL Instructions . . . . . 124
6.3.4.4 RETURN Instruction . . . . . 124
6.3.4.5 DONE and RETRY Instructions . . . . . 124
6.3.4.6 Trap Instruction (Tcc) . . . . . 125
6.3.4.7 DCTI Couples . . . . . 125
6.3.5 Conditional Move Instructions . . . . . 125
6.3.6 Register Window Management Instructions. . . . . 127
6.3.6.1 SAVE Instruction. . . . . 127
6.3.6.2 RESTORE Instruction . . . . . 127

6.3.6.3 SAVED Instruction . . . 128
6.3.6.4 RESTORED Instruction . . . 128
6.3.6.5 Flush Windows Instruction . . . 129
6.3.7 Ancillary State Register (ASR) Access . . . 129
6.3.8 Privileged Register Access . . . 129
6.3.9 Floating-Point Operate (FPop) Instructions . . . 129
6.3.10 Implementation-Dependent Instructions . . . 130
6.3.11 Reserved Opcodes and Instruction Fields . . . 130

**7 Instructions . . . 133**

7.31.1 FMUL8x16 Instruction . . . 205
7.31.2 FMUL8x16AU Instruction . . . 206
7.31.3 FMUL8x16AL Instruction . . . 206
7.31.4 FMUL8SUx16 Instruction . . . 207
7.31.5 FMUL8ULx16 Instruction . . . 207
7.31.6 FMULD8SUx16 Instruction . . . 208
7.31.7 FMULD8ULx16 Instruction . . . 209
7.34.1 FPACK16 . . . 214
7.34.2 FPACK32 . . . 215
7.34.3 FPACKFIX . . . 217
7.47.1 IMPDEP1 Opcodes . . . 239
7.47.1.1 Opcode Formats . . . 239
7.47.2 IMDEP2B Opcodes . . . 240
7.63.1 Memory Synchronization . . . 279
7.63.2 Synchronization of the Virtual Processor . . . 280
7.63.3 TSO Ordering Rules affecting Use of MEMBAR . . . 280
7.74.1 Exceptions . . . 299
7.74.2 Weak versus Strong Prefetches . . . 300
7.74.3 Prefetch Variants . . . 301
7.74.3.1 Prefetch for Several Reads (fcn = 0, $20(14_{16})$) . . . 302
7.74.3.2 Prefetch for One Read (fcn = 1, $21(15_{16})$) . . . 302
7.74.3.3 Prefetch for Several Writes (and Possibly Reads) (fcn = 2, $22(16_{16})$)302
7.74.3.4 Prefetch for One Write (fcn = 3, $23(17_{16})$) . . . 303
7.74.3.5 Prefetch Page (fcn = 4) . . . 303
7.74.3.6 Prefetch to Nearest Unified Cache (fcn = $17(11_{16})$) . . . 303
7.74.4 Implementation-Dependent Prefetch Variants (fcn = 16, 18, 19, and 24–31) 304
7.74.5 Additional Notes . . . 304

**8 IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007 . . . 391**

8.1 Traps Inhibiting Results . . . 391
8.2 Underflow Behavior . . . 392
8.2.1 Trapped Underflow Definition (ufm = 1) . . . 393
8.2.2 Untrapped Underflow Definition (ufm = 0) . . . 393
8.3 Integer Overflow Definition . . . 393

8.4 Floating-Point Nonstandard Mode . . . 394
8.5 Arithmetic Result Tables . . . 395
8.5.1 Floating-Point Add (FADD) . . . 396
8.5.2 Floating-Point Subtract (FSUB) . . . 397
8.5.3 Floating-Point Multiply . . . 397
8.5.4 Floating-Point Multiply-Add (FMADD) . . . 398
8.5.5 Floating-Point Negative Multiply-Add (FNMADD) . . . 399
8.5.6 Floating-Point Multiply-Subtract (FMSUB) . . . 400
8.5.7 Floating-Point Negative Multiply-Subtract (FNMSUB) . . . 401
8.5.8 Floating-Point Divide (FDIV) . . . 402
8.5.9 Floating-Point Square Root (FSQRT) . . . 402
8.5.10 Floating-Point Compare (FCMP, FCMPE) . . . 403
8.5.11 Floating-Point to Floating-Point Conversions (F<s | d | q>TO<s | d | q>) 404
8.5.12 Floating-Point to Integer Conversions (F<s | d | q>TO<i | x>) . 405
8.5.13 Integer to Floating-Point Conversions (F<i | x>TO<s | d | q>) . 406

**9 Memory . . . 407**

9.1 Memory Location Identification . . . 408
9.2 Memory Accesses and Cacheability . . . 408
9.2.1 Coherence Domains . . . 408
9.2.1.1 Cacheable Accesses . . . 409
9.2.1.2 Noncacheable Accesses . . . 409
9.2.1.3 Noncacheable Accesses with Side-Effect . . . 409
9.3 Memory Addressing and Alternate Address Spaces . . . 411
9.3.1 Memory Addressing Types . . . 411
9.3.2 Memory Address Spaces . . . 412
9.3.3 Address Space Identifiers . . . 413
9.4 SPARC V9 Memory Model . . . 415
9.4.1 SPARC V9 Program Execution Model . . . 415
9.4.2 Virtual Processor/Memory Interface Model . . . 417
9.5 The UltraSPARC Architecture Memory Model — TSO . . . 418
9.5.1 Memory Model Selection . . . 419
9.5.2 Programmer-Visible Properties of the UltraSPARC Architecture TSO Model 420
9.5.3 TSO Ordering Rules . . . 421
9.5.4 Hardware Primitives for Mutual Exclusion . . . 422
9.5.4.1 Compare-and-Swap (CASA, CASXA) . . . 423
9.5.4.2 Swap (SWAP) . . . 423
9.5.4.3 Load Store Unsigned Byte (LDSTUB) . . . 423
9.5.5 Memory Ordering and Synchronization . . . 423
9.5.5.1 Ordering MEMBAR Instructions . . . 424
9.5.5.2 Sequencing MEMBAR Instructions . . . 425
9.5.5.3 Synchronizing Instruction and Data Memory . . . 426
9.6 Nonfaulting Load . . . 427

9.7 Store Coalescing . . . . . 428

**10 Address Space Identifiers (ASIs) . . . . . 429**

10.1 Address Space Identifiers and Address Spaces . . . . . 429
10.2 ASI Values . . . . . 429
10.3 ASI Assignments . . . . . 430
10.3.1 Supported ASIs . . . . . 431
10.4 Special Memory Access ASIs . . . . . 444
10.4.1 ASIs $10_{16}$, $11_{16}$, $16_{16}$, $17_{16}$ and $18_{16}$ (`ASI_*AS_IF_USER_*`) . . 445
10.4.2 ASIs $18_{16}$, $19_{16}$, $1E_{16}$, and $1F_{16}$ (`ASI_*AS_IF_USER_*_LITTLE`) . 445
10.4.3 ASI $14_{16}$ (`ASI_REAL`) . . . . . 446
10.4.4 ASI $15_{16}$ (`ASI_REAL_IO`) . . . . . 447
10.4.5 ASI $1C_{16}$ (`ASI_REAL_LITTLE`) . . . . . 447
10.4.6 ASI $1D_{16}$ (`ASI_REAL_IO_LITTLE`) . . . . . 447
10.4.7 ASIs $22_{16}$, $23_{16}$, $27_{16}$, $2A_{16}$, $2B_{16}$, $2F_{16}$ (Privileged Load Integer Twin Extended Word) 447
10.4.8 ASIs $26_{16}$ and $2E_{16}$ (Privileged Load Integer Twin Extended Word, Real Addressing) 448
10.4.9 ASIs $30_{16}$, $31_{16}$, $36_{16}$, $38_{16}$, $39_{16}$, $3E_{16}$ (`ASI_AS_IF_PRIV_*`) . 449
10.4.10 ASIs $E2_{16}$, $E3_{16}$, $EA_{16}$, $EB_{16}$ (Nonprivileged Load Integer Twin Extended Word) 450
10.4.11 Block Load and Store ASIs . . . . . 451
10.4.12 Partial Store ASIs . . . . . 452
10.4.13 Short Floating-Point Load and Store ASIs . . . . . 452
10.5 ASI-Accessible Registers . . . . . 452
10.5.1 Privileged Scratchpad Registers (`ASI_SCRATCHPAD`) . . . . . 453
10.5.2 Hyperprivileged Scratchpad Registers (`ASI_HYP_SCRATCHPAD`) 454
10.5.3 CMT Registers Accessed Through ASIs . . . . . 454
10.5.4 ASI Changes in the UltraSPARC Architecture . . . . . 454

**11 Performance Instrumentation . . . . . 457**

11.1 High-Level Requirements . . . . . 457
11.1.1 Usage Scenarios . . . . . 457
11.1.2 Metrics . . . . . 459
11.1.3 Accuracy Requirements . . . . . 459
11.2 Performance Counters and Controls . . . . . 460
11.2.1 Counter Overflow . . . . . 460

**12 Traps . . . . . 463**

12.1 Virtual Processor Privilege Modes . . . . . 464
12.2 Virtual Processor States, Normal Traps, and `RED_state` Traps . . . . . 466
12.2.1 `RED_state` . . . . . 467
12.2.1.1 `RED_state` Execution Environment . . . . . 468

12.2.1.2 RED_state Entry Traps . . . . . 469
12.2.1.3 RED_state Software Considerations . . . . . 470
12.2.1.4 Usage of Trap Levels . . . . . 470
12.2.2 error_state . . . . . 470
12.3 Trap Categories . . . . . 471
12.3.1 Precise Traps . . . . . 471
12.3.2 Deferred Traps . . . . . 471
12.3.3 Disrupting Traps . . . . . 473
12.3.3.1 Disrupting versus Precise and Deferred Traps . . . . . 473
12.3.3.2 Causes of Disrupting Traps . . . . . 474
12.3.3.3 Conditioning of Disrupting Traps . . . . . 474
12.3.3.4 Trap Handler Actions for Disrupting Traps . . . . . 475
12.3.3.5 Clearing Requirement for Disrupting Traps . . . . . 475
12.3.4 Reset Traps . . . . . 476
12.3.5 Uses of the Trap Categories . . . . . 477
12.4 Trap Control . . . . . 477
12.4.1 PIL Control . . . . . 478
12.4.2 FSR.tem Control . . . . . 478
12.5 Trap-Table Entry Addresses . . . . . 479
12.5.1 Trap-Table Entry Address to Privileged Mode . . . . . 479
12.5.2 Privileged Trap Table Organization . . . . . 480
12.5.3 Trap-Table Entry Address to Hyperprivileged Mode . . . . . 480
12.5.4 Hyperprivileged Trap Table Organization . . . . . 481
12.5.5 Trap Table Entry Address to RED_state . . . . . 481
12.5.6 RED_state Trap Table Organization . . . . . 482
12.5.7 Trap Type (TT) . . . . . 482
12.5.7.1 Trap Type for Spi ll/Fill Traps . . . . . 496
12.5.8 Trap Priorities . . . . . 496
12.6 Trap Processing . . . . . 497
12.6.1 Normal Trap Processing . . . . . 499
12.6.2 RED_state Trap Processing . . . . . 501
12.6.2.1 Nonreset Traps with TL = *MAXTL* – 1 . . . . . 501
12.6.2.2 Power-On Reset (POR) Traps . . . . . 503
12.6.2.3 Watchdog Reset (WDR) Traps . . . . . 504
12.6.2.4 Externally Initiated Reset (XIR) Traps . . . . . 505
12.6.2.5 Software-Initiated Reset (SIR) Traps . . . . . 506
12.6.2.6 Nonreset Traps When the Virtual Processor Is in RED_state 507
12.7 Exception and Interrupt Descriptions . . . . . 508
12.7.1 SPARC V9 Traps Not Used in UltraSPARC Architecture 2007 521
12.8 Register Window Traps . . . . . 522
12.8.1 Window Spill and Fill Traps . . . . . 522
12.8.2 *clean_window* Trap . . . . . 523
12.8.3 Vectoring of Fill/Spill Traps . . . . . 523
12.8.4 CWP on Window Traps . . . . . 524
12.8.5 Window Trap Handlers . . . . . 524

**13 Interrupt Handling . . . . . 525**

13.1 Interrupt Packets . . . . . 526
13.2 Software Interrupt Register (SOFTINT) . . . . . 526
13.2.1 Setting the Software Interrupt Register . . . . . 526
13.2.2 Clearing the Software Interrupt Register . . . . . 527
13.3 Interrupt Queues . . . . . 527
13.3.1 Interrupt Queue Registers . . . . . 527
13.4 Interrupt Traps . . . . . 529
13.5 Strand Interrupt ID Register (STRAND_INTR_ID) . . . . . 530

**14 Memory Management . . . . . 531**

14.1 Virtual Address Translation . . . . . 532
14.2 Hyperprivileged Memory Management Architecture . . . . . 536
14.2.1 Partition ID . . . . . 536
14.2.2 Real Address Translation . . . . . 537
14.3 Context ID . . . . . 537
14.4 TSB Translation Table Entry (TTE) . . . . . 539
14.5 Translation Storage Buffer (TSB) . . . . . 543
14.5.1 TSB Indexing Support . . . . . 544
14.5.2 TSB Cacheability and Consistency . . . . . 544
14.5.3 TSB Organization . . . . . 544
14.5.4 TSB Configuration . . . . . 545
14.6 Hardware Support for TSB Access . . . . . 546
14.6.1 Hardware Tablewalk . . . . . 546
14.6.1.1 Typical Hardware Tablewalk Sequence . . . . . 547
14.6.2 Typical TLB Software Miss Sequence . . . . . 548
14.7 Faults and Traps . . . . . 549
14.8 MMU Operation Summary . . . . . 552
14.9 ASI Value, Context ID, and Endianness Selection for Translation . . . . . 555
14.10 Translation . . . . . 558
14.10.1 MMU Behavior During Reset and Upon Entering RED_state 563
14.10.1.1 MMU Bypass . . . . . 564
14.10.1.2 MMU Disabled Behavior . . . . . 564
14.11 SPARC V9 “MMU Attributes” . . . . . 565
14.12 MMU Internal Registers and ASI Operations . . . . . 566
14.12.1 Accessing MMU Registers . . . . . 566
14.12.2 Context ID Registers . . . . . 567
14.12.3 Partition ID Register . . . . . 569
14.12.4 MMU Real Range Registers . . . . . 569
14.12.5 MMU Physical Offset Registers . . . . . 571
14.12.6 TSB Configuration Registers . . . . . 572
14.12.7 I/D/U TSB Pointer Registers . . . . . 573
14.12.8 Synchronous Fault Addresses . . . . . 575
14.12.8.1 DMMU Synchronous Fault Address Register . . . . . 575

14.12.8.2 Instruction Synchronous Fault Address . . . . . 576
14.12.9 I/D/U TLB Tag Access, Data In, Data Access, and Tag Read Registers 576
14.12.9.1 I/D/U MMU TLB Tag Access Registers . . . . . 578
14.12.9.2 I/D/UMMU TLB Data In Register . . . . . 580
14.12.9.3 I/D/U MMU TLB Data Access Register . . . . . 581
14.12.9.4 I/D/UMMU TLB Tag Read Register . . . . . 582
14.12.10 I/D/UMMU TLB Tag Target Registers . . . . . 585
14.12.11 I/D/UMMU Demap . . . . . 585
14.12.12 Tablewalk Pending Registers . . . . . 588
14.12.12.1 Tablewalk Pending Control Register . . . . . 589
14.12.12.2 Tablewalk Pending Status Register . . . . . 589
14.13 Translation Lookaside Buffer Hardware . . . . . 590
14.13.1 TLB Operations . . . . . 590

**15 Chip-Level Multithreading (CMT) . . . . . 593**

15.1 Overview of CMT . . . . . 593
15.1.1 CMT Definition . . . . . 594
15.1.1.1 Background Terminology . . . . . 594
15.1.1.2 CMT Definition . . . . . 596
15.1.2 General CMT Behavior . . . . . 597
15.2 Accessing CMT Registers . . . . . 597
15.2.1 Classes of CMT Registers . . . . . 598
15.2.2 Accessing CMT Registers Through ASIs . . . . . 599
15.3 CMT Registers . . . . . 600
15.3.1 Strand ID Register (STRAND_ID) . . . . . 601
15.3.1.1 Exposing Stranding . . . . . 602
15.3.2 Strand Interrupt ID Register (STRAND_INTR_ID) . . . . . 602
15.3.2.1 Assigning an Interrupt ID . . . . . 603
15.3.2.2 Dispatching and Receiving Interrupts . . . . . 603
15.3.2.3 Updating the Strand Interrupt ID Register . . . . . 604
15.4 Disabling and Parking Virtual Processors . . . . . 604
15.4.1 Strand Available Register (STRAND_AVAILABLE) . . . . . 605
15.4.2 Enabling and Disabling Virtual Processors . . . . . 605
15.4.2.1 Strand Enable Status Register (STRAND_ENABLE_STATUS) 606
15.4.2.2 Strand Enable Register (STRAND_ENABLE) . . . . . 606
15.4.2.3 Dynamically Enabling/Disabling Virtual Processors 608
15.4.3 Parking and Unparking Virtual Processors . . . . . 608
15.4.3.1 Strand Running Register (STRAND_RUNNING) . . 609
15.4.3.2 Strand Running Status Register (STRAND_RUNNING_STATUS) 612
15.4.4 Virtual Processor Standby (or Wait) State . . . . . 613
15.5 Reset and Trap Handling . . . . . 614
15.5.1 Per-Strand Resets (SIR and WDR Resets) . . . . . 614
15.5.2 Full-Processor Resets (POR and WRM Resets) . . . . . 614

15.5.2.1 Boot Sequence . . . 615
15.5.3 Partial Processor Resets (XIR Reset) . . . 615
15.5.3.1 XIR Steering Register (XIR_STEERING) . . . 616
15.6 Error Handling in CMT Processors . . . 617
15.6.1 Virtual-Processor-Specific Error Reporting . . . 617
15.6.2 Reporting Errors on Shared Structures . . . 618
15.6.2.1 Error Steering . . . 618
15.6.2.2 Reporting Non-Virtual-Processor-Specific Errors . . . 621
15.7 Additional CMT Software Interfaces . . . 622
15.7.1 Diagnostic/RAS Registers . . . 622
15.7.2 Configuration Registers . . . 622
15.7.3 Performance Registers . . . 622
15.7.4 Booting Support . . . 622
15.8 Performance Issues for CMT Processors . . . 623
15.9 Recommended Subset for Single-Strand Processors . . . 623
15.10 Machine State Summary . . . 625

**16 Resets . . . 627**

16.1 Resets . . . 627
16.1.1 Power-on Reset (POR) . . . 628
16.1.2 Warm Reset (WMR) . . . 629
16.1.3 Externally Initiated Reset (XIR) . . . 629
16.1.4 Watchdog Reset (WDR) . . . 630
16.1.5 Software-Initiated Reset (SIR) . . . 630
16.2 Machine States . . . 630
16.2.1 Machines States for CMT . . . 634

**17 Error Handling . . . 637**

17.1 Error Reporting . . . 637
17.1.1 Precise Traps . . . 638
17.1.2 Deferred Traps . . . 638
17.1.3 Disrupting Exceptions . . . 639
17.1.3.1 Disrupting Traps . . . 639
17.1.4 Fatal Error Signaling . . . 640
17.2 NotData Overview . . . 641
17.2.1 Notdata Requirement . . . 641
17.3 Error Status Registers . . . 642
17.3.1 Elements of an Event Status Register (ESR) . . . 643
17.4 Protection, Detection, Reporting, and Handling of Errors . . . 646
17.4.1 L1 (Level-1) Caches . . . 646
17.4.2 TLB Errors . . . 647
17.4.2.1 Hardware-Corrected TLB Errors . . . 647
17.4.2.2 Software-Corrected TLB Errors . . . 648
17.4.3 Register File Errors . . . 648
17.4.4 Execution Unit Errors . . . 649

17.4.5 Other Core Errors Associated with Instruction Processing Before Instruction Retirement 649
17.4.6 Store Errors . . . . . 649
17.4.7 Errors Not Associated with Instruction Processing . . . . . 650
17.4.8 L2 Cache Errors . . . . . 651
17.4.9 External Interface and Bus Errors . . . . . 652
17.5 Error Handling for Common Processor Errors . . . . . 653

**A Opcode Maps . . . . . 657**

**B Implementation Dependencies . . . . . 669**

B.1 Definition of an Implementation Dependency . . . . . 669
B.2 Hardware Characteristics . . . . . 670
B.3 Implementation Dependency Categories . . . . . 670
B.4 List of Implementation Dependencies . . . . . 671

**C Assembly Language Syntax . . . . . 695**

C.1 Notation Used . . . . . 695
C.1.1 Register Names . . . . . 696
C.1.2 Special Symbol Names . . . . . 697
C.1.3 Values . . . . . 699
C.1.4 Labels . . . . . 700
C.1.5 Other Operand Syntax . . . . . 700
C.1.6 Comments . . . . . 701
C.2 Syntax Design . . . . . 702
C.3 Synthetic Instructions . . . . . 702

**. . . . . Indexi**

# Preface

First came the 32-bit SPARC Version 7 (V7) architecture, publicly released in 1987. Shortly after, the SPARC V8 architecture was announced and published in book form. The 64-bit SPARC V9 architecture was released in 1994. Now, the UltraSPARC Architecture specification provides the first significant update in over 10 years to Sun's SPARC processor architecture.

## What's New?

UltraSPARC Architecture 2007 pulls together in one document all parts of the architecture:

- the nonprivilged (Level 1) architecture from SPARC V9
- most of the privileged (Level 2) architecture from SPARC V9
- more in-depth coverage of all SPARC V9 features

Plus, it includes all of Sun's now-standard architectural extensions (beyond SPARC V9), developed through the processor generations of UltraSPARC III, IV, IV+, and T1:

- the VIS™ 1 and VIS 2 instruction set extensions and the associated GSR register
- multiple levels of global registers, controlled by the GL register
- Sun's 64-bit MMU architecture
- privileged instructions ALLCLEAN, OTHERW, NORMALW, and INVALW
- access to the VER register is now hyperprivileged (and VER was renamed the HVER register)
- the SIR instruction is now hyperprivileged
- new hyperprivileged instructions RDHPR and WRHPR

- the new Hyperprivileged mode
- Chip-level Multithreading (CMT) architecture

UltraSPARC Architecture 2007 includes the following changes since :

- replacement of *instruction_address_exception* and *data_acess_exception* exceptions by multiple *IAE_** and *DAE_** exceptions
- FSR.ftt = 3 (unimplemented_FPop) has been retired; all unimplemented FPops now generate the *illegal_instruction* exception instead of *fp_exception_other* with FSR.ftt = 3 (unimplemented_FPop).

In addition, architectural features are now tagged with Software Classes and Implementation Classes[1]. Software Classes provide a new, high-level view of the expected architectural longevity and portability of software that references those features. Implementation Classes give an indication of how efficiently each feature is likely to be implemented across current and future UltraSPARC Architecture processor implementations. This information provides guidance that should be particularly helpful to programmers who write in assembly language or those who write tools that generate SPARC instructions. It also provides the infrastructure for defining clear procedures for adding and removing features from the architecture over time, with minimal software disruption.

# Acknowledgements

This specification builds upon all previous SPARC specifications — SPARC V7, V8, and especially, SPARC V9. It therefore owes a debt to all the pioneers who developed those architectures.

SPARC V7 was developed by the SPARC ("Sunrise") architecture team at Sun Microsystems, with special assistance from Professor David Patterson of University of California at Berkeley.

The enhancements present in SPARC V8 were developed by the nine member companies of the SPARC International Architecture Committee: Amdahl Corporation, Fujitsu Limited, ICL, LSI Logic, Matsushita, Philips International, Ross Technology, Sun Microsystems, and Texas Instruments.

SPARC V9 was also developed by the SPARC International Architecture Committee, with key contributions from the individuals named in the Editor's Notes section of *The SPARC Architecture Manual-Version 9*.

1. although most features in this specification are already tagged with Software Classes, the full description of those Classes does not appear in this version of the specification. Please check back (`http://opensparc.sunsource.net/nonav/opensparct1.html`) for a later release of this document, which *will* include that description

The voluminous enhancements and additions present in this *UltraSPARC Architecture 2007* specification are the result of **years** of deliberation, review, and feedback from readers of earlier Sun-internal revisions. I would particularly like to acknowledge the following people for their key contributions:

- The UltraSPARC Architecture working group, who reviewed dozens of drafts of this specification and strived for the highest standards of accuracy and completeness; its active members included: Hendrik-Jan Agterkamp, Paul Caprioli, Steve Chessin, Hunter Donahue, Greg Grohoski, John (JJ) Johnson, Paul Jordan, Jim Laudon, Jim Lewis, Bob Maier, Wayne Mesard, Greg Onufer, Seongbae Park, Joel Storm, David Weaver, and Tom Webber.
- Robert (Bob) Maier, for expansion of exception descriptions in every page of the Instructions chapter, major re-writes of 7 chapters and appendices (*Memory*, *Memory Management*, *Performance Instrumentation*, *Error Handling*, *Resets*, and *Interrupt Handling*), significant updates to 5 other chapters, and tireless efforts to infuse commonality wherever possible across implementations.
- Steve Chessin and Joel Storm, "ace" reviewers — the two of them spotted more typographical errors and small inconsistencies than all other reviewers combined
- Jim Laudon (an UltraSPARC T1 architect and author of that processor's implementation specification), for numerous descriptions of new features which were merged into this specicification
- The working group responsible for developing the system of Software Classes and Implementation Classes, comprising: Steve Chessin, Yuan Chou, Peter Damron, Q. Jacobson, Nicolai Kosche, Bob Maier, Ashley Saulsbury, Lawrence Spracklen, and David Weaver.
- Lawrence Spracklen, for his advice and numerous contributions regarding descriptions of VIS instructions
- Tom Webber, for providing descriptions of several new features in UltraSPARC Architecture 2007

I hope you find the *UltraSPARC Architecture 2007* specification more complete, accurate, and readable than its predecessors.

— *David Weaver*
UltraSPARC Architecture coordinator and specification editor

Corrections and other comments regarding this specification can be emailed to:
`UA-editor@sun.com`

CHAPTER 1

# Document Overview

This chapter discusses:


- **Navigating UltraSPARC Architecture 2007** on page 1.
- **Fonts and Notational Conventions** on page 2.
- **Reporting Errors in this Specification** on page 5.


## 1.1 Navigating *UltraSPARC Architecture 2007*

If you are new to the SPARC architecture, read Chapter 3, *Architecture 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 the SPARC V9 architecture but not UltraSPARC Architecture 2007, note that UltraSPARC Architecture 2007 conforms to the SPARC V9 Level 1 architecture (and most of Level 2), with numerous extensions — particularly with respect to CMT features, VIS instructions, and support for hyperprivileged-mode operation.

This specfication is structured as follows:

- Chapter 2, *Definitions*, which defines key terms used throughout the specification
- Chapter 3, *Architecture Overview*, provides an overview of UltraSPARC Architecture 2007
- Chapter 4, *Data Formats*, describes the supported data formats
- Chapter 5, *Registers*, describes the register set
- Chapter 6, *Instruction Set Overview*, provides a high-level description of the UltraSPARC Architecture 2007 instruction set

- Chapter 7, *Instructions*, describes the UltraSPARC Architecture 2007 instruction set in great detail
- Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*, describes the trap model
- Chapter 9, *Memory* describes the supported memory model
- Chapter 10, *Address Space Identifiers (ASIs)*, provides a complete list of supported ASIs
- Chapter 11, *Performance Instrumentation* describes the architecture for performance monitoring hardware
- Chapter 12, *Traps*, describes the trap model
- Chapter 13, *Interrupt Handling*, describes how interrupts are handled
- Chapter 14, *Memory Management*, describes MMU operation
- Chapter 15, *Chip-Level Multithreading (CMT)*, describes the new CMT features
- Chapter 16, *Resets*, describes resets, `RED_state`, and `error_state`.
- Chapter 17, *Error Handling*, describes handling of detected errors
- Appendix A, *Opcode Maps*, provides the overall picture of how the instruction set is mapped into opcodes
- Appendix B, *Implementation Dependencies*, describes all implementation dependencies
- Appendix C, *Assembly Language Syntax*, describes extensions to the SPARC assembly language syntax; in particular, synthetic instructions are documented in this appendix

## 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 terms where substitution is expected, for example, "`fcc`*n*", "virtual processor *n*", or "*reg_plus_imm*".
- *Italic sans serif* font is used for exception and trap names. For example, "The *privileged_action* exception...."
- lowercase helvetica font is used for register field names (named bits) and instruction field names, for example: "The rs1 field contains...."
- UPPERCASE HELVETICA font is used for register names; for example, FSR.

- TYPEWRITER (Courier) font is used for literal values, such as code (assembly language, C language, ASI names) and for state names. For example: %f0, ASI_PRIMARY, execute_state.
- When a register field is shown along with its containing register name, they are separated by a period ('.'), for example, "FSR.cexc".
- UPPERCASE words are acronyms or instruction names. Some common acronyms appear in the glossary in Chapter 2, *Definitions*. **Note:** Names of some instructions contain both upper- and lower-case letters.
- An underscore character joins words in register, register field, exception, and trap names. **Note:** Such words may 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:

- The left arrow symbol ( ← ) is the assignment operator. For example, "PC ← PC + 1" means that the Program Counter (PC) is incremented by 1.
- Square brackets ( [ ] ) are used in two different ways, distinguishable by the context in which they are used:
  - Square brackets indicate indexing into an array. For example, TT[TL] means the element of the Trap Type (TT) array, as indexed by the contents of the Trap Level (TL) register.
  - Square brackets are also used to indicate optional additions/extensions to symbol names. For example, "ST[D|Q]F" expands to all three of "STF", "STDF", and "STQF". Similarly, ASI_PRIMARY[_LITTLE] indicates two related address space identifiers, ASI_PRIMARY and ASI_PRIMARY_LITTLE. (Contrast with the use of angle brackets, below)
- Angle brackets ( < > ) indicate mandatory additions/extensions to symbol names. For example, "ST<D|Q>F" expands to mean "STDF" and "STQF". (Contrast with the second use of square brackets, above)
- Curly braces ( { } ) indicate a bit field within a register or instruction. For example, CCR{4} refers to bit 4 in the Condition Code Register.
- A consecutive set of values is indicated by specifying the upper and lower limit of the set separated by a colon ( : ), for example, CCR{3:0} refers to the set of four least significant bits of register CCR. (Contrast with the use of double periods, below)
- A double period ( .. ) indicates any *single* intermediate value between two given end values is possible. For example, NAME[2..0] indicates four forms of NAME exist: NAME, NAME2, NAME1, and NAME0; whereas NAME<2..0> indicates that three forms exist: NAME2, NAME1, and NAME0. (Contrast with the use of the colon, above)

- A vertical bar ( | ) separates mutually exclusive alternatives inside square brackets ( [ ] ), angle brackets ( < > ), or curly braces ( { } ). For example, “NAME[A|B]” expands to “NAME, NAMEA, NAMEB” and “NAME<A|B>” expands to "NAMEA, NAMEB".
- The asterisk ( * ) is used as a wild card, encompassing the full set of valid values. For example, FCMP* refers to FCMP with all valid suffixes (in this case, FCMP<s|d|q> and FCMPE<s|d|q>). An asterisk is typically used when the full list of valid values either is not worth listing (because it has little or no relevance in the given context) or the valid values are too numerous to list in the available space.
- The slash ( / ) is used to separate paired or complementary values in a list, for example, “the LDBLOCKF/STBLOCKF instruction pair ....”
- The double colon (**::**) is an operator that indicates concatenation (typically, of bit vectors). Concatenation strictly strings the specified component values into a single longer string, in the order specified. The concatenation operator performs no arithmetic operation on any of the component values.

## 1.2.1 Implementation Dependencies

Implementors of UltraSPARC Architecture 2007 processors are allowed to resolve some aspects of the architecture in machine-dependent ways.

The *definition* of each implementation dependency is indicated by the notation “**IMPL. DEP. #*nn*-*XX***: Some descriptive text”. The number ***nn*** provides an index into the complete list of dependencies in Appendix B, *Implementation Dependencies*.

A *reference* to (but not definition of) an implementation dependency is indicated by the notation “(impl. dep. #*nn*)”.

## 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 hexadecimal numbers within the text have spaces inserted every four characters to improve readability. Within C language 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:

**Note** | General notes contain incidental information relevant to the paragraph preceding the note.

**Programming Note** | Programming notes contain incidental information about how software can use an architectural feature.

**Implementation Note** | An Implementation Note contains incidental information, describing how an UltraSPARC Architecture 2007 processor might implement an architectural feature.

**V9 Compatibility Note** | Note containing information about possible differences between UltraSPARC Architecture 2007 and SPARC V9 implementations. Such information is relevant to UltraSPARC Architecture 2007 implementations and might not apply to other SPARC V9 implementations.

**Forward Compatibility Note** | Note containing information about how the UltraSPARC Architecture is expected to evolve in the future. Such notes are not intended as a guarantee that the architecture will evolve as indicated, but as a guide to features that should not be depended upon to remain the same, by software intended to run on both current and future implementations.

## 1.3 Reporting Errors in this Specification

This specification has been reviewed for completeness and accuracy. Nonetheless, as with any document this size, errors and omissions may occur, and reports of such are welcome. Please send "bug reports" and other comments on this document to the email address: `UA-editor@sun.com`

CHAPTER 2

# Definitions

This chapter defines concepts and terminology common to all implementations of UltraSPARC Architecture 2007.

**address space** A range of $2^{64}$ locations that can be addressed by instruction fetches and load, store, or load-store instructions. See also **address space identifier (ASI)**.

**address space identifier (ASI)** An 8-bit value that identifies a particular address space. An ASI is (implicitly or explicitly) associated with every instruction access or data access. See also **implicit ASI**.

**aliased** Said of each of two virtual or real addresses that refer to the same underlying memory location.

**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.

**available (virtual processor)** A virtual processor that is physically present and functional, that can be enabled and used.

**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** (Obsolete) abbreviation for Block Load instruction; replaced by LDBLOCKF.

**BST** (Obsolete) abbreviation for Block Store instruction; replaced by STBLOCKF.

**byte** Eight consecutive bits of data, aligned on an 8-bit boundary.

**CCR** Abbreviation for Condition Codes Register.

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

**cleared** A term applied to an error when the originating incorrect signal or datum is set to a value that is not in error. An originating incorrect signal that is stored in a memory (a stored error) may be cleared automatically by hardware action or may need software action to clear it. An originating incorrect signal that is not stored in any memory needs no action to clear it. (For this definition, "memory" includes caches, registers, flip-flops, latches, and any other mechanism for storing information, and not just what is usually considered to be system memory.)

CMT Chip-level MultiThreading (or, as an adjective, Chip-level MultiThreaded). Refers to a physical processor containing more than one virtual processor.

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

**completed (memory operation)** Said of a memory transaction 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.

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

**context ID** A numeric value that uniquely identifies a particular context.

**copyback** The process of sending a copy of the data from a cache line owned by a physical processor core, in response to a snoop request from another device.

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

**core** In an UltraSPARC Architecture processor, may refer to either a virtual processor or a physical processor core.

**correctable** A term applied to an error when at the time the error occurs, the error detector knows that enough information exists, either accompanying the incorrect signal or datum or elsewhere in the system, to correct the error. Examples include parity errors on clean L1s, which are corrected by invalidation of the line and refetching of the data from higher up in the memory hierarchy, and correctable errors on L2s. See also **uncorrectable**.

**corrected** A term applied to an error when the incorrect signal or datum is replaced by the correct signal or datum, perhaps in a downstream location. Depending on the circuit, correcting an error may or may not clear it.

**cross-call** An interprocessor call in a system containting multiple virtual processors.

**CTI** Abbreviation for **control-transfer instruction**.

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

**data access (instruction)** A load, store, load-store, or FLUSH instruction.

**DCTI** Delayed control transfer instruction.

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

**denormalized number** Synonym for **subnormal number**.

**deprecated** The term applied to an architectural feature (such as an instruction or register) for which an UltraSPARC Architecture 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 UltraSPARC Architecture software and may not be supported in future versions of the architecture.

**disable (core)** The process of changing the state of a virtual processor to `Disabled`, during which all other processor state (including cache coheriency) may be lost and all interrupts to that virtual processor will be discarded. See also **park** and **enable**.

**disabled (core)** A virtual processor that is out of operation (not executing instructions, not participating in cache coherency, and discarding interrupts). See also **parked** and **enabled**.

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

**D-SFAR** Data Synchronous Fault Address register.

**enable (core)** The process of moving a virtual processor from `Disabled` to `Enabled` state and preparing it for operation. See also **disable** and **park**.

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

**error** A signal or datum that is wrong. The error can be created by some problem internal to the processor, or it can appear at inputs to the processor. An error can propagate through fault-free circuitry and appear as an error at the output. It can be stored in a memory, whether program-visible or not, and can later be either read out of the memory or overwritten.

**ESR** Abbreviation for Error Status Register.

**even parity** The mode of parity checking in which each combination of data bits plus a parity bit contains an even number of '1' bits.

**exception** A condition that makes it impossible for the processor to continue executing the current instruction stream. Some exceptions may be masked (that is, trap generation disabled — for example, floating-point exceptions masked by FSR.tem) 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**.

**explicit ASI** An ASI that that is provided by a load, store, or load-store alternate instruction (either from its imm_asi field or from the ASI register).

**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.

**fault** A physical condition that causes a device, a component, or an element to fail to perform in a required manner; for example, a short-circuit, a broken wire, or an intermittent connection.

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

**FGU** Floating-point and Graphics Unit (which most implementations specify as a superset of **FPU**).

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

**F register** A floating-point register. The SPARC V9 architecture includes single-, double-, and quad-precision F registers.

**floating-point operate instructions** Instructions that perform floating-point calculations, as defined in *Floating-Point Operate (FPop) Instructions* on page 129. FPop instructions do not include FBfcc instructions, loads and stores between memory and the F registers, or non-floating-point operations that read or write F registers.

**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** Abbreviation for **floating-point operate** (instructions).

**FPRS** Floating-Point Register State register.

**FPU** Floating-Point Unit.

**FSR** Floating-Point Status register.

**GL** Global Level register.

**GSR** General Status register.

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

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

**hypervisor (software)** A layer of software that executes in hyperprivileged processor state. One purpose of hypervisor software (also referred to as "the hypervisor") is to provide greater isolation between operating system ("supervisor") software and the underlying processor implementation.

**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 UltraSPARC Architecture that can legitimately vary among implementations. In many cases, the permitted range of variation is specified. When a range is specified, compliant implementations must not deviate from that range.

**implicit ASI** An address space identifier that is implicitly supplied by the virtual processor on all instruction accesses and on data accesses that do not explicitly provide an ASI value (from either an imm_asi instruction field or the ASI register).

**initiated** Synonym for **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 UltraSPARC Architecture 2007 instruction set architecture.

**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.

**inter-strand** Describes an operation that crosses virtual processor (strand) boundaries.

**intra-strand** Describes an operation that occurs entirely within one virtual processor (strand).

**invalid (ASI or address)** Undefined, reserved, or illegal.

**ISA** Instruction set architecture.

**issued** 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 for **initiated**.

**IU** Integer Unit.

**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* includes loads into integer or floating-point registers, block loads, 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** The address translation hardware in an UltraSPARC Architecture implementation that translates 64-bit virtual address into underlying physical addresses. The MMU is composed of the TLBs, ASRs, and ASI registers used to manage address translation. See also **context**, **physical address**, **real address**, and **virtual address**.

**MMU** Abbreviation for **Memory Management Unit**.

**multiprocessor system** A system containing more than one processor.

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

**next program counter** Conceptually, 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 regular load triggers an exception whereas a nonfaulting load appears to ignore the exception and loads its destination register with a value of zero (on an UltraSPARC Architecture processor, hardware treats regular and nonfaulting loads identically; the distinction is made in trap handler software). Contrast with **speculative load**.

**nonprivileged** An adjective that describes
(1) the state of the virtual processor when PSTATE.priv = 0 and HPSTATE.hpriv = 0, that is, when it is in nonprivileged mode;
(2) virtual processor state information that is accessible to software regardless of the current privilege mode; for example, nonprivileged registers, nonprivileged ASRs, or, in general, nonprivileged state;
(3) an instruction that can be executed in any privilege mode (hyperprivileged, privileged, or nonprivileged).

**nonprivileged mode** The mode in which a virtual processor is operating when executing application software (at the lowest privilege level). Nonprivileged mode is defined by PSTATE.priv = 0 and HSTATE.hpriv = 0. See also **privileged** and **hyperprivileged**.

**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.

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

**NPC** Next program counter.

**npt** Nonprivileged trap.

**nucleus software** Privileged software running at a trap level greater than 0 (TL> 0).

**NUMA** Nonuniform memory access.

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

**octlet** Eight bytes (64 bits) of data. Not to be confused with "octet," which has been commonly used to describe eight bits of data. In this document, the term *byte*, rather than octet, is used to describe eight bits of data.

**odd parity** The mode of parity checking in which each combination of data bits plus a parity bit together contain an odd number of '1' bits.

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

**optional** A feature not required for UltraSPARC Architecture 2007 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 a reset) is required to complete the parking process. See also **disable** and **unpark**.

**parked** Said of a virtual processor that is suspended from operation. When parked, a virtual processor does not issue instructions for execution but still maintains cache coherency. See also **disabled, enabled,** and **running**.

**PC** Program counter.

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

**physical core** The term *physical processor core*, or just *physical core*, is similar to the term *pipeline* but represents a broader collection of hardware that are required for performing the execution of instructions from one or more software threads. For a detailed definition of this term, see page 595. See also **pipeline**, **processor**, **strand**, **thread**, *and* **virtual processor**.

**physical processor** *Synonym for* **processor**; used when an explicit contrast needs to be drawn between **processor** and virtual processor. See also **processor** and **virtual processor**.

**PIL** Processor Interrupt Level register.

**pipeline** Refers to an execution pipeline, the basic collection of hardware needed to execute instructions. For a detailed definition of this term, see page 595. See also **physical core**, **processor**, **strand**, **thread**, and **virtual processor**.

**PIPT** Physically indexed, physically tagged (cache).

**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 also **side effect**.

**privileged** An adjective that describes:
(1) the state of the virtual processor when PSTATE.priv = 1 and HPSTATE.hpriv = 0,that is, when the virtual processor is in privileged mode;
(2) processor state that is only accessible to software while the virtual processor is in hyperprivileged or 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 hyperprivileged or privileged mode.

**privileged mode** The mode in which a processor is operating when PSTATE.priv = 1 and HPSTATE.hpriv = 0. See also **nonprivileged** and **hyperprivileged**.

**processor** The unit on which a shared interface is provided to control the configuration and execution of a collection of strands; a physical module that plugs into a system. *Synonym for* **processor module**. For a detailed definition of this term, see page 595. See also **pipeline**, **physical core**, **strand**, **thread**, and **virtual processor**.

**processor core** Synonym for **physical core**.

**processor module** Synonym for **processor**.

**program counter** A register that contains the address of the instruction currently being executed.

**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.

**RA** Real address.

**RAS** Reliability, Availability, and Serviceability

**RAW** Read After Write (hazard)

**rd** Rounding direction.

**real address** An address produced by a virtual processor that refers to a particular software-visible memory location, as viewed from privileged mode. Virtual addresses are usually translated by a combination of hardware and software to real

addresses, which can be used to access real memory. Real addresses, in turn, are usually translated to physical addresses, which can be used to access physical memory. See also **physical address** and **virtual address**.

**recoverable** A term applied to an error when enough information exists elsewhere in the system for software to recover from an uncorrectable error. Examples include uncorrectable errors on clean L2 lines, which are recovered by software invalidating the line and initiating a refetch from memory. See also **unrecoverable**.

**RED_state** Reset, Error, and Debug state. The virtual processor state when HPSTATE.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.

*A reserved instruction field* must read as 0, unless the implementation supports extended instructions within the field. The behavior of an UltraSPARC Architecture 2007 virtual processor when it encounters a nonzero value in a reserved instruction field is as defined in *Reserved Opcodes and Instruction Fields* on page 130.

*A reserved bit combination within an instruction field* is defined in Chapter 7, *Instructions*. In all cases, an UltraSPARC Architecture 2007 processor must decode and trap on such reserved bit combinations.

*A reserved field within a register* reads as 0 in current implementations and, when written by software, should always be written with values of that field previously read from that register or with the value zero (as described in *Reserved Register Fields* on page 48).

Throughout this specification, figures and tables illustrating registers and instruction encodings indicate reserved fields and reserved bit combinations with a wide ("em") dash (—).

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

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

**retired** An instruction is said to be "retired" when one of the following two events has occurred:
(1) A precise trap has been taken, with TPC containing the instruction's address (the instruction has not changed architectural state in this case).
(2) The instruction's execution has progressed to a point at which architectural state affected by the instruction has been updated such that all three of the following are true:

- The PC has advanced beyond the instruction.

- Except for deferred trap handlers, no consumer in the same instruction stream can see the old values and all consumers in the same instruction stream will see the new values.
- Stores are visible to all loads in the same instruction stream, including stores to noncacheable locations.

**RMO** Abbreviation for Relaxed Memory Order (a memory model).

**RTO** Read to Own (a type of transaction, used to request ownership of a cache line).

**RTS** Read to Share (a type of transaction, used to request read-only access to a cache line).

**running** A state of a virtual processor in which it is in operation (maintaining cache coherency and issuing instructions for execution) and not `Parked`.

**service processor** A device external to the processor that can examine and alter internal processor state. A service processor may be used to control/coordinate a multiprocessor system and aid in error recovery.

**shall** Synonym for **must**.

**should** A keyword indicating flexibility of choice with a strongly preferred implementation. Synonym for **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. Each cache controller monitors (snoops) the bus to determine whether it needs to copy back or invalidate its copy of each 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** The hardware state that must be maintained in order to execute a software thread. For a detailed definition of this term, see page 594. See also **pipeline**, **physical core**, **processor**, **thread**, and **virtual processor**.

**subnormal 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.

**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** Synonym for **park**.

**suspended** Synonym for **parked**.

**synchronization** An operation that causes the processor to wait until the effects of all previous instructions are completely visible before any subsequent instructions are executed.

**system** A set of virtual processors that share a common physical address space.

**taken** A control-transfer instruction (CTI) is *taken* when the CTI writes the target address value into NPC.

A trap is *taken* when the control flow changes in response to an exception, reset, Tcc instruction, or interrupt. An exception must be detected and recognized before it can cause a trap to be taken.

**TBA** Trap base address.

**thread** A software entity that can be executed on hardware. For a detailed definition of this term, see page 594. See also **pipeline**, **physical core**, **processor**, **strand**, and **virtual processor**.

**TLB** Abbreviation for **Translation Lookaside Buffer.**

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

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

**TNPC** Trap-saved next program counter.

**TPC** Trap-saved program counter.

**Translation Lookaside Buffer** A cache within an MMU that contains recently-used Translation Table Entries (TTEs). TLBs speed up translations by often eliminating the need to reread 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 more-privileged software through a table, the address of which is specified by the privileged Trap Base Address (TBA) register or the Hyperprivileged Trap Base Address (HTBA) 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 virtual-to-real address mappings.

**TSO** Total Store Order (a memory model).

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

**UA-2007** UltraSPARC Architecture 2007

**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 and may or may not cause a trap. An undefined feature may vary among implementations, and may also vary over time on a given implementation.

Notwithstanding any of the above, undefined aspects of the architecture shall not cause security holes (such as changing the privilege state or allowing circumvention of normal restrictions imposed by the privilege state), put a virtual processor into a more-privileged mode, or put the 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.

**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. See also **disable** and **park**.

**unparked** Synonym for **running**.

**unpredictable** Synonym for **undefined**.

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

**uncorrectable** A term applied to an error when not enough information accompanies the incorrect signal or datum to allow correction of the error, and it is not known by the error detector whether enough such information exists elsewhere in the

system. Examples include uncorrectable errors on L2s. Uncorrectable errors can be further divided into two types: re**coverable** *and* **unrecoverable**. See also **correctable**.

**unrecoverable** A term applied to an error when not enough information exists elsewhere in the system for software to recover from an uncorrectable error. Examples include uncorrectable errors on dirty L2 lines. See also **recoverable**.

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

**user application program** Synonym for **application program**.

**VA** Abbreviation for **virtual address**.

**virtual address** An address produced by a virtual processor that refers to a particular software-visible memory location. Virtual addresses usually are translated by a combination of hardware and software to physical addresses, which can be used to access physical memory. See also **physical address** and **real address.**

**virtual core, virtual processor core** Synonyms for **virtual processor**.

**virtual processor** The term *virtual processor*, or *virtual processor core*, is used to identify each strand in a processor. At any given time, an operating system can have a different thread scheduled on each virtual processor. For a detailed definition of this term, see page 595. See also **pipeline**, **physical core**, **processor**, **strand**, and **thread**.

**VIS** Abbreviation for VIS™ Instruction Set.

**VP** Abbreviation for **virtual processor**.

**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.

**XIR** Externally initiated reset.

CHAPTER 3

# Architecture Overview

The UltraSPARC Architecture supports 32-bit and 64-bit integer and 32-bit, 64-bit, and 128-bit floating-point as its principal data types. The 32-bit 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.

The *UltraSPARC Architecture 2007* specification describes a processor architecture to which Sun Microsystem's SPARC processor implementations (beginning with UltraSPARC T1) comply. Future implementations are expected to comply with either this document or a later revision of this document.

The UltraSPARC Architecture 2007 is a descendant of the SPARC V9 architecture and complies fully with the "Level 1" (nonprivileged) SPARC V9 specification.

Nonprivileged (application) software that is intended to be portable across all SPARC V9 processors should be written to adhere to *The SPARC Architecture Manual-Version 9*.

Material in this document specific to UltraSPARC Architecture 2007 processors may not apply to SPARC V9 processors produced by other vendors.

In this specification, the word *architecture* refers to the processor features that are visible to an assembly language programmer or to a compiler code generator. It does not include details of the implementation that are not visible or easily observable by software, nor those that only affect timing (performance).

# 3.1 The UltraSPARC Architecture 2007

This section briefly describes features, attributes, and components of the UltraSPARC Architecture 2007 and, further, describes correct implementation of the architecture specification and SPARC V9-compliance levels.

## 3.1.1 Features

The UltraSPARC Architecture 2007, like its ancestor 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), and 16 quad-precision (128-bit) overlayed registers.
- **Fast trap handlers** — Traps are vectored through a table.
- **Multiprocessor synchronization instructions** — Multiple variations of atomic load-store memory operations are supported.
- **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.

In addition, UltraSPARC Architecture 2007 includes the following features that were not present in the SPARC V9 specification:

- **Hyperprivileged mode**, which simplifies porting of operating systems, supports far greater portability of operating system (privileged) software, supports the ability to run multiple simultaneous guest operating systems, and provides more robust handling of error conditions.
- **Multiple levels of global registers** — Instead of the two 8-register sets of global registers specified in the SPARC V9 architecture, UltraSPARC Architecture 2007 provides multiple sets; typically, one set is used at each trap level.
- **Extended instruction set** — UltraSPARC Architecture 2007 provides many instruction set extensions, including the VIS instruction set for "vector" (SIMD) data operations.
- **More detailed, specific instruction descriptions** — UltraSPARC Architecture 2007 provides many more details regarding what exceptions can be generated by each instruction and the specific conditions under which those exceptions can occur. Also, detailed lists of valid ASIs are provided for each load/store instruction from/to alternate space.
- **Detailed MMU architecture** — Although some details of the UltraSPARC MMU architecture are necessarily implementation-specifc, UltraSPARC Architecture 2007 provides a blueprint for the UltraSPARC MMU, including software view (TTEs and TSBs) and MMU hardware control registers.
- **Chip-Level Multithreading (CMT)** — UltraSPARC Architecture 2007 provides a control architecture for highly-threaded processor implementations.

## 3.1.2 Attributes

UltraSPARC Architecture 2007 is a processor *instruction set architecture* (ISA) derived from SPARC V8 and SPARC V9, which in turn come from a reduced instruction set computer (RISC) lineage. As an architecture, UltraSPARC Architecture 2007 allows 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.

### 3.1.2.1 Design Goals

The UltraSPARC Architecture 2007 architecture is designed to be a target for optimizing compilers and high-performance hardware implementations. This specification documents the UltraSPARC Architecture 2007 and provides a design spec against which an implementation can be verified, using appropriate verification software.

#### 3.1.2.2 Register Windows

The UltraSPARC Architecture 2007 architecture is derived from the SPARC architecture, which was formulated at Sun Microsystems in 1984 through1987. The SPARC architecture is, in turn, 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 privileged software, not user programs, manages the register windows. Privileged software can save a minimum number of registers (approximately 24) during a context switch, thereby optimizing context-switch latency.

### 3.1.3 System Components

The UltraSPARC Architecture 2007 allows for a spectrum of subarchitectures, such as cache system, I/O, and memory management unit (MMU).

#### 3.1.3.1 Binary Compatibility

The most important mandate for the UltraSPARC Architecture is compatibility across implementations of the architecture for application (nonprivileged) software, down to the binary level. Binaries executed in nonprivileged mode should behave identically on all UltraSPARC Architecture 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 UltraSPARC Architecture 2007 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 9, *Memory*, for more information.

Additionally, UltraSPARC Architecture 2007 is binary upward-compatible from SPARC V9 for applications running in nonprivileged mode that conform to the SPARC V9 ABI and upward-compatible from SPARC V8 for applications running in nonprivileged mode that conform to the SPARC V8 ABI.

#### 3.1.3.2 UltraSPARC Architecture 2007 MMU

Although the SPARC V9 architecture allows its implementations freedom in their MMU designs, UltraSPARC Architecture 2007 defines a common MMU architecture (see Chapter 14, *Memory Management*) with some specifics left to implementations (see processor implementation documents).

#### 3.1.3.3 Privileged Software

UltraSPARC Architecture 2007 does not assume that all implementations must execute identical privileged software (operating systems) or hyperprivileged software (hypervisors). Thus, certain traits that are visible to privileged software may be tailored to the requirements of the system.

### 3.1.4 Architectural Definition

The UltraSPARC Architecture 2007 is defined by the chapters and appendixes of this specification. A correct implementation of the architecture interprets a program strictly according to the rules and algorithms specified in the chapters and appendixes.

UltraSPARC Architecture 2007 defines a set of implementations that conform to the SPARC V9 architecture, Level 1.

### 3.1.5 UltraSPARC Architecture 2007 Compliance with SPARC V9 Architecture

UltraSPARC Architecture 2007 fully complies with SPARC V9 Level 1 (nonprivileged). It partially complies with SPARC V9 Level 2 (privileged).

### 3.1.6 Implementation Compliance with UltraSPARC Architecture 2007

Compliant implementations must not add to or deviate from this standard except in aspects described as implementation dependent. Appendix B, *Implementation Dependencies* lists all UltraSPARC Architecture 2007, SPARC V9, and SPARC V8 implementation dependencies. Documents for specific UltraSPARC Architecture 2007 processor implementations describe the manner in which implementation dependencies have been resolved in those implementations.

**IMPL. DEP. #1-V8:** Whether an instruction complies with UltraSPARC Architecture 2007 by being implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.

## 3.2 Processor Architecture

An UltraSPARC Architecture 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.

An UltraSPARC Architecture virtual processor can run in *nonprivileged* mode, *privileged* mode, or *hyperprivileged* mode. In hyperprivileged mode, the processor can execute any instruction, including privileged instructions. In privileged mode, the processor can execute nonprivileged and privileged instructions. In nonprivileged mode, the processor can only execute nonprivileged instructions. In nonprivileged or privileged mode, an attempt to execute an instruction requiring greater privilege than the current mode causes a trap to hyperprivileged software.

### 3.2.1 Integer Unit (IU)

An UltraSPARC Architecture 2007 implementation's 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.

**IMPL. DEP. #2-V8:** An UltraSPARC Architecture implementation may contain from 72 to 640 general-purpose 64-bit R registers. This corresponds to a grouping of the registers into *MAXGL* + 1 sets of global R registers plus a circular stack of *N_REG_WINDOWS* sets of 16 registers each, known as register windows. The number of register windows present (*N_REG_WINDOWS*) is implementation dependent, within the range of 3 to 32 (inclusive).

### 3.2.2 Floating-Point Unit (FPU)

An UltraSPARC Architecture 2007 implementation's 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.

If no FPU is present, then it appears to software as if the FPU is permanently disabled.

If the FPU is not enabled, then an attempt to execute a floating-point instruction generates an *fp_disabled* trap and the *fp_disabled* trap handler software must either

- Enable the FPU (if present) and reexecute the trapping instruction, or
- Emulate the trapping instruction in software.

## 3.3 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
- SIMD (single instruction, multiple data) instructions

These classes are discussed in the following subsections.

### 3.3.1 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 extended-word (64-bit) accesses. There are versions of integer load instructions that perform either sign-extension or zero-extension on 8-bit, 16-bit, 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[1] memory accesses.

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

**Note** | The SWAP instruction is also specified, but it is deprecated and should not be used in newly developed software.

1. No UltraSPARC Architecture processor currently implements the LDQF instruction in hardware; it generates an exception and is emulated in hyperprivileged software.

The (nonportable) LDTXA instruction supplies an atomic 128-bit (16-byte) load that is important in certain system software applications.

### 3.3.1.1 Memory Alignment Restrictions

A memory access on an UltraSPARC Architecture 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 in instruction may trigger an exception and cause a subsequent trap. For details, see *Memory Alignment Restrictions* on page 112.

### 3.3.1.2 Addressing Conventions

The UltraSPARC Architecture 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.

The UltraSPARC Architecture also supports 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 data unit being accessed.

Addressing conventions are illustrated in FIGURE 6-2 on page 115 and FIGURE 6-3 on page 117.

### 3.3.1.3 Addressing Range

**IMPL. DEP. #405-S10:** An UltraSPARC Architecture implementation may support a full 64-bit virtual address space or a more limited range of virtual addresses. In an implementation that does not support a full 64-bit virtual address space, the supported range of virtual addresses is restricted to two equal-sized ranges at the extreme upper and lower ends of 64-bit addresses; that is, for $n$-bit virtual addresses, the valid address ranges are 0 to $2^{n-1} - 1$ and $2^{64} - 2^{n-1}$ to $2^{64} - 1$.

### 3.3.1.4 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 $00_{16}$–$2F_{16}$ is restricted to privileged and hyperprivileged software, access to alternate spaces $30_{16}$–$7F_{16}$ is restricted to hyperprivileged software, and access to alternate spaces $80_{16}$–$FF_{16}$ is unrestricted. Some of the ASIs are available for implementation-dependent uses. Privileged and hyperprivileged software can use the implementation-dependent ASIs to access special protected

registers, such as MMU control registers, cache control registers, virtual processor state registers, and other processor-dependent or system-dependent values. See *Address Space Identifiers (ASIs)* on page 117 for more information.

Alternate space addressing is also provided for the atomic memory access instructions LDSTUBA, CASA, and CASXA.

**Note** | The SWAPA instruction is also specified, but it is deprecated and should not be used in newly developed software.

### 3.3.1.5 Separate Instruction and Data Memories

The interpretation of addresses 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 caching and translation mechanism and data references use another, although the same underlying main memory is shared.

In such split-memory systems, the coherency mechanism may be split, so a write[1] into data memory is not immediately reflected in instruction memory. For this reason, programs that modify their own instruction stream (self-modifying code[2]) and that wish to be portable across all UltraSPARC Architecture (and 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.

An UltraSPARC Architecture virtual processor may or may not have coherent instruction and data caches. Even if an implementation does have coherent instruction and data caches, a FLUSH instruction is required for self-modifying code — not for cache coherency, but to flush pipeline instruction buffers that contain unmodified instructions which may have been subsequently modified.

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

The UltraSPARC Architecture 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-V9:** The semantic effect of accessing input/output (I/O) locations is implementation dependent.

**IMPL. DEP. #6-V8:** Whether the I/O registers can be accessed by nonprivileged code is implementation dependent.

**IMPL. DEP. #7-V8:** The addresses and contents of I/O registers are implementation dependent.

1. this includes use of store instructions (executed on the same or another virtual processor) that write to instruction memory, or any other means of writing into instruction memory (for example, DMA)

2. this is practiced, for example, by software such as debuggers and dynamic linkers

#### 3.3.1.7 Memory Synchronization

Two instructions are used for synchronization of memory operations: FLUSH and MEMBAR. Their operation is explained in *Flush Instruction Memory* on page 187 and *Memory Barrier* on page 277, respectively.

**Note** | STBAR is also available, but it is deprecated and should not be used in newly developed software.

### 3.3.2 Integer 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 other arithmetic and/or logical instructions to create a constant in an R register.

Shift instructions shift the contents of an R register left or right by a given number of bits ("shift count"). The shift distance is specified by a constant in the instruction or by the contents of an R register.

### 3.3.3 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. Setting the *annul bit* in a conditional delayed control-transfer instruction causes the delay instruction to be annulled (that is, to have no effect) if and only if the branch is not taken. Setting the annul bit in an *un*conditional delayed control-transfer instruction ("branch always") causes the delay instruction to be always annulled.

**Note** | The SPARC V8 architecture specified that the delay instruction was always fetched, even if annulled, and that an annulled instruction could not cause any traps. The SPARC V9 architecture 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 (± $2^{31}$ bytes).

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

## 3.3.4 State Register Access

### 3.3.4.1 Ancillary State Registers

The read and write ancillary state register instructions read and write the contents of ancillary state registers visible to nonprivileged software (Y, CCR, ASI, PC, TICK, and FPRS) and some registers visible only to privileged and hyperprivileged software (SOFTINT, TICK_CMPR, and STICK_CMPR).

**IMPL. DEP. #8-V8-Cs20:** Ancillary state registers (ASRs) in the range 0–27 that are not defined in UltraSPARC Architecture 2007 are reserved for future architectural use. ASRs in the range 28–31 are available to be used for implementation-dependent purposes.

**IMPL. DEP. #9-V8-Cs20:** The privilege level required to execute each of the implementation-dependent read/write ancillary state register instructions (for ASRs 28–31) is implementation dependent.

### 3.3.4.2 PR State Registers

The read and write privileged register instructions (RDPR and WRPR) read and write the contents of state registers visible only to privileged and hyperprivileged software (TPC, TNPC, TSTATE, TT, TICK, TBA, PSTATE, TL, PIL, CWP, CANSAVE, CANRESTORE, CLEANWIN, OTHERWIN, and WSTATE).

### 3.3.4.3 HPR State Registers

The read and write hyperprivileged register instructions (RDHPR and WRHPR) read and write the contents of state registers visible only to hyperprivileged software (HPSTATE, HTSTATE, HINTP, HVER, and HSTICK_CMPR).

### 3.3.5 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. FPops compute a result that is a function of one , two, or three source operands. The groups of instructions that are considered FPops are listed in *Floating-Point Operate (FPop) Instructions* on page 129.

### 3.3.6 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 on the contents of an integer register. These instructions can be used to reduce the number of branches in software.

### 3.3.7 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.

### 3.3.8 SIMD

UltraSPARC Architecture 2007 includes SIMD (single instruction, multiple data) instructions, also known as "vector" instructions, which allow a single instruction to perform the same operation on multiple data items, totalling 64 bits, such as eight 8-bit, four 16-bit, or two 32-bit data items. These operations are part of the “VIS” extensions.

## 3.4 Traps

A *trap* is a vectored transfer of control to privileged or hyperprivileged 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, or the Hyperprivileged Trap Base Register, HTBA). The displacement within the table is

encoded in the type number of each trap and the level of the trap. Part of the trap table is reserved for hardware traps, and part of it is reserved for software traps generated by trap (Tcc) instructions.

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 supported trap levels. A trap causes hyperprivileged state to be saved in the HTSTATE trap stack. A trap also sets bits in the PSTATE (and, in some cases, HPSTATE) register and typically increments the GL register. 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 12, *Traps*, for a complete description of traps.

## 3.5 Chip-Level Multithreading (CMT)

An UltraSPARC Architecture 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, referred to as chip-level multithreaded (CMT) processors. CMT-specific hyperprivileged registers are used for identification and configuration of CMT processors.

The CMT programming model describes a common interface between hardware (CMT registers) and software

The common CMT registers and the CMT programming model are described in Chapter 15, *Chip-Level Multithreading (CMT)*.

CHAPTER 4

# Data Formats

The UltraSPARC 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: Uint8 SIMD (32 bits), Int16 SIMD (64 bits), and Int32 SIMD (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/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 36.
- **Floating-Point Data Formats** on page 40.
- **SIMD Data Formats** on page 43.


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


- **Signed Integer Doubleword (64 bits)** on page 37.
- **Unsigned Integer Doubleword (64 bits)** on page 39.
- **Floating Point, Double Precision (64 bits)** on page 41.
- **Floating Point, Quad Precision (128 bits)** on page 42.

# 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 doubleword/extended-word | 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 doubleword/extended-word | 64 | 0 to $2^{64} - 1$ |
| Integer tagged word | 32 | 0 to $2^{30} - 1$ |

TABLE 4-2 describes the memory and register alignment for multiword integer data. All registers in the integer register file are 64 bits wide, but can be used to contain smaller (narrower) data sizes. Note that there is no difference between integer extended-words and doublewords in memory; the only difference is how they are represented in registers.

**TABLE 4-2** Integer Doubleword/Extended-word Alignment

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

1. 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 integer extended-word

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

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

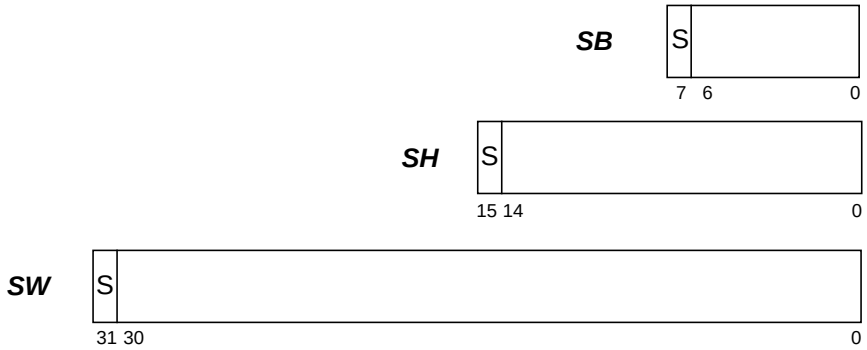



**FIGURE 4-1** Signed Integer Byte, Halfword, and Word Data Formats

### 4.1.1.2 Signed Integer Doubleword (64 bits)

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

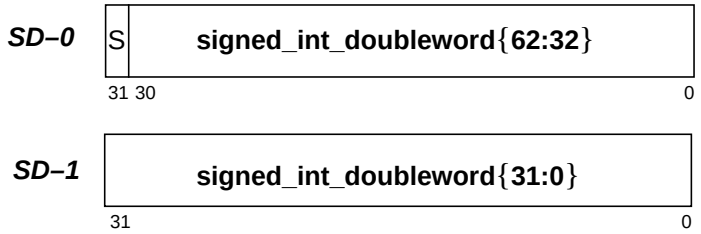



**FIGURE 4-2** Signed Integer Double Data Format

#### 4.1.1.3 Signed Integer Extended-Word (64 bits)

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

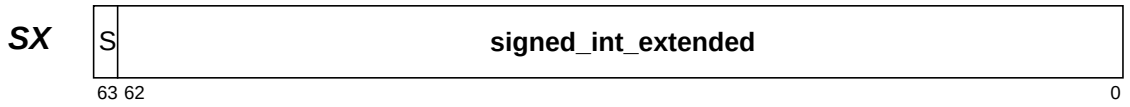



**FIGURE 4-3** Signed Integer Extended-Word Data Format

### 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 integer extended-word

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

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

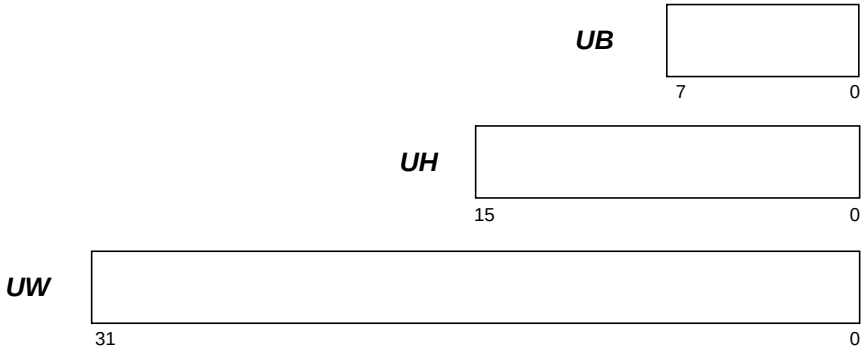



**FIGURE 4-4** Unsigned Integer Byte, Halfword, and Word Data Formats

#### 4.1.2.2 Unsigned Integer Doubleword (64 bits)

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

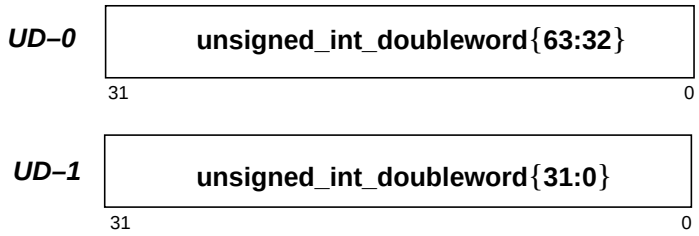



**FIGURE 4-5** Unsigned Integer Double Data Format

#### 4.1.2.3 Unsigned Extended Integer (64 bits)

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

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

### 4.1.3 Tagged Word (32 bits)

FIGURE 4-7 illustrates the tagged word data format.

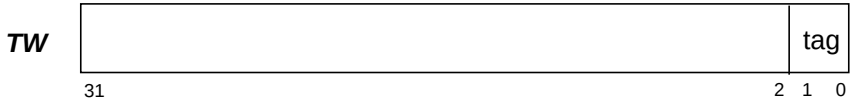



**FIGURE 4-7** Tagged Word Data Format

# 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 (32 bits)

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

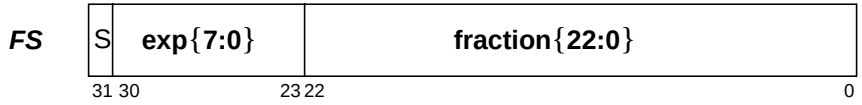



**FIGURE 4-8** Floating-Point Single-Precision Data Format

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

| s = sign (1 bit)<br>e = biased exponent (8 bits)<br>f = fraction (23 bits)<br>*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, f = 0) | $(-1)^s \times 0$ |
| Signalling NaN | s = u; e = 255 (max); f = .0uu--uu<br>(At least one bit of the fraction must be nonzero) |
| Quiet NaN | s = u; e = 255 (max); f = .1uu--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 (64 bits)

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.

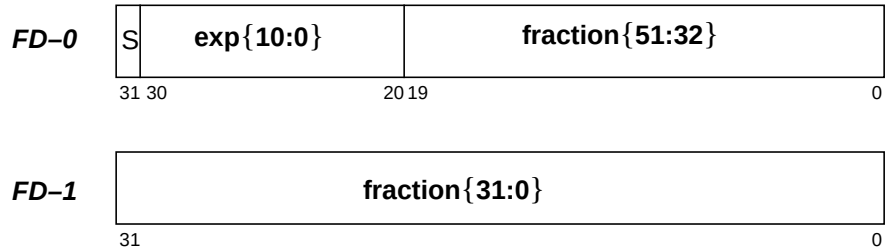



**FIGURE 4-9** Floating-Point Double-Precision Data Format

**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, f = 0) | $(-1)^s \times 0$ |
| Signalling NaN | s = u; e = 2047 (max); f = .0uu--uu<br>(At least one bit of the fraction must be nonzero) |
| Quiet NaN | s = u; e = 2047 (max); f = .1uu--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 (128 bits)

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.

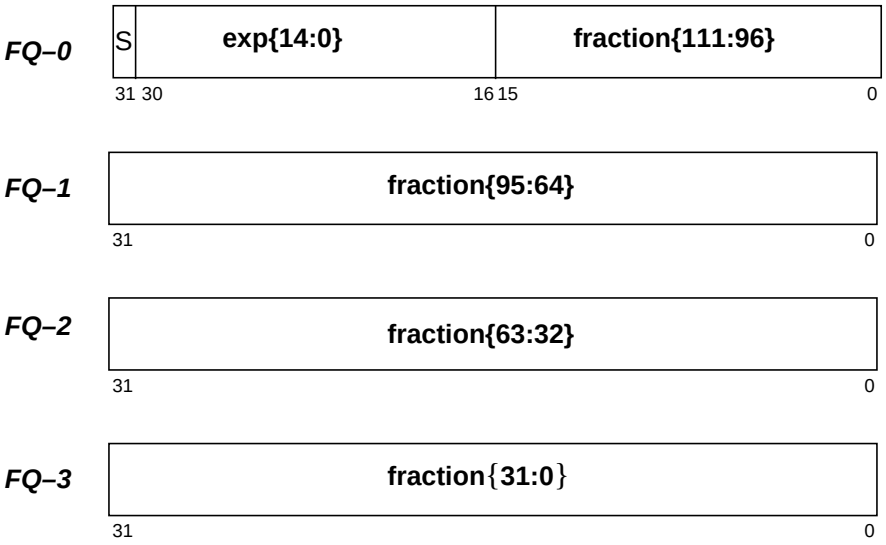



**FIGURE 4-10** Floating-Point Quad-Precision Data Format

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

| s = sign (1 bit)<br>e = biased exponent (15 bits)<br>f = fraction (112 bits)<br>*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, f = 0) | $(-1)^s \times 0$ |
| Signalling NaN | s = u; e = 32767 (max); f = .0uu--uu<br>(At least one bit of the fraction must be nonzero) |
| Quiet NaN | s = u; e = 32767 (max); f = .1uu--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).

◊ Note that this 32-bit floating-point register is only directly addressable in the lower half of the register file (that is, if its register number is ≤ 31).

## 4.3 SIMD 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.

Conversion between the different SIMD data formats can be achieved through SIMD multiplication or by the use of the SIMD data formatting instructions.

**Programming Note**

The SIMD data formats can be used in graphics calculations to represent intensity values for an image (e.g., α, B, G, R).

Intensity values are typically grouped in one of two ways, when using SIMD data formats:

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

## 4.3.1 Uint8 SIMD Data Format

The Uint8 SIMD data format consists of four unsigned 8-bit integers contained in a 32-bit word (see FIGURE 4-11).

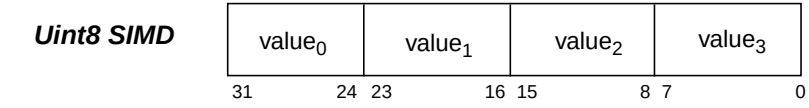



**FIGURE 4-11** Uint8 SIMD Data Format

## 4.3.2 Int16 SIMD Data Formats

The Int16 SIMD data format consists of four signed 16-bit integers contained in a 64-bit word (see FIGURE 4-12).

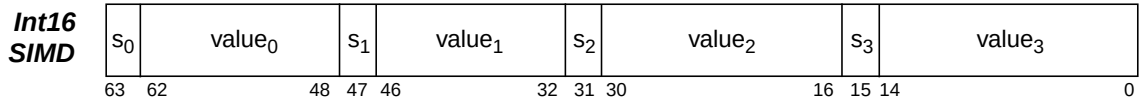



**FIGURE 4-12** Int16 SIMD Data Format

## 4.3.3 Int32 SIMD Data Format

The Int32 SIMD data format consists of two signed 32-bit integers contained in a 64-bit word (see FIGURE 4-13).

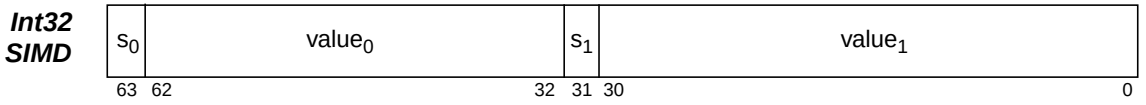



**FIGURE 4-13** Int32 SIMD Data Format

**Programming Note** The integer SIMD data formats can be used to hold fixed-point data. The position of the binary point in a SIMD datum is implied by the programmer and does not influence the computations performed by instructions that operate on that SIMD data format.

CHAPTER 5

# Registers

The following registers are described in this chapter:

- **General-Purpose R Registers on page 48.**
- **Floating-Point Registers** on page 54.
- **Floating-Point State Register (FSR)** on page 60.
- **Ancillary State Registers** on page 69. The following registers are included in this category:
  - **32-bit Multiply/Divide Register (Y) (ASR 0)** on page 70.
  - **Integer Condition Codes Register (CCR) (ASR 2)** on page 71.
  - **Address Space Identifier (ASI) Register (ASR 3)** on page 73.
  - **Tick (TICK) Register (ASR 4)** on page 73.
  - **Program Counters (PC, NPC) (ASR 5)** on page 75.
  - **Floating-Point Registers State (FPRS) Register (ASR 6)** on page 75.
  - **General Status Register (GSR) (ASR 19)** on page 76.
  - **SOFTINT$^{P}$ Register (ASRs 20, 21, 22)** on page 77.
  - **SOFTINT_SET$^{P}$ Pseudo-Register (ASR 20)** on page 78.
  - **SOFTINT_CLR$^{P}$ Pseudo-Register (ASR 21)** on page 79.
  - **Tick Compare (TICK_CMPR$^{P}$) Register (ASR 23)** on page 79.
  - **System Tick (STICK) Register (ASR 24)** on page 80.
  - **System Tick Compare (STICK_CMPR$^{P}$) Register (ASR 25)** on page 81.
- **Register-Window PR State Registers** on page 82. The following registers are included in this subcategory:
  - **Current Window Pointer (CWP$^{P}$) Register (PR 9)** on page 83.
  - **Savable Windows (CANSAVE$^{P}$) Register (PR 10)** on page 83.
  - **Restorable Windows (CANRESTORE$^{P}$) Register (PR 11)** on page 84.
  - **Clean Windows (CLEANWIN$^{P}$) Register (PR 12)** on page 84.
  - **Other Windows (OTHERWIN$^{P}$) Register (PR 13)** on page 84.
  - **Window State (WSTATE$^{P}$) Register (PR 14)** on page 85.
- **Non-Register-Window PR State Registers** on page 87. The following registers are included in this subcategory:
  - **Trap Program Counter (TPC$^{P}$) Register (PR 0)** on page 87.
  - **Trap Next PC (TNPC$^{P}$) Register (PR 1)** on page 88.
  - **Trap State (TSTATE$^{P}$) Register (PR 2)** on page 89.
  - **Trap Type (TT$^{P}$) Register (PR 3)** on page 90.

- **Trap Base Address (TBA$^{P}$) Register (PR 5)** on page 91.
- **Processor State (PSTATE$^{P}$) Register (PR 6)** on page 91.
- **Trap Level Register (TL$^{P}$) (PR 7)** on page 97.
- **Processor Interrupt Level (PIL$^{P}$) Register (PR 8)** on page 99.
- **Global Level Register (GL$^{P}$) (PR 16)** on page 99.

- **HPR State Registers** on page 101. The following registers are included in this category.
  - **Hyperprivileged State (HPSTATE$^{H}$) Register (HPR 0)** on page 102.
  - **Hyperprivileged Trap State (HTSTATE$^{H}$) Register (HPR 1)** on page 103.
  - **Hyperprivileged Interrupt Pending (HINTP$^{H}$) Register (HPR 3)** on page 104.
  - **Hyperprivileged Implementation Version (HVER$^{H}$) Register (HPR 6)** on page 106.
  - **Hyperprivileged System Tick Compare (HSTICK_CMPR$^{H}$) Register (HPR 31)** on page 107.

There are additional registers that may be accessed through ASIs; those registers are described in Chapter 10, *Address Space Identifiers (ASIs)*.

# 5.1 Reserved Register Fields

For convenience, some registers in this chapter are illustrated as fewer than 64 bits wide. Any bits not shown (or explicitly marked as reserved) are reserved for future extensions to the architecture.

Such a reserved field within a register reads as zero in current implementations and, when written by software, should only be written with the value of that field previously read from that register or with the value zero.

**Programming Note** | Software intended to run on future versions of the UltraSPARC Architecture should not assume that reserved register fields will read as 0 or any other particular value.

# 5.2 General-Purpose R Registers

An UltraSPARC Architecture virtual processor contains an array of general-purpose 64-bit R registers. The array is partitioned into *MAXGL* + 1 sets of eight *global* registers, plus *N_REG_WINDOWS* groups of 16 registers each. The value of *N_REG_WINDOWS* in an UltraSPARC Architecture implementation falls within the range 3 to 32 (inclusive).

One set of 8 global registers is always visible. At any given time, a group of 24 registers, known as a *register window*, is also visible. A register window comprises the 16 registers from the current 16-register group (referred to as 8 *in* registers and 8 *local* registers), plus half of the registers from the next 16-register group (referred to as 8 *out* registers). See FIGURE 5-1.

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 full set of R registers are visible is described in the following sections. The visible 32 R registers are named R[0] through R[31], illustrated in FIGURE 5-1.

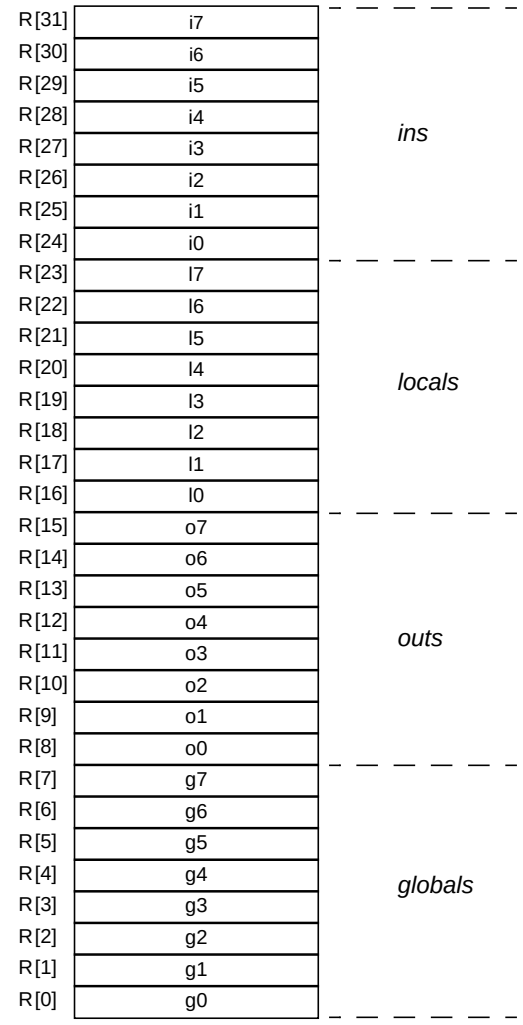



**FIGURE 5-1** General-Purpose Registers (as Visible at Any Given Time)

## 5.2.1 Global R Registers (A1)

Registers R[0]–R[7] refer to a set of eight registers called the *global* registers (labeled g0 through g7). At any time, one of *MAXGL* +1 sets of eight registers is enabled and can be accessed as the current set of global registers. The currently enabled set of global registers is selected by the GL register. See *Global Level Register (GL$^P$) (PR 16)* on page 99.

Global register zero (G0) always reads as zero; writes to it have no software-visible effect.

## 5.2.2 Windowed R Registers (A1)

A set of 24 R registers that is visible as R[8]–R[31] at any given time 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 5-2.

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.

**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] |

**V9 Compatibility Note** In the SPARC V9 architecture, the number of 16-register windowed register sets, *N_REG_WINDOWS*, ranges from 3 to 32 (impl. dep. #2-V8). The maximum global register set index in the UltraSPARC Architecture, *MAXGL*, ranges from 2 to 15. The number of implemented global register sets is *MAXGL* + 1. The total number of R registers in a given UltraSPARC Architecture implementation is:

$$(N_REG_WINDOWS \times 16) + ((MAXGL + 1) \times 8)$$

Therefore, an UltraSPARC Architecture processor may contain from 72 to 640 R registers.

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

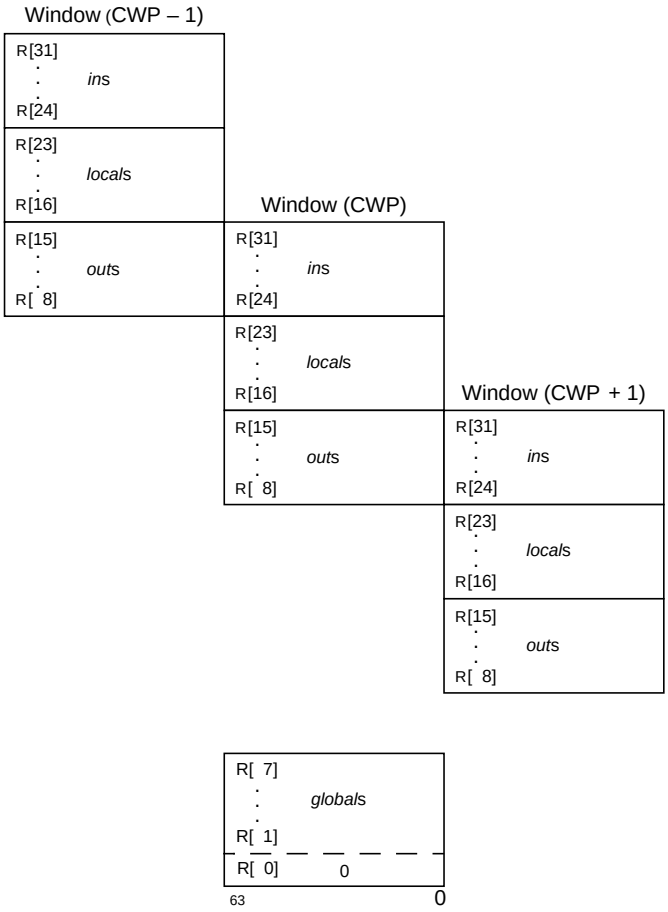



**FIGURE 5-2** Three Overlapping Windows and Eight Global Registers

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

Register address *o*, where $8 \le o \le 15$, refers to exactly the same *out* register before the register window is advanced by a SAVE instruction (CWP is incremented by 1 (**modulo** *N_REG_WINDOWS*)) as does register address $o+16$ after the register window is advanced. Likewise, register address *i*, where $24 \le i \le 31$, refers to exactly the same

*in* register before the register window is restored by a RESTORE instruction (CWP is decremented by 1 (**modulo** *N_REG_WINDOWS*)) as does register address *i*−16 after the window is restored. See FIGURE 5-2 on page 51 and FIGURE 5-3 on page 53.

To application software, the virtual processor appears to provide an infinitely-deep stack of register windows.

**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 the section "Leaf-Procedure Optimization" in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes*

Since CWP arithmetic is performed modulo *N_REG_WINDOWS*, the highest-numbered implemented window overlaps with window 0. The *out*s of window *N_REG_WINDOWS* − 1 are the *in*s of window 0. Implemented windows are numbered contiguously from 0 through *N_REG_WINDOWS* −1.

Because the windows overlap, the number of windows available to software is 1 less than the number of implemented windows; that is, *N_REG_WINDOWS* – 1. When the register file is full, the *outs* of the newest window are the *ins* of the oldest window, which still contains valid data.

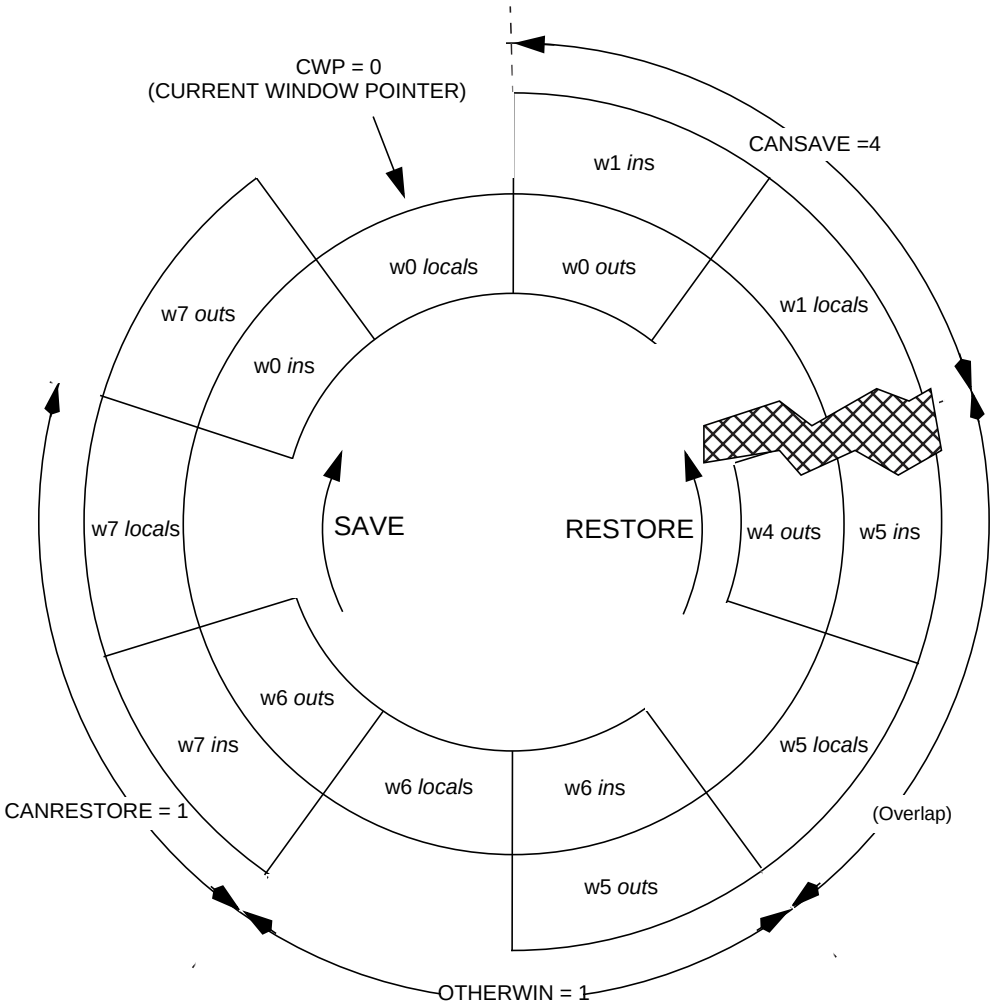
Window overflow is detected by the CANSAVE register, and window underflow is detected by the CANRESTORE register, both of which are controlled by privileged software. A window overflow (underflow) condition causes a window spill (fill) trap.

When a new register window is made visible through use of a SAVE instruction, the *local* and *out* registers are guaranteed to contain either zeroes or valid data from the current context. If software executes a RESTORE and later executes a SAVE, then the contents of the resulting window's *local* and *out* registers are not guaranteed to be preserved between the RESTORE and the SAVE[1]. Those registers may even have been written with "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$^P$) Register (PR 12)* on page 84, *Savable Windows (CANSAVE$^P$) Register (PR 10)* on page 83, and *Restorable Windows (CANRESTORE$^P$) Register (PR 11)* on page 84.

**Implementation Note** An UltraSPARC Architecture virtual processor supports the guarantee in the preceding paragraph of "either zeroes or valid data from the current context"; it may do so either in hardware or in a combination of hardware and system software.

1. For example, any of those 16 registers might be altered due to the occurrence of a trap between the RESTORE and the SAVE, or might be altered during the RESTORE operation due to the way that register windows are implemented. After a RESTORE instruction executes, software must assume that the values of the affected 16 registers from before the RESTORE are unrecoverable.

*Register Window Management Instructions* on page 127 describes how the windowed integer registers are managed.




CANSAVE + CANRESTORE + OTHERWIN = *N_REG_WINDOWS* – 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 *N_REG_WINDOWS* = 8

In FIGURE 5-3, *N_REG_WINDOWS* = 8. The eight *global* registers 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.

### 5.2.3 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.** LDTW, LDTWA, STTW, and STTWA instructions access a pair of words ("twin 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.

When the R[0]–R[1] register pair is used as a destination in LDTW or LDTWA, only R[1] is modified. When the R[0]–R[1] register pair is used as a source in STTW or STTWA, 0 is read from R[0], so 0 is written to the 32-bit word at the lowest 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 anLDTW, LDTWA, STTW, or STTWA instruction that refers to a misaligned (odd) destination register number causes an *illegal_instruction* trap.

## 5.3 Floating-Point Registers (A1)

The floating-point register set consists of sixty-four 32-bit registers, which may be accessed as follows:

- Sixteen 128-bit quad-precision registers, referenced as $F_Q[0]$, $F_Q[4]$, …, $F_Q[60]$
- Thirty-two 64-bit double-precision registers, referenced as $F_D[0]$, $F_D[2]$, …, $F_D[62]$
- Thirty-two 32-bit single-precision registers, referenced as $F_S[0]$, $F_S[1]$, …, $F_S[31]$ (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. 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** Floating-Point Registers, with Aliasing *(1 of 3)*

| Single Precision (32-bit) | | Double Precision (64-bit) | | | Quad Precision (128-bit) | | |
|---|---|---|---|---|---|---|---|
| Register | Assembly Language | Bits | Register | Assembly Language | Bits | Register | Assembly Language |
| $F_S[0]$ | %f0 | 63:32 | $F_D[0]$ | %d0 | 127:64 | $F_Q[0]$ | %q0 |
| $F_S[1]$ | %f1 | 31:0 | | | 63:0 | | |
| $F_S[2]$ | %f2 | 63:32 | $F_D[2]$ | %d2 | | | |
| $F_S[3]$ | %f3 | 31:0 | | | | | |
| $F_S[4]$ | %f4 | 63:32 | $F_D[4]$ | %d4 | 127:64 | $F_Q[4]$ | %q4 |
| $F_S[5]$ | %f5 | 31:0 | | | 63:0 | | |
| $F_S[6]$ | %f6 | 63:32 | $F_D[6]$ | %d6 | | | |
| $F_S[7]$ | %f7 | 31:0 | | | | | |
| $F_S[8]$]] | %f8 | 63:32 | $F_D[8]$ | %d8 | 127:64 | $F_Q[8]$ | %q8 |
| $F_S[9]$ | %f9 | 31:0 | | | 63:0 | | |
| $F_S[10]$ | %f10 | 63:32 | $F_D[10]$ | %d10 | | | |
| $F_S[11]$ | %f11 | 31:0 | | | | | |
| $F_S[12]$ | %f12 | 63:32 | $F_D[12]$ | %d12 | 127:64 | $F_Q[12]$ | %q12 |
| $F_S[13]$ | %f13 | 31:0 | | | 63:0 | | |
| $F_S[14]$ | %f14 | 63:32 | $F_D[14]$ | %d14 | | | |
| $F_S[15]$ | %f15 | 31:0 | | | | | |
| $F_S[16]$ | %f16 | 63:32 | $F_D[16]$ | %d16 | 127:64 | $F_Q[16]$ | %q16 |
| $F_S[17]$ | %f17 | 31:0 | | | 63:0 | | |
| $F_S[18]$ | %f18 | 63:32 | $F_D[18]$ | %d18 | | | |
| $F_S[19]$ | %f19 | 31:0 | | | | | |
| $F_S[20]$ | %f20 | 63:32 | $F_D[20]$ | %d20 | 127:64 | $F_Q[20]$ | %q20 |
| $F_S[21]$ | %f21 | 31:0 | | | 63:0 | | |
| $F_S[22]$ | %f22 | 63:32 | $F_D[22]$ | %d22 | | | |
| $F_S[23]$ | %f23 | 31:0 | | | | | |

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

| Single Precision (32-bit) | | Double Precision (64-bit) | | | Quad Precision (128-bit) | | |
|---|---|---|---|---|---|---|---|
| Register | Assembly Language | Bits | Register | Assembly Language | Bits | Register | Assembly Language |
| $F_S[24]$ | %f24 | 63:32 | $F_D[24]$ | %d24 | 127:64 | $F_Q[24]$ | %q24 |
| $F_S[25]$ | %f25 | 31:0 | | | 63:0 | | |
| $F_S[26]$ | %f26 | 63:32 | $F_D[26]$ | %d26 | | | |
| $F_S[27]$ | %f27 | 31:0 | | | | | |
| $F_S[28]$ | %f28 | 63:32 | $F_D[28]$ | %d28 | 127:64 | $F_Q[28]$ | %q28 |
| $F_S[29]$ | %f29 | 31:0 | | | 63:0 | | |
| $F_S[30]$ | %f30 | 63:32 | $F_D[30]$ | %d30 | | | |
| $F_S[31]$ | %f31 | 31:0 | | | | | |
| | | 63:32 | $F_D[32]$ | %d32 | 127:64 | $F_Q[32]$ | %q32 |
| | | 31:0 | | | 63:0 | | |
| | | 63:32 | $F_D[34]$ | %d34 | | | |
| | | 31:0 | | | | | |
| | | 63:32 | $F_D[36]$ | %d36 | 127:64 | $F_Q[36]$ | %q36 |
| | | 31:0 | | | 63:0 | | |
| | | 63:32 | $F_D[38]$ | %d38 | | | |
| | | 31:0 | | | | | |
| | | 63:32 | $F_D[40]$ | %d40 | 127:64 | $F_Q[40]$ | %q40 |
| | | 31:0 | | | 63:0 | | |
| | | 63:32 | $F_D[42]$ | %d42 | | | |
| | | 31:0 | | | | | |
| | | 63:32 | $F_D[44]$ | %d44 | 127:64 | $F_Q[44]$ | %q44 |
| | | 31:0 | | | 63:0 | | |
| | | 63:32 | $F_D[46]$ | %d46 | | | |
| | | 31:0 | | | | | |
| | | 63:32 | $F_D[48]$ | %d48 | 127:64 | $F_Q[48]$ | %q48 |
| | | 31:0 | | | 63:0 | | |
| | | 63:32 | $F_D[50]$ | %d50 | | | |
| | | 31:0 | | | | | |

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

| Single Precision (32-bit) | | Double Precision (64-bit) | | | Quad Precision (128-bit) | | |
|---|---|---|---|---|---|---|---|
| Register | Assembly Language | Bits | Register | Assembly Language | Bits | Register | Assembly Language |
| | | 63:32<br>31:0 | $F_D$[52] | %d52 | 127:64<br>63:0 | $F_Q$[52] | %q52 |
| | | 63:32<br>31:0 | $F_D$[54] | %d54 | | | |
| | | 63:32<br>31:0 | $F_D$[56] | %d56 | 127:64<br>63:0 | $F_Q$[56] | %q56 |
| | | 63:32<br>31:0 | $F_D$[58] | %d58 | | | |
| | | 63:32<br>31:0 | $F_D$[60] | %d60 | 127:64<br>63:0 | $F_Q$[60] | %q60 |
| | | 63:32<br>31:0 | $F_D$[62] | %d62 | | | |

## 5.3.1 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 5-3.

**TABLE 5-3** Floating-Point Register Number Encoding

| Register Operand Type | Full 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} |

**SPARC V8 Compatibility Note**

In the SPARC V8 architecture, 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 an UltraSPARC Architecture virtual processor, using the encoding in TABLE 5-3.

## 5.3.2 Double and Quad Floating-Point Operands

A single 32-bit 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** The upper 16 double-precision (upper 8 quad-precision) floating-point registers cannot be directly loaded by 32-bit load instructions. Therefore, double- or quad-precision data that is only word-aligned in memory cannot be directly loaded into the upper registers with LDF[A] instructions. The following guidelines are recommended:

1. Whenever possible, align floating-point data in memory on proper address boundaries. If access to a datum is required to be atomic, the datum *must* be properly aligned.
2. If a double- or quad-precision datum is not properly aligned in memory or is still aligned on a 4-byte boundary, and access to the datum in memory is not required to be atomic, then software should attempt to allocate a register for it in the lower half of the floating-point register file so that the datum can be loaded with multiple LDF[A] instructions.
3. If the only available registers for such a datum are located in the upper half of the floating-point register file and access to the datum in memory is not required to be atomic, the word-aligned datum can be loaded into them by one of two methods:
   - Load the datum into an upper register by using multiple LDF[A] instructions to first load it into a double- or quad-precision register in the lower half of the floating-point register file, then copy that register to the desired destination register in the upper half
   - Use an LDDF[A] or LDQF[A] instruction to perform the load directly into the upper floating-point register, understanding that use of these instructions on poorly aligned data can cause a trap (*LDDF_mem_not_aligned*) on some implementations, possibly slowing down program execution significantly.

**Programming Note** If an UltraSPARC Architecture 2007 implementation does not implement a particular quad floating-point arithmetic operation in hardware and an invalid quad register operand is specified, the *illegal_instruction* trap occurs because it has higher priority.

**Implementation Note** UltraSPARC Architecture 2009 implementations do not implement any quad floating-point arithmetic operations in hardware. Therefore, an attempt to execute any of them results in a trap on the *illegal_instruction* exception.

# 5.4 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 (deprecated) STFSR and LDFSR instructions, respectively. The 64-bit FSR register is read by the STXFSR instruction and written by the LDXFSR instruction. The ver, ftt, qne, unimplemented (for example, ns), and reserved ("—") fields of FSR are not modified by either LDFSR or LDXFSR.

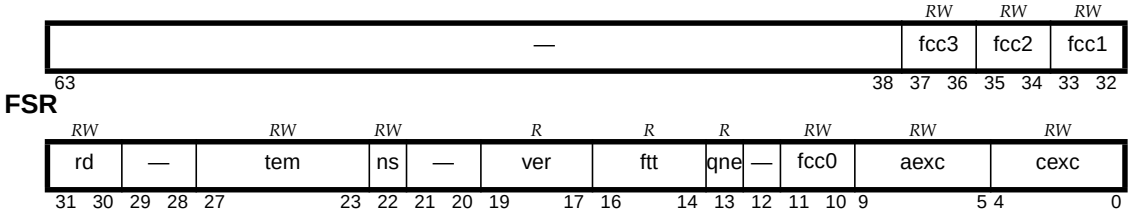



**FIGURE 5-4** FSR Fields

Bits 63–38, 29–28, 21–20, and 12 of FSR are reserved. When read by an STXFSR instruction, these bits always read as zero

**Programming Note** For future compatibility, software should issue LDXFSR instructions only with zero values in these bits or values of these bits exactly as read by a previous STXFSR.

The subsections on pages 60 through 69 describe the remaining fields in the FSR.

## 5.4.1 Floating-Point Condition Codes (fcc0, fcc1, fcc2, fcc3) A1

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

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 compare instruction. The fcc*n* fields are read by STXFSR and written by LDXFSR. The fcc0 field can also be read and written by STFSR and LDFSR, respectively. FBfcc and FBPfcc instructions base their control transfers on the content of these fields. The MOVcc and FMOVcc instructions can conditionally copy a register, based on the contents of these fields.

In TABLE 5-5, $f_{rs1}$ 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 5-4** 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*) |

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

| | Content of *fccn* | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| **Indicated Relation (FCMP*, FCMPE*)** | F[rs1] = F[rs2] | F[rs1] < F[rs2] | F[rs1] > F[rs2] | F[rs1] ? F[rs2] (*unordered*) |

## 5.4.2 Rounding Direction (rd) A1

Bits 31 and 30 select the rounding direction for floating-point results according to IEEE Std 754-1985. TABLE 5-6 shows the encodings.

**TABLE 5-6** Rounding Direction (rd) Field of FSR

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

If the interval mode bit of the General Status register has a value of 1 (GSR.im = 1), then the value of FSR.rd is ignored and floating-point results are instead rounded according to GSR.irnd. See *General Status Register (GSR) (ASR 19)* on page 76 for further details.

## 5.4.3 Trap Enable Mask (tem) A1

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). See FIGURE 5-6 on page 67. If a floating-point 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.

## 5.4.4 Nonstandard Floating-Point (ns) A1

When FSR.ns = 1, it causes a SPARC V9 virtual processor to produce implementation-defined results that may or may not correspond to IEEE Std 754-1985 (impl. dep. #18-V8).

For an implementation in which no nonstandard floating-point mode exists, the ns bit of FSR should always read as 0 and writes to it should be ignored.

For detailed requirements for the case when an UltraSPARC Architecture processor elects to implement floating-point nonstandard mode, see *Floating-Point Nonstandard Mode* on page 394.

## 5.4.5 FPU Version (ver) A1

**IMPL. DEP. #19-V8**: Bits 19 through 17 identify one or more particular implementations of the FPU architecture.

For each SPARC V9 IU implementation (as identified by its HVER.impl field), there may be one or more FPU implementations, or none. FSR.ver 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.

FSR.ver = 7 is reserved to indicate that no hardware floating-point controller is present.

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

## 5.4.6 Floating-Point Trap Type (ftt) A1

Several conditions can cause a floating-point exception trap. When a floating-point exception trap occurs, FSR.ftt (FSR{16:14}) identifies the cause of the exception, the "floating-point trap type." After a floating-point exception occurs, FSR.ftt encodes the type of the floating-point exception until it is cleared (set to 0) by execution of an STFSR, STXFSR, or FPop that does not cause a trap due to a floating-point exception.

The FSR.ftt field can be read by a STFSR or STXFSR instruction. The LDFSR and LDXFSR instructions do not affect FSR.ftt.

Privileged software that handles floating-point traps must execute an STFSR (or STXFSR) to determine the floating-point trap type. STFSR and STXFSR set FSR.ftt to zero after the store completes without error. If the store generates an error and does not complete, FSR.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 FSR.ftt. The ftt field remains zero until the next FPop instruction completes execution.

FSR.ftt encodes the primary condition ("floating-point trap type") that caused the generation of an *fp_exception_other* or *fp_exception_ieee_754* exception. It is possible for more than one such condition to occur simultaneously; in such a case, only the highest-priority condition will be encoded in FSR.ftt. The conditions leading to *fp_exception_other* and *fp_exception_ieee_754* exceptions, their relative priorities, and the corresponding FSR.ftt values are listed in TABLE 5-7. Note that the FSR.ftt values 4 and 5 were defined in the SPARC V9 architecture but are not currently in use, and that the value 7 is reserved for future architectural use.

**TABLE 5-7** FSR Floating-Point Trap Type (ftt) Field

| Condition Detected During Execution of an FPop | Relative Priority (1 = highest) | Result | |
|---|---|---|---|
| | | FSR.ftt Set to Value | Exception Generated |
| invalid_fp_register | 20 | 6 | *fp_exception_other* |
| unfinished_FPop | 30 | 2 | *fp_exception_other* |
| IEEE_754_exception | 40 | 1 | *fp_exception_ieee_754* |
| *Reserved* | — | 3, 4, 5, 7 | — |
| (none detected) | — | 0 | — |

The IEEE_754_exception and unfinished_FPop conditions 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 user 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 *fp_exception_ieee_754* exception or after recovery from an unfinished_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 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 = 1 (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 unchanged.

**ftt = 2 (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. In cases where exceptions have occurred, the cexc field is unchanged.

**Implementation Note** Implementations are encouraged to support standard IEEE 754 floating-point arithmetic with reasonable performance (that is, without generating *fp_exception_other* with FSR.ftt=unfinished_FPop) in all cases, even if some cases are slower than others.

**IMPL. DEP. #248-U3:** 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 FSR.ftt = unfinished_FPop under a different (but specified) set of conditions.

**ftt = 3 (Reserved).**

**SPARC V9 Compatibility Note** In SPARC V9, FSR.ftt = 3 was defined to be "unimplemented_FPop". All conditions which used to cause cause *fp_exception_other* with FSR.ftt = 3 now cause an *illegal_instruction* exception, instead. FSR.ftt = 3 is now reserved and available for other future uses.

**ftt = 4 (Reserved).**

**SPARC V9 Compatibility Note** In the SPARC V9 architecture, FSR.ftt = 4 was defined to be "sequence_error", for use with certain error conditions associated with a floating-point queue (FQ). Since UltraSPARC Architecture implementations generate precise (rather than deferred) traps for floating-point operations, an FQ is not needed; therefore sequence_error conditions cannot occur and ftt =4 has been returned to the pool of reserved ftt values.

**ftt = 5 (Reserved).**

**SPARC V9 Compatibility Note** — In the SPARC V9 architecture, FSR.ftt = 5 was defined to be "hardware_error", for use with hardware error conditions associated with an external floating-point unit (FPU) operating asynchronously to the main processor (IU). Since UltraSPARC Architecture processors are now implemented with an integral FPU, a hardware error in the FPU can generate an exception directly, rather than indirectly report the error through FSR.ftt (as was required when FPUs were external to IUs). Therefore, ftt = 5 has been returned to the pool of reserved ftt values.

**ftt = 6 (invalid_fp_register).** This trap type indicates that one or more F register operands of an FPop are misaligned; that is, a quad-precision register number is not 0 **mod** 4. An implementation generates an *fp_exception_other* trap with FSR.ftt = invalid_fp_register in this case.

**Implementation Note** — If an UltraSPARC Architecture 2007 processor does not implement a particular quad FPop in hardware, that FPop generates an *illegal_instruction* exception instead of *fp_exception_other* with FSR.ftt = 6 (invalid_fp_register), regardless of the specified F registers.

## 5.4.7 FQ Not Empty (qne) Y2

Since UltraSPARC Architecture 2007 virtual processors do not implement a floating-point queue, FSR.qne always reads as zero and writes to FSR.qne are ignored.

## 5.4.8 Accrued Exceptions (aexc) A1

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 5-7 on page 68.

After an FPop completes with ftt = 0, the tem and cexc fields are logically **and**ed 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 **or**ed into the aexc field and no trap is generated. Thus, while (and only while) traps are masked, exceptions are accumulated in the aexc field.

FSR.aexc can be set to a specific value when an LDFSR or LDXFSR instruction is executed.

## 5.4.9 Current Exception (cexc) (A1)

FSR.cexc (FSR{4:0}) indicates whether 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 (set to 0). See FIGURE 5-6 on page 67.

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

The above behavior is summarized in TABLE 5-8 (where "✔" indicates "exception was detected" and "x" indicates "don't care"):

**TABLE 5-8** 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 |
| - | - | - | x | x | x | no | 0 | 0 | 0 |
| - | - | ✔ | x | x | 0 | no | 0 | 0 | 1 |
| - | ✔[1] | ✔[1] | x | 0 | 0 | no | 0 | 1 | 1 |
| ✔[2] | - | ✔[2] | 0 | x | 0 | no | 1 | 0 | 1 |
| - | - | ✔ | x | x | 1 | yes | 0 | 0 | 1 |
| - | ✔[1] | ✔[1] | x | 0 | 1 | yes | 0 | 0 | 1 |
| - | ✔ | - | x | 1 | x | yes | 0 | 1 | 0 |
| - | ✔ | ✔ | x | 1 | x | yes | 0 | 1 | 0 |
| ✔[2] | - | ✔[2] | 1 | x | x | yes | 1 | 0 | 0 |
| ✔[2] | - | ✔[2] | 0 | x | 1 | yes | 0 | 0 | 1 |

Notes: [1] When the underflow trap is disabled (FSR.tem.ufm = 0) 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.

## 5.4.10 Floating-Point Exception Fields (A1)

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):

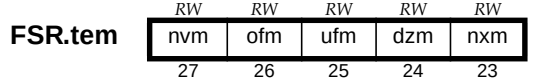



**FIGURE 5-6** Trap Enable Mask (tem) Fields of FSR

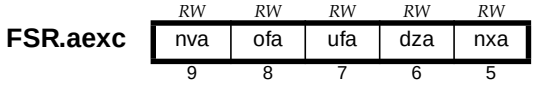



**FIGURE 5-7** Accrued Exception Bits (aexc) Fields of FSR

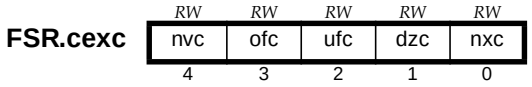



**FIGURE 5-8** Current Exception Bits (aexc) Fields of FSR

**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).

**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.

**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, when the correct unrounded result is not 0:

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

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

The SPARC V9 architecture allows tininess to be detected either before or after rounding. However, in all cases and regardless of the setting of FSR.tem.ufm, an UltraSPARC Architecture strand detects tininess before rounding (impl. dep. #55-V8-Cs10). See *Trapped Underflow Definition (ufm = 1)* on page 393 and *Untrapped Underflow Definition (ufm = 0)* on page 393 for additional details.

**Division by zero (dzc, dza).** An infinite result is produced exactly from finite operands. For example, X ÷ 0.0, where X is subnormal or normalized; 1 = division by zero, 0 = no division by zero.

**Inexact (nxc, nxa).** The rounded result of an operation differs from the infinitely precise unrounded result; 1 = inexact result, 0 = exact result.

### 5.4.11 FSR Conformance

An UltraSPARC Architecture implementation implements the tem, cexc, and aexc fields of FSR in hardware, conforming to IEEE Std 754-1985 (impl. dep. #22-V8).

**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 *fp_exception_other* (with FSR.ftt = 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.

## 5.5 Ancillary State Registers

The SPARC V9 architecture defines several optional ancillary state registers (ASRs) and allows for additional ones. Access to a particular ASR may be privileged or nonprivileged.

An ASR 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 left ASRs numbered 16–31 available for implementation-dependent uses. UltraSPARC Architecture virtual processors implement the ASRs summarized in TABLE 5-9 and defined in the following subsections.

Each virtual processor contains its own set of ASRs; ASRs are not shared among virtual processors.

**TABLE 5-9** ASR Register Summary

| ASR number | ASR name | Register | Read by Instruction(s) | Written by Instruction(s) |
|---|---|---|---|---|
| 0 | $Y^{D}$ | Y register (deprecated) | $RDY^{D}$ | $WRY^{D}$ |
| 1 | — | *Reserved* | — | — |
| 2 | CCR | Condition Codes register | RDCCR | WRCCR |
| 3 | ASI | ASI register | RDASI | WRASI |
| 4 | $TICK^{P_{npt}}$ | TICK register | $RDTICK^{P_{npt}}$, $RDPR^{P}$ (TICK) | $WRPR^{P}$ (TICK) |
| 5 | PC | Program Counter (PC) | RDPC | (all instructions) |
| 6 | FPRS | Floating-Point Registers Status register | RDFPRS | WRFPRS |

**TABLE 5-9** ASR Register Summary *(Continued)*

| ASR number | ASR name | Register | Read by Instruction(s) | Written by Instruction(s) |
|---|---|---|---|---|
| 7–14 | — | *Reserved* | — | — |
| 15 | — | *Reserved* | — | — |
| 16–31 | | non-SPARC V9 ASRs | — | — |
| 16-18 | — | Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | — | — |
| 19 | GSR | General Status register (GSR) | RDGSR, FALIGNDATA, many VIS and floating-point instructions | WRGSR, BMASK, SIAM |
| 20 | SOFTINT_SET$^{P}$ | (pseudo-register, for "Write 1s Set" to SOFTINT register, ASR 22) | — | WRSOFTINT_SET$^{P}$ |
| 21 | SOFTINT_CLR$^{P}$ | (pseudo-register, for "Write 1s Clear" to SOFTINT register, ASR 22) | — | WRSOFTINT_CLR$^{P}$ |
| 22 | SOFTINT$^{P}$ | per-virtual processor Soft Interrupt register | RDSOFTINT$^{P}$ | WRSOFTINT$^{P}$ |
| 23 | TICK_CMPR$^{P}$ (N2) | Tick Compare register | RDTICK_CMPR$^{P}$ (N2) | WRTICK_CMPR$^{P}$ (N2) |
| 24 | STICK$^{P_{npt}}$ | System Tick register | RDSTICK$^{P_{npt}}$ | WRSTICK$^{H}$ |
| 25 | STICK_CMPR$^{P}$ | System Tick Compare register | RDSTICK_CMPR$^{P}$ | WRSTICK_CMPR$^{P}$ |
| 26–31 | — | Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | — | — |

## 5.5.1 32-bit Multiply/Divide Register (Y) (ASR 0) (E3)

> 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 pages 331 and page 379; for the multiply step instruction, see page 287; for division instructions, see pages 323 and 377; for the read instruction, see page 306; and for the write instruction, see page 382.

The low-order 32 bits of the Y register, illustrated in FIGURE 5-9, 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.

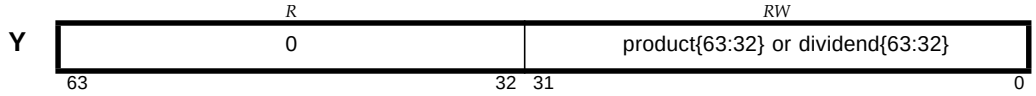



**FIGURE 5-9** Y Register

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

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

## 5.5.2 Integer Condition Codes Register (CCR) (ASR 2) (A1)

The Condition Codes Register (CCR), shown in FIGURE 5-10, contains the integer condition codes. The CCR register may be explicitly read and written by the RDCCR and WRCCR instructions, respectively.

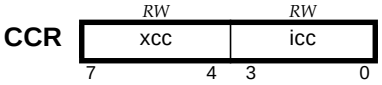



**FIGURE 5-10** Condition Codes Register

### 5.5.2.1 Condition Codes (CCR.xcc and CCR.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 5-11.

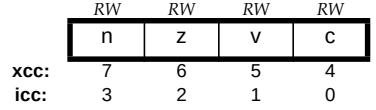



**FIGURE 5-11** Integer Condition Codes (CCR.icc and CCR.xcc)

The n bits indicate whether the two'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) two'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 = borrow, 0 = no borrow (see TABLE 5-10).

**TABLE 5-10** Setting of Carry (Borrow) bits for Subtraction That Sets CCs

| Unsigned Comparison of Operand Values | Setting of Carry bits in CCR |
|---|---|
| R[rs1]{31:0} ≥ R[rs2]{31:0} | CCR.icc.c ← 0 |
| R[rs1]{31:0} < R[rs2]{31:0} | CCR.icc.c ← 1 |
| R[rs1]{63:0} ≥ R[rs2]{63:0} | CCR.xcc.c ← 0 |
| R[rs1]{63:0} < R[rs2]{63:0} | CCR.xcc.c ← 1 |

Both fields of CCR (xcc and icc) are modified by 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 contents of TSTATE.ccr. The behavior of the following instructions are conditioned by the contents of CCR.icc or CCR.xcc:

- BPcc and Tcc instructions (conditional transfer of control)
- Bicc (conditional transfer of control, based on CCR.icc only)
- MOVcc instruction (conditionally move the contents of an integer register)
- FMOVcc instruction (conditionally move the contents of a floating-point register)

**Extended (64-bit) integer condition 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.

**32-bit Integer condition 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.

## 5.5.3 Address Space Identifier (ASI) Register (ASR 3) A1

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.

The ASI register may be explicitly read and written by the RDASI and WRASI instructions, respectively.

Software (executing in any privilege mode) may write any value into the ASI register. However, values in the range $00_{16}$ to $7F_{16}$ are "restricted" ASIs; an attempt to perform an access using an ASI in that range is restricted to software executing in a mode with sufficient privileges for the ASI. When an instruction executing in nonprivileged mode attempts an access using an ASI in the range $00_{16}$ to $7F_{16}$ or an instruction executing in privileged mode attempts an access using an ASI the range $30_{16}$ to $7F_{16}$, a *privileged_action* exception is generated. See Chapter 10, *Address Space Identifiers (ASIs)* for details.

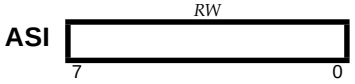



**FIGURE 5-12** Address Space Identifier Register

## 5.5.4 Tick (TICK) Register (ASR 4) A1

FIGURE 5-13 illustrates the TICK register.

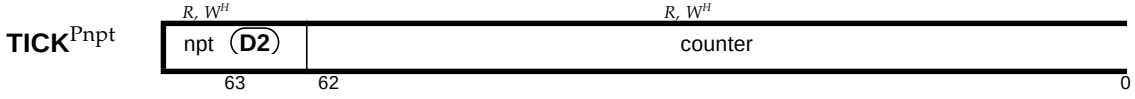



**FIGURE 5-13** TICK Register

The counter field of the TICK register is a 63-bit counter that counts strand clock cycles.

Bit 63 of the TICK register D2 is the nonprivileged trap (npt) bit, which controls access to the TICK register by nonprivileged software.

Hyperprivileged software can always read the TICK register, with either the RDPR or RDTICK instruction.

Hyperprivileged software can always write to the TICK register with the WRPR instruction (there is no distinct WRTICK instruction).

Privileged software can always read the TICK register, with either the RDPR or RDTICK instruction.

Privileged software cannot write to the TICK register; an attempt to do so (with the WRPR instruction) results in an *illegal_instruction* exception.

Nonprivileged software can read the TICK register by using the RDTICK instruction, but only when nonprivileged access to TICK is enabled (TICK.npt = 0) by hyperprivileged software. If nonprivileged access is disabled (TICK.npt = 1), an attempt by nonprivileged software to read the TICK register using the RDTICK instruction causes a *privileged_action* exception.

An attempt by nonprivileged software at any time to read the TICK register using the privileged RDPR instruction causes a *privileged_opcode* exception.

Nonprivileged software cannot write the TICK register. An attempt by nonprivileged software to write the TICK register using the privileged WRPR instruction causes a *privileged_opcode* exception.

TICK.npt is set to 1 by a power-on reset trap. The value of TICK.counter is reset to 0 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 counter. The number of counts between a write and a subsequent read does not accurately reflect the number of strand 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.

The difference between the values read from the TICK register on two reads is intended to reflect the number of strand cycles executed between the reads.

**Programming Note** If a single TICK register is shared among multiple virtual processors, then the difference between subsequent reads of TICK.counter reflects a shared cycle count, not a count specific to the virtual processor reading the TICK register.

**IMPL. DEP. #105-V9:** (a) If an accurate count cannot always be returned when TICK is read, any inaccuracy should be small, bounded, and documented.
(b) 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 nonprivileged 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.

## 5.5.5 Program Counters (PC, NPC) (ASR 5) A1

The PC contains the address of the instruction currently being executed. The least-significant two bits of PC always contain zeroes.

The PC can be read directly with the RDPC instruction. PC cannot be explicitly written by any instruction (including Write State Register), but is implicitly written by control transfer instructions. A WRasr to ASR 5 causes an *illegal_instruction* exception.

The Next Program Counter, NPC, is a pseudo-register that contains the address of the next instruction to be executed if a trap does not occur. The least-significant two bits of NPC always contain zeroes.

NPC is written implicitly by control transfer instructions. However, NPC cannot be read or written explicitly by any instruction.

PC and NPC can be indirectly set by privileged software that writes to TPC[TL] and/or TNPC[TL] and executes a RETRY instruction.

See Chapter 6, *Instruction Set Overview*, for details on how PC and NPC are used.

## 5.5.6 Floating-Point Registers State (FPRS) Register (ASR 6) A1

The Floating-Point Registers State (FPRS) register, shown in FIGURE 5-14, contains control information for the floating-point register file; this information is readable and writable by nonprivileged software.

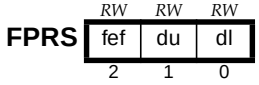



**FIGURE 5-14** Floating-Point Registers State Register

The FPRS register may be explicitly read and written by the RDFPRS and WRFPRS instructions, respectively.

**Enable FPU (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 (FPRS.fef = 1) but the PSTATE.pef bit is not set (PSTATE.pef = 0), then executing a floating-point instruction causes an *fp_disabled* exception; that is, both FPRS.fef and PSTATE.pef must be set to 1 to enable floating-point operations.

**Programming Note** FPRS.fef can be used by application software to notify system software that the application does not require the contents of the F registers to be preserved. Depending on system software, this may provide some performance benefit, for example, the F registers would not have to be saved or restored during context switches to or from that application. Once an application sets FPRS.fef to 0, it must assume that the values in all F registers are volatile (may change at any time).

**Dirty Upper Registers (du).** Bit 1 is the "dirty" bit for the upper half of the floating-point registers; that is, F[32]–F[62]. It is set to 1 whenever any of the upper floating-point registers is modified. The du bit is cleared only by software.

An UltraSPARC Architecture 2007 virtual processor may set FPRS.du pessimistically; that is, it may be set whenever an FPop executes, even though an exception may occur that prevents the instruction from completing so no destination F register was actually modified (impl. dep. #403-S10). Note that if the FPop triggers *fp_disabled*, FPRS.du is *not* modified.

**Dirty Lower Registers (dl).** Bit 0 is the "dirty" bit for the lower 32 floating-point registers; that is, F[0]–F[31]. It is set to 1 whenever any of the lower floating-point registers is modified. The dl bit is cleared only by software.

An UltraSPARC Architecture 2007 virtual processor may set FPRS.dl pessimistically; that is, it may be set whenever an FPop executes, even though an exception may occur that prevents the instruction from completing so no destination F register was actually modified (impl. dep. #403-S10). Note that if the FPop triggers *fp_disabled*, FPRS.dl is *not* modified.

## 5.5.7 General Status Register (GSR) (ASR 19) A1

The General Status Register[1] (GSR) is a nonprivileged read/write register that is implicitly referenced by many VIS instructions. The GSR can be read by the RDGSR instruction (see *Read Ancillary State Register* on page 305) and written by the WRGSR instruction (see *Write Ancillary State Register* on page 381).

If the FPU is disabled (PSTATE.pef = 0 or FPRS.fef = 0), an attempt to access this register using an otherwise-valid RDGSR or WRGSR instruction causes an *fp_disabled* trap.

The GSR is illustrated in FIGURE 5-15 and described in TABLE 5-11.

1. This register was (inaccurately) referred to as the "Graphics Status Register" in early UltraSPARC implementations

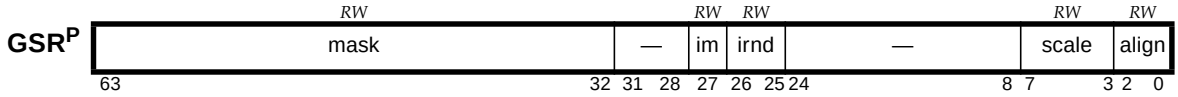



**FIGURE 5-15** General Status Register (GSR) (ASR 19)

**TABLE 5-11** GSR Bit Description

| Bit | Field | Description |
|---|---|---|
| 63:32 | mask | This 32-bit 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: If GSR.im = 0, rounding is performed according to FSR.rd; if GSR.im = 1, rounding is performed according to GSR.irnd. |
| 26:25 | irnd | IEEE Std 754-1985 rounding direction to use in Interval Mode (GSR.im = 1), as follows: |
| 24:8 | — | *Reserved.* |
| 7:3 | scale | 5-bit shift count in the range 0–31, used by the FPACK instructions for formatting. |
| 2:0 | align | Least three significant bits of the address computed by the last-executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction. |

| irnd | Round toward ... |
|---|---|
| 0 | Nearest (even, if tie) |
| 1 | 0 |
| 2 | + ∞ |
| 3 | – ∞ |

## 5.5.8 SOFTINT$^{P}$ Register (ASRs 20 (D2), 21 (D2), 22 (D1))

Software uses the privileged, read/write SOFTINT register (ASR 22) to schedule interrupts (via *interrupt_level_n* exceptions).

SOFTINT can be read with a RDSOFTINT instruction (see *Read Ancillary State Register* on page 305) and written with a WRSOFTINT, WRSOFTINT_SET, or WRSOFTINT_CLR instruction (see *Write Ancillary State Register* on page 381). An attempt to access to this register in nonprivileged mode causes a *privileged_opcode* exception.

**Programming Note** | To atomically modify the set of pending software interrupts, use of the SOFTINT_SET and SOFTINT_CLR ASRs is recommended.

The SOFTINT register is illustrated in FIGURE 5-16 and described in TABLE 5-12.

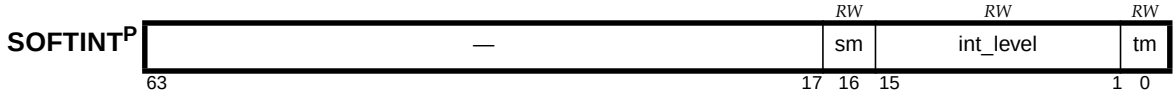



**FIGURE 5-16** SOFTINT Register (ASR 22)

**TABLE 5-12** SOFTINT Bit Description

| Bit | Field | Description |
|---|---|---|
| 16 | sm | When the STICK_CMPR (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 SOFTINT.sm ("STICK match") is set to 1 and a level 14 interrupt (*interrupt_level_14*) is generated. See *System Tick Compare (STICK_CMPR^P) Register (ASR 25)* on page 81 for details. SOFTINT.sm can also be directly written to 1 by software. |
| 15:1 | int_level | When SOFTINT.int_level{*n–1*} (SOFTINT{*n*}) is set to 1, an *interrupt_level_n* exception is generated.<br>**Notes:** A level-14 interrupt (*interrupt_level_14*) can be triggered by SOFTINT.sm, SOFTINT.tm, or a write to SOFTINT.int_level{13} (SOFTINT{14}).<br>A level-15 interrupt (*interrupt_level_15*) can be triggered by a write to SOFTINT.int_level{14} (SOFTINT{15}), or possibly by other implementation-dependent mechanisms.<br>An *interrupt_level_n* exception will only cause a trap if (PIL < *n*) and (PSTATE.ie = 1) and (HPSTATE.hpriv = 0). |
| 0 | tm (N2) | When the TICK_CMPR (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 tm ("TICK match") field in SOFTINT is set to 1 and a level-14 interrupt (*interrupt_level_14*) is generated. See *Tick Compare (TICK_CMPR^P) Register (ASR 23)* on page 79 for details. SOFTINT.tm can also be directly written to 1 by software. |

Setting any of SOFTINT.sm, SOFTINT.int_level{13} (SOFTINT{14}), or SOFTINT.tm to 1 causes a level-14 interrupt (*interrupt_level_14*). However, those three bits are independent; setting any one of them does not affect the other two.

See *Software Interrupt Register (SOFTINT)* on page 526 for additional information regarding the SOFTINT register.

### 5.5.8.1 SOFTINT_SET^P Pseudo-Register (ASR 20) (D2)

A Write State register instruction to ASR 20 (WRSOFTINT_SET) atomically sets selected bits in the privileged SOFTINT Register (ASR 22) (see page 77). That is, bits 16:0 of the write data are **or**ed 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 and write-only. There is no instruction to read this pseudo-register. An attempt to write to ASR 20 in non-privileged mode, using the WRasr instruction, causes a *privileged_opcode* exception.

**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 5-17 illustrates the SOFTINT_SET pseudo-register.

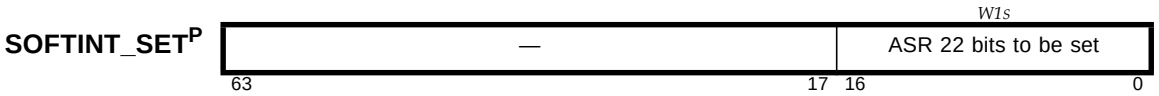



**FIGURE 5-17** SOFTINT_SET Pseudo-Register (ASR 20)

### 5.5.8.2 SOFTINT_CLR$^P$ Pseudo-Register (ASR 21) (D2)

A Write State register instruction to ASR 21 (WRSOFTINT_CLR) atomically clears selected bits in the privileged SOFTINT register (ASR 22) (see page 77). That is, bits 16:0 of the write data are inverted and **and**ed 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 and write-only. There is no instruction to read this pseudo-register. An attempt to write to ASR 21 in non-privileged mode, using the WRasr instruction, causes a *privileged_opcode* exception.

**Programming Note** | There is no actual "register" (machine state) corresponding to ASR 21; it is just a programming interface to conveniently clear (set to '0') selected bits in the SOFTINT register, ASR 22.

FIGURE 5-18 illustrates the SOFTINT_CLR pseudo-register.

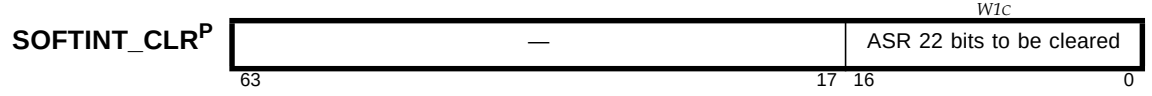



**FIGURE 5-18** SOFTINT_CLR Pseudo-Register (ASR 21))

## 5.5.9 Tick Compare (TICK_CMPR$^P$) Register (ASR 23) (N2)

The privileged TICK_CMPR 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* exception (see *Exception and Interrupt Descriptions* on page 508).

After a power-on reset trap, the int_dis bit is set to 1 (disabling Tick Compare interrupts) and the value of the tick_cmpr field is undefined.

The TICK_CMPR register is illustrated in FIGURE 5-19 and described in TABLE 5-13.

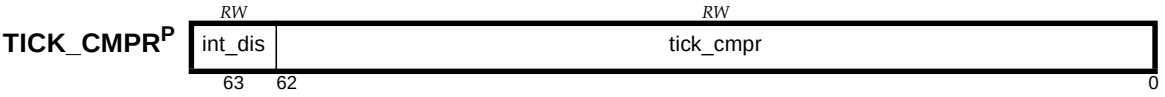



**FIGURE 5-19** TICK_CMPR Register

**TABLE 5-13** TICK_CMPR Register Description

| Bit | Field | Description |
|---|---|---|
| 63 | int_dis | Interrupt Disable. If int_dis = 0, TICK compare interrupts are enabled and if int_dis = 1, TICK compare interrupts are disabled. |
| 62:0 | tick_cmpr | Tick Compare Field. When this field exactly matches the value in TICK.counter and TICK_CMPR.int_dis = 0, SOFTINT.tm is set to 1. This has the effect of posting a level-14 interrupt to the virtual processor, which causes an *interrupt_level_14* trap when (PIL < 14) and (PSTATE.ie = 1 and HPSTATE.hpriv = 0). The level-14 interrupt handler must check SOFTINT{14}, SOFTINT{0} (tm), and SOFTINT{16} (sm) to determine the source of the level-14 interrupt. |

## 5.5.10 System Tick (STICK) Register (ASR 24) (D1)

The System Tick (STICK) register provides a counter that is synchronized across a system, useful 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 (D2) is the nonprivileged trap (npt) bit, which controls access to the TICK register by nonprivileged software.

The STICK register is illustrated in FIGURE 5-20 and described below.

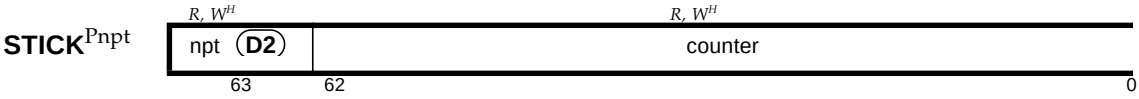



**FIGURE 5-20** STICK Register

Hyperprivileged software can always read the STICK register with the RDSTICK instruction and write it with the WRSTICK instruction.

Privileged software can always read the STICK register with the RDSTICK instruction.

Privileged software cannot write the STICK register; an attempt to execute the WRSTICK instruction in privileged mode results in an *illegal_instruction* exception.

Nonprivileged software can read the STICK register by using the RDSTICK instruction, but only when nonprivileged access to STICK is enabled (STICK.npt = 0) by hyperprivileged software. If nonprivileged access is disabled (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; an attempt to execute the WRSTICK instruction in nonprivileged mode results in an *illegal_instruction* exception.

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 counter.

**IMPL. DEP. #442-S10:** (a) If an accurate count cannot always be returned when STICK is read, any inaccuracy should be small, bounded, and documented. (b) An implementation may implement fewer than 63 bits in STICK.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.

After a power-on reset trap, STICK.npt is set to 1 and the value of STICK.counter is undefined.

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

## 5.5.11 System Tick Compare (STICK_CMPR$^{P}$) Register (ASR 25) D2

The privileged STICK_CMPR 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* exception (see *Exception and Interrupt Descriptions* on page 508).

After a power-on reset trap, the int_dis bit is set to 1 (disabling System Tick Compare interrupts), and the stick_cmpr field is undefined.

The System Tick Compare Register is illustrated in FIGURE 5-21 and described in TABLE 5-14.

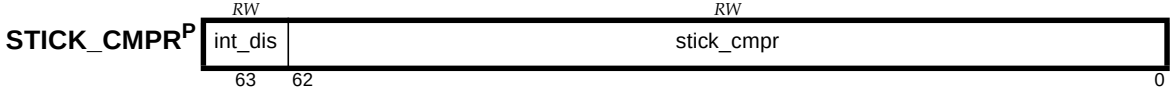



**FIGURE 5-21** STICK_CMPR Register

**TABLE 5-14** STICK_CMPR Register Description

| Bit | Field | Description |
|---|---|---|
| 63 | int_dis | Interrupt Disable. If set to 1, STICK_CMPR interrupts are disabled. |
| 62:0 | stick_cmpr | System Tick Compare Field. When this field exactly matches STICK.counter and STICK_CMPR.int_dis = 0, SOFTINT.sm is set to 1. This has the effect of posting a level-14 interrupt to the virtual processor, which causes an *interrupt_level_14* trap when (PIL < 14) and (PSTATE.ie = 1). The level-14 interrupt handler must check SOFTINT{14}, SOFTINT{0} (tm), and SOFTINT{16} (sm) to determine the source of the level-14 interrupt. |

# 5.6 Register-Window PR State Registers

The state of the register windows is determined by the contents of a set of privileged registers. These state registers can be read/written by privileged software using the RDPR/WRPR instructions. An attempt by nonprivileged software to execute a RDPR or WRPR instruction causes a *privileged_opcode* exception. 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.

**IMPL. DEP. #126-V9-Ms10:** Privileged registers CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN contain values in the range 0 to *N_REG_WINDOWS* – 1. An attempt to write a value greater than *N_REG_WINDOWS* – 1 to any of these registers causes an implementation-dependent value between 0 and *N_REG_WINDOWS* – 1 (inclusive) to be written to the register. Furthermore, an attempt to write a value greater than *N_REG_WINDOWS* – 2 violates the register window state definition in *Register Window State Definition* on page 85.
Although the width of each of these five registers is architecturally 5 bits, the width is implementation dependent and shall be between $\lceil \log_2(N_REG_WINDOWS) \rceil$ 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.
For UltraSPARC Architecture 2007 processors, *N_REG_WINDOWS* = 8. Therefore, each register window state register is implemented with 3 bits, the maximum value for CWP and CLEANWIN is 7, and the maximum value for CANSAVE, CANRESTORE, and OTHERWIN is 6. When these registers are written by the WRPR instruction, bits 63:3 of the data written are ignored.

For details of how the window-management registers are used, see *Register Window Management Instructions* on page 127.

**Programming Note** | CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN must never be set to a value greater than *N_REG_WINDOWS* – 2 on an UltraSPARC Architecture virtual processor. Setting any of these to a value greater than *N_REG_WINDOWS* – 2 violates the register window state definition in *Register Window State Definition* on page 85. Hardware is not required to enforce this restriction; it is up to system software to keep the window state consistent.

**Implementation Note** | A write to any privileged register, including PR state registers, may drain the CPU pipeline.

## 5.6.1 Current Window Pointer (CWP$^P$) Register (PR 9)

D1

The privileged CWP register, shown in FIGURE 5-22, is a counter that identifies the current window into the array of integer registers. See *Register Window Management Instructions* on page 127 and Chapter 12, *Traps*, for information on how hardware manipulates the CWP register.

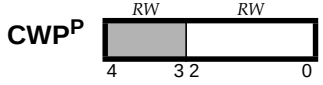



**FIGURE 5-22** Current Window Pointer Register

## 5.6.2 Savable Windows (CANSAVE$^P$) Register (PR 10)

D1

The privileged CANSAVE register, shown in FIGURE 5-23, 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.

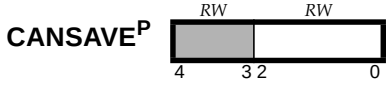



**FIGURE 5-23** CANSAVE Register, Figure 5-24, page 88

## 5.6.3 Restorable Windows (CANRESTORE$^{P}$) Register (PR 11) (D1)

The privileged CANRESTORE register, shown in FIGURE 5-24, 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.

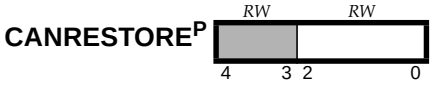



**FIGURE 5-24** CANRESTORE Register

## 5.6.4 Clean Windows (CLEANWIN$^{P}$) Register (PR 12) (D1)

The privileged CLEANWIN register, shown in FIGURE 5-25, contains the number of windows that can be used by the SAVE instruction without causing a *clean_window* exception.

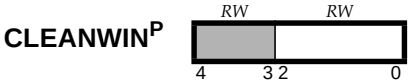



**FIGURE 5-25** CLEANWIN Register

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.

## 5.6.5 Other Windows (OTHERWIN$^{P}$) Register (PR 13) (D1)

The privileged OTHERWIN register, shown in FIGURE 5-26, 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.

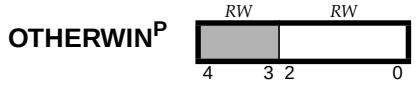



**FIGURE 5-26** OTHERWIN Register

## 5.6.6 Window State (WSTATE$^P$) Register (PR 14) (D1)

The privileged WSTATE register, shown in FIGURE 5-27, 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 State Definition*, below, for details of the semantics of OTHERWIN.

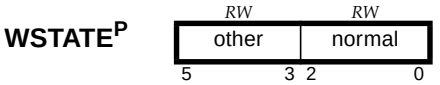



**FIGURE 5-27** WSTATE Register

## 5.6.7 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 82. Those registers are affected by the instructions described in *Register Window Management Instructions* on page 127. Privileged software can read/write these state registers directly by using RDPR/WRPR instructions.

### 5.6.7.1 Register Window State Definition

For the state of the register windows to be consistent, the following must always be true:

$$\text{CANSAVE} + \text{CANRESTORE} + \text{OTHERWIN} = N_REG_WINDOWS - 2$$

FIGURE 5-3 on page 53 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 5-3, these are windows 0 and 5, respectively. They are always present and account for the "2" subtracted from *N_REG_WINDOWS* in the right-hand 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 5-3) 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 5-3) 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 5-3) 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 *N_REG_WINDOWS* – 1.

### 5.6.7.2 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.

See *Register Window Traps* on page 522 for a detailed description of how fill, spill, and *clean_window* traps support register windowing.

# 5.7 Non-Register-Window PR State Registers

The registers described in this section are visible only to software running in privileged or hyperprivileged mode (that is, when PSTATE.priv = 1 **or** HPSTATE.hpriv = 1), and may be accessed with the WRPR and RDPR instructions. (An attempt to execute a WRPR or RDPR instruction in nonprivileged mode causes a *privileged_opcode* exception.)

Each virtual processor provides a full set of these state registers.

**Implementation Note** | A write to any privileged register, including PR state registers, may drain the CPU pipeline.

## 5.7.1 Trap Program Counter ($TPC^P$) Register (PR 0) D1

The privileged Trap Program Counter register (TPC; FIGURE 5-28) contains the program counter (PC) from the previous trap level. There are *MAXTL* instances of the TPC, 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.

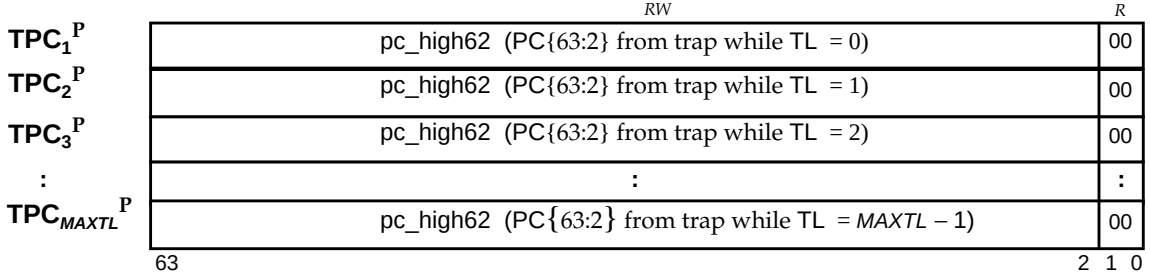



**FIGURE 5-28** Trap Program Counter Register Stack

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

TABLE 5-15 lists the events that cause TPC to be read or written.

**TABLE 5-15** Events that involve TPC, when executing with TL = *n*.

| 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 |

## 5.7.2 Trap Next PC ($TNPC^P$) Register (PR 1) D1

The privileged Trap Next Program Counter register (TNPC; FIGURE 5-28) is the next program counter (NPC) from the previous trap level. There are *MAXTL* instances 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.

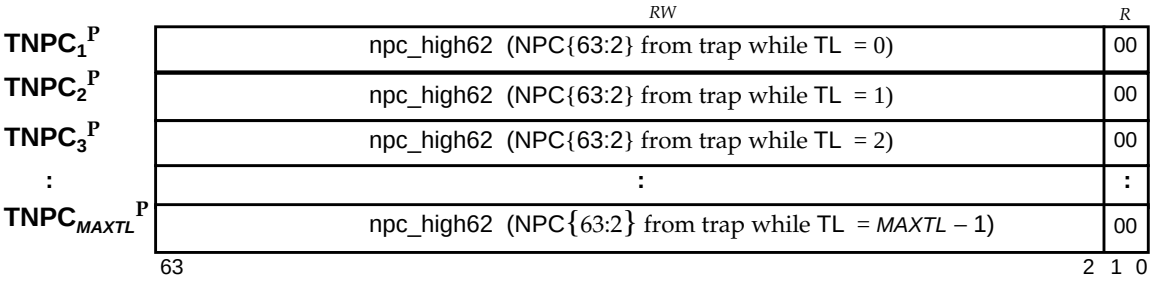



**FIGURE 5-29** Trap Next Program Counter Register Stack

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

TABLE 5-16 lists the events that cause TNPC to be read or written.

**TABLE 5-16** Events that involve TNPC, when executing with TL = *n*.

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

## 5.7.3 Trap State (TSTATE$^{P}$) Register (PR 2) D1

The privileged Trap State register (TSTATE; FIGURE 5-30) contains the state from the previous trap level, comprising the contents of the GL, CCR, ASI, CWP, and PSTATE registers from the previous trap level. There are *MAXTL* instances 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.

| | *RW* | *RW* | *RW* | *R* | *RW* | *R* | *RW* |
|---|---|---|---|---|---|---|---|
| **TSTATE$_1$$^{P}$** | gl (GL from TL = 0) | ccl (CCR from TL = 0) | asi (ASI from TL = 0) | — | pstate (PSTATE from TL = 0) | — | cwp (CWP from TL = 0) |
| **TSTATE$_2$$^{P}$** | gl (GL from TL = 1) | ccl (CCR from TL = 1) | asi (ASI from TL = 1 | — | pstate (PSTATE from TL = 1) | — | cwp (CWP from TL = 1) |
| **TSTATE$_3$$^{P}$** | gl (GL from TL = 2) | ccr (CCR from TL = 2) | asi (ASI from TL = 2 | — | pstate (PSTATE from TL = 2) | — | cwp (CWP from TL = 2) |
| **:** $^{P}$ | : | : | : | : | : | : | : |
| **TSTATE$_{MAXPTL}$$^{P}$** | gl (GL from TL = *MAXPTL* − 1) | ccr (CCR from TL = *MAXPTL* − 1) | asi (ASI from TL = *MAXPTL* − 1) | — | pstate (PSTATE from TL = *MAXPTL* − 1) | — | cwp (CWP from TL = *MAXPTL* − 1) |
| **TSTATE$_{MAXPTL+1}$$^{H}$** | gl (GL from TL = *MAXPTL*) | ccr (CCR from TL = *MAXPTL*) | asi (ASI from TL = *MAXPTL*) | — | pstate (PSTATE from TL = *MAXPTL*) | — | cwp (CWP from TL = *MAXPTL*) |
| **:** $^{H}$ | : | : | : | : | : | : | : |
| **TSTATE$_{MAXTL}$$^{H}$** | gl (GL from TL = *MAXTL* − 1) | ccr (CCR from TL = *MAXTL* − 1) | asi (ASI from TL = *MAXTL* − 1) | — | pstate (PSTATE from TL = *MAXTL* − 1) | — | cwp (CWP from TL = *MAXTL* − 1) |
| | 42 40 | 39 32 | 31 24 | 23 21 | 20 8 | 7 5 | 4 0 |

**TABLE 5-17**

**FIGURE 5-30** Trap State (TSTATE) Register Stack

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.

**V9 Compatibility Note** Because of the addition of additional bits in the PSTATE register in the UltraSPARC Architecture, a 13-bit PSTATE value is stored in TSTATE instead of the 10-bit value specified in the SPARC V9 architecture.

TABLE 5-18 lists the events that cause TSTATE to be read or written.

**TABLE 5-18** Events That Involve TSTATE, When Executing with TL = *n*

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

## 5.7.4 Trap Type ($TT^P$) Register (PR 3) D1

The privileged Trap Type register (TT; see FIGURE 5-31) 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 12-2 on page 470). There are *MAXTL* 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.

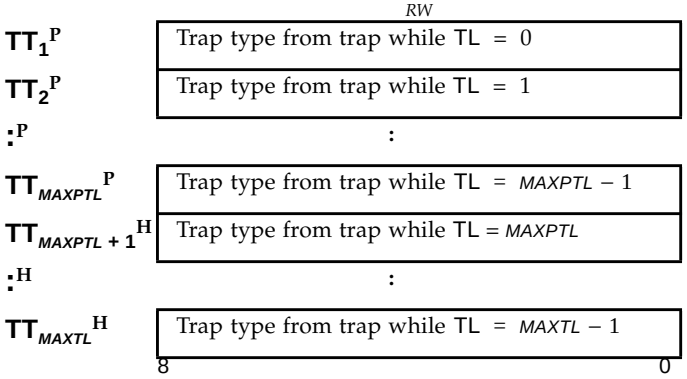



**FIGURE 5-31** Trap Type Register Stack

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*], where *n* is greater than the current trap level (*n* > TL), is undefined.

TABLE 5-19 lists the events that cause TT to be read or written.

**TABLE 5-19** Events that involve TT, when executing with TL = *n*.

| Event | Effect |
|---|---|
| Trap | TT[*n* + 1] ← (trap type) |
| RDPR (TT) | R[rd] ← TT[*n*] |
| WRPR (TT) | TT[*n*] ← *value* |
| Power-on reset (*POR*) | TT values TT[1] through TT[*MAXTL* − 1] are left undefined; TT[*MAXTL*] ← $001_{16}$. |

## 5.7.5 Trap Base Address ($TBA^P$) Register (PR 5) D1

The privileged Trap Base Address register (TBA), shown in FIGURE 5-32, provides the upper 49 bits (bits 63:15) of the virtual address used to select the trap vector for a trap that is to be delivered to privileged mode. The lower 15 bits of the TBA always read as zero, and writes to them are ignored.

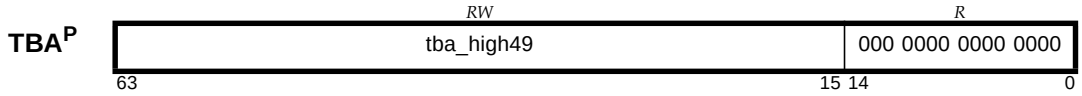



**FIGURE 5-32** Trap Base Address Register

Details on how the full address for a trap vector is generated, using TBA and other state, are provided in *Trap-Table Entry Address to Privileged Mode* on page 479.

## 5.7.6 Processor State ($PSTATE^P$) Register (PR 6) D1

The privileged Processor State register (PSTATE), shown in FIGURE 5-33, contains control fields for the current state of the virtual processor. There is only one instance of the PSTATE register per virtual processor.

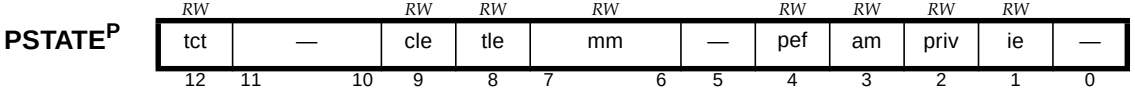



**FIGURE 5-33** PSTATE Field

Writes to PSTATE are 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.

The following subsections describe the fields of the PSTATE register.

**Trap on Control Transfer (tct).** PSTATE.tct enables the Trap-on-Control-Transfer feature.When PSTATE.tct = 1, the virtual processor monitors each control transfer instruction (CTI) to determine whether a *control_transfer_instruction* exception should be generated. If the virtual processor is executing a CTI, PSTATE.tct = 1, and a successful control transfer is going to occur as a result of execution of that CTI, the processor generates a *control_transfer_instruction* exception instead of completing execution of the control transfer instruction.

When the trap is taken, the address of the CTI (the value of PC when the CTI began execution) is saved in TPC[TL] and the value of NPC when the CTI began execution is saved in TNPC[TL].

During initial trap processing, before trap handler code is executed, the virtual processor sets PSTATE.tct to 0 (so that control transfers within the trap handler don't cause additional traps).

**Programming Note** Trap handler software for a *control_transfer_instruction* trap should take care when returning to the software that caused the trap. Execution of DONE or RETRY causes PSTATE.tct to be restored from TSTATE, normally setting PSTATE.tct back to 1. If trap handler software intends for *control_transfer_instruction* exceptions to be reenabled, then it must emulate the trapped control transfer instruction.

**IMPL. DEP. #450-S20:** Availability of the *control_transfer_instruction* exception feature is implementation dependent. If not implemented, trap type $074_{16}$ is unused, PSTATE.tct always reads as zero, and writes to PSTATE.tct are ignored.

For the purposes of the *control_transfer_instruction* exception, a discontinuity in instruction-fetch addresses caused by a WRPR to PSTATE that changes the value of PSTATE.am (and thus, potentially the more-significant 32 bits of the address of the next instruction; see page 96) is *not* considered a control transfer. Only explicit CTIs can generate a *control_transfer_instruction* exception.

**Current Little Endian (cle).** This bit affects the endianness of data accesses performed using an implicit ASI. 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. Specific ASIs used are shown in TABLE 6-3 on page 118. Note that the endianness of a data access may be further affected by TTE.ie used by the MMU.

Instruction accesses are unaffected by PSTATE.cle and are always performed in big-endian byte order.

**Trap Little Endian (tle).** When a trap is taken, the current PSTATE register is pushed onto the trap stack.

During a virtual processor trap to privileged mode, 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. During a virtual processor trap to hyperprivileged mode, the PSTATE.tle bit is *not* copied into PSTATE.cle of the new PSTATE register and is unused.

**Memory Model (mm).** This 2-bit field determines the memory model in use by the virtual processor. The defined values for an UltraSPARC Architecture virtual processor are listed in TABLE 5-20.

**TABLE 5-20** PSTATE.mm Encodings

| mm Value | Selected Memory Model |
|---|---|
| 00 | Total Store Order (TSO) |
| 01 | *Reserved* |
| 10 | *Implementation dependent* (impl. dep. #113-V9-Ms10) |
| 11 | *Implementation dependent* (impl. dep. #113-V9-Ms10) |

The current memory model is determined by the value of PSTATE.mm. Software should refrain from writing the values $01_2$, $10_2$, or $11_2$ to PSTATE.mm because they are implementation-dependent or reserved for future extensions to the architecture, and in any case not currently portable across implementations.

- **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 or stores.

**IMPL. DEP. #113-V9-Ms10:** Whether memory models represented by PSTATE.mm = $10_2$ or $11_2$ are supported in an UltraSPARC Architecture processor is implementation dependent. If the $10_2$ model is supported, then when PSTATE.mm = $10_2$ the implementation must correctly execute software that adheres to the RMO model described in *The SPARC Architecture Manual-Version 9*. If the $11_2$ model is supported, its definition is implementation dependent.

**IMPL. DEP. #119-Ms10**: The effect of writing an unimplemented memory model designation into PSTATE.mm is implementation dependent.

**SPARC V9 Compatibility Notes**

The PSO memory model described in SPARC V8 and SPARC V9 architecture specifications was never implemented in a SPARC V9 implementation and is not included in the UltraSPARC Architecture specification.

The RMO memory model described in the SPARC V9 specification was implemented in some non-Sun SPARC V9 implementations, but is not directly supported in UltraSPARC Architecture 2007 implementations. All software written to run correctly under RMO will run correctly under TSO on an UltraSPARC Architecture 2007 implementation.

**Enable FPU (pef).** When set to 1, the PSTATE.pef bit enables the floating-point unit. This allows privileged software to manage the FPU. For the FPU to be usable, both PSTATE.pef and FPRS.fef must be set to 1. Otherwise, any floating-point instruction that tries to reference the FPU causes an *fp_disabled* trap.

If an implementation does not contain a hardware FPU, PSTATE.pef always reads as 0 and writes to it are ignored.

**Address Mask (am).** The PSTATE.am bit is provided to allow 32-bit SPARC software to run correctly on a 64-bit SPARC V9 processor, by masking out (zeroing) bits 63:32 of virtual addresses at appropriate times.

When PSTATE.am = 0, the full 64 bits of all instruction and data addresses are *preserved* at all times.

When PSTATE.am = 1, bits 63:32 of instruction and data virtual addresses are masked out (treated as 0).

**Programming Note**

It is the responsibility of privileged and hyperprivileged software to manage the setting of the PSTATE.am bit, since hardware masks virtual addresses when PSTATE.am = 1.

Misuse of the PSTATE.am bit can result in undesirable behavior. PSTATE.am should *not* be set to 1 in privileged or hyperprivileged mode.

The PSTATE.am bit should always be set to 1 when 32-bit software is executed.

Instances in which the more-significant 32 bits of a virtual address **are masked** include:

- Before any data address is sent out of the virtual processor (notably, to the memory system, which includes MMU, internal caches, and external caches).

- Before any instruction address is sent out of the virtual processor (notably, to the memory system, which includes MMU, internal caches, and external caches)

  **Forward Compatibility Note** | This behavior is expected to be expanded in the next revision of the architecture to include real and physical addresses, as well as virtual addresses.

- When the value of PC is stored to a general-purpose register by a CALL, JMPL, or RDPC instruction (closed impl.dep. #125-V9-Cs10)
- When the values of PC and NPC are written to TPC[TL] and TNPC[TL] (respectively) during a trap (closed impl.dep. #125-V9-Cs10)
- Before any virtual address is sent to a watchpoint comparator

  **Programming Note** | A 64-bit comparison is always used when performing a masked watchpoint address comparison with the Instruction or Data VA watchpoint register. When PSTATE.am = 1, the more significant 32 bits of the VA watchpoint register must be zero for a match (and resulting trap) to occur.

- When an exception occurs and an address is written to the Data Synchronous Fault Address register (D-SFAR) (impl.dep. #241-U3)

  **Programming Note** | If a memory access is initiated when PSTATE.am = 1, the memory system will only see a 32-bit memory address. Therefore, if such a memory access causes an exception or error, the memory system will (is only able to) report a 32-bit address in the D-SFAR register (64-bit address with the more-significant 32 bits set to 0).

When PSTATE.am = 1, the more-significant 32 bits of a virtual address **are explicitly preserved and *not* masked** out in the following cases:

- When a target address is written to NPC by a control transfer instruction

  **Forward Compatibility Note** | This behavior is expected to change in the next revision of the architecture, such that implementations will explicitly mask out (not preserve) the more-significant 32 bits, in this case.

- When NPC is incremented to NPC + 4 during execution of an instruction that is not a taken control transfer

  **Forward Compatibility Note** | This behavior is expected to change in the next revision of the architecture, such that implementations will explicitly mask out (not preserve) the more-significant 32 bits, in this case.

- When a WRPR instruction writes to TPC[TL] or TNPC[TL]

**Programming Note** Since writes to PSTATE are nondelayed (see page 91), a change to PSTATE.am can affect which instruction is executed immediately after the write to PSTATE.am. Specifically, if a WRPR to the PSTATE register changes the value of PSTATE.am from '0' to '1', and NPC{63:32} when the WRPR began execution was nonzero, then the next instruction executed after the WRPR will be from the address indicated in NPC{31:0} (with the more-significant 32 address bits set to zero).

- When a RDPR instruction reads from TPC[TL] or TNPC[TL]

If *(1)* TSTATE[TL].pstate.am = 1 and *(2)* a DONE or RETRY instruction is executed[1], it is implementation dependent whether the DONE or RETRY instruction masks (zeroes) the more-significant 32 bits of the values it places into PC and NPC (impl. dep. #417-S10).

**IMPL. DEP. #443-S10:** In hyperprivileged mode, when PSTATE.am = 1 and physical addressing is being used, it is implementation-dependent whether the more-significant 32 bits of addresses are masked (treated as zero).

**Programming Note** Because of implementation dependency #417-S10, great care must be taken in trap handler software if TSTATE[TL].pstate.am = 1 and the trap handler wishes to write a nonzero value to the more-significant 32 bits of TPC[TL] or TNPC[TL].

**Privileged Mode (priv).** When PSTATE.priv = 1 and HPSTATE.hpriv = 0, the virtual processor is operating in privileged mode.

When PSTATE.priv = 0 and HPSTATE.hpriv = 0, the processor is operating in nonprivileged mode

When HPSTATE.hpriv = 1, the virtual processor is operating in hyperprivileged mode, independent of the state of PSTATE.priv. Hyperprivileged mode provides a superset of the capabilities of privileged mode.

**PSTATE_interrupt_enable (ie).** PSTATE.ie controls when the virtual processor can take traps due to disrupting exceptions (such as interrupts or errors unrelated to instruction processing).

1. which sets PSTATE.am to '1', by restoring the value from TSTATE[TL].pstate.am to PSTATE.am

Outstanding disrupting exceptions that are destined for privileged mode can only cause a trap when the virtual processor is in nonprivileged or privileged mode and PSTATE.ie = 1. At all other times, they are held pending. For more details, see *Conditioning of Disrupting Traps* on page 474.

Outstanding disrupting exceptions that are destined for hyperprivileged mode can only cause a trap when the virtual processor is not in hyperprivileged mode, or when it is in hyperprivileged mode and PSTATE.ie = 1. At all other times, they are held pending. For more details, see *Conditioning of Disrupting Traps* on page 474

**SPARC V9 Compatibility Note** Since the UltraSPARC Architecture provides a more general "alternate globals" facility (through use of the GL register) than does SPARC V9, an UltraSPARC Architecture processor does not implement the SPARC V9 PSTATE.ag bit.

## 5.7.7 Trap Level Register ($TL^P$) (PR 7) D1

The privileged Trap Level register (TL; FIGURE 5-34) 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.

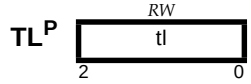



**FIGURE 5-34** Trap Level Register

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.

**IMPL. DEP. #101-V9-CS10:** The architectural parameter *MAXPTL* is a constant for each implementation; its legal values are from 2 to 6 (supporting from 2 to 6 levels of saved trap state visible to privileged software). In a typical implementation *MAXPTL* = *MAXPGL* (see impl. dep. #401-S10). The architectural parameter *MAXTL* is a constant for each implementation; its legal values are from 4 to 7 (supporting from 4 to 7 levels of saved trap state). Architecturally, *MAXPTL* must be ≥ 2, *MAXTL* must be ≥ 4, and *MAXTL* must be > *MAXPTL*.

In an UltraSPARC Architecture 2007 implementation, *MAXPTL* = 2 and *MAXTL* = 6. See Chapter 12, *Traps*, for more details regarding the TL register.

; see processor-specific documentation for the value of *MAXTL* on a particular implementationAfter a power-on rest (POR), TL is set to *MAXTL*.

The effect of writing to TL with a WRPR instruction is summarized in TABLE 5-21.

**TABLE 5-21** Effect of WRPR of Value *x* to Register TL

| Value *x* Written with WRPR | Privilege Level when Executing WRPR | | |
|---|---|---|---|
| | **Nonprivileged** | **Privileged** | **Hyperprivileged** |
| $x \leq MAXPTL$ | *privileged_opcode* exception | TL ← *x* | TL ← *x* |
| $MAXPTL < x \leq MAXTL$ | | TL ← *MAXPTL* (no exception generated) | |
| $x > MAXTL$ | | | TL ← *MAXTL* (no exception generated) |

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

**Programming Note** An UltraSPARC Architecture implementation only needs to implement sufficient bits in the TL register to encode the maximum trap level value. In an implementation where$MAXTL \leq 7$, bits 63:3 of data written to the TL register using the WRPR instruction are ignored; only the least-significant three bits (bits 2:0) of TL are actually written. For example, if $MAXTL = 6$, writing a value of $09_{16}$ to the TL register causes a value of $1_{16}$ to actually be stored in TL.

**Implementation Note** *MAXPTL* =2 for all UltraSPARC Architecture 2007 processors. Writing a value between 3 and 7 to the TL register in privileged mode causes a 2 to be stored in TL.

**Programming Note** Although it is possible for hyperprivileged software to set TL > *MAXPTL* for privileged or nonprivileged software†, an UltraSPARC Architecture virtual processor's behavior when executing with TL > *MAXPTL* outside of hyperprivileged mode is undefined.

Although it is possible for privileged or hyperprivileged software to set TL > 0 for nonprivileged software†, an UltraSPARC Architecture virtual processor's behavior when executing with TL > 0 in nonprivileged mode is undefined.

† by executing a WRPR to TSTATE followed by DONE instruction or RETRY instruction or a JMPL/WRHPR instruction pair.

## 5.7.8 Processor Interrupt Level ($PIL^P$) Register (PR 8) D1

The privileged Processor Interrupt Level register (PIL; see FIGURE 5-35) specifies the interrupt level above which the virtual processor will accept an *interrupt_level_n* interrupt. Interrupt priorities are mapped so that interrupt level 2 has greater priority than interrupt level 1, and so on. See TABLE 12-4 on page 485 for a list of exception and interrupt priorities.

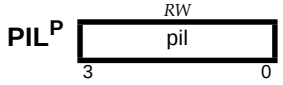



**FIGURE 5-35** Processor Interrupt Level Register

**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.

## 5.7.9 Global Level Register ($GL^P$) (PR 16) D1

The privileged Global Level (GL) register selects which set of global registers is visible at any given time.

FIGURE 5-36 illustrates the Global Level register.

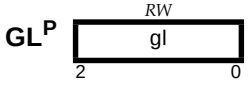



**FIGURE 5-36** Global Level Register, GL

When a trap occurs, GL is stored in TSTATE[TL].gl, GL is incremented, and a new set of global registers (R[1] through R[7]) becomes visible. A DONE or RETRY instruction restores the value of GL from TSTATE[TL].

The valid range of values that the GL register may contain is *MAXGL*, where *MAXGL* is one fewer than the number of global register sets available to the virtual processor.

**IMPL. DEP. #401-S10:** The architectural parameter *MAXPGL* is a constant for each implementation; its legal values are from 2 to 7 (supporting from 3 to 8 sets of global registers visible to privileged software). In a typical implementation *MAXPGL* = *MAXPTL* (see impl. dep. #101-V9-CS10). The architectural parameter *MAXGL* is a constant for each implementation; its legal values are from 3 to 7 (supporting from 4 to 8 sets of global registers). Architecturally, *MAXPGL* must be ≥ 2 and *MAXGL* must be > *MAXPGL*.

*MAXPGL* = 2 in all UltraSPARC Architecture 2007 implementations and *MAXGL* is implementation-dependent. (impl. dep. #401-S10).

**IMPL. DEP. #400-S10:** Although GL is defined as a 3-bit register, an implementation may implement any subset of those bits sufficient to encode the values from 0 to *MAXGL* for that implementation. If any bits of GL are not implemented, they read as zero and writes to them are ignored.

GL operates similarly to TL, in that it increments during entry to a trap, but the values of GL and TL are independent. That is, TL = *n* does not imply that GL = *n*, and GL = *n* does not imply that TL = *n*. Furthermore, there may be a different total number of global levels (register sets) than there are trap levels; that is, *MAXTL* and *MAXGL* are not necessarily equal.

The GL register can be accessed directly with the RDPR and WRPR instructions (as privileged register number 16). Writing the GL register directly with WRPR will change the set of global registers visible to all instructions subsequent to the WRPR.

In privileged mode, attempting to write a value greater than *MAXPGL* to the GL register causes *MAXPGL* to be written to GL.

In hyperprivileged mode, attempting to write a value greater than *MAXGL* to the GL register causes *MAXGL* to be written to GL.

When a DONE or RETRY instruction is executed and HTSTATE[TL].hpstate.hpriv = 0 (which will cause the DONE or RETRY to return the virtual processor to nonprivileged or privileged mode), the value of GL restored from TSTATE[TL] saturates at *MAXPGL*. That is, if the value in TSTATE[TL].gl is greater than *MAXPGL*, then *MAXPGL* is substituted and written to GL. This protects against non-hyperprivileged software executing with GL > *MAXPGL*.

**Programming Note** Although it is possible for hyperprivileged software to set GL > *MAXPGL* for privileged or nonprivileged software†, executing with GL > *MAXPGL* outside of hyperprivileged mode is an illegal state and the behavior of a virtual processor in that state is undefined.

† by executing a WRPR that modifies GL, followed by a JMPL/WRHPR instruction pair (it is not possible to set GL > *MAXPGL* using DONE/RETRY)

The effect of writing to GL with a WRPR instruction is summarized in TABLE 5-22.

**TABLE 5-22** Effect of WRPR to Register GL

| Value $x$ Written with WRPR | Privilege Level when WRPR Is Executed | | |
|---|---|---|---|
| | Nonprivileged | Privileged | Hyperprivileged |
| $x \le MAXPGL$ | *privileged_opcode* exception | GL ← $x$ | GL ← $x$ |
| $MAXPGL < x \le MAXGL$ | | GL ← *MAXPGL* (no exception generated) | |
| $x > MAXGL$ | | | GL ← *MAXGL* (no exception generated) |

If *MAXGL* < *MAXTL*, then there are fewer sets of global registers than trap levels. In this case, if a trap occurs while GL = *MAXGL*, GL will have the same value before the trap occurs and in the software that handles the trap. Trap handler software must detect this case and safely save any global register before the trap handler writes to it. The Hyperprivileged Scratchpad registers (see *Privileged Scratchpad Registers (ASI_SCRATCHPAD)* on page 453) may be useful in such cases.

**Programming Note** An UltraSPARC Architecture implementation only needs to implement sufficient bits in the GL register to encode the maximum global level value. In an implementation where *MAXGL* ≤ 7, bits 63:3 of data written to the GL register using the WRPR instruction are ignored; only the least-significant three bits (bits 2:0) are actually written to GL. For example, if *MAXGL* = 7, writing a value of $9_{16}$ to the TL register causes a value of $1_{16}$ to actually be stored in GL.

Since TSTATE itself is software-accessible, it is possible that when a DONE or RETRY is executed to return from a trap handler, the value of GL restored from TSTATE[TL] will be different from that which was saved into TSTATE[TL] when the trap occurred.

During power-on reset (POR), the value of GL is set to *MAXGL*. During all other resets, GL is incremented (the same behavior as TL).

# 5.8 HPR State Registers

The registers described in this section can be directly accessed with the hyperprivileged WRHPR and RDHPR instructions.

An attempt to read or write any HPR state register (using RDHPR or WRHPR) in privileged or nonprivileged modes (that is, when HPSTATE.hpriv = 0) causes an *illegal_instruction* exception.

## 5.8.1 Hyperprivileged State (HPSTATE$^{H}$) Register (HPR 0) (D1)

The Hyperprivileged State register (HPSTATE), shown in FIGURE 5-37, contains hyperprivileged control fields for the virtual processor. There is one instance of the HPSTATE register per virtual processor.

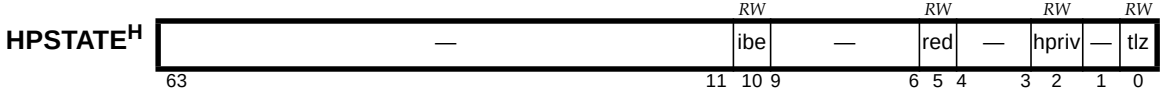



**FIGURE 5-37** HPSTATE Fields

Writing HPSTATE is nondelayed; that is, new machine state written to HPSTATE is visible to the next instruction executed. The hyperprivileged RDHPR and WRHPR instructions are used to read and write HPSTATE, respectively.

The following subsections describe the fields contained in the HPSTATE register.

**Instruction Breakpoint Enable (ibe).** When HPSTATE.ibe = 1, the Instruction Breakpoint feature is enabled, allowing an *instr_breakpoint* exception to occur. When an *instr_breakpoint* exception trap occurs, the virtual processor sets HPSTATE.ibe to 0 before entering trap handler software, to guarantee that no additional *instr_breakpoint* exception can occur in the instruction breakpoint trap handler unless the trap handler explicitly reenables instruction breakpointing by setting HPSTATE.ibe to 1.

**RED_state (red).** When HPSTATE.red is set to 1, the virtual processor is operating in `RED_state` (Reset, Error, and Debug state). See ***`RED_state`*** on page 467. The virtual processor sets HPSTATE.red when any hardware reset occurs. HPSTATE.red is also set to 1 when a trap is taken while TL = (*MAXTL* − 1). Software can reliably exit `RED_state` by one of two methods:

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

**Programming Note** Software should not write 0 to HPSTATE.red in the delay slot of a DCTI (e.g. JMPL instruction). Exiting `RED_state` using a DONE or RETRY instruction avoids this problem entirely.

**Programming Note** | HPSTATE.hpriv = 0 and HPSTATE.red = 1 is an undefined operational state. Therefore, care should be taken never to write that combination of values to HPSTATE.

**Hyperprivileged mode (hpriv).** When HPSTATE.hpriv = 1, the virtual processor is operating in hyperprivileged mode and ignores PSTATE.priv.

When HPSTATE.hpriv = 0, the processor is operating in privileged or nonprivileged mode, as determined by PSTATE.priv.

See the Programming Note on page 385, recommending that a WRHPR instruction that changes HPSTATE.priv never be executed in the delay slot of a DCTI instruction.

**Trap Level Zero trap enable (tlz).** When HPSTATE.tlz = 0, generation of *trap_level_zero* exceptions is disabled. When all three of the following conditions exist, a *trap_level_zero* exception is generated:

- HPSTATE.tlz = 1 (generation of *trap_level_zero* is enabled)
- the virtual processor is in nonprivileged or privileged mode (HPSTATE.hpriv = 0)
- the trap level (TL) register's value is zero (TL = 0)

**Programming Note** | The purpose of *trap_level_zero* is to improve efficiency when descheduling a virtual processor. When a descheduling event occurs and the virtual processor is executing in privileged mode at TL > 0, hyperprivileged software can choose to enable the *trap_level_zero* exception (set HPSTATE.tlz ← 1) and return to privileged mode, enabling privileged software to complete its TL > 0 processing. When privileged code returns to TL = 0, this exception enables the hyperprivileged code to regain control and deschedule the virtual processor with low overhead.

## 5.8.2 Hyperprivileged Trap State (HTSTATE$^H$) Register (HPR 1) D1

The Hyperprivileged Trap State register (HTSTATE; FIGURE 5-38) contains the hyperprivileged state from the previous trap level, comprising the contents of the HPSTATE register from the previous trap level. There are *MAXTL* instances of the HTSTATE register, but only one is accessible at a time. The current value in the TL register determines which instance of HTSTATE is accessible.

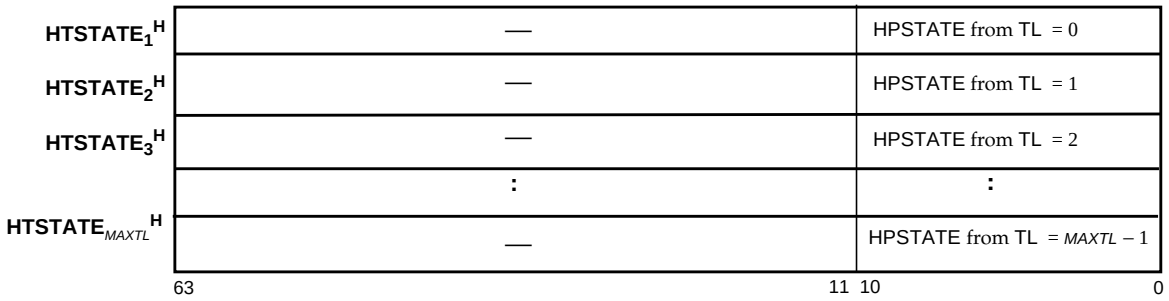



**FIGURE 5-38** Hyperprivileged Trap State Register

An attempt to read or write the HTSTATE register when TL = 0 causes an *illegal_instruction* exception.

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

TABLE 5-23 lists the events that cause HTSTATE to be read or written.

**TABLE 5-23** Events that involve HTSTATE, when executing with TL = *n*.

| Event | Effect |
|---|---|
| Trap | HTSTATE[*n* + 1]{10:0} ← HPSTATE |
| DONE instruction | HPSTATE ← HTSTATE[*n*]{10:0} |
| RETRY instruction | HPSTATE ← HTSTATE[*n*]{10:0} |
| RDHPR (HTSTATE) | R[rd] ← HTSTATE[*n*] |
| WRHPR (HTSTATE) | HTSTATE[*n*] ← *value* |
| Power-on reset (*POR*) | All HTSTATE values are left undefined |

## 5.8.3 Hyperprivileged Interrupt Pending (HINTP$^H$) Register (HPR 3) (D1)

The hyperprivileged HINTP register provides a mechanism for hyperprivileged software to determine that an *hstick_match* interrupt is pending while PSTATE.ie = 0 and to clear the interrupt without having to first set PSTATE.ie = 1 and take a disrupting trap.

When HINTP.hsp = 1, a match between STICK and HSTICK_CMPR has occurred while match interrupt generation was enabled (HSTICK_CMPR.int_dis = 0, see *System Tick Compare (STICK_CMPR$^P$) Register (ASR 25)* on page 81), causing an *hstick_match* exception to be generated.

**Programming Note** | A pending *hstick_match* exception can also be generated if software directly writes a '1' to HINTP.hsp.

When HINTP.hsp = 0, no interrupt is pending due to a match between STICK and HSTICK_CMPR. Software can clear a pending *hstick_match* interrupt (indicated by HINTP.hsp = 1) by writing 0 to HINTP.hsp.

The format of the HINTP register is illustrated in FIGURE 5-39.

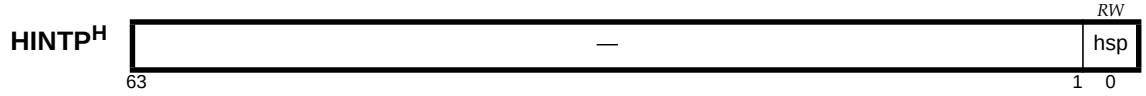



**FIGURE 5-39** Hyperprivileged Interrupt Pending (HINTP) Register Format

## 5.8.4 Hyperprivileged Trap Base Address (HTBA$^H$) Register (HPR 5) D1

The Hyperprivileged Trap Base Address register (HTBA), shown in FIGURE 5-40, provides the most significant 50 bits (bits 63:14) of the physical address used to select the trap vector for a trap that is to be serviced in hyperprivileged mode. The least significant 14 bits of HTBA always read as zero, and writes to them are ignored.

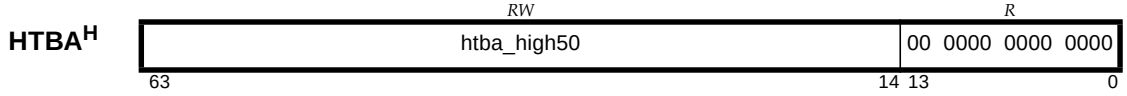



**FIGURE 5-40** Hyperprivileged Trap Base Address Register

Details on how the full address for a trap vector is generated, using HTBA and other state, are provided in *Trap-Table Entry Address to Hyperprivileged Mode* on page 480.

**IMPL. DEP. #406-S10:** It is implementation dependent whether all 50 bits of HTBA{63:14} are implemented or if only bits *n–1*:14 are implemented. If the latter, writes to bits 63:*n* are ignored and when HTBA is read, bits 63:*n* read as sign-extended copies of the most significant implemented bit, HTBA{*n – 1*}.

See Chapter 12, *Traps*, for more details on trap vectors.

## 5.8.5 Hyperprivileged Implementation Version ($HVER^H$) Register (HPR 6) D2

The Hyperprivileged Implementation Version register, shown in FIGURE 5-41, specifies the fixed parameters pertaining to a particular processor implementation and mask set. The HVER register is read-only, readable by the RDHPR instruction in hyperprivileged mode.

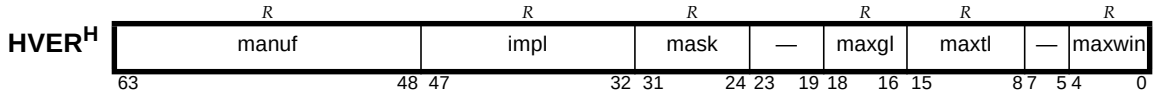



**FIGURE 5-41** Hyperprivileged Implementation Version Register

**IMPL. DEP. #104-V9:** HVER.manuf contains a 16-bit manufacturer code. This field is optional and if not present shall read as 0. HVER.manuf may indicate the original supplier of a second-sourced processor. It is intended that the contents of HVER.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 HVER.manuf.

**IMPL. DEP. #13-V8:** HVER.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.

HVER.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 1 for consecutive releases.

**Implementation Note** Conventionally, this field is die-specific, with bits 31:28 indicating the major mask revision number and bits 27:24 indicating the minor mask revision number.

HVER.maxgl contains the maximum number of levels of global register sets supported by an implementation (impl. dep. #401-S10), that is, *MAXGL*, the maximum value that the GL register may contain.

HVER.maxtl contains the maximum number of trap levels supported by an implementation (impl. dep. #101-V9-CS10), that is, *MAXTL*, the maximum value of the contents of the TL register.

HVER.maxwin contains the maximum index number available for use as a valid CWP value in an implementation; that is, HVER.maxwin contains the value *N_REG_WINDOWS* – 1 (impl. dep. #2-V8).

**SPARC V9 Compatibility Note** The SPARC V9 VER register was replaced in the UltraSPARC Architecture by the hyperprivileged HVER register.

## 5.8.6 Hyperprivileged System Tick Compare (HSTICK_CMPR$^{H}$) Register (HPR 31) D2

The Hyperprivileged System Tick Compare (HSTICK_CMPR) register allows hyperprivileged software to set up so that an *hstick_match* interrupt will occur when the STICK register reaches a specified value while HSTICK_CMPR.int_dis = 0. While executing in hyperprivileged mode and PSTATE.ie = 0, the interrupt is masked.

The Hyperprivileged System Tick Compare Register is illustrated in FIGURE 5-42.

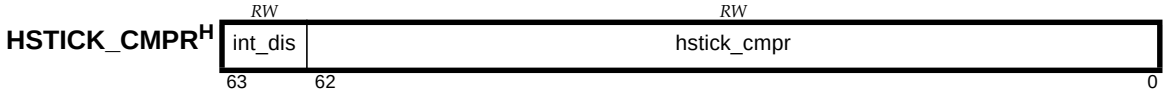



**FIGURE 5-42** HSTICK_CMPR Register

The fields of HSTICK_CMPR are described in TABLE 5-24.

**TABLE 5-24** Bit Description of HSTICK_CMPR Register

| Bit(s) | Field Name | Description |
|---|---|---|
| 63 | int_dis | If int_dis = 0, a match between HSTICK_CMPR.hstick_cmpr and STICK will cause hardware to set HINTP.hsp to 1. If int_dis = 1, this behavior is disabled; when a match occurs, HINTP.hsp will not be changed. |
| 62:0 | hstick_cmpr | Hyperprivileged System Tick Compare Field. When HSTICK_CMPR.int_dis = 0 and the value in HSTICK_CMPR.hstick_cmpr exactly matches the value in STICK.counter, HINTP.hsp is set to 1. After that, if HINTP.hsp remains set to 1, the next time that hyperprivileged interrupts are unmasked (HPSTATE.hpriv = 0 or PSTATE.ie = 1), an *hstick_match* exception will occur. |

**Programming Note** When int_dis = 1, an *hstick_match* interrupt can still occur if HINTP.hsp is set to 1 by software and the other prerequisite conditions for triggering *hstick_match* are met.

**Programming Note** HINTP.hsp must be set to 0 between the time an *hstick_match* trap occurs and the *hstick_match* trap handler returns. Otherwise, a return from the trap handler could immediately trigger another *hstick_match* trap. Refer to implementation-specific documentation regarding whether hardware sets HINTP.hsp to 0 when the *hstick_match* trap is taken or HINTP.hsp must be set to 0 by hyperprivileged software in the *hstick_match* trap handler.

After a power-on reset trap, the int_dis bit is set to 1 (disabling Hyperprivileged System Tick Compare interrupts), and the value of HSTICK_CMPR.hstick_cmpr is undefined.

CHAPTER 6

# Instruction Set Overview

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. Instructions are described in the following sections:


- **Instruction Execution** on page 109.
- **Instruction Formats** on page 110.
- **Instruction Categories** on page 111.


## 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 12, *Traps*, for a detailed description of exception and trap processing.

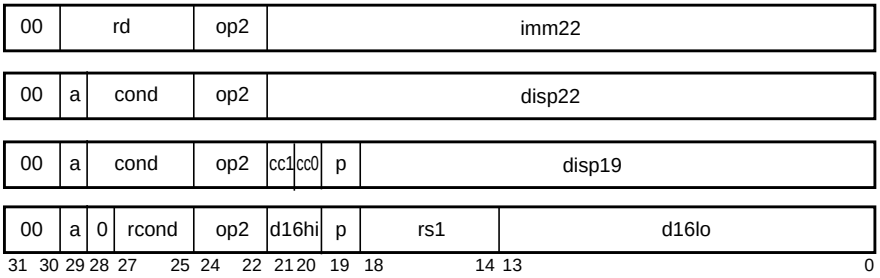
If a trap does not occur and the instruction is not a control transfer, the next program counter is copied into the PC, and the NPC is incremented by 4 (ignoring arithmetic overflow if any). There are two types of control-transfer instructions (CTIs): delayed and immediate. For a delayed CTI, at the end of the execution of the instruction, NPC is copied 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, an 8-bit address space identifier (ASI) is appended to the 64-bit memory address. Load/store alternate instructions (see *Address Space Identifiers (ASIs)* on page 117) can provide an arbitrary ASI with their data addresses or can use the ASI value currently contained in the ASI register.

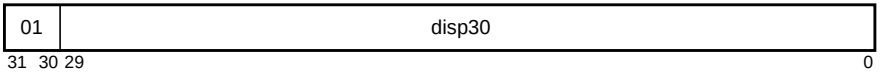
# 6.2 Instruction Formats

Every instruction is encoded in a single 32-bit word. Their most typical 32-bit formats formats are shown in FIGURE 6-1. For detailed formats for specific instructions, see individual instruction descriptions in the *Instructions* chapter.

*op = $00_2$: SETHI, Branches, and ILLTRAP*




*op = $01_2$: CALL*




*op = $10_2$ or $11_2$: Arithmetic, Logical, Moves, Tcc, Loads, Stores, Prefetch, and Misc*

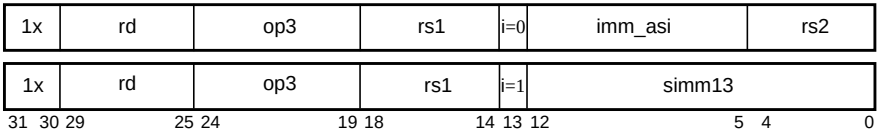



**FIGURE 6-1** Summary of Instruction Formats

## 6.3 Instruction Categories

UltraSPARC Architecture 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

These categories are 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 memory access 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, an ASI is appended 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. LDFSR/LDXFSR and STFSR/STXFSR are special load and store instructions that load or store the floating-point status register, FSR, instead of acting on an R or F register.

The destination field of a PREFETCH instruction (fcn) is used to encode the type of the prefetch.

Memory is byte (8-bit) addressable. Integer load and store instructions support byte, halfword (2 bytes), word (4 bytes), and doubleword/extended-word (8 bytes) accesses. Floating-point load and store instructions support word, doubleword, and quadword memory accesses. LDSTUB accesses bytes, SWAP accesses words, CASA accesses words, and CASXA accesses doublewords. The LDTXA (load twin-extended-word) instruction accesses a quadword (16 bytes) in memory. Block loads and stores access 64-byte aligned data. PREFETCH accesses at least 64 bytes.

**Programming Note** For some instructions, by use of simm13, any location in the lowest or highest 4 Kbytes of an address space can be accessed without the use of 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 twin word (LDTW, LDTWA, STTW, STTWA ) access must be aligned on an 8-byte boundary,an integer twin-extended-word (LDTXA) access must be aligned on a 16-byte boundary, and a Block Load (LDBLOCKF) or Store (STBLOCKF) 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 that is 4-byte aligned but not 8-byte aligned may result in less efficient and nonatomic access (causes a trap and is emulated in software (impl. dep. #109-V9-Cs10)), 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 that is 4-byte or 8-byte aligned but not 16-byte aligned may result in less efficient and nonatomic access (causes a trap and is emulated in software (impl. dep. #111-V9-Cs10)), 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 may cause an *LDDF_mem_address_not_aligned* exception (impl. dep. #109-V9-Cs10).
- An STDF or STDFA instruction accessing an address that is word aligned but not doubleword aligned may cause an *STDF_mem_address_not_aligned* exception (impl. dep. #110-V9-Cs10).

- An LDQF or LDQFA instruction accessing an address that is word aligned but not quadword aligned may cause an *LDQF_mem_address_not_aligned* exception (impl. dep. #111-V9-Cs10a).

  **Implementation Note** | Although the architecture provides for the *LDQF_mem_address_not_aligned* exception,UltraSPARC Architecture 2007 implementations do not currently generate it.

- An STQF or STQFA instruction accessing an address that is word aligned but not quadword aligned may cause an *STQF_mem_address_not_aligned* exception (impl. dep. #112-V9-Cs10a).

  **Implementation Note** | Although the architecture provides for the *STQF_mem_address_not_aligned* exception, UltraSPARC Architecture 2007 implementations do not currently generate it.

### 6.3.1.2 Addressing Conventions

An UltraSPARC Architecture 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 use of selected ASIs. It is also possible to change the default byte order for implicit data accesses. See *Processor State (PSTATE$^{P}$) Register (PR 6)* on page 91 for more information.[1]

**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 described in TABLE 6-1 and illustrated in FIGURE 6-2.

**TABLE 6-1** Big-endian Addressing Conventions

| Term | Definition |
|---|---|
| **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. |

1. Readers interested in 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.

**TABLE 6-1** Big-endian Addressing Conventions

| Term | Definition |
|---|---|
| **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.<br>For the deprecated integer load/store twin word instructions (LDTW, LDTWA†, STTW, STTWA), 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.<br>†Note that the LDTXA instruction, which is not an LDTWA operation but does share LDTWA's opcode, is ***not*** deprecated. |
| **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. |

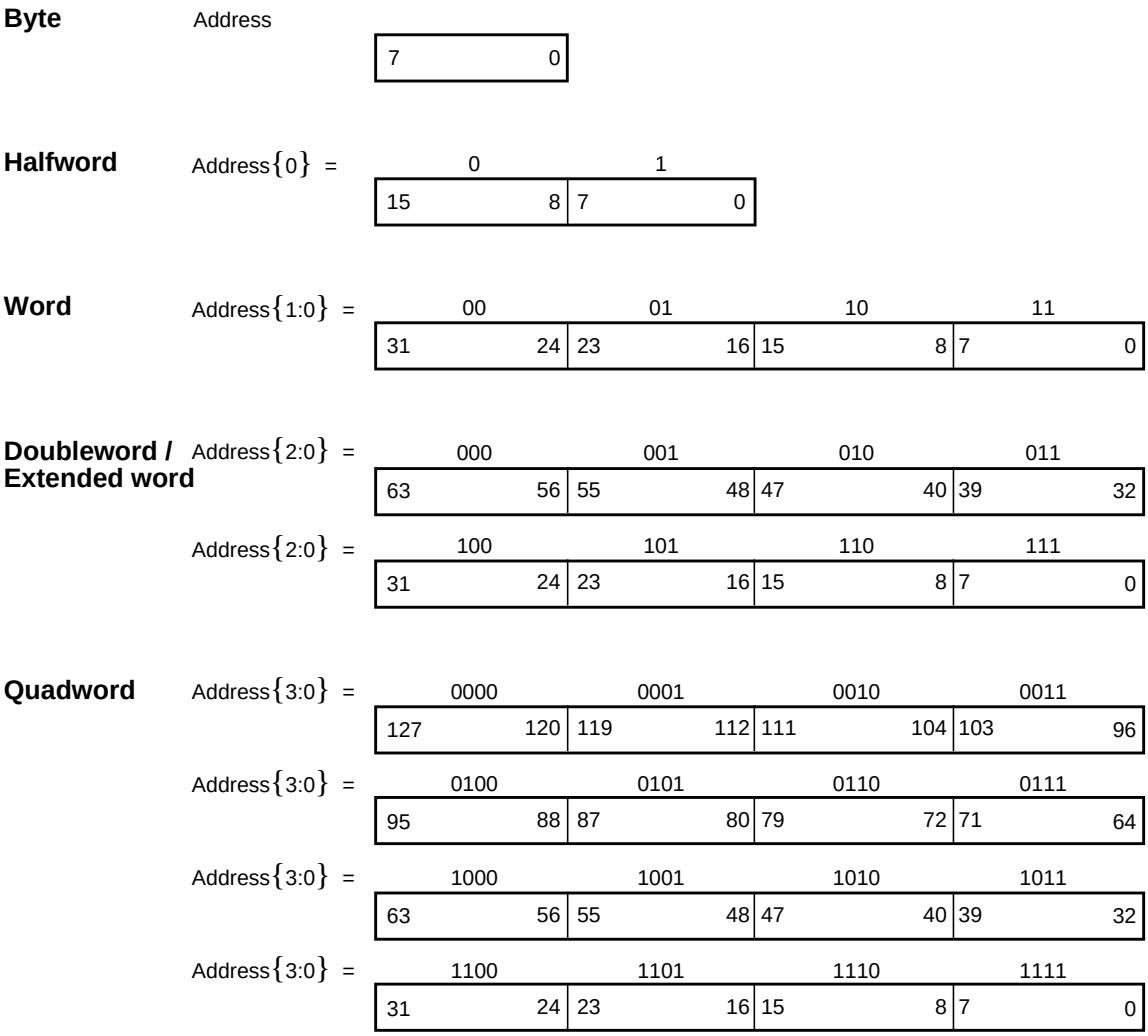



**FIGURE 6-2** Big-endian Addressing Conventions

**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 defined in TABLE 6-2 and illustrated in FIGURE 6-3.

**TABLE 6-2** Little-endian Addressing Convention

| Term | Definition |
|---|---|
| **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.<br>For the deprecated integer load/store twin word instructions (LDTW, LDTWA†, STTW, STTWA), 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 LDTW/LDTWA (STTW/STTWA) 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).<br>†Note that the LDTXA instruction, which is not an LDTWA operation but does share LDTWA's opcode, is ***not*** deprecated. |
| **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. |

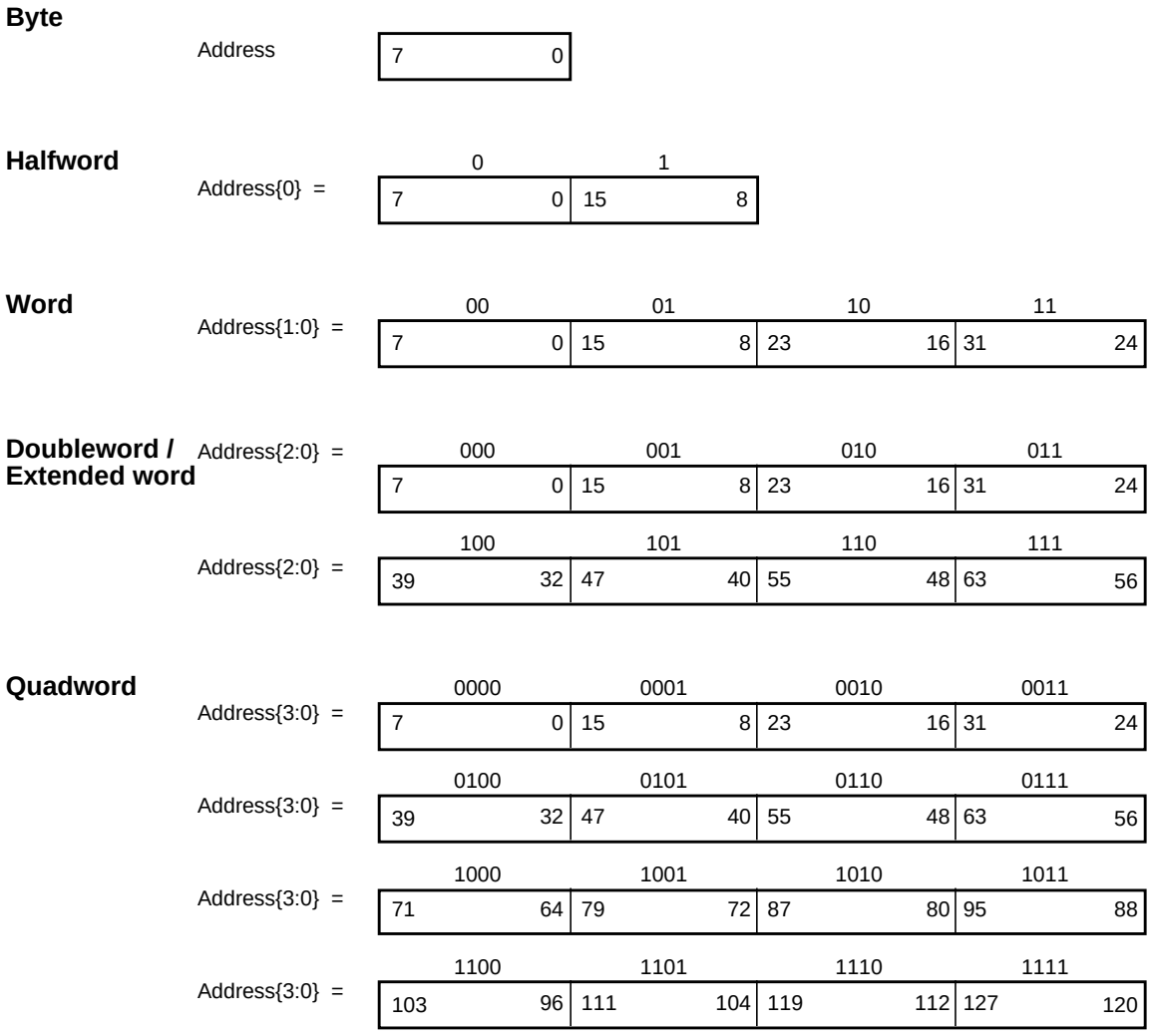



**FIGURE 6-3** Little-endian Addressing Conventions

### 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 6-3. Note that in hyperprivileged mode, all accesses are performed using physical addresses, so there is no implicit ASI in hyperprivileged mode (see *ASI Value, Context ID, and Endianness Selection for Translation* on page 555 for details).

**TABLE 6-3** ASIs Used for Data Accesses and Instruction Fetches in Nonprivileged and Privileged Modes

| 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 *Memory Addressing and Alternate Address Spaces* on page 411.

ASIs $00_{16}$-$7F_{16}$ are restricted; only software with sufficient privilege is allowed to access them. ASIs $00_{16}$-$2F_{16}$ are accessible by both privileged and hyperprivileged software, while ASIs $30_{16}$-$7F_{16}$ are accessible only by hyperprivileged software. An attempt to access a restricted ASI by insufficiently-privileged software results in a *privileged_action* exception (impl. dep #103-V9-Ms10(6)). ASIs $80_{16}$ through $FF_{16}$ are unrestricted; software is allowed to access them regardless of the virtual processor's privilege mode, as summarized in TABLE 6-4.

**TABLE 6-4** Allowed Accesses to ASIs

| Value | Access Type | Processor Mode (HPSTATE.hpriv, PSTATE.priv) | Result of ASI Access |
|---|---|---|---|
| $00_{16}$–$2F_{16}$ | Restricted (Privileged) | Nonprivileged (0,0) | *privileged_action* exception |
| | | Privileged (0,1) | Valid access |
| | | Hyperprivileged (1,x) | Valid access |
| $30_{16}$–$7F_{16}$ | Restricted (Hyperprivileged) | Nonprivileged (0,0) | *privileged_action* exception |
| | | Privileged (0,1) | *privileged_action* exception |
| | | Hyperprivileged (1,x) | Valid access |
| $80_{16}$–$FF_{16}$ | Unrestricted | Nonprivileged (0,0) | Valid access |
| | | Privileged (0,1) | Valid access |
| | | Hyperprivileged (1,x) | Valid access |

**IMPL. DEP. #29-V8:** Some UltraSPARC Architecture 2007 ASIs are implementation dependent. See TABLE 10-1 on page 431 for details.

**V9 Compatibility Note** | In SPARC V9, many ASIs were defined to be implementation dependent.

An UltraSPARC Architecture implementation decodes all 8 bits of ASI specifiers (impl. dep. #30-V8-Cu3).

**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.

**Programming Note** 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.

## 6.3.2 Memory Synchronization Instructions

Two forms of memory barrier (MEMBAR) instructions allow programs to manage the order and completion of memory references. Ordering MEMBARs induce a partial ordering between sets of loads and stores and future loads and stores. Sequencing MEMBARs 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.

## 6.3.3 Integer Arithmetic and Logical Instructions

The integer arithmetic and logical instructions generally compute a result that is a function of two source operands and either write the result in a third (destination) register R[rd] or discard it. The first source operand is R[rs1]. The second source operand depends on the i bit in the instruction; if i = 0, then the second operand is R[rs2]; if i = 1, then the second operand is the constant simm10, simm11, or simm13 from the instruction itself, sign-extended to 64 bits.

**Note** The value of R[0] always reads as zero, and writes to it are ignored.

### 6.3.3.1 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 702 for details.

### 6.3.3.2 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.

#### 6.3.3.3 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.

#### 6.3.3.4 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 processors, $32 \times 32 \rightarrow 64$-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.

#### 6.3.3.5 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) of these instructions are deprecated. See *Tagged Add* on page 368 and *Tagged Subtract* on page 374 for details.

### 6.3.4 Control-Transfer Instructions (CTIs)

The basic control-transfer instruction types are as follows:

- 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 NPC is changed, the effect of the transfer of control is delayed by one instruction. Most control transfers 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, based on the setting of an *annul bit* in the instruction. The effect of the annul bit depends upon whether the transfer is taken or not taken and whether the branch is conditional or unconditional. Annulled delay instructions neither affect 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. On an UltraSPARC Architecture implementation, 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 6-5 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 6-5** Control-Transfer Characteristics *(1 of 2)*

| 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 |
| BA | PC-relative | No | Yes | 1 | EA | EA + 4 |
| BN | PC-relative | Yes | No | 0 | NPC | NPC + 4 |

**TABLE 6-5** Control-Transfer Characteristics *(Continued) (2 of 2)*

| Instruction Group | Address Form | Delayed | Taken | Annul Bit | New PC | New NPC |
|---|---|---|---|---|---|---|
| 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 6-5, 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 2 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 or 8 bits of R[rs1] + R[rs2] if i = 0, or as the least significant 7 or 8 bits of R[rs1] + *imm_trap#* if i = 1. Whether 7 or 8 bits are used depends on the privilege level — 7 bits are used in nonprivileged mode and 8 bits are used in privileged and hyperprivileged modes. The trap level, TL, is incremented. The hardware trap type is computed as 256 + the software trap number and stored in TT[TL]. The effective address is generated by combining the contents of the TBA register with the trap type and other data; see *Trap Processing* on page 497 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].

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

### 6.3.4.1 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 annuling behavior of a taken conditional branch is different from that of an unconditional branch.

### 6.3.4.2 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.

### 6.3.4.3 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.

### 6.3.4.4 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.

### 6.3.4.5 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 stack.

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.

#### 6.3.4.6 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 in 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 a trap table pointed to by a trap base address register.

A Tcc instruction can specify one of 256 software trap types (128 when in nonprivileged mode). When a Tcc is taken, 256 plus the 7 (in nonprivileged mode) or 8 (in privileged or hyperprivileged mode) least significant bits of the Tcc's second source operand are 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 12, *Traps*, for more information.

**Programming Note** Tcc can be used to implement breakpointing, tracing, and calls to privileged or hyperprivileged software. Tcc can also be used for runtime checks, such as out-of-range array index checks or integer overflow checks.

#### 6.3.4.7 DCTI Couples (E2)

A delayed control transfer instruction (DCTI) in the delay slot of another DCTI is referred to as a "DCTI couple". The use of DCTI couples is deprecated in the UltraSPARC Architecture; no new software should place a DCTI in the delay slot of another DCTI, because on future UltraSPARC Architecture implementations DCTI couples may execute either slowly or differently than the programmer assumes it will.

**SPARC V8 and SPARC V9 Compatibility Note** The SPARC V8 architecture left behavior undefined for a DCTI couple. The SPARC V9 architecture defined behavior in that case, but as of UltraSPARC Architecture 2005, *use of DCTI couples was deprecated*.

### 6.3.5 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 can be any of the conditions allowed in conditional delayed control-transfer instructions. This condition is tested against one of the six 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 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
```

to eliminate 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 6-6.

**TABLE 6-6** 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 (treated as a signed value) 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.

## 6.3.6 Register Window Management Instructions

This subsection describes the instructions that manage register windows in the UltraSPARC Architecture. The privileged registers affected by these instructions are described in *Register-Window PR State Registers* on page 82.

### 6.3.6.1 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, (CLEANWIN – 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 operation 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).

### 6.3.6.2 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 execution of a CALL (or a JMPL) instruction. If the procedure requires a register window, it executes a SAVE instruction in its prologue code. 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. This optimization, called "Leaf-Procedure Optimization", is routinely performed by SPARC compilers.

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.

### 6.3.6.3 SAVED Instruction

SAVED is a privileged instruction used by a spill trap handler to indicate that a window spill has completed successfully. It increments CANSAVE and decrements either OTHERWIN or CANRESTORE, depending on the conditions at the time SAVED is executed.

See *SAVED* on page 321 for details.

### 6.3.6.4 RESTORED Instruction

RESTORED is a privileged instruction, used by a fill trap handler to indicate that a window has been filled successfully. It increments CANRESTORE and decrements either OTHERWIN or CANSAVE, depending on the conditions at the time RESTORED is executed. RESTORED also manipulates CLEANWIN, which is used to ensure that no address space's data become visible to another address space through windowed registers.

See *RESTORED* on page 313 for details.

#### 6.3.6.5 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:

*N_REG_WINDOWS* – 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.

### 6.3.7 Ancillary State Register (ASR) 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 Ancillary State register instruction is privileged only if the accessed register is privileged.

The supported RDasr and WRasr instructions are described in *Ancillary State Registers* on page 69.

### 6.3.8 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.

### 6.3.9 Floating-Point Operate (FPop) Instructions

Floating-point operate instructions (FPops) compute a result that is a function of one , two, or three source operands and place the result in one or more destination F registers, with one exception: floating-point compare operations do not write to an F register but instead update one of the fcc*n* fields of the FSR.

The term “FPop” refers to instructions in the FPop1, FMAf (IMPDEP2B), and FPop2 opcode spaces. FPop instructions do not include FBfcc instructions, loads and stores between memory and the F registers, or non-floating-point operations that read or write F registers.

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 125.

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 126.

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-V8:** 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. In this case, software running in a mode with greater privileges must emulate any functionality not present in the hardware.

See *ftt = 2 (unfinished_FPop)* on page 64 to see which instructions can produce an *fp_exception_other* exception (with FSR.ftt = unfinished_FPop).

## 6.3.10 Implementation-Dependent Instructions

The SPARC V9 architecture provided two instruction spaces that are entirely implementation dependent: IMPDEP1 and IMPDEP2.

In the UltraSPARC Architecture, the IMPDEP1 opcode space is used by VIS instructions.

In the UltraSPARC Architecture, IMPDEP2 is subdivided into IMPDEP2A and IMPDEP2B. IMPDEP2A remains implementation dependent. The IMPDEP2B opcode space is reserved for implementation of floating-point multiply-add/multiply-subtract instructions.

## 6.3.11 Reserved Opcodes and Instruction Fields

If a conforming UltraSPARC Architecture 2007 implementation attempts to execute an instruction bit pattern that is not specifically defined in this specification, it behaves as follows:

- If the instruction bit pattern 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, then the instruction bit pattern is invalid and causes an *illegal_instruction* exception.

See Appendix A, *Opcode Maps*, for an enumeration of the reserved instruction bit patterns (opcodes).

**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" ("—").

CHAPTER 7

# Instructions

*UltraSPARC Architecture 2007* extends the standard SPARC V9 instruction set with additional classes of instructions:

- Enhanced functionality:
  - Instructions for alignment (*Align Address* on page 146)
  - Array handling (*Three-Dimensional Array Addressing* on page 149)
  - Byte-permutation instructions (*Byte Mask and Shuffle* on page 155)
  - Edge handling (*Edge Handling Instructions* on pages 168 and 170)
  - Logical operations on floating-point registers (*F Register Logical Operate (1 operand)* on page 227)
  - Partitioned arithmetic (*Fixed-point Partitioned Add* on page 219*Fixed-point Partitioned Subtract (64-bit)* on page 224)
  - Pixel manipulation (*FEXPAND* on page 185, *FPACK* on page 213, and *FPMERGE* on page 222)
  - Access to hyperprivileged  state (such asRDHPR and WRHPR instructions)
- Efficient memory access
  - Partial store (*Store Partial Floating-Point* on page 351)
  - Short floating-point loads and stores (*Store Short Floating-Point* on page 355)
  - Block load and store (*Block Load* on page 249 and *Block Store* on page 338)
- Efficient interval arithmetic: SIAM (*Set Interval Arithmetic Mode* on page 327) and all instructions that reference GSR.im
- Floating-point Multiply-Add and Multiply-Subtract (FMA) instructions (*Floating-Point Multiply-Add and Multiply-Subtract (fused)* on page 192)

TABLE 7-2 provides a quick index of instructions, alphabetically by architectural instruction name.

TABLE 7-3 summarizes the instruction set, listed within functional categories.

Within these tables and throughout the rest of this chapter, and in Appendix A, *Opcode Maps*, certain opcodes are marked with mnemonic superscripts. The superscripts and their meanings are defined in TABLE 7-1.

**TABLE 7-1** Instruction Superscripts

| Superscript | Meaning |
|---|---|
| D | Deprecated instruction (do not use in new software) |
| H | Hyperprivileged instruction |
| N | Nonportable instruction |
| P | Privileged instruction |
| $P_{ASI}$ | Privileged action if bit 7 of the referenced ASI is 0 |
| $P_{ASR}$ | Privileged instruction if the referenced ASR register is privileged |
| $P_{npt}$ | Privileged action if in nonprivileged mode (PSTATE.priv = 0 **and** HPSTATE.hpriv = 0) and nonprivileged access is disabled ((S)TICK.npt = 1) |

**TABLE 7-2** UltraSPARC Architecture 2007 Instruction Set - Alphabetical *(1 of 3)*

| Page | Instruction | Page | Instruction | Page | Instruction |
|---|---|---|---|---|---|
| 145 | ADD (ADDcc) | 210 | FdMULq | | |
| 145 | ADDC (ADDCcc) | 185 | FEXPAND | 222 | FPMERGE |
| | | 186 | FiTO<s \| d \| q> | | |
| 146 | ALIGNADDRESS[_LITTLE] | | | 224 | FPSUB<16,32>[S] |
| 147 | ALLCLEAN | 187 | FLUSH | | |
| 148 | AND (ANDcc) | | | | |
| 149 | ARRAY<8 \| 16 \| 32> | 191 | FLUSHW | | |
| 153 | Bicc | 192 | FMADD(s,d) | 210 | FsMULd |
| 155 | BMASK | | | 231 | FSQRT<s \| d \| q> |
| 156 | BPcc | 195 | FMOV<s \| d \| q> | 228 | FSRC<1 \| 2>[s] |
| 159 | BPr | 196 | FMOV<s \| d \| q>cc | 236 | FSUB<s \| d \| q> |
| 155 | BSHUFFLE | 201 | FMOV<s \| d \| q>R | | |
| 161 | CALL | 192 | FMSUB(s,d) | | |
| 162 | CASA$^{P_{ASI}}$ | 210 | FMUL<s \| d \| q> | | |
| 162 | CASXA$^{P_{ASI}}$ | 204 | FMUL8[SU \| UL]x16 | 230 | FXNOR[s] |
| | | 204 | FMUL8x16 | 230 | FXOR[s] |
| | | 204 | FMUL8x16[AU \| AL] | 237 | FxTO<s \| d \| q> |
| | | 204 | FMULD8[SU \| UL]x16 | 227 | FZERO[s] |
| 165 | DONE$^{P}$ | | | 238 | ILLTRAP |
| 168 | EDGE<8 \| 16 \| 32>[L]cc | 230 | FNAND[s] | 239 | IMPDEP2A |
| 170 | EDGE<8 \| 16 \| 32>[L]N | 212 | FNEG<s \| d \| q> | 239 | IMPDEP2B |
| 234 | F<s \| d \| q>TO<s \| d \| q> | | | 241 | INVALW |
| 232 | F<s \| d \| q>TOi | 192 | FNMADD | 242 | JMPL |
| 232 | F<s \| d \| q>TOx | 192 | FNMSUB | | |
| 171 | FABS<s \| d \| q> | | | 249 | LDBLOCKF |
| 172 | FADD<s \| d \| q> | 230 | FNOR[s] | 253 | LDDF |
| 173 | FALIGNDATA | 228 | FNOT<1 \| 2>[s] | 256 | LDDFA$^{P_{ASI}}$ |
| 230 | FANDNOT<1 \| 2>[s] | | | 253 | LDF |
| 230 | FAND[s] | | | 256 | LDFA$^{P_{ASI}}$ |
| 174 | FBfcc$^{D}$ | | | 260 | LDFSR$^{D}$ |
| 176 | FBPfcc | 227 | FONE[s] | 253 | LDQF |
| | | 230 | FORNOT<1 \| 2>[s] | 256 | LDQFA$^{P_{ASI}}$ |
| 182 | FCMP<s \| d \| q> | 230 | FOR[s] | 244 | LDSB |
| 179 | FCMP*<16,32> | 213 | FPACK<16 \| 32 \| FIX> | 246 | LDSBA$^{P_{ASI}}$ |
| 182 | FCMPE<s \| d \| q> | | | 244 | LDSH |
| 184 | FDIV<s \| d \| q> | 219 | FPADD<16,32>[S] | 246 | LDSHA$^{P_{ASI}}$ |
| 262 | LDSHORTF | 305 | RDASI | 338 | STBLOCKF |

**TABLE 7-2** UltraSPARC Architecture 2007 Instruction Set - Alphabetical *(2 of 3)*

| Page | Instruction | | | | |
|---|---|---|---|---|---|
| 264 | LDSTUB | 305 | RDasr$^{P_{ASR}}$ | 342 | STDF |
| 265 | LDSTUBA$^{P_{ASI}}$ | 305 | RDCCR | 345 | STDFA$^{P_{ASI}}$ |
| 244 | LDSW | 305 | RDFPRS | 342 | STF |
| 246 | LDSWA$^{P_{ASI}}$ | 305 | RDGSR | 345 | STFA$^{P_{ASI}}$ |
| 272 | LDTXA$^{N}$ | 308 | RDHPR$^{H}$ | 349 | STFSR$^{D}$ |
| 267 | LDTW$^{D}$ | 305 | RDPC | 333 | STH |
| 269 | LDTWA$^{D, P_{ASI}}$ | | | 335 | STHA$^{P_{ASI}}$ |
| 264 | LDUB | | | 351 | STPARTIALF |
| 246 | LDUBA$^{P_{ASI}}$ | 309 | RDPR$^{P}$ | 342 | STQF |
| 244 | LDUH | 305 | RDSOFTINT$^{P}$ | 345 | STQFA$^{P_{ASI}}$ |
| 246 | LDUHA$^{P_{ASI}}$ | 305 | RDSTICK_CMPR$^{P}$ | 355 | STSHORTF |
| 244 | LDUW | 305 | RDSTICK$^{P_{npt}}$ | 357 | STTW$^{D}$ |
| 246 | LDUWA$^{P_{ASI}}$ | 305 | RDTICK_CMPR$^{P}$ | 359 | STTWA$^{D, P_{ASI}}$ |
| 244 | LDX | 305 | RDTICK$^{P_{npt}}$ | 333 | STW |
| 246 | LDXA$^{P_{ASI}}$ | 313 | RESTORED$^{P}$ | 335 | STWA$^{P_{ASI}}$ |
| | | 311 | RESTORE$^{P}$ | 333 | STX |
| 275 | LDXFSR | 315 | RETRY$^{P}$ | 335 | STXA$^{P_{ASI}}$ |
| | | 317 | RETURN | 362 | STXFSR |
| 277 | MEMBAR | 321 | SAVED$^{P}$ | 364 | SUB (SUBcc) |
| 281 | MOVcc | 319 | SAVE$^{P}$ | 364 | SUBC (SUBCcc) |
| | | 323 | SDIV$^{D}$ (SDIVcc$^{D}$) | 366 | SWAPA$^{D, P_{ASI}}$ |
| | | 289 | SDIVX | 365 | SWAP$^{D}$ |
| 285 | MOVr | 325 | SETHI | 368 | TADDcc |
| 287 | MULScc$^{D}$ | 326 | SHUTDOWN$^{D,P}$ | 369 | TADDccTV$^{D}$ |
| 289 | MULX | 327 | SIAM | 371 | Tcc |
| 290 | NOP | 328 | SIR$^{H}$ | 374 | TSUBcc |
| 291 | NORMALW | 329 | SLL | 375 | TSUBccTV$^{D}$ |
| 292 | OR (ORcc) | 329 | SLLX | 377 | UDIV$^{D}$ (UDIVcc$^{D}$) |
| 292 | ORN (ORNcc) | 331 | SMUL$^{D}$ (SMULcc$^{D}$) | 289 | UDIVX |
| 293 | OTHERW | 329 | SRA | 379 | UMUL$^{D}$ (UMULcc$^{D}$) |
| 294 | PDIST | 329 | SRAX | | |
| | | 329 | SRL | 381 | WRASI |
| 295 | POPC | 329 | SRLX | 381 | WRasr$^{P_{ASR}}$ |
| 297 | PREFETCH | 333 | STB | 381 | WRCCR |
| 297 | PREFETCHA$^{P_{ASI}}$ | 335 | STBA$^{P_{ASI}}$ | 381 | WRFPRS |
| | | | | 381 | WRGSR |
| 384 | WRHPR$^{H}$ | 381 | WRSOFTINT_SET$^{P}$ | 381 | WRY$^{D}$ |
| | | 381 | WRSOFTINT$^{P}$ | | |

**TABLE 7-2** UltraSPARC Architecture 2007 Instruction Set - Alphabetical *(3 of 3)*

| Page | Instruction | | | | |
|---|---|---|---|---|---|
| | | 381 | WRSTICK_CMPR[P] | 389 | XNOR (XNORcc) |
| 384 | WRPR[P] | 381 | WRSTICK[P] | 389 | XOR (XORcc) |
| 381 | WRSOFTINT_CLR[P] | 381 | WRTICK_CMPR[P] | | |

**TABLE 7-3** Instruction Set - by Functional Category *(1 of 6)*

| Instruction | Category and Function | Page | Ext. to V9? |
|---|---|---|---|
| | ***Data Movement Operations, Between R Registers*** | | |
| MOVcc | Move integer register if condition is satisfied | 281 | |
| MOVr | Move integer register on contents of integer register | 285 | |
| | ***Data Movement Operations, Between F Registers*** | | |
| FMOV<s\|d\|q> | Floating-point move | 195 | |
| FMOV<s\|d\|q>cc | Move floating-point register if condition is satisfied | 196 | |
| FMOV<s\|d\|q>R | Move f-p reg. if integer reg. contents satisfy condition | 201 | |
| FSRC<1\|2>[s] | Copy source | 228 | **VIS 1** |
| | ***Data Conversion Instructions*** | | |
| FiTO<s\|d\|q> | Convert 32-bit integer to floating-point | 186 | |
| F<s\|d\|q>TOi | Convert floating point to integer | 232 | |
| F<s\|d\|q>TOx | Convert floating point to 64-bit integer | 232 | |
| F<s\|d\|q>TO<s\|d\|q> | Convert between floating-point formats | 234 | |
| FxTO<s\|d\|q> | Convert 64-bit integer to floating-point | 237 | |
| | ***Logical Operations on R Registers*** | | |
| AND (ANDcc) | Logical **and** (and modify condition codes) | 148 | |
| OR (ORcc) | Inclusive-**or** (and modify condition codes) | 292 | |
| ORN (ORNcc) | Inclusive-**or not** (and modify condition codes) | 292 | |
| XNOR (XNORcc) | Exclusive-**nor** (and modify condition codes) | 389 | |
| XOR (XORcc) | Exclusive-**or** (and modify condition codes) | 389 | |
| | ***Logical Operations on F Registers*** | | |
| FAND[s] | Logical **and** operation | 230 | **VIS 1** |
| FANDNOT<1\|2>[s] | Logical **and** operation with one inverted source | 230 | **VIS 1** |
| FNAND[s] | Logical **nand** operation | 230 | **VIS 1** |
| FNOR[s] | Logical **nor** operation | 230 | **VIS 1** |
| FNOT<1\|2>[s] | Copy negated source | 228 | **VIS 1** |
| FONE[s] | One fill | 227 | **VIS 1** |
| FOR[s] | Logical **or** operation | 230 | **VIS 1** |
| FORNOT<1\|2>[s] | Logical **or** operation with one inverted source | 230 | **VIS 1** |
| FXNOR[s] | Logical **xnor** operation | 230 | **VIS 1** |
| FXOR[s] | Logical **xor** operation | 230 | **VIS 1** |
| FZERO[s] | Zero fill | 227 | **VIS 1** |
| | ***Shift Operations on R Registers*** | | |
| SLL | Shift left logical | 329 | |
| SLLX | Shift left logical, extended | 329 | |
| SRA | Shift right arithmetic | 329 | |
| SRAX | Shift right arithmetic, extended | 329 | |

**TABLE 7-3** Instruction Set - by Functional Category *(2 of 6)*

| Instruction | Category and Function | Page | Ext. to V9? |
|---|---|---|---|
| SRL | Shift right logical | 329 | |
| SRLX | Shift right logical, extended | 329 | |
| | ***Special Addressing Operations*** | | |
| ALIGNADDRESS[_LITTLE] | Calculate address for misaligned data | 146 | **VIS 1** |
| ARRAY<8\|16\|32> | 3-D array addressing instructions | 149 | **VIS 1** |
| FALIGNDATA | Perform data alignment for misaligned data | 173 | **VIS 1** |
| | ***Control Transfers*** | | |
| Bicc | Branch on integer condition codes | 153 | |
| BPcc | Branch on integer condition codes with prediction | 156 | |
| BPr | Branch on contents of integer register with prediction | 159 | |
| CALL | Call and link | 161 | |
| DONE$^{P}$ | Return from trap | 165 | |
| FBfcc$^{D}$ | Branch on floating-point condition codes | 174 | |
| FBPfcc | Branch on floating-point condition codes with prediction | 176 | |
| ILLTRAP | Illegal instruction | 238 | |
| JMPL | Jump and link | 242 | |
| RETRY$^{P}$ | Return from trap and retry | 315 | |
| RETURN | Return | 317 | |
| SIR$^{H}$ | Software-initiated reset | 328 | |
| Tcc | Trap on integer condition codes | 371 | |
| | ***Byte Permutation*** | | |
| BMASK | Set the GSR.mask field | 155 | **VIS 2** |
| BSHUFFLE | Permute bytes as specified by GSR.mask | 155 | **VIS 2** |
| | ***Data Formatting Operations on F Registers*** | | |
| FEXPAND | Pixel expansion | 185 | **VIS 1** |
| FPACK<16\|32\|FIX> | Pixel packing | 213 | **VIS 1** |
| FPMERGE | Pixel merge | 222 | **VIS 1** |
| | ***Memory Operations to/from F Registers*** | | |
| LDBLOCKF | Block loads | 249 | **VIS 1** |
| STBLOCKF | Block stores | 338 | **VIS 1** |
| LDDF | Load double floating-point | 253 | |
| LDDFA$^{P_{ASI}}$ | Load double floating-point from alternate space | 256 | |
| LDF | Load floating-point | 253 | |
| LDFA$^{P_{ASI}}$ | Load floating-point from alternate space | 256 | |
| LDQF | Load quad floating-point | 253 | |
| LDQFA$^{P_{ASI}}$ | Load quad floating-point from alternate space | 256 | |
| LDSHORTF | Short floating-point loads | 262 | **VIS 1** |

**TABLE 7-3** Instruction Set - by Functional Category *(3 of 6)*

| Instruction | Category and Function | Page | Ext. to V9? |
|---|---|---|---|
| STDF | Store double floating-point | 342 | |
| STDFA$^{P_{ASI}}$ | Store double floating-point into alternate space | 345 | |
| STF | Store floating-point | 342 | |
| STFA$^{P_{ASI}}$ | Store floating-point into alternate space | 345 | |
| STPARTIALF | Partial Store instructions | 351 | **VIS 1** |
| STQF | Store quad floating point | 342 | |
| STQFA$^{P_{ASI}}$ | Store quad floating-point into alternate space | 345 | |
| STSHORTF | Short floating-point stores | 355 | **VIS 1** |
| | ***Memory Operations — Miscellaneous*** | | |
| LDFSR$^{D}$ | Load floating-point state register (lower) | 260 | |
| LDXFSR | Load floating-point state register | 275 | |
| MEMBAR | Memory barrier | 277 | |
| PREFETCH | Prefetch data | 297 | |
| PREFETCHA$^{P_{ASI}}$ | Prefetch data from alternate space | 297 | |
| STFSR$^{D}$ | Store floating-point state register (lower) | 349 | |
| STXFSR | Store floating-point state register | 362 | |
| | ***Atomic (Load-Store) Memory Operations to/from R Registers*** | | |
| CASA$^{P_{ASI}}$ | Compare and swap word in alternate space | 162 | |
| CASXA$^{P_{ASI}}$ | Compare and swap doubleword in alternate space | 162 | |
| LDSTUB | Load-store unsigned byte | 264 | |
| LDSTUBA$^{P_{ASI}}$ | Load-store unsigned byte in alternate space | 265 | |
| SWAP$^{D}$ | Swap integer register with memory | 365 | |
| SWAPA$^{D, P_{ASI}}$ | Swap integer register with memory in alternate space | 366 | |
| | ***Memory Operations to/from R Registers*** | | |
| LDSB | Load signed byte | 244 | |
| LDSBA$^{P_{ASI}}$ | Load signed byte from alternate space | 246 | |
| LDSH | Load signed halfword | 244 | |
| LDSHA$^{P_{ASI}}$ | Load signed halfword from alternate space | 246 | |
| LDSW | Load signed word | 244 | |
| LDSWA$^{P_{ASI}}$ | Load signed word from alternate space | 246 | |
| LDTXA$^{N}$ | Load integer twin extended word from alternate space | 272 | **VIS 2+** |
| LDTW$^{D, P_{ASI}}$ | Load integer twin word | 267 | |
| LDTWA$^{D, P_{ASI}}$ | Load integer twin word from alternate space | 269 | |
| LDUB | Load unsigned byte | 264 | |
| LDUBA$^{P_{ASI}}$ | Load unsigned byte from alternate space | 246 | |
| LDUH | Load unsigned halfword | 244 | |
| LDUHA$^{P_{ASI}}$ | Load unsigned halfword from alternate space | 246 | |

**TABLE 7-3** Instruction Set - by Functional Category *(4 of 6)*

| Instruction | Category and Function | Page | Ext. to V9? |
|---|---|---|---|
| LDUW | Load unsigned word | 244 | |
| LDUWA$^{P_{ASI}}$ | Load unsigned word from alternate space | 246 | |
| LDX | Load extended | 244 | |
| LDXA$^{P_{ASI}}$ | Load extended from alternate space | 246 | |
| STB | Store byte | 333 | |
| STBA$^{P_{ASI}}$ | Store byte into alternate space | 335 | |
| STTW$^{D}$ | Store twin word | 357 | |
| STTWA$^{D, P_{ASI}}$ | Store twin word into alternate space | 359 | |
| STH | Store halfword | 333 | |
| STHA$^{P_{ASI}}$ | Store halfword into alternate space | 335 | |
| STW | Store word | 333 | |
| STWA$^{P_{ASI}}$ | Store word into alternate space | 335 | |
| STX | Store extended | 333 | |
| STXA$^{P_{ASI}}$ | Store extended into alternate space | 335 | |
| | ***Floating-Point Arithmetic Operations*** | | |
| FABS<s \| d \| q> | Floating-point absolute value | 171 | |
| FADD<s \| d \| q> | Floating-point add | 172 | |
| FDIV<s \| d \| q> | Floating-point divide | 184 | |
| FdMULq | Floating-point multiply double to quad | 210 | |
| FMADD(s,d) | Floating-point multiply-add single/double (fused) | 192 | |
| FMSUB(s,d) | Floating-point multiply-subtract single/double (fused) | 192 | |
| FMUL<s \| d \| q> | Floating-point multiply | 210 | |
| FNMADD(s,d) | Floating-point negative multiply-add single/double (fused) | 192 | |
| FNEG<s \| d \| q> | Floating-point negate | 212 | |
| FNMSUB(s,d) | Floating-point negative multiply-subtract single/double (fused) | 192 | |
| FsMULd | Floating-point multiply single to double | 210 | |
| FSQRT<s \| d \| q> | Floating-point square root | 231 | |
| FSUB<s \| d \| q> | Floating-point subtract | 236 | |
| | ***Floating-Point Comparison Operations*** | | |
| FCMP*<16,32> | Compare four 16-bit signed values or two 32-bit signed values | 179 | **VIS 1** |
| FCMP<s \| d \| q> | Floating-point compare | 182 | |
| FCMPE<s \| d \| q> | Floating-point compare (exception if unordered) | 182 | |
| | ***Register-Window Control Operations*** | | |
| ALLCLEAN | Mark all register window sets as “clean” | 147 | |
| INVALW | Mark all register window sets as “invalid” | 241 | |
| FLUSHW | Flush register windows | 191 | |

**TABLE 7-3** Instruction Set - by Functional Category *(5 of 6)*

| Instruction | Category and Function | Page | Ext. to V9? |
|---|---|---|---|
| NORMALW | "Other" register windows become "normal" register windows | 291 | |
| OTHERW | "Normal" register windows become "other" register windows | 293 | |
| RESTORE$^{P}$ | Restore caller's window | 311 | |
| RESTORED$^{P}$ | Window has been restored | 313 | |
| SAVE$^{P}$ | Save caller's window | 319 | |
| SAVED$^{P}$ | Window has been saved | 321 | |
| | ***Miscellaneous Operations*** | | |
| FLUSH | Flush instruction memory | 187 | |
| IMPDEP2A | Implementation-dependent instructions | 239 | |
| IMPDEP2B | Implementation-dependent instructions (reserved) | 239 | |
| NOP | No operation | 290 | |
| SHUTDOWN$^{D,P}$ | Shut down the virtual processor | 326 | **VIS 1** |
| | ***Integer SIMD Operations on F Registers*** | | |
| FPADD<16,32>[S] | Fixed-point partitioned add | 219 | **VIS 1** |
| FPSUB<16,32>[S] | Fixed-point partitioned subtract | 224 | **VIS 1** |
| | ***Integer Arithmetic Operations on R Registers*** | | |
| ADD (ADDcc) | Add (and modify condition codes) | 145 | |
| ADDC (ADDCcc) | Add with carry (and modify condition codes) | 145 | |
| MULScc$^{D}$ | Multiply step (and modify condition codes) | 287 | |
| MULX | Multiply 64-bit integers | 289 | |
| SDIV$^{D}$ (SDIVcc$^{D}$) | 32-bit signed integer divide (and modify condition codes) | 323 | |
| SDIVX | 64-bit signed integer divide | 289 | |
| SMUL$^{D}$ (SMULcc$^{D}$) | Signed integer multiply (and modify condition codes) | 331 | |
| SUB (SUBcc) | Subtract (and modify condition codes) | 364 | |
| SUBC (SUBCcc) | Subtract with carry (and modify condition codes) | 364 | |
| TADDcc | Tagged add and modify condition codes (trap on overflow) | 368 | |
| TADDccTV$^{D}$ | Tagged add and modify condition codes (trap on overflow) | 369 | |
| TSUBcc | Tagged subtract and modify condition codes (trap on overflow) | 374 | |
| TSUBccTV$^{D}$ | Tagged subtract and modify condition codes (trap on overflow) | 375 | |
| UDIV$^{D}$ (UDIVcc$^{D}$) | Unsigned integer divide (and modify condition codes) | 377 | |
| UDIVX | 64-bit unsigned integer divide | 289 | |
| UMUL$^{D}$ (UMULcc$^{D}$) | Unsigned integer multiply (and modify condition codes) | 379 | |
| | ***Integer Arithmetic Operations on F Registers*** | | |
| FMUL8x16 | 8x16 partitioned product | 204 | **VIS 1** |
| FMUL8x16[AU\|AL] | 8x16 upper/lower α partitioned product | 204 | **VIS 1** |
| FMUL8[SU\|UL]x16 | 8x16 upper/lower partitioned product | 204 | **VIS 1** |
| FMULD8[SU\|UL]x16 | 8x16 upper/lower partitioned product | 204 | **VIS 1** |

**TABLE 7-3** Instruction Set - by Functional Category *(6 of 6)*

| Instruction | Category and Function | Page | Ext. to V9? |
|---|---|---|---|
| | ***Miscellaneous Operations on R Registers*** | | |
| POPC | Population count | 295 | |
| SETHI | Set high 22 bits of low word of integer register | 325 | |
| | ***Miscellaneous Operations on F Registers*** | | |
| EDGE<8\|16\|32>[L]cc | Edge handling instructions (and modify condition codes) | 168 | **VIS 1** |
| EDGE<8\|16\|32>[L]N | Edge handling instructions | 170 | **VIS 2** |
| PDIST | Pixel component distance | 294 | **VIS 1** |
| | ***Control and Status Register Access*** | | |
| RDASI | Read ASI register | 305 | |
| RDasr$^{P_{ASR}}$ | Read ancillary state register | 305 | |
| RDCCR | Read Condition Codes register (CCR) | 305 | |
| RDFPRS | Read Floating-Point Registers State register (FPRS) | 305 | |
| RDGSR | Read General Status register (GSR) | 305 | |
| RDPC | Read Program Counter register (PC) | 305 | |
| RDHPR$^{H}$ | Read hyperprivileged register | 308 | |
| RDPR$^{P}$ | Read privileged register | 309 | |
| RDSOFTINT$^{P}$ | Read per-virtual processor Soft Interrupt register (SOFTINT) | 305 | |
| RDSTICK$^{P_{npt}}$ | Read System Tick register (STICK) | 305 | |
| RDSTICK_CMPR$^{P}$ | Read System Tick Compare register (STICK_CMPR) | 305 | |
| RDTICK$^{P_{npt}}$ | Read Tick register (TICK) | 305 | |
| RDTICK_CMPR$^{P}$ | Read Tick Compare register (TICK_CMPR) | 305 | |
| SIAM | Set interval arithmetic mode | 327 | **VIS 2** |
| WRASI | Write ASI register | 381 | |
| WRasr$^{P_{ASR}}$ | Write ancillary state register | 381 | |
| WRCCR | Write Condition Codes register (CCR) | 381 | |
| WRFPRS | Write Floating-Point Registers State register (FPRS) | 381 | |
| WRGSR | Write General Status register (GSR) | 381 | |
| WRHPR$^{H}$ | Write hyperprivileged register | 384 | |
| WRPR$^{P}$ | Write privileged register | 384 | |
| WRSOFTINT$^{P}$ | Write per-virtual processor Soft Interrupt register (SOFTINT) | 381 | |
| WRSOFTINT_CLR$^{P}$ | Clear bits of per-virtual processor Soft Interrupt register (SOFTINT) | 381 | |
| WRSOFTINT_SET$^{P}$ | Set bits of per-virtual processor Soft Interrupt register (SOFTINT) | 381 | |
| WRTICK_CMPR$^{P}$ | Write Tick Compare register (TICK_CMPR) | 381 | |
| WRSTICK$^{P}$ | Write System Tick register (STICK) | 381 | |
| WRSTICK_CMPR$^{P}$ | Write System Tick Compare register (STICK_CMPR) | 381 | |
| WRY$^{D}$ | Write Y register | 381 | |

In the remainder of this chapter, related instructions are grouped into subsections. Each subsection consists of the following sets of information:

**(1) Instruction Table.** This lists the instructions that are defined in the subsection, including the values of the field(s) that uniquely identify the instruction(s), assembly language syntax, and software and implementation classifications for the instructions. *(description of the Software Classes [letters] and Implementation Classes [digits] will be provided in a later update to this specification)*

**Note** | Instruction classes will be defined in a later draft of this document and in the meantime are subject to change.

**(2) Illustration of Instruction Format(s).** These illustrations show how the instruction is encoded in a 32-bit word in memory. In them, a dash (—) indicates that the field is *reserved* for future versions of the architecture and must be 0 in any instance of the instruction. If a conforming UltraSPARC Architecture implementation encounters nonzero values in these fields, its behavior is as defined in *Reserved Opcodes and Instruction Fields* on page 130.

**(3) Description.** This subsection describes the operation of the instruction, its features, restrictions, and exception-causing conditions.

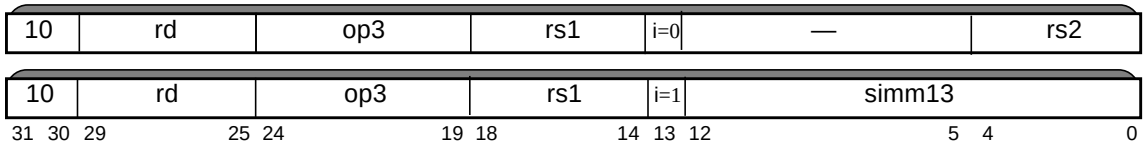
**(4) Exceptions.** The exceptions that can occur as a consequence of attempting to execute the instruction(s). Exceptions due to an *IAE_**, *fast_instruction_access_MMU_miss*, *instruction_access_error*, *fast_ECC_error*, *ECC_error* (corrected *ECC_error*), *WDR*, and interrupts are not listed because they can occur on any instruction. An instruction not implemented in hardware generates an *illegal_instruction* exception and therefore will not generate any of the other exceptions listed. Exceptions are listed in order of trap priority (see *Trap Priorities* on page 496), from highest to lowest priority.

**(5) See Also.** A list of related instructions (on selected pages).

**Note** | This specification does not contain any timing information (in either cycles or elapsed time), since timing is always implementation dependent.

# 7.1 Add

| Instruction | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| ADD | 00 0000 | Add | `add` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| ADDcc | 01 0000 | Add and modify cc's | `addcc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| ADDC | 00 1000 | Add with 32-bit Carry | `addc` | reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| ADDCcc | 01 1000 | Add with 32-bit Carry and modify cc's | `addccc` | reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |




*Description* If i = 0, ADD and ADDcc compute "R[rs1] + R[rs2]". If i = 1, they compute "R[rs1] + **sign_ext**( simm13)". In either case, the sum is written to R[rd].

ADDC and ADDCcc ("ADD with carry") also add the CCR register's 32-bit carry (icc.c) bit. That is, if i = 0, they compute "R[rs1] + R[rs2] + icc.c" and if i = 1, they compute "R[rs1] + **sign_ext**( simm13) + icc.c". In either case, the sum is written to 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 from that of the operands.

**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).

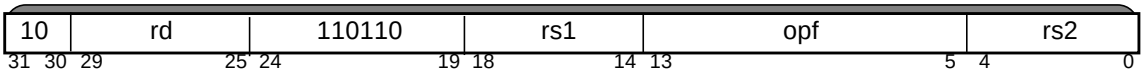
**SPARC V8 Compatibility Note** | ADDC and ADDCcc were previously named ADDX and ADDXcc, respectively, in SPARC V8.

An attempt to execute an ADD, ADDcc, ADDC or ADDCcc instruction when i = 0 and reserved instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

# 7.2 Align Address VIS 1

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| ALIGNADDRESS | 0 0001 1000 | Calculate address for misaligned data access | `alignaddr` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **A1** |
| ALIGNADDRESS_LITTLE | 0 0001 1010 | Calculate address for misaligned data access little-endian | `alignaddrl` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **A1** |




*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 two's complement of the least significant 3 bits of the result is stored in GSR.align.

**Note** | ALIGNADDRESS_LITTLE generates the opposite-endian byte ordering for a subsequent FALIGNDATA operation.

A byte-aligned 64-bit load can be performed as shown below.

```
alignaddr   Address, Offset, Address  !set GSR.align
ldd         [Address], %d0
ldd         [Address + 8], %d2
faligndata  %d0, %d2, %d4         !use GSR.align to select bytes
```

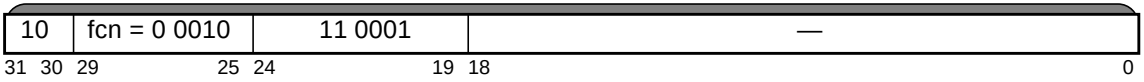
If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction causes an *fp_disabled* exception.

*Exceptions* *fp_disabled*

*See Also* Align Data on page 173

# 7.3 Mark All Register Window Sets “Clean”

| Instruction | Operation | Assembly Language Syntax | Class |
|---|---|---|---|
| ALLCLEAN[P] | Mark all register window sets as “clean” | `allclean` | **A1** |




*Description*

The ALLCLEAN instruction marks all register window sets as “clean”; specifically, it performs the following operation:

CLEANWIN ← (*N_REG_WINDOWS* – 1)

**Programming Note**

ALLCLEAN is used to indicate that all register windows are “clean”; that is, do not contain data belonging to other address spaces. It is needed because the value of *N_REG_WINDOWS* is not known to privileged software.

This instruction allows window manipulations to be atomic, without the value of *N_REG_WINDOWS* being visible to privileged software and without an assumption that *N_REG_WINDOWS* is constant (since hyperprivileged software can migrate a thread among virtual processors, across which *N_REG_WINDOWS* may vary).

An attempt to execute an ALLCLEAN instruction when reserved instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute an ALLCLEAN instruction in nonprivileged mode (PSTATE.priv = 0 and HSTATE.hpriv = 0) causes a *privileged_opcode* exception.

*Exceptions*

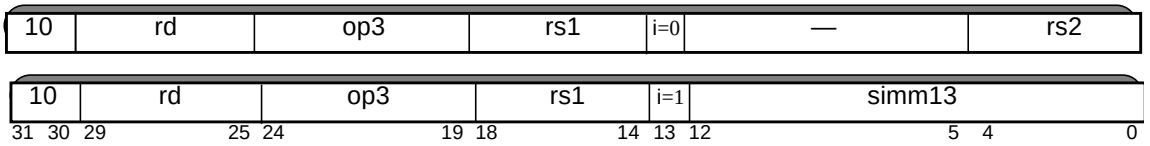
*illegal_instruction*
*privileged_opcode*

*See Also*

INVALW on page 241
NORMALW on page 291
OTHERW on page 293
RESTORED on page 313
SAVED on page 321

# 7.4 AND Logical Operation

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| AND | 00 0001 | **and** | `and` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| ANDcc | 01 0001 | **and** and modify cc's | `andcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| ANDN | 00 0101 | **and not** | `andn` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| ANDNcc | 01 0101 | **and not** and modify cc's | `andncc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |




*Description* These instructions implement bitwise logical **and** 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 and ANDNcc 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 and ANDNcc logically negate their second operand before applying the main (**and**) operation.

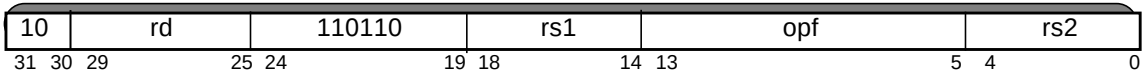
An attempt to execute an AND, ANDcc, ANDN or ANDNcc instruction when i = 0 and reserved instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

# 7.5 Three-Dimensional Array Addressing

**VIS 1**

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| ARRAY8 | 0 0001 0000 | Convert 8-bit 3D address to blocked byte address | `array8` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| ARRAY16 | 0 0001 0010 | Convert 16-bit 3D address to blocked byte address | `array16` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| ARRAY32 | 0 0001 0100 | Convert 32-bit 3D address to blocked byte address | `array32` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |




*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 R[rs2] and their meanings are shown in TABLE 7-4. Illegal values produce undefined results in the destination register, R[rd].

**TABLE 7-4** 3D R[rs2] Array X and Y Dimensions

| R[rs2] Value *(n)* | Number of Elements |
|---|---|
| 0 | 64 |
| 1 | 128 |
| 2 | 256 |
| 3 | 512 |
| 4 | 1024 |
| 5 | 2048 |

**Implementation Note** Architecturally, an illegal R[rs2] value (>5) causes the array instructions to produce undefined results. For historic reference, past implementations of these instructions have ignored R[rs2]{63:3} and have treated R[rs2] values of 6 and 7 as if they were 5.

The array instructions facilitate 3D texture mapping and volume rendering by computing a memory address for data lookup based on fixed-point x, y, and z coordinates. The data are laid out in a blocked fashion, so that points which are near one another have their data stored in nearby memory locations.

If the texture data were laid out in the obvious fashion (the z = 0 plane, followed by the z = 1 plane, etc.), then even small changes in z would result in references to distant pages in memory. The resulting lack of locality would tend to result in TLB misses and poor performance. The three versions of the array instruction, ARRAY8, ARRAY16, and ARRAY32, differ only in the scaling of the computed memory offsets. ARRAY16 shifts its result left by one position and ARRAY32 shifts left by two in order to handle 16- and 32-bit texture data.

When using the array instructions, a "blocked-byte" data formatting structure is imposed. The N × N × M volume, where N = $2^n \times 64$, M = $m \times 32$, $0 \le n \le 5$, $1 \le m \le 16$ should be composed of 64 × 64 × 32 smaller volumes, which in turn should be composed of 4 × 4 × 2 volumes. This data structure is optimal for 16-bit data. For 16-bit data, the 4 × 4 × 2 volume has 64 bytes of data, which is ideal for reducing cache-line misses; the 64 × 64 × 32 volume will have 256 Kbytes of data, which is good for improving the TLB hit rate. FIGURE 7-1 illustrates how the data has to be organized, where the origin (0,0,0) is assumed to be at the lower-left front corner and the x coordinate varies faster than y than z. That is, when traversing the volume from the origin to the upper right back, you go from left to right, front to back, bottom to top.

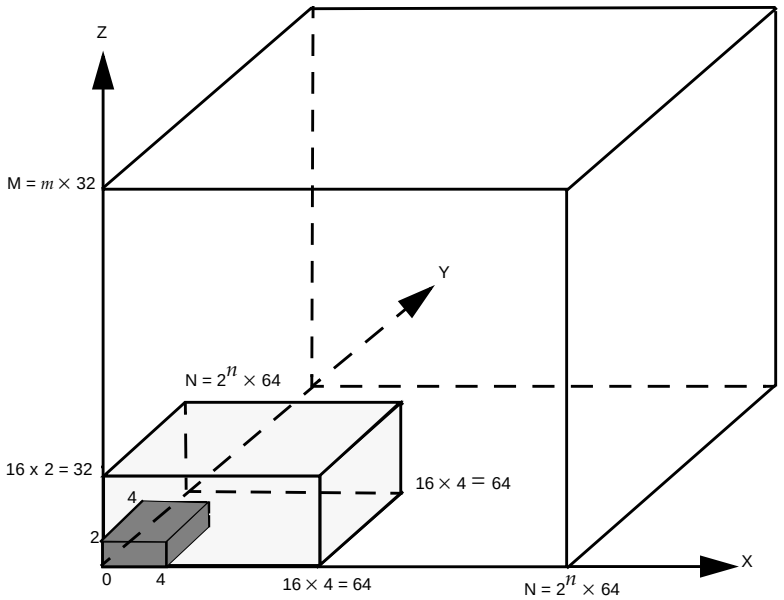



**FIGURE 7-1** Blocked-Byte Data Formatting Structure

The array instructions have 2 inputs:

The (x,y,z) coordinates are input via a single 64-bit integer organized in R[rs1] as shown in FIGURE 7-2.

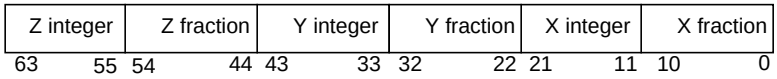



**FIGURE 7-2** Three-Dimensional Array Fixed-Point Address Format

Note that z has only 9 integer bits, as opposed to 11 for x and y. Also note that since (x,y,z) are all contained in one 64-bit register, they can be incremented or decremented simultaneously with a single add or subtract instruction (ADD or SUB).

So for a $512 \times 512 \times 32$ or a $512 \times 512 \times 256$ volume, the size value is 3. Note that the x and y size of the volume must be the same. The z size of the volume is a multiple of 32, ranging between 32 and 512.

The array instructions generate an integer memory offset, that when added to the base address of the volume, gives the address of the volume element (voxel) and can be used by a load instruction. The offset is correct only if the data has been reformatted as specified above.

The integer parts of x, y, and z are converted to the following blocked-address formats as shown in FIGURE 7-3 for ARRAY8, FIGURE 7-4 for ARRAY16, and FIGURE 7-5 for ARRAY32.

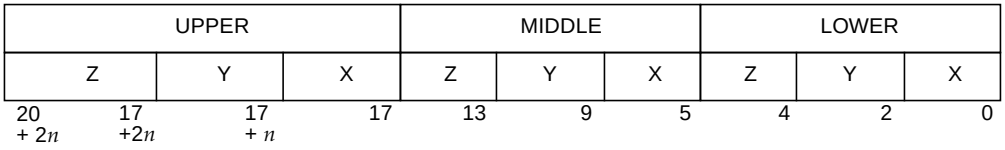



**FIGURE 7-3** Three-Dimensional Array Blocked-Address Format (ARRAY8)

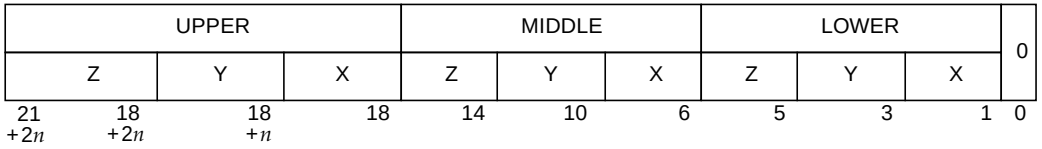



**FIGURE 7-4** Three-Dimensional Array Blocked-Address Format (ARRAY16)

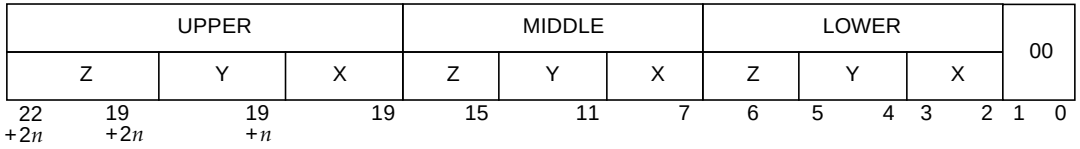



**FIGURE 7-5** Three Dimensional Array Blocked-Address Format (ARRAY32)

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.

**TABLE 7-5** ARRAY8 Description

| Result (R[rd]) Bits | Source (R[rs1] Bits | Field Information |
|---|---|---|
| 1:0 | 12:11 | X_integer{1:0} |
| 3:2 | 34:33 | Y_integer{1:0} |
| 4 | 55 | Z_integer{0} |
| 8:5 | 16:13 | X_integer{5:2} |
| 12:9 | 38:35 | Y_integer{5:2} |
| 16:13 | 59:56 | Z_integer{4:1} |
| 17+*n*-1:17 | 17+*n*-1:17 | X_integer{6+*n*-1:6} |
| 17+2*n*-1:17+*n* | 39+*n*-1:39 | Y_integer{6+*n*-1:6} |
| 20+2*n*:17+2*n* | 63:60 | Z_integer{8:5} |
| 63:20+2*n*+1 | n/a | 0 |

In the above description, if $n = 0$, there are 64 elements, so X_integer{6} and Y_integer{6} are not defined. That is, result{20:17} equals Z_integer{8:5}.

**Note** To maximize reuse of external cache and TLB data, software should block array references of a large image to the 64-Kbyte level. This means processing elements within a $32 \times 32 \times 64$ block.

The code fragment below shows assembly of components along an interpolated line at the rate of one component per clock.

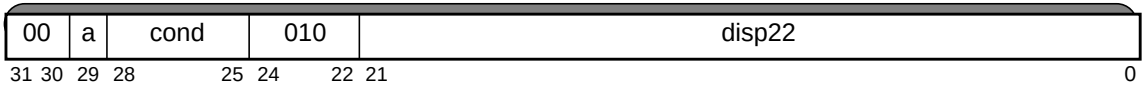
```
add             Addr, DeltaAddr, Addr
array8          Addr, %g0, bAddr
ldda            [bAddr] #ASI_FL8_PRIMARY, data
faligndata      data, accum, accum
```

*Exceptions* None

# 7.6 Branch on Integer Condition Codes (Bicc)

| Opcode | cond | Operation | *icc* Test | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|
| BA | 1000 | Branch Always | 1 | `ba{,a}` | *label* | **A1** |
| BN | 0000 | Branch Never | 0 | `bn{,a}` | *label* | **A1** |
| BNE | 1001 | Branch on Not Equal | **not** Z | `bne`[†]`{,a}` | *label* | **A1** |
| BE | 0001 | Branch on Equal | Z | `be`[‡]`{,a}` | *label* | **A1** |
| BG | 1010 | Branch on Greater | **not** (Z **or** (N **xor** V)) | `bg{,a}` | *label* | **A1** |
| BLE | 0010 | Branch on Less or Equal | Z **or** (N **xor** V) | `ble{,a}` | *label* | **A1** |
| BGE | 1011 | Branch on Greater or Equal | **not** (N **xor** V) | `bge{,a}` | *label* | **A1** |
| BL | 0011 | Branch on Less | N **xor** V | `bl{,a}` | *label* | **A1** |
| BGU | 1100 | Branch on Greater Unsigned | **not** (C **or** Z) | `bgu{,a}` | *label* | **A1** |
| BLEU | 0100 | Branch on Less or Equal Unsigned | C **or** Z | `bleu{,a}` | *label* | **A1** |
| BCC | 1101 | Branch on Carry Clear (Greater Than or Equal, Unsigned) | **not** C | `bcc`[◊]`{,a}` | *label* | **A1** |
| BCS | 0101 | Branch on Carry Set (Less Than, Unsigned) | C | `bcs`[▽]`{,a}` | *label* | **A1** |
| BPOS | 1110 | Branch on Positive | **not** N | `bpos{,a}` | *label* | **A1** |
| BNEG | 0110 | Branch on Negative | N | `bneg{,a}` | *label* | **A1** |
| BVC | 1111 | Branch on Overflow Clear | **not** V | `bvc{,a}` | *label* | **A1** |
| BVS | 0111 | Branch on Overflow Set | V | `bvs{,a}` | *label* | **A1** |

† *synonym:* `bnz` ‡ *synonym:* `bz` ◊ *synonym:* `bgeu` ▽ *synonym:* `blu`




**Programming Note** To set the annul (`a`) 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.

Unconditional branches and `icc`-conditional branches are described below:

- **Unconditional branches (**BA, BN) — If its annul bit is 0 (`a` = 0), a BN (Branch Never) instruction is treated as a NOP. If its annul bit is 1 (`a` = 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 (a) bit of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul bit is 0 (a = 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 annul bit is 1 (a = 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, *Instruction Set Overview*.

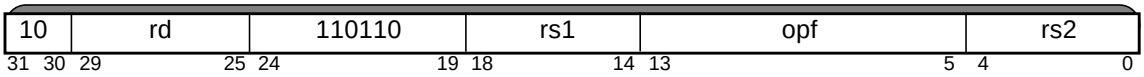
If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1 and the Bicc instruction will cause a transfer of control (BA or taken conditional branch), then Bicc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the Bicc instruction) is stored in TPC[TL] and the value of NPC from before the Bicc was executed is stored in TNPC[TL].

Note that BN never causes a *control_transfer_instruction* exception.

*Exceptions* *control_transfer_instruction* (impl. dep. #450-S20)

# 7.7 Byte Mask and Shuffle VIS 2

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| BMASK | 0 0001 1001 | Set the GSR.mask field in preparation for a subsequent BSHUFFLE instruction | `bmask` *reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$ | **B1** |
| BSHUFFLE | 0 0100 1100 | Permute 16 bytes as specified by GSR.mask | `bshuffle` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |




*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 $F_D$[rs1] (more significant half) and $F_D$[rs2] (less significant half) to form a 128-bit (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 $F_D$[rd]. Bytes in $F_D$[rd] 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 to generate each byte in the destination register, $F_D$[rd].

| Destination Byte (in F[rd]) | Source Byte |
|---|---|
| 0 (most significant) | ($F_D$[rs1] :: $F_D$[[rs2]){GSR.mask{31:28}} |
| 1 | ($F_D$[[rs1] :: $F_D$[[rs2]){GSR.mask{27:24}} |
| 2 | ($F_D$[[rs1] :: $F_D$[[rs2]){GSR.mask{23:20}} |
| 3 | ($F_D$[[rs1] :: $F_D$[[rs2]){GSR.mask{19:16}} |
| 4 | ($F_D$[[rs1] :: $F_D$[[rs2]){GSR.mask{15:12}} |
| 5 | ($F_D$[[rs1] :: $F_D$[[rs2]){GSR.mask{11:8}} |
| 6 | ($F_D$[[rs1] :: $F_D$[[rs2]){GSR.mask{7:4}} |
| 7 (least significant) | ($F_D$[[rs1] :: $F_D$[[rs2]){GSR.mask{3:0}} |

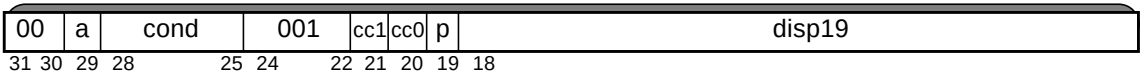
If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a BMASK or BSHUFFLE instruction causes an *fp_disabled* exception.

*Exceptions* *fp_disabled*

# 7.8 Branch on Integer Condition Codes with Prediction (BPcc)

| Instruction | cond | Operation | cc Test | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| BPA | 1000 | Branch Always | 1 | `ba{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPN | 0000 | Branch Never | 0 | `bn{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPNE | 1001 | Branch on Not Equal | **not** Z | `bne†{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPE | 0001 | Branch on Equal | Z | `be‡{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPG | 1010 | Branch on Greater | **not** (Z **or** (N **xor** V)) | `bg{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPLE | 0010 | Branch on Less or Equal | Z **or** (N **xor** V) | `ble{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPGE | 1011 | Branch on Greater or Equal | **not** (N **xor** V) | `bge{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPL | 0011 | Branch on Less | N **xor** V | `bl{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPGU | 1100 | Branch on Greater Unsigned | **not** (C **or** Z) | `bgu{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPLEU | 0100 | Branch on Less or Equal Unsigned | C **or** Z | `bleu{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPCC | 1101 | Branch on Carry Clear (Greater than or Equal, Unsigned) | **not** C | `bcc◊{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPCS | 0101 | Branch on Carry Set (Less than, Unsigned) | C | `bcs∇{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPPOS | 1110 | Branch on Positive | **not** N | `bpos{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPNEG | 0110 | Branch on Negative | N | `bneg{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPVC | 1111 | Branch on Overflow Clear | **not** V | `bvc{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |
| BPVS | 0111 | Branch on Overflow Set | V | `bvs{,a}{,pt\|,pn}` *i_or_x_cc, label* | **A1** |

† *synonym:* `bnz` ‡ *synonym:* `bz` ◊ *synonym:* `bgeu` ∇ *synonym:* `blu`




| cc1 | cc0 | Condition Code |
|---|---|---|
| 0 | 0 | icc |
| 0 | 1 | — |
| 1 | 0 | xcc |
| 1 | 1 | — |

# BPcc

**Programming Note** To set the annul (a) 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 defaults 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) may be used in the SPARC V9 architecture 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 bit is 1 (a = 1), then the following (delay) instruction is annulled (not executed). If the annul bit is 0 (a = 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 bit of the branch instruction is 1 (a = 1), then the delay instruction is annulled (not executed). If the annul bit is 0 (a = 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 (a) bit. If a conditional branch is not taken and the annul bit is 1 (a = 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) 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, *Instruction Set Overview*.

An attempt to execute a BPcc instruction with cc0 = 1 (a reserved value) causes an *illegal_instruction* exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1 and the BPcc instruction will cause a transfer of control (BPA or taken conditional branch), then BPcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the BPcc) is stored in TPC[TL] and the value of NPC from before the BPcc was executed is stored in TNPC[TL].

Note that BPN never causes a *control_transfer_instruction* exception.

*Exceptions*

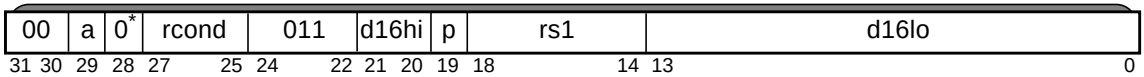
*illegal_instruction*
*control_transfer_instruction* (impl. dep. #450-S20)

*See Also*

Branch on Integer Register with Prediction (BPr) on page 159

# 7.9 Branch on Integer Register with Prediction (BPr)

| Instruction | rcond | Operation | Register Contents Test | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| — | 000 | *Reserved* | — | | — |
| BRZ | 001 | Branch on Register Zero | R[rs1] = 0 | `brz{,a}{,pt|,pn}` $reg_{rs1}$, *label* | **A1** |
| BRLEZ | 010 | Branch on Register Less Than or Equal to Zero | R[rs1] ≤ 0 | `brlez{,a}{,pt|,pn}` $reg_{rs1}$, *label* | **A1** |
| BRLZ | 011 | Branch on Register Less Than Zero | R[rs1] < 0 | `brlz{,a}{,pt|,pn}` $reg_{rs1}$, *label* | **A1** |
| — | 100 | *Reserved* | — | | — |
| BRNZ | 101 | Branch on Register Not Zero | R[rs1] ≠ 0 | `brnz{,a}{,pt|,pn}` $reg_{rs1}$, *label* | **A1** |
| BRGZ | 110 | Branch on Register Greater Than Zero | R[rs1] > 0 | `brgz{,a}{,pt|,pn}` $reg_{rs1}$, *label* | **A1** |
| BRGEZ | 111 | Branch on Register Greater Than or Equal to Zero | R[rs1] ≥ 0 | `brgez{,a}{,pt|,pn}` $reg_{rs1}$, *label* | **A1** |




* Although SPARC V9 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.

**Programming Note** To set the annul (a) 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 defaults 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 (a) bit. If the branch is not taken and the annul bit is 1 (a = 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. If p = 1, the branch is expected to be taken; p = 0 indicates that the branch is expected not to be taken.

An attempt to execute a BPr instruction when instruction bit 28 = 1 or rcond is a reserved value ($000_2$ or $100_2$) causes an *illegal_instruction* exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1 and the BPr instruction will cause a transfer of control (taken conditional branch), then BPr generates a *control_transfer_instruction* exception instead of causing a control transfer.

Annulment, delay instructions, prediction, and delayed control transfers are described further in Chapter 6, *Instruction Set Overview*.

**Implementation Note** If this instruction is implemented by tagging each register value with an N (negative) bit and Z (zero) bit, the table below can be used 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*

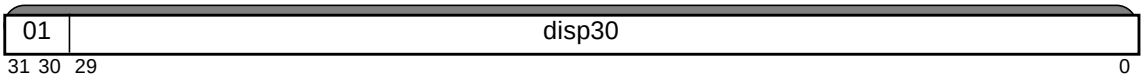
*illegal_instruction*
*control_transfer_instruction* (impl. dep. #450-S20)

*See Also*

Branch on Integer Condition Codes with Prediction (BPcc) on page 156

# 7.10 Call and Link

| Instruction | op | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| CALL | 01 | Call and Link | `call` | *label* | **A1** |




*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).

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system and in the address written into R[15]. (closed impl. dep. #125-V9-Cs10)

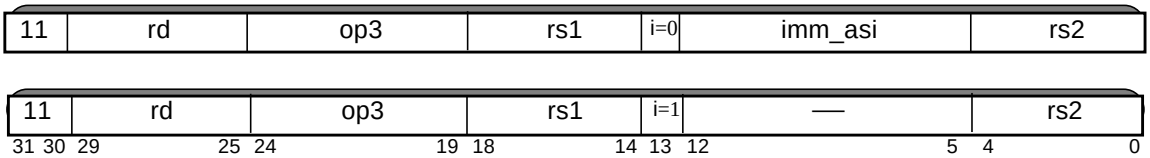
If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then CALL generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the CALL instruction) is stored in TPC[TL] and the value of NPC from before the CALL was executed is stored in TNPC[TL]. The full 64-bit (nonmasked) PC and NPC values are stored in TPC[TL] and TNPC[TL], regardless of the value of PSTATE.am.

*Exceptions* *control_transfer_instruction* (impl. dep. #450-S20)

*See Also* JMPL on page 242

# 7.11 Compare and Swap

| Instruction | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| CASA$^{P_{ASI}}$ | 11 1100 | Compare and Swap Word from Alternate Space | `casa` [$reg_{rs1}$] *imm_asi*, $reg_{rs2}$, $reg_{rd}$<br>`casa` [$reg_{rs1}$] `%asi`, $reg_{rs2}$, $reg_{rd}$ | | **A1** |
| CASXA$^{P_{ASI}}$ | 11 1110 | Compare and Swap Extended from Alternate Space | `casxa` [$reg_{rs1}$] *imm_asi*, $reg_{rs2}$, $reg_{rd}$<br>`casxa` [$reg_{rs1}$] `%asi`, $reg_{rs2}$, $reg_{rd}$ | | **A1** |




*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 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.

An attempt to execute a CASXA or CASA instruction when i = 1 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

A *mem_address_not_aligned* exception is generated if the address in R[rs1] is not properly aligned.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, CASXA and CASA cause a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, CASXA and CASA cause a *privileged_action* exception.

**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 536 for the syntax of these synthetic instructions.

The compare-and-swap instructions do not affect the condition codes.

The compare-and-swap instructions can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

**ASIs valid for CASA and CASXA instructions**

| | |
|---|---|
| ASI_AS_IF_PRIV_PRIMARY | ASI_AS_IF_PRIV_PRIMARY_LITTLE |
| ASI_AS_IF_PRIV_SECONDARY | ASI_AS_IF_PRIV_SECONDARY_LITTLE |
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |

**ASIs valid for CASA and CASXA instructions**

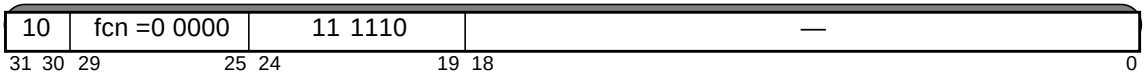
| | |
|---|---|
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |

*Exceptions*

*illegal_instruction*
*mem_address_not_aligned*
*privileged_action*
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nc_page* (attempted access to noncacheable page)
*DAE_nfo_page* (attempted access to non-faulting-only page)
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

# 7.12 DONE

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| DONE[P] | 11 1110 | Return from Trap (skip trapped instruction) | done | **A1** |




*Description* The DONE instruction restores the saved state from TSTATE[TL] (GL, CCR, ASI, PSTATE, and CWP), HTSTATE[TL] (HPSTATE), sets PC and NPC, and decrements TL. DONE sets PC←TNPC[TL] and NPC←TNPC[TL]+4 (normally, the value of NPC saved at the time of the original trap and address of the instruction immediately after the one referenced by the NPC).

**Programming Notes**

The DONE and RETRY instructions are used to return from privileged trap handlers.

Unlike RETRY, DONE ignores the contents of TPC[TL].

If the saved TNPC[TL] was not altered by trap handler software, DONE causes execution to resume immediately *after* the instruction that originally caused the trap (as if that instruction was "done" executing).

Execution of a DONE instruction in the delay slot of a control-transfer instruction produces undefined results.

When a DONE instruction is executed and HTSTATE[TL].hpstate.hpriv = 0 (which will cause the DONE to return the virtual processor to nonprivileged or privileged mode), the value of GL restored from TSTATE[TL] saturates at *MAXPGL*. That is, if the value in TSTATE[TL].gl is greater than *MAXPGL*, then *MAXPGL* is substituted and written to GL. This protects against non-hyperprivileged software executing with GL > *MAXPGL*.

If software writes invalid or inconsistent state to TSTATE or HTSTATE before executing DONE, virtual processor behavior during and after execution of the DONE instruction is undefined.

The DONE instruction does not provide an error barrier, as MEMBAR #Sync does (impl. dep. #215-U3).

Note that since PSTATE.tct is automatically set to 0 during entry to a trap handler, execution of a DONE instruction at the end of a trap handler will not cause a *control_transfer_instruction* exception unless trap handler software has explicitly set PSTATE.tct to 1. During execution of the DONE instruction, the value of PSTATE.tct is restored from TSTATE.

**Programming Notes**

If *control_transfer_instruction* traps are to be re-enabled (PSTATE.tct ← 1, restored from TSTATE[TL].pstate.tct) when trap handler software for the *control_transfer_instruction* trap returns, the trap handler must
(1) emulate the trapped CTI, setting TPC[TL] and TNPC[TL] appropriately, remembering to compensate for annul bits) and
(2) use a DONE (not RETRY) instruction to return.

If the CTI that caused the *control_transfer_instruction* trap was a DONE (RETRY) instruction, the trap handler must carefully emulate the trapped DONE (RETRY) (decrementing TL may suffice) before the trap handler returns using its own DONE (RETRY) instruction.

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system.

**IMPL. DEP. #417-S10**: If *(1)* TSTATE[TL].pstate.am = 1 and *(2)* a DONE instruction is executed (which sets PSTATE.am to '1' by restoring the value from TSTATE[TL].pstate.am to PSTATE.am), it is implementation dependent whether the DONE instruction masks (zeroes) the more-significant 32 bits of the values it places into PC and NPC.

**Exceptions.** In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0) or hyperprivileged mode (HPSTATE.hpriv = 1), an attempt to execute DONE while TL = 0 causes an *illegal_instruction* exception. An attempt to execute DONE (in any mode) with instruction bits 18:0 nonzero causes an *illegal_instruction* exception.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), an attempt to execute DONE causes a *privileged_opcode* exception.

**Implementation Note**

In nonprivileged mode, *illegal_instruction* exception due to TL = 0 does not occur. The *privileged_opcode* exception occurs instead, regardless of the current trap level (TL).

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then DONE generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the DONE instruction) is stored in TPC[TL] and the value of NPC from before the DONE was executed is stored in TNPC[TL]. The full 64-bit (nonmasked) PC and NPC values are stored in TPC[TL] and TNPC[TL], regardless of the value of PSTATE.am.

# DONE

A *trap_level_zero* disrupting trap can occur upon the *completion* of a DONE instruction, if the following three conditions are true after DONE has executed:

- *trap_level_zero* exceptions are enabled (HPSTATE.tlz = 1),
- the virtual processor is in nonprivileged or privileged mode (HPSTATE.hpriv = 0), and
- the trap level (TL) register's value is zero (TL = 0)

*Exceptions*

*illegal_instruction*
*privileged_opcode*
*control_transfer_instruction* (impl. dep. #450-S20)
*trap_level_zero*

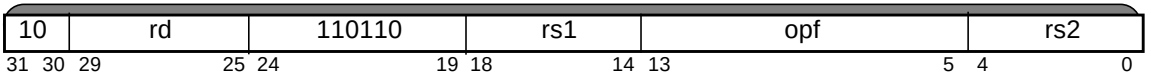
*See Also*

RETRY on page 315

# 7.13 Edge Handling Instructions VIS 1

| Instruction | opf | Operation | Assembly Language Syntax † | | Class |
|---|---|---|---|---|---|
| EDGE8cc | 0 0000 0000 | Eight 8-bit edge boundary processing | `edge8cc` | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE8Lcc | 0 0000 0010 | Eight 8-bit edge boundary processing, little-endian | `edge8lcc` | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE16cc | 0 0000 0100 | Four 16-bit edge boundary processing | `edge16cc` | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE16Lcc | 0 0000 0110 | Four 16-bit edge boundary processing, little-endian | `edge16lcc` | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE32cc | 0 0000 1000 | Two 32-bit edge boundary processing | `edge32cc` | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE32Lcc | 0 0000 1010 | Two 32-bit edge boundary processing, little-endian | `edge32lcc` | $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |

† The original assembly language mnemonics for these instructions did not include the "`cc`" suffix, as appears in the names of all other instructions that set the integer condition codes. The old, non-"`cc`" mnemonics are deprecated. Over time, assemblers will support the new mnemonics for these instructions. In the meantime, some older assemblers may recognize only the mnemonics, without "`cc`".




*Description* These instructions handle the boundary conditions for parallel pixel scan line loops, where R[rs1] is the address of the next pixel to render and R[rs2] is the address of the last pixel in the scan line.

EDGE8Lcc, EDGE16Lcc, and EDGE32Lcc are little-endian versions of EDGE8cc, EDGE16cc, and EDGE32cc. 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 Partial Store instruction (see *Partial Store* on page 298) on little-endian data.

A 2-bit (EDGE32cc), 4-bit (EDGE16cc), or 8-bit (EDGE8cc) 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 of R[rs1] and the right edge mask is computed from the 3 least significant bits of R[rs2], according to TABLE 7-6.

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 **and**ed 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 **and**ed 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 *Subtract* on page 303).

TABLE 7-6 lists edge mask specifications.

**TABLE 7-6** Edge Mask Specification

| Edge Size | R[rs*n*] {2:0} | Big Endian | | Little Endian | |
|---|---|---|---|---|---|
| | | Left Edge | Right Edge | Left Edge | Right Edge |
| 8 | 000 | 1111 1111 | 1000 0000 | 1111 1111 | 0000 0001 |
| 8 | 001 | 0111 1111 | 1100 0000 | 1111 1110 | 0000 0011 |
| 8 | 010 | 0011 1111 | 1110 0000 | 1111 1100 | 0000 0111 |
| 8 | 011 | 0001 1111 | 1111 0000 | 1111 1000 | 0000 1111 |
| 8 | 100 | 0000 1111 | 1111 1000 | 1111 0000 | 0001 1111 |
| 8 | 101 | 0000 0111 | 1111 1100 | 1110 0000 | 0011 1111 |
| 8 | 110 | 0000 0011 | 1111 1110 | 1100 0000 | 0111 1111 |
| 8 | 111 | 0000 0001 | 1111 1111 | 1000 0000 | 1111 1111 |
| 16 | 00x | 1111 | 1000 | 1111 | 0001 |
| 16 | 01x | 0111 | 1100 | 1110 | 0011 |
| 16 | 10x | 0011 | 1110 | 1100 | 0111 |
| 16 | 11x | 0001 | 1111 | 1000 | 1111 |
| 32 | 0xx | 11 | 10 | 11 | 01 |
| 32 | 1xx | 01 | 11 | 10 | 11 |

*Exceptions* None

*See Also* EDGE<8|16|32>[L]N on page 170

# 7.14 Edge Handling Instructions (no CC) VIS 2

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| EDGE8N | 0 0000 0001 | Eight 8-bit edge boundary processing, no CC | `edge8n` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE8LN | 0 0000 0011 | Eight 8-bit edge boundary processing, little-endian, no CC | `edge8ln` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE16N | 0 0000 0101 | Four 16-bit edge boundary processing, no CC | `edge16n` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE16LN | 0 0000 0111 | Four 16-bit edge boundary processing, little-endian, no CC | `edge16ln` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE32N | 0 0000 1001 | Two 32-bit edge boundary processing, no CC | `edge32n` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |
| EDGE32LN | 0 0000 1011 | Two 32-bit edge boundary processing, little-endian, no CC | `edge32ln` $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | **B1** |

*Description* EDGE8[L]N, EDGE16[L]N, and EDGE32[L]N operate identically to EDGE8[L]cc, EDGE16[L]cc, and EDGE32[L]cc, respectively, but do not set the integer condition codes.

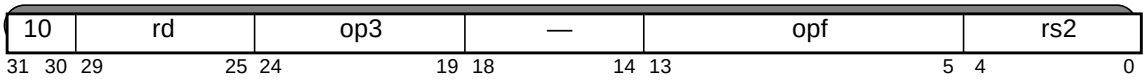
See *Edge Handling Instructions* on page 168 for details.

*Exceptions* None

*See Also* EDGE<8,16,32>[L]cc on page 168

# 7.15 Floating-Point Absolute Value

| Instruction | op3 | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| FABSs | 11 0100 | 0 0000 1001 | Absolute Value Single | `fabss` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FABSd | 11 0100 | 0 0000 1010 | Absolute Value Double | `fabsd` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FABSq | 11 0100 | 0 0000 1011 | Absolute Value Quad | `fabsq` $freg_{rs2}$, $freg_{rd}$ | **C3** |




*Description*

FABS copies the source floating-point register(s) to the destination floating-point register(s), with the sign bit cleared (set to 0).

FABSs operates on single-precision (32-bit) floating-point registers, FABSd operates on double-precision (64-bit) floating-point register pairs, and FABSq operates on quad-precision (128-bit) floating-point register quadruples.

These instructions clear (set to 0) both FSR.cexc and FSR.ftt. They do not round, do not modify FSR.aexc, and do not treat floating-point NaN values differently from other floating-point values.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FABSq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FABS instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FABS instruction causes an *fp_disabled* exception.

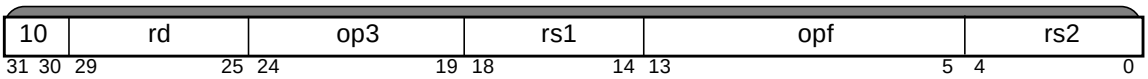
An attempt to execute an FABSq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FABSq only))

# 7.16 Floating-Point Add

| Instruction | op3 | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| FADDs | 11 0100 | 0 0100 0001 | Add Single | `fadds` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FADDd | 11 0100 | 0 0100 0010 | Add Double | `faddd` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FADDq | 11 0100 | 0 0100 0011 | Add Quad | `faddq` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **C3** |




*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.

Rounding is performed as specified by FSR.rd.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FADDq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FADD instruction causes an *fp_disabled* exception.

An attempt to execute an FADDq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) or (rd{1:0} ≠ 0) causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

**Note** | An *fp_exception_other* with FSR.ftt = unfinished_FPop can occur if the operation detects unusual, implementation-specific conditions.

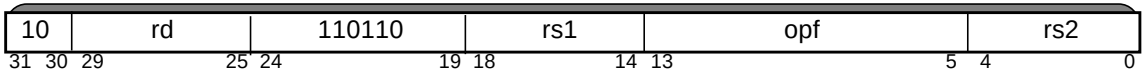
For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FADDq only))
*fp_exception_other* (FSR.ftt = unfinished_FPop)
*fp_exception_ieee_754* (OF, UF, NX, NV)

# 7.17 Align Data VIS 1

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| FALIGNDATA | 0 0100 1000 | Perform data alignment for misaligned data | `faligndata` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |




*Description* 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 is numbered GSR.align+7).

GSR.align is normally set by a previous ALIGNADDRESS instruction.

GSR.align 101

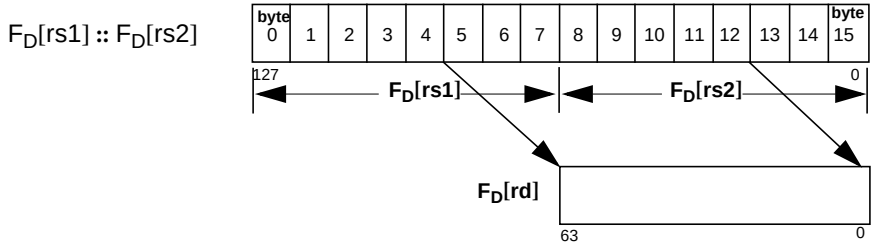



**FIGURE 7-6** FALIGNDATA

A byte-aligned 64-bit load can be performed as shown below.

```
alignaddr   Address, Offset, Address !set GSR.align
ldd         [Address], %d0
ldd         [Address + 8], %d2
faligndata  %d0, %d2, %d4        !use GSR.align to select bytes
```

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FALIGNDATA instruction causes an *fp_disabled* exception.

*Exceptions* *fp_disabled*

*See Also* Align Address on page 146

# 7.18 Branch on Floating-Point Condition Codes (FBfcc)

| Opcode | cond | Operation | fcc Test | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|
| FBA[D] | 1000 | Branch Always | 1 | `fba{,a}` | *label* | **A1** |
| FBN[D] | 0000 | Branch Never | 0 | `fbn{,a}` | *label* | **A1** |
| FBU[D] | 0111 | Branch on Unordered | U | `fbu{,a}` | *label* | **A1** |
| FBG[D] | 0110 | Branch on Greater | G | `fbg{,a}` | *label* | **A1** |
| FBUG[D] | 0101 | Branch on Unordered or Greater | G **or** U | `fbug{,a}` | *label* | **A1** |
| FBL[D] | 0100 | Branch on Less | L | `fbl{,a}` | *label* | **A1** |
| FBUL[D] | 0011 | Branch on Unordered or Less | L **or** U | `fbul{,a}` | *label* | **A1** |
| FBLG[D] | 0010 | Branch on Less or Greater | L **or** G | `fblg{,a}` | *label* | **A1** |
| FBNE[D] | 0001 | Branch on Not Equal | L **or** G **or** U | `fbne`[†]`{,a}` | *label* | **A1** |
| FBE[D] | 1001 | Branch on Equal | E | `fbe`[‡]`{,a}` | *label* | **A1** |
| FBUE[D] | 1010 | Branch on Unordered or Equal | E **or** U | `fbue{,a}` | *label* | **A1** |
| FBGE[D] | 1011 | Branch on Greater or Equal | E **or** G | `fbge{,a}` | *label* | **A1** |
| FBUGE[D] | 1100 | Branch on Unordered or Greater or Equal | E **or** G **or** U | `fbuge{,a}` | *label* | **A1** |
| FBLE[D] | 1101 | Branch on Less or Equal | E **or** L | `fble{,a}` | *label* | **A1** |
| FBULE[D] | 1110 | Branch on Unordered or Less or Equal | E **or** L **or** U | `fbule{,a}` | *label* | **A1** |
| FBO[D] | 1111 | Branch on Ordered | E **or** L **or** G | `fbo{,a}` | *label* | **A1** |

[†] *synonym:* `fbnz` [‡] *synonym:* `fbz`

**Programming Note** To set the annul (a) 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 (a) bit 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 (a) bit. If a conditional branch is not taken and the annul bit is 1 (a = 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.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FBfcc instruction causes an *fp_disabled* exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1 and the FBfcc instruction will cause a transfer of control (FBA or taken conditional branch), then FBfcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the FBfcc instruction) is stored in TPC[TL] and the value of NPC from before the FBfcc was executed is stored in TNPC[TL]. Note that FBN never causes a *control_transfer_instruction* exception.

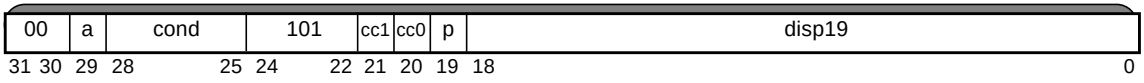
*Exceptions*

*fp_disabled*
*control_transfer_instruction* (impl. dep. #450-S20)

# 7.19 Branch on Floating-Point Condition Codes with Prediction (FBPfcc)

| Instruction | cond | Operation | fcc Test | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|
| FBPA | 1000 | Branch Always | 1 | `fba{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPN | 0000 | Branch Never | 0 | `fbn{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPU | 0111 | Branch on Unordered | U | `fbu{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPG | 0110 | Branch on Greater | G | `fbg{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPUG | 0101 | Branch on Unordered or Greater | G **or** U | `fbug{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPL | 0100 | Branch on Less | L | `fbl{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPUL | 0011 | Branch on Unordered or Less | L **or** U | `fbul{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPLG | 0010 | Branch on Less or Greater | L **or** G | `fblg{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPNE | 0001 | Branch on Not Equal | L **or** G **or** U | `fbne`†`{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPE | 1001 | Branch on Equal | E | `fbe`‡`{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPUE | 1010 | Branch on Unordered or Equal | E **or** U | `fbue{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPGE | 1011 | Branch on Greater or Equal | E **or** G | `fbge{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPUGE | 1100 | Branch on Unordered or Greater or Equal | E **or** G **or** U | `fbuge{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPLE | 1101 | Branch on Less or Equal | E **or** L | `fble{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPULE | 1110 | Branch on Unordered or Less or Equal | E **or** L **or** U | `fbule{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |
| FBPO | 1111 | Branch on Ordered | E **or** L **or** G | `fbo{,a}{,pt\|,pn}` | `%fcc`*n, label* | **A1** |

† *synonym:* `fbnz` ‡ *synonym:* `fbz`




| cc1 | cc0 | Condition Code |
|---|---|---|
| 0 | 0 | `fcc0` |
| 0 | 1 | `fcc1` |
| 1 | 0 | `fcc2` |
| 1 | 1 | `fcc3` |

**Programming Note** | To set the annul (a) 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 defaults to ",pt". To select the appropriate floating-point condition code, include "%fcc0", "%fcc1", "%fcc2", or "%fcc3" before the label.

*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 (a) bit 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 (a) bit 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 (a) bit. If a conditional branch is not taken and the annul bit is 1 (a = 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, *Instruction Set Overview*.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FBPfcc instruction causes an *fp_disabled* exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1 and the FBfcc instruction will cause a transfer of control (FBPA or taken conditional branch), then FBPfcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction*

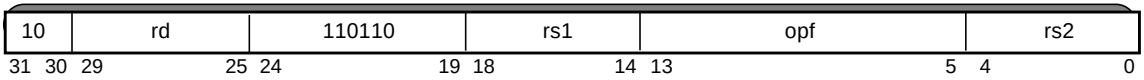
trap occurs, PC (the address of the FBPfcc instruction) is stored in TPC[TL] and the value of NPC from before the FBPfcc was executed is stored in TNPC[TL]. Note that FBPN never causes a *control_transfer_instruction* exception.

*Exceptions*

*fp_disabled*
*control_transfer_instruction* (impl. dep. #450-S20)

# 7.20 SIMD Signed Compare VIS 1

| Instruction | opf | Operation | s1 | s2 | d | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|---|
| FCMPLE16 | 0 0010 0000 | Four 16-bit compare; set R[rd] if *src1* ≤ *src2* | f64 | f64 | i64 | `fcmple16` $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ | **B1** |
| FCMPNE16 | 0 0010 0010 | Four 16-bit compare; set R[rd] if *src1* ≠ *src2* | f64 | f64 | i64 | `fcmpne16` $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ | **B1** |
| FCMPLE32 | 0 0010 0100 | Two 32-bit compare; set R[rd] if *src1* ≤ *src2* | f64 | f64 | i64 | `fcmple32` $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ | **B1** |
| FCMPNE32 | 0 0010 0110 | Two 32-bit compare; set R[rd] if *src1* ≠ *src2* | f64 | f64 | i64 | `fcmpne32` $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ | **B1** |
| FCMPGT16 | 0 0010 1000 | Four 16-bit compare; set R[rd] if *src1* > *src2* | f64 | f64 | i64 | `fcmpgt16` $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ | **B1** |
| FCMPEQ16 | 0 0010 1010 | Four 16-bit compare; set R[rd] if *src1* = *src2* | f64 | f64 | i64 | `fcmpeq16` $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ | **B1** |
| FCMPGT32 | 0 0010 1100 | Two 32-bit compare; set R[rd] if *src1* > *src2* | f64 | f64 | i64 | `fcmpgt32` $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ | **B1** |
| FCMPEQ32 | 0 0010 1110 | Two 32-bit compare; set R[rd] if *src1* = *src2* | f64 | f64 | i64 | `fcmpeq32` $freg_{rs1}$, $freg_{rs2}$, $reg_{rd}$ | **B1** |




*Description* Either four 16-bit signed values or two 32-bit signed values in $F_D$[rs1] and $F_D$[rs2] are compared. The 4-bit or 2-bit condition-code results are stored in the least significant bits of the integer register R[rd]. The least significant 16-bit or 32-bit compare result corresponds to bit zero of R[rd].

**Note** | Bits 63:4 of the destination register R[rd] are set to zero for 16-bit compares. Bits 63:2 of the destination register R[rd] are set to zero for 32-bit compares.

For FCMPGT{16,32}, each bit in the result is set to 1 if the corresponding signed value in $F_D$[rs1] is greater than the signed value in $F_D$[rs2]. Less-than comparisons are made by swapping the operands.

For FCMPLE{16,32}, each bit in the result is set to 1 if the corresponding signed value in $F_D$[rs1] is less than or equal to the signed value in $F_D$[rs2]. Greater-than-or-equal comparisons are made by swapping the operands.

For FCMPEQ{16,32}, each bit in the result is set to 1 if the corresponding signed value in $F_D$[rs1] is equal to the signed value in $F_D$[rs2].

For FCMPNE{16,32}, each bit in the result is set to 1 if the corresponding signed value in $F_D$[rs1] is not equal to the signed value in $F_D$[rs2].

FIGURE 7-7 and FIGURE 7-8 illustrate 16-bit and 32-bit pixel comparison operations, respectively.

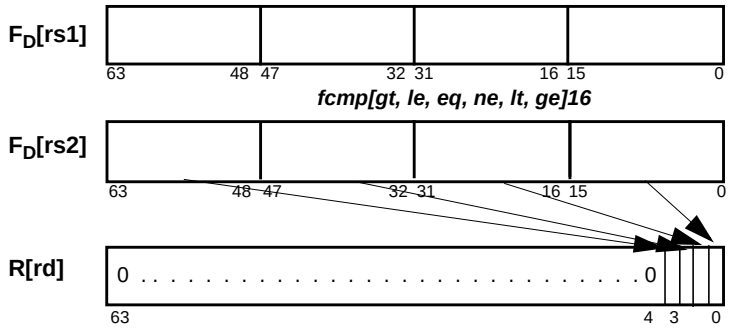



**FIGURE 7-7** Four 16-bit Signed Fixed-point SIMD Comparison Operations

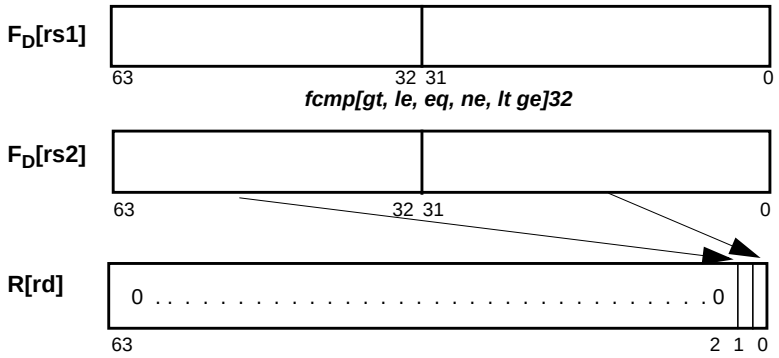



**FIGURE 7-8** Two 32-bit Signed Fixed-point SIMD Comparison Operation

In all comparisons, if a compare condition is not true, the corresponding bit in the result is set to 0.

**Programming Note** The results of a SIMD signed compare operation can be used directly by both integer operations (for example, partial stores) and partitioned conditional moves.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a SIMD signed compare instruction causes an *fp_disabled* exception.

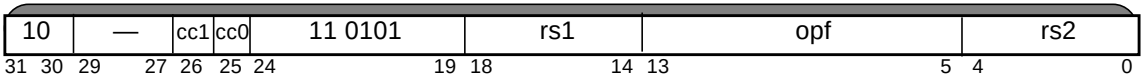
# FCMP*<16|32> (SIMD)

*Exception* *fp_disabled*

*See Also* STPARTIALF on page 351

# 7.21 Floating-Point Compare

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| FCMPs | 0 0101 0001 | Compare Single | `fcmps` %`fcc`*n*, *freg*$_{rs1}$, *freg*$_{rs2}$ | **A1** |
| FCMPd | 0 0101 0010 | Compare Double | `fcmpd` %`fcc`*n*, *freg*$_{rs1}$, *freg*$_{rs2}$ | **A1** |
| FCMPq | 0 0101 0011 | Compare Quad | `fcmpq` %`fcc`*n*, *freg*$_{rs1}$, *freg*$_{rs2}$ | **C3** |
| FCMPEs | 0 0101 0101 | Compare Single and Exception if Unordered | `fcmpes` %`fcc`*n*, *freg*$_{rs1}$, *freg*$_{rs2}$ | **A1** |
| FCMPEd | 0 0101 0110 | Compare Double and Exception if Unordered | `fcmped` %`fcc`*n*, *freg*$_{rs1}$, *freg*$_{rs2}$ | **A1** |
| FCMPEq | 0 0101 0111 | Compare Quad and Exception if Unordered | `fcmpeq` %`fcc`*n*, *freg*$_{rs1}$, *freg*$_{rs2}$ | **C3** |




| cc1 | cc0 | Condition Code |
|---|---|---|
| 0 | 0 | `fcc0` |
| 0 | 1 | `fcc1` |
| 1 | 0 | `fcc2` |
| 1 | 1 | `fcc3` |

*Description* These instructions compare F[rs1] with F[rs2] , and set the selected floating-point condition code (`fcc`*n*) as follows

| Relation | Resulting fcc value |
|---|---|
| *freg*$_{rs1}$ = *freg*$_{rs2}$ | 0 |
| *freg*$_{rs1}$ < *freg*$_{rs2}$ | 1 |
| *freg*$_{rs1}$ > *freg*$_{rs2}$ | 2 |
| *freg*$_{rs1}$ ? *freg*$_{rs2}$ (unordered) | 3 |

The “?” in the preceding table means that the compared values are unordered. The unordered condition occurs when one or both of the operands to the comparison 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<s|d|q> / FCMPE<s|d|q>

FCMP causes an invalid (NV) exception if either operand is a signalling NaN.

**V8 Compatibility Note** Unlike the SPARC V8 architecture, SPARC V9 and the UltraSPARC Architecture do 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 rd = 0. In SPARC V9 and the UltraSPARC Architecture, bits 26 and 25 of the instruction (rd{1:0}) specify the floating-point condition code to be set. Legal SPARC V8 code will work on SPARC V9 and the UltraSPARC Architecture because the zeroes in the R[rd] field are interpreted as `fcc0` and the FBfcc instruction branches based on the value of `fcc0`.

An attempt to execute an FCMP instruction when instruction bits 29:27 are nonzero causes an *illegal_instruction* exception.

**Note** UltraSPARC Architecture 2007 processors do not implement in hardware the instructions that refer to quad-precision floating-point registers. An attempt to execute FCMPq or FCMPEq generates *illegal_instruction*, which causes a trap, allowing privileged software to emulate the instruction.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FCMP or FCMPE instruction causes an *fp_disabled* exception.

An attempt to execute an FCMPq or FCMPEq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_ieee_754* (NV)
*fp_exception_other* (FSR.ftt = invalid_fp_register (FCMPq, FCMPEq only))

# 7.22 Floating-Point Divide

| Instruction | op3 | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| FDIVs | 11 0100 | 0 0100 1101 | Divide Single | `fdivs` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FDIVd | 11 0100 | 0 0100 1110 | Divide Double | `fdivd` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FDIVq | 11 0100 | 0 0100 1111 | Divide Quad | `fdivq` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **C3** |

*Description*

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 FSR.rd.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware the instructions that refer to quad-precision floating-point registers. An attempt to execute an FDIVq instruction generates an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FCMP or FCMPE instruction causes an *fp_disabled* exception.

An attempt to execute an FADDq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

**Note** | For FDIVs and FDIVd, an *fp_exception_other* with FSR.ftt = unfinished_FPop can occur if the divide unit detects unusual, implementation-specific conditions.

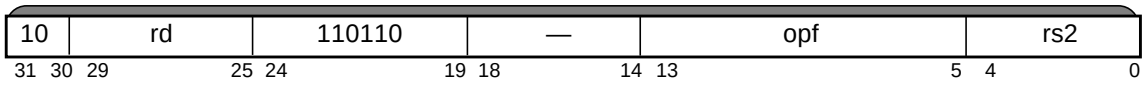
For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FDIVq only)
*fp_exception_other* (FSR.ftt = unfinished_FPop (FDIVs, FDIV))
*fp_exception_ieee_754* (OF, UF, DZ, NV, NX)

# 7.23 FEXPAND VIS 1

| Instruction | opf | Operation | s1 | s2 | d | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|---|
| FEXPAND | 0 0100 1101 | Four 16-bit expands | — | f32 | f64 | `fexpand` *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |




*Description* FEXPAND takes four 8-bit unsigned integers from $F_S$[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 $F_D$[rd]. FIGURE 7-10 illustrates the operation.

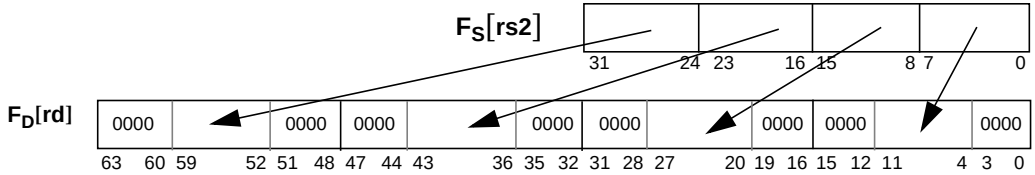



**FIGURE 7-9** FEXPAND Operation

This operation is carried out as follows:

1. Left-shift each 8-bit value by 4 and zero-extend each result to a 16-bit fixed value.
2. Store the result in the destination register, $F_D$[rd].

**Programming Note** | FEXPAND performs the inverse of the FPACK16 operation.

An attempt to execute an FEXPAND instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FEXPAND instruction causes an *fp_disabled* exception.

*Exceptions*
*illegal_instruction*
*fp_disabled*

*See Also*
FPMERGE on page 222
FPACK on page 213

# 7.24 Convert 32-bit Integer to Floating Point

| Instruction | op3 | opf | Operation | s1 | s2 | d | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|---|---|
| FiTOs | 11 0100 | 0 1100 0100 | Convert 32-bit Integer to Single | — | f32 | f32 | `fitos` *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FiTOd | 11 0100 | 0 1100 1000 | Convert 32-bit Integer to Double | — | f32 | f64 | `fitod` *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FiTOq | 11 0100 | 0 1100 1100 | Convert 32-bit Integer to Quad | — | f32 | f128 | `fitoq` *freg*$_{rs2}$, *freg*$_{rd}$ | **C3** |

*Description*

FiTOs, FiTOd, and FiTOq convert the 32-bit signed integer operand in floating-point register $F_S$[rs2] into a floating-point number in the destination format. All write their result into the floating-point register(s) specified by rd.

The value of FSR.rd determines how rounding is performed by FiTOs.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FiTOq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FiTO<s | d | q> instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FiTO<s | d | q> instruction causes an *fp_disabled* exception.

An attempt to execute an FiTOq instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

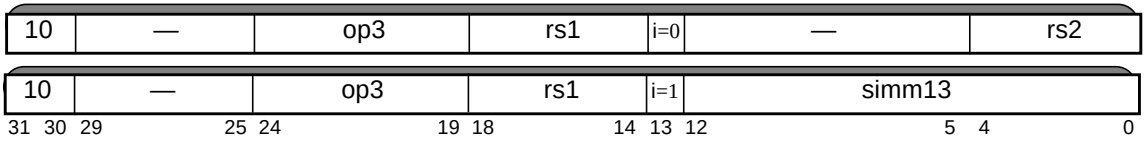
*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FiTOq))
*fp_exception_ieee_754* (NX (FiTOs only))

# 7.25 Flush Instruction Memory

| Instruction | op3 | Operation | Assembly Language Syntax† | Class |
|---|---|---|---|---|
| FLUSH | 11 1011 | Flush Instruction Memory | `flush` [*address*] | **A1** |

† The original assembly language syntax for a FLUSH instruction ("`flush` *address*") has been deprecated because of inconsistency with other SPARC assembly language syntax. Over time, assemblers will support the new syntax for this instruction. In the meantime, some existing assemblers may only recognize the original syntax.




*Description*

FLUSH ensures that the aligned doubleword specified by the effective address is consistent across any local caches and, in a multiprocessor system, will eventually (impl. dep. #122-V9) become consistent everywhere.

The SPARC V9 instruction set architecture does not guarantee consistency between instruction memory and data memory. When software writes[1] to a memory location that may be executed as an instruction (self-modifying code[2]), a potential memory consistency problem arises, which is addressed by the `FLUSH` instruction. Use of FLUSH after instruction memory has been modified ensures that instruction and data memory are synchronized for the processor that issues the FLUSH instruction.

The virtual processor waits until all previous (cacheable) stores have completed before issuing a FLUSH instruction. For the purpose of memory ordering, a FLUSH instruction behaves like a store instruction.

In the following discussion $P_{FLUSH}$ refers to the virtual processor that executed the FLUSH instruction.

FLUSH causes a synchronization within a virtual processor which 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 in the system. With respect to MEMBAR-induced orderings, FLUSH behaves as if it is a store operation (see *Memory Barrier* on page 277).

1. this includes use of store instructions (executed on the same or another virtual processor) that write to instruction memory, or any other means of writing into instruction memory (for example, DMA transfer)

2. practiced, for example, by software such as debuggers and dynamic linkers

# FLUSH

Given any store $S_A$ to address A, that precedes in memory order a FLUSH $F_A$ to address A, that in turn precedes in memory order a store $S_B$ to address B; if any instruction $I_B$ fetched from address B executes the instruction created by store $S_B$, then any instruction $I_A$ that fetched from address A and that follows $I_B$ in program order cannot execute any version of the instruction from address A that existed prior to the store $S_A$.

The preceeding statement defines an ordering requirement to which UltraSPARC Architecture processors comply. By using a FLUSH instruction between two stores that modify instructions, atomicity between the two stores is guaranteed such that any virtual processor executing the instruction modified by the later store will never fetch and/or execute the instruction before it was modified by the earlier store.

If i = 0, the effective address operand for the FLUSH instruction is "R[rs1] + R[rs2]"; if i = 1, it is "R[rs1] + **sign_ext** (simm13)". The three least-significant bits of the effective address are ignored; that is, the effective address always refers to an aligned doubleword.

See implementation-specific documentation for details on specific implementations of the FLUSH instruction.

On an UltraSPARC Architecture processor:

- A FLUSH instruction causes a synchronization within the virtual processor on which the FLUSH is executed, which flushes its instruction pipeline to ensure that no instruction already fetched has subsequently been modified in memory. Any other virtual processors on the same physical processor are unaffected by a FLUSH.
- Coherency between instruction and data memories may or may not be maintained by hardware.

**IMPL. DEP. #409-S10-Cs20**: The implementation of the FLUSH instruction is implementation dependent. If the implementation automatically maintains consistency between instruction and data memory,
(1) the FLUSH address is ignored and
(2) the FLUSH instruction cannot cause any data access exceptions, because its effective address operand is not translated or used by the MMU.
On the other hand, if the implementation does *not* maintain consistency between instruction and data memory, the FLUSH address is used to access the MMU and the FLUSH instruction can cause data access exceptions.

**Programming Note** | For portability across all SPARC V9 implementations, software must always supply the target effective address in FLUSH instructions.

- If the implementation contains instruction prefetch buffers:
  - the instruction prefetch buffer(s) are invalidated

- instruction prefetching is suspended, but may resume starting with the instruction immediately following the FLUSH

**Programming Notes**

1.Typically, FLUSH is used in self-modifying code. The use of self-modifying code is discouraged.

2. If a program includes self-modifying code, to be portable it *must* issue a FLUSH instruction for each modified doubleword of instructions (or make a call to privileged software that has an equivalent effect) after storing into the instruction stream.

3. 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.

4. 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.

5. FLUSH may be a time-consuming operation. (see the Implementation Note below)

6. In a multiprocessor system, the effects of a FLUSH operation will be globally visible before any subsequent store becomes globally visible.

7. FLUSH is designed to act on a doubleword. On some implementations, FLUSH may trap to system software. For these reasons, system software should provide a service routine, callable by nonprivileged software, 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.

8. FLUSH operates using the current (implicit) context. Therefore, a FLUSH executed in privileged or hyperprivileged mode will use the nucleus context and will not necessarily affect instruction cache lines containing data from a user (nonprivileged) context.

**Implementation Note**

In a multiprocessor configuration, FLUSH requires all processors that may be referencing the addressed doubleword to flush their instruction caches, which is a potentially disruptive activity.

**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.

An attempt to execute a FLUSH instruction when instruction bits 29:25 are nonzero causes an *illegal_instruction* exception.

*Exceptions*

*illegal_instruction*
*DAE_nfo_page*
*fast_data_access_MMU_miss* (TLB miss with hardware tablewalk disabled) (impl. dep. #409-S10-Cs20)
*data_access_MMU_miss* (TLB miss with hardware tablewalk enabled) (impl. dep. #409-S10-Cs20)
*fast_data_access_protection*

# 7.26 Flush Register Windows

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| FLUSHW | 10 1011 | Flush Register Windows | `flushw` | **A1** |

*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 flush register windows to memory so that it can switch memory stacks or examine register contents from previous stack frames.

FLUSHW acts as a NOP if CANSAVE = *N_REG_WINDOWS* − 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** *N_REG_WINDOWS*). See *Register Window Management Instructions* on page 127.

**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.

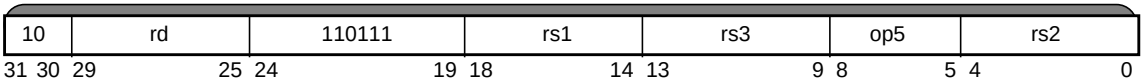
An attempt to execute a FLUSHW instruction when instruction bits 29:25, 18:14, or 12:0 are nonzero causes an *illegal_instruction* exception.

*Exceptions*

*illegal_instruction*
*spill_n_normal*
*spill_n_other*

# 7.27 Floating-Point Multiply-Add and Multiply-Subtract (fused)

| Instruction | op5 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| FMADDs | 00 01 | Multiply-Add Single | `fmadds` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$ | **C3** |
| FMADDd | 00 10 | Multiply-Add Double | `fmaddd` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$ | **C3** |
| FMSUBs | 01 01 | Multiply-Subtract Single | `fmsubs` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$ | **C3** |
| FMSUBd | 01 10 | Multiply-Subtract Double | `fmsubd` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$ | **C3** |
| FNMSUBs | 10 01 | Negative Multiply-Subtract Single | `fnmsubs` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$ | **C3** |
| FNMSUBd | 10 10 | Negative Multiply-Subtract Double | `fnmsubd` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$ | **C3** |
| FNMADDs | 11 01 | Negative Multiply-Add Single | `fnmadds` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$ | **C3** |
| FNMADDd | 11 10 | Negative Multiply-Add Double | `fnmaddd` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$ | **C3** |




| Instruction | Implementation |
|---|---|
| Multiply-Add | F[rd] ← (F[rs1] x F[rs2]) + F[rs3] |
| Multiply-Subtract | F[rd] ← (F[rs1] x F[rs2]) – F[rs3] |
| Negative Multiply-Add | F[rd] ← – ((F[rs1] x F[rs2]) + F[rs3]) |
| Negative Multiply-Subtract | F[rd] ← – ((F[rs1] x F[rs2]) – F[rs3]) |

*Description*

The fused floating-point multiply-add instructions, FMADD<s | d>, multiply the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, add that product to the register(s) specified by rs3, round the result, and write the result into the floating-point register(s) specified by rd.

The fused floating-point multiply-subtract instructions, FMSUB<s | d>, multiply the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, subtract from that product the register(s) specified by rs3, round the result, and write the result into the floating-point register(s) specified by rd.

The fused floating-point negative multiply-add instructions, FNMADD<s | d>, multiply the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, add to the product the register(s) specified by rs3, negate the result, round the result, and write the result into the floating-point register(s) specified by rd.

# FMAf

The fused floating-point negative multiply-subtract instructions, FNMSUB<s | d>, multiply the floating-point register(s) specified by the rs1 field and the floating-point register(s) specified by the rs2 field, subtract from the product the register(s) specified by the rs3 field, negate the result, round the result, and write the result into the floating-point register(s) specified by the rd field.

All of the above instructions are "fused" operations; no rounding is performed between the multiplication operation and the subsequent addition (or subtraction). Therefore, at most one rounding step occurs.

The negative fused multiply-add/subtract instructions (FNM*) treat NaN values as follows:

- A source QNaN propagates with its sign bit unchanged
- A generated (default response) QNaN result has a sign bit of zero
- A source SNaN that is converted to a QNaN result retains the sign bit of the source SNaN

**Exceptions.** If an FMAf instruction is not implemented in hardware, it generates an *illegal_instruction* exception, so that privileged software can emulate the instruction.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMAf instruction causes an *fp_disabled* exception.

Overflow, underflow, and inexact exception bits within FSR.cexc and FSR.aexc are updated based on the final result of the operation and not on the intermediate result of the multiplication. The invalid operation exception bits within FSR.cexc and FSR.aexc are updated as if the multiplication and the addition/subtraction were performed using two individual instructions. An invalid operation exception is detected when any of the following conditions are true:

- A source operand (F[rs1], F[rs2], or F[rs3]) is a SNaN
- $\infty \times 0$
- $\infty - \infty$

If the instruction generates an IEEE-754 exception or exceptions for which the corresponding trap enable mask (FSR.tem) bits are set, an *fp_exception_ieee_754* exception and subsequent trap is generated.

If either the multiply or the add/subtract operation detects an unfinished_FPop condition (for example, due to a subnormal operand or final result), the Multiply-Add/Subtract instruction generates an *fp_exception_other* exception with FSR.ftt = unfinished_FPop. An *fp_exception_other* exception with FSR.ftt = unfinished_FPop always takes precedence over an *fp_exception_ieee_754* exception. That is, if an *fp_exception_other* exception occurs due to an unfinished_FPop condition, the FSR.cexc and FSR.aexc fields remain unchanged even if a floating point IEEE 754 exception occurs during the multiply operation (regardless whether traps are enabled, via FSR.tem, for the IEEE exception) and the unfinished_FPop condition occurs during the subsequent add/subtract operation.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

## *Semantic Definitions*

FMADD:

```
(1) tmp ← F[rs1] x F[rs2]
(2) tmp ← tmp + F[rs3]

(3) F[rd] ← round(tmp)
```

FNMADD:

```
(1) tmp ← F[rs1] × F[rs2]
(2) tmp ← tmp + F[rs3]
(3) tmp ← – tmp
(4) F[rd] ← round(tmp)
```

FMSUB:

```
(1) tmp ← F[rs1] x F[rs2]
(2) tmp ← tmp – F[rs3]

(3) F[rd] ← round(tmp)
```

FNMSUB:

```
(1) tmp ← F[rs1] × F[rs2]
(2) tmp ← tmp – F[rs3]
(3) tmp ← – tmp
(4) F[rd] ← round(tmp)
```

## *Exceptions*

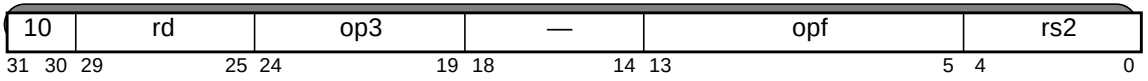
*fp_disabled*
*fp_exception_ieee_754* (OF, UF, NX, NV)
*fp_exception_other* (FSR.ftt = unfinished_FPop)

## *See Also*

FMUL on page 210
FADD on page 172
FSUB on page 224

# 7.28 Floating-Point Move

| Instruction | op3 | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| FMOVs | 11 0100 | 0 0000 0001 | Move (copy) Single | `fmovs` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMOVd | 11 0100 | 0 0000 0010 | Move (copy) Double | `fmovd` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMOVq | 11 0100 | 0 0000 0011 | Move (copy) Quad | `fmovq` $freg_{rs2}$, $freg_{rd}$ | **C3** |




*Description*

FMOV copies the source floating-point register(s) to the destination floating-point register(s), unaltered.

FMOVs, FMOVd, and FMOVq perform 32-bit, 64-bit, and 128-bit operations, respectively.

These instructions clear (set to 0) both FSR.cexc and FSR.ftt. They do not round, do not modify FSR.aexc, and do not treat floating-point NaN values differently from other floating-point values.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMOVq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FMOV instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMOV instruction causes an *fp_disabled* exception.

An attempt to execute an FMOVq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

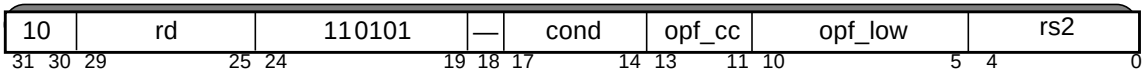
*illegal_instruction*
*fp_disabled* *fp_exception_other* (FSR.ftt = invalid_fp_register (FMOVq only))

*See Also*

*F Register Logical Operate (2 operand)* on page 228

# 7.29 Move Floating-Point Register on Condition (FMOVcc)

| Instruction | opf_low | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| FMOVSicc | 00 0001 | Move Floating-Point Single, based on 32-bit integer condition codes | `fmovs`*icc* `%icc,` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMOVDicc | 00 0010 | Move Floating-Point Double, based on 32-bit integer condition codes | `fmovd`*icc* `%icc,` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMOVQicc | 00 0011 | Move Floating-Point Quad, based on 32-bit integer condition codes | `fmovq`*icc* `%icc,` $freg_{rs2}$, $freg_{rd}$ | **C3** |
| FMOVSxcc | 00 0001 | Move Floating-Point Single, based on 64-bit integer condition codes | `fmovs`*xcc* `%xcc,` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMOVDxcc | 00 0010 | Move Floating-Point Double, based on 64-bit integer condition codes | `fmovd`*xcc* `%xcc,` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMOVQxcc | 00 0011 | Move Floating-Point Quad, based on 64-bit integer condition codes | `fmovq`*xcc* `%xcc,` $freg_{rs2}$, $freg_{rd}$ | **C3** |
| FMOVSfcc | 00 0001 | Move Floating-Point Single, based on floating-point condition codes | `fmovs`*fcc* `%fcc`*n*`,` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMOVDfcc | 00 0010 | Move Floating-Point Double, based on floating-point condition codes | `fmovd`*fcc* `%fcc`*n*`,` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMOVQfcc | 00 0011 | Move Floating-Point Quad, based on floating-point condition codes | `fmovq`*fcc* `%fcc`*n*`,` $freg_{rs2}$, $freg_{rd}$ | **C3** |



# FMOVcc

*Encoding of the* cond *Field for F.P. Moves Based on Integer Condition Codes (*icc *or* xcc*)*

| cond | Operation | icc / xcc Test | *icc*/*xcc* name(s) in Assembly Language Mnemonics |
|---|---|---|---|
| 1000 | Move Always | 1 | a |
| 0000 | Move Never | 0 | n |
| 1001 | Move if Not Equal | **not** Z | ne (or nz) |
| 0001 | Move if Equal | Z | e (or z) |
| 1010 | Move if Greater | **not** (Z **or** (N **xor** V)) | g |
| 0010 | Move if Less or Equal | Z **or** (N **xor** V) | le |
| 1011 | Move if Greater or Equal | **not** (N **xor** V) | ge |
| 0011 | Move if Less | N **xor** V | l |
| 1100 | Move if Greater Unsigned | **not** (C **or** Z) | gu |
| 0100 | Move if Less or Equal Unsigned | (C **or** Z) | leu |
| 1101 | Move if Carry Clear (Greater or Equal, Unsigned) | **not** C | cc (or geu) |
| 0101 | Move if Carry Set (Less than, Unsigned) | C | cs (or lu) |
| 1110 | Move if Positive | **not** N | pos |
| 0110 | Move if Negative | N | neg |
| 1111 | Move if Overflow Clear | **not** V | vc |
| 0111 | Move if Overflow Set | V | vs |

*Encoding of the* cond *Field for F.P. Moves Based on Floating-Point Condition Codes (*fcc*n)*

| cond | Operation | fcc*n* Test | *fcc* name(s) in Assembly Language Mnemonics |
|---|---|---|---|
| 1000 | Move Always | 1 | a |
| 0000 | Move Never | 0 | n |
| 0111 | Move if Unordered | U | u |
| 0110 | Move if Greater | G | g |
| 0101 | Move if Unordered or Greater | G **or** U | ug |
| 0100 | Move if Less | L | l |
| 0011 | Move if Unordered or Less | L **or** U | ul |
| 0010 | Move if Less or Greater | L **or** G | lg |
| 0001 | Move if Not Equal | L **or** G **or** U | ne (or nz) |
| 1001 | Move if Equal | E | e (or z |
| 1010 | Move if Unordered or Equal | E **or** U | ue |
| 1011 | Move if Greater or Equal | E **or** G | ge |
| 1100 | Move if Unordered or Greater or Equal | E **or** G **or** U | uge |
| 1101 | Move if Less or Equal | E **or** L | le |
| 1110 | Move if Unordered or Less or Equal | E **or** L **or** U | ule |
| 1111 | Move if Ordered | E **or** L **or** G | o |

*Encoding of* opf_cc *Field (also see* TABLE E-10 *on page 484)*

| opf_cc | Instruction | Condition Code to be Tested |
|---|---|---|
| $100_2$ | FMOV<s \| d \| q>icc | icc |
| $110_2$ | FMOV<s \| d \| q>xcc | xcc |
| $000_2$ | FMOV<s \| d \| q>fcc | fcc0 |
| $001_2$ | | fcc1 |
| $010_2$ | | fcc2 |
| $011_2$ | | fcc3 |
| $101_2$ | (*illegal_instruction* exception) | |
| $111_2$ | | |

# FMOVcc

*Description*

The FMOVcc 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 floating-point condition code field in FSR. 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 read, but do not modify, any condition codes.

These instructions clear (set to 0) both FSR.cexc and FSR.ftt. They do not round, do not modify FSR.aexc, and do not treat floating-point NaN values differently from other floating-point values.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMOVQicc, FMOVQxcc, or FMOVQfcc instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FMOVcc instruction when instruction bit 18 is nonzero or opf_cc = $101_2$ or $111_2$ causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMOVQicc, FMOVQxcc, or FMOVQfcc instruction causes an *fp_disabled* exception.

An attempt to execute an FMOVQicc, FMOVQxcc, or FMOVQfcc instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

**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 instructiononly executed if the
        ! preceding branch was 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
```

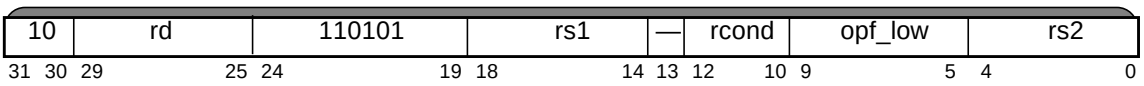
This code also takes four instructions but requires no branches and may boost performance significantly. Use MOVcc and FMOVcc instead of branches wherever these instructions would improve performance.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FMOVQ instructions))

# 7.30 Move Floating-Point Register on Integer Register Condition (FMOVR)

| Instruction | rcond | opf_low | Operation | Test | Class |
|---|---|---|---|---|---|
| — | 000 | 0 0101 | *Reserved* | — | — |
| FMOVRsZ | 001 | 0 0101 | Move Single if Register = 0 | R[rs1] = 0 | **A1** |
| FMOVRsLEZ | 010 | 0 0101 | Move Single if Register ≤ 0 | R[rs1] ≤ 0 | **A1** |
| FMOVRsLZ | 011 | 0 0101 | Move Single if Register < 0 | R[rs1] < 0 | **A1** |
| — | 100 | 0 0101 | *Reserved* | — | — |
| FMOVRsNZ | 101 | 0 0101 | Move Single if Register ≠ 0 | R[rs1] ≠ 0 | **A1** |
| FMOVRsGZ | 110 | 0 0101 | Move Single if Register > 0 | R[rs1] > 0 | **A1** |
| FMOVRsGEZ | 111 | 0 0101 | Move Single if Register ≥ 0 | R[rs1] ≥ 0 | **A1** |
| — | 000 | 0 0110 | *Reserved* | — | — |
| FMOVRdZ | 001 | 0 0110 | Move Double if Register = 0 | R[rs1] = 0 | **A1** |
| FMOVRdLEZ | 010 | 0 0110 | Move Double if Register ≤ 0 | R[rs1] ≤ 0 | **A1** |
| FMOVRdLZ | 011 | 0 0110 | Move Double if Register < 0 | R[rs1] < 0 | **A1** |
| — | 100 | 0 0110 | *Reserved* | — | — |
| FMOVRdNZ | 101 | 0 0110 | Move Double if Register ≠ 0 | R[rs1] ≠ 0 | **A1** |
| FMOVRdGZ | 110 | 0 0110 | Move Double if Register > 0 | R[rs1] > 0 | **A1** |
| FMOVRdGEZ | 111 | 0 0110 | Move Double if Register ≥ 0 | R[rs1] ≥ 0 | **A1** |
| — | 000 | 0 0111 | *Reserved* | — | — |
| FMOVRqZ | 001 | 0 0111 | Move Quad if Register = 0 | R[rs1] = 0 | **C3** |
| FMOVRqLEZ | 010 | 0 0111 | Move Quad if Register ≤ 0 | R[rs1] ≤ 0 | **C3** |
| FMOVRqLZ | 011 | 0 0111 | Move Quad if Register < 0 | R[rs1] < 0 | **C3** |
| — | 100 | 0 0111 | *Reserved* | — | — |
| FMOVRqNZ | 101 | 0 0111 | Move Quad if Register ≠ 0 | R[rs1] ≠ 0 | **C3** |
| FMOVRqGZ | 110 | 0 0111 | Move Quad if Register > 0 | R[rs1] > 0 | **C3** |
| FMOVRqGEZ | 111 | 0 0111 | Move Quad if Register ≥ 0 | R[rs1] ≥ 0 | **C3** |



# FMOVR

| Assembly Language Syntax | | |
|---|---|---|
| `fmovr{s,d,q}z` | *reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | (*synonym:* `fmovr{s,d,q}e`) |
| `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}nz` | *reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | (*synonym:* `fmovr{s,d,q}ne`) |
| `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.

These instructions clear (set to 0) both FSR.cexc and FSR.ftt. They do not round, do not modify FSR.aexc, and do not treat floating-point NaN values differently from other floating-point values.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMOVRq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FMOVR instruction when instruction bit 13 is nonzero or rcond = $000_2$ or $100_2$ causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMOVR instruction causes an *fp_disabled* exception.

An attempt to execute an FMOVRq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

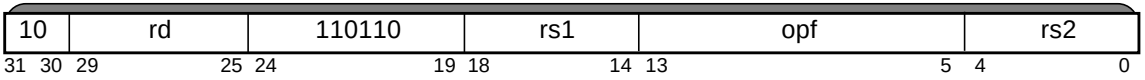
**Implementation Note** | If this instruction is implemented by tagging each register value with an N (negative) and a Z (zero) condition bit, use the following table to determine whether rcond is TRUE:

| Branch | Test |
|---|---|
| FMOVRNZ | **not** Z |
| FMOVRZ | Z |
| FMOVRGEZ | **not** N |
| FMOVRLZ | N |
| FMOVRLEZ | N **or** Z |
| FMOVRGZ | N n**or** Z |

*Exceptions* *illegal_instruction*
*fp_disabled* *fp_exception_other* (FSR.ftt = invalid_fp_register (FMOVRq instructions))

# 7.31 Partitioned Multiply Instructions VIS 1

| Instruction | opf | Operation | s1 | s2 | d | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|---|
| FMUL8x16 | 0 0011 0001 | Unsigned 8-bit by signed 16-bit partitioned product | f32 | f64 | f64 | `fmul8x16` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |
| FMUL8x16AU | 0 0011 0011 | Unsigned 8-bit by signed 16-bit upper α partitioned product | f32 | f32 | f64 | `fmul8x16au` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |
| FMUL8x16AL | 0 0011 0101 | Unsigned 8-bit by signed 16-bit lower α partitioned product | f32 | f32 | f64 | `fmul8x16al` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |
| FMUL8SUx16 | 0 0011 0110 | Signed upper 8-bit by signed 16-bit partitioned product | f64 | f64 | f64 | `fmul8sux16` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |
| FMUL8ULx16 | 0 0011 0111 | Unsigned lower 8-bit by signed 16-bit partitioned product | f64 | f64 | f64 | `fmul8ulx16` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |
| FMULD8SUx16 | 0 0011 1000 | Signed upper 8-bit by signed 16-bit partitioned product | f32 | f32 | f64 | `fmuld8sux16` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |
| FMULD8ULx16 | 0 0011 1001 | Unsigned lower 8-bit by signed 16-bit partitioned product | f32 | f32 | f64 | `fmuld8ulx16` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |




**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.

*Description* The following sections describe the versions of partitioned multiplies.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an partitioned multiply instruction causes an *fp_disabled* exception.

*Exceptions* *fp_disabled*

## 7.31.1 FMUL8x16 Instruction

FMUL8x16 multiplies each unsigned 8-bit value (for example, a pixel component) in the 32-bit floating-point register $F_S$[rs1] by the corresponding (signed) 16-bit fixed-point integer in the 64-bit floating-point register $F_D$[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 $F_D$[rd]. FIGURE 7-10 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.

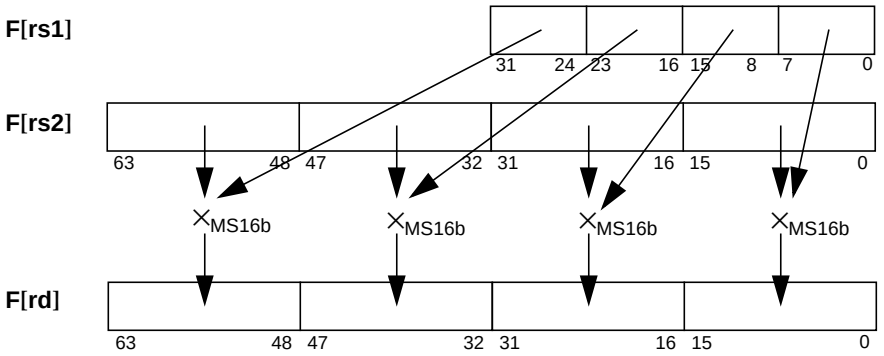



**FIGURE 7-10** FMUL8x16 Operation

## 7.31.2 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 $F_S$[rs2] (typically an α pixel component value). FIGURE 7-11 illustrates the operation.

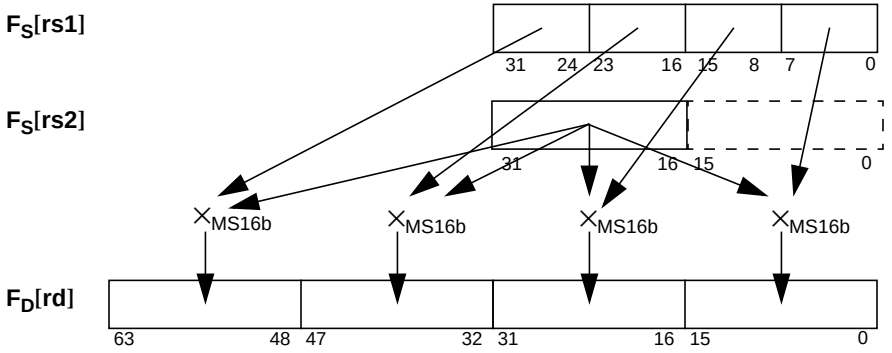



**FIGURE 7-11** FMUL8x16AU Operation

## 7.31.3 FMUL8x16AL Instruction

FMUL8x16AL is the same as FMUL8x16AU, except that the least significant ("lower") 16 bits of the 32-bit register $F_S$[rs2] register are used as a multiplier. FIGURE 7-12 illustrates the operation.

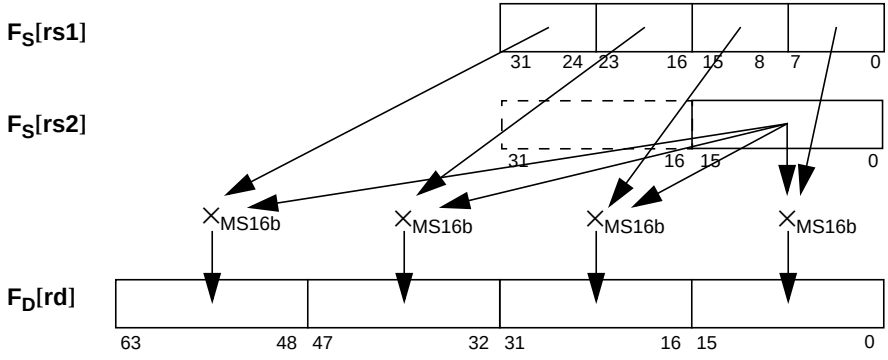



**FIGURE 7-12** FMUL8x16AL Operation

## 7.31.4 FMUL8SUx16 Instruction

FMUL8SUx16 multiplies the most significant (“upper”) 8 bits of each 16-bit signed value in the 64-bit floating-point register $F_D$[rs1] by the corresponding signed, 16-bit, fixed-point, signed integer in the 64-bit floating-point register $F_D$[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 $F_D$[rd]. If the product is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE 7-13 illustrates the operation.

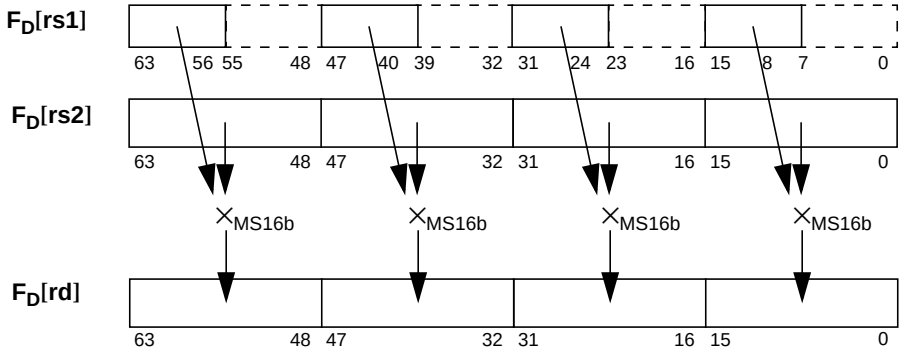



**FIGURE 7-13** FMUL8SUx16 Operation

## 7.31.5 FMUL8ULx16 Instruction

FMUL8ULx16 multiplies the unsigned least significant (“lower”) 8 bits of each 16-bit value in the 64-bit floating-point register $F_D$[rs1] by the corresponding fixed-point signed 16-bit integer in the 64-bit floating-point register $F_D$[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 $F_D$[rd]. If the result is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE 7-14 illustrates the operation; CODE EXAMPLE 7-1 exemplifies the operation.

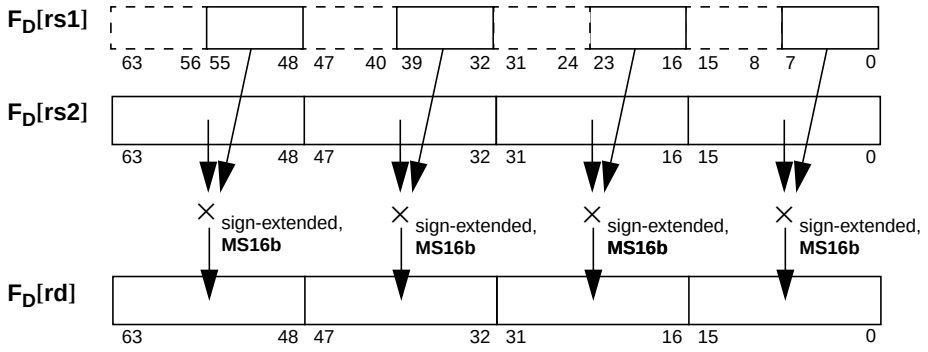



**FIGURE 7-14** FMUL8ULx16 Operation

**CODE EXAMPLE 7-1** 16-bit × 16-bit 16-bit Multiply

```
    fmul8sux16   %f0, %f1, %f2
    fmul8ulx16   %f0, %f1, %f3
    fpadd16      %f2, %f3, %f4
```

## 7.31.6 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 7-15 illustrates the operation.

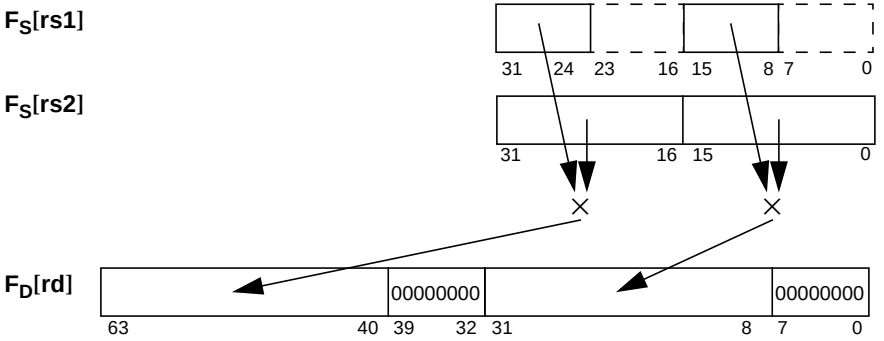



**FIGURE 7-15** FMULD8SUx16 Operation

## 7.31.7 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 7-16 illustrates the operation; CODE EXAMPLE 7-2 exemplifies the operation.

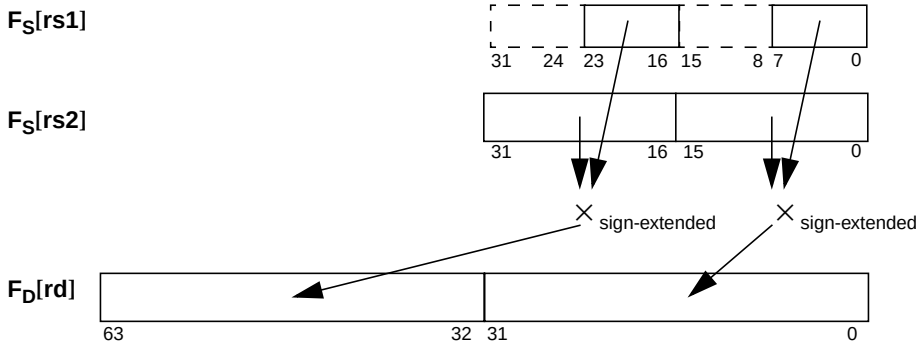


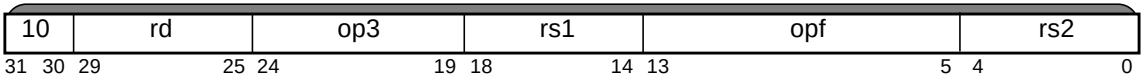

**FIGURE 7-16** FMULD8ULx16 Operation

**CODE EXAMPLE 7-2** 16-bit x 16-bit 32-bit Multiply

```
    fmuld8sux16  %f0, %f1, %f2
    fmuld8ulx16  %f0, %f1, %f3
    fpadd32      %f2, %f3, %f4
```

# 7.32 Floating-Point Multiply

| Instruction | op3 | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| FMULs | 11 0100 | 0 0100 1001 | Multiply Single | `fmuls` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMULd | 11 0100 | 0 0100 1010 | Multiply Double | `fmuld` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FMULq | 11 0100 | 0 0100 1011 | Multiply Quad | `fmulq` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **C3** |
| FsMULd | 11 0100 | 0 0110 1001 | Multiply Single to Double | `fsmuld` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FdMULq | 11 0100 | 0 0110 1110 | Multiply Double to Quad | `fdmulq` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **C3** |




*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.

Rounding is performed as specified by FSR.rd.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMULq or FdMULq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute any FMUL instruction causes an *fp_disabled* exception.

An attempt to execute an FMULq instruction when rs1{1} ≠ 0 or rs2{1} ≠ 0 or rd{1:0} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

An attempt to execute an FdMULq instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

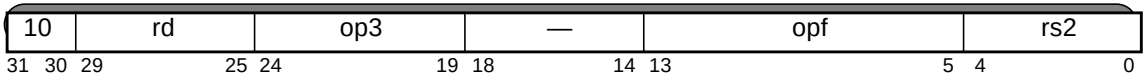
# FMUL<s|d|q>

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FMULq and FdMULq only))
*fp_exception_other* (FSR.ftt = unfinished_FPop)
*fp_exception_ieee_754* (any: NV; FMUL<s | d | q> only: OF, UF, NX)

# 7.33 Floating-Point Negate

| Instruction | op3 | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| FNEGs | 11 0100 | 0 0000 0101 | Negate Single | `fnegs` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FNEGd | 11 0100 | 0 0000 0110 | Negate Double | `fnegd` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FNEGq | 11 0100 | 0 0000 0111 | Negate Quad | `fnegq` $freg_{rs2}$, $freg_{rd}$ | **C3** |




*Description*

FNEG copies the source floating-point register(s) to the destination floating-point register(s), with the sign bit complemented.

These instructions clear (set to 0) both FSR.cexc and FSR.ftt. They do not round, do not modify FSR.aexc, and do not treat floating-point NaN values differently from other floating-point values.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FNEGq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FNEG instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FNEG instruction causes an *fp_disabled* exception.

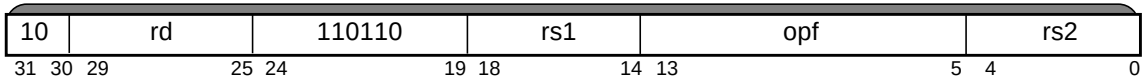
An attempt to execute an FNEGq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FNEGq only))

# 7.34 FPACK VIS 1

| Instruction | opf | Operation | s1 | s2 | d | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|---|
| FPACK16 | 0 0011 1011 | Four 16-bit packs into 8 unsigned bits | — | f64 | f32 | `fpack16` *freg$_{rs2}$*, *freg$_{rd}$* | **B1** |
| FPACK32 | 0 0011 1010 | Two 32-bit packs into 8 unsigned bits | f64 | f64 | f64 | `fpack32` *freg$_{rs1}$*, *freg$_{rs2}$*, *freg$_{rd}$* | **B1** |
| FPACKFIX | 0 0011 1101 | Four 16-bit packs into 16 signed bits | — | f64 | f32 | `fpackfix` *freg$_{rs2}$*, *freg$_{rd}$* | **B1** |




*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. See the subsections on following pages for more detailed descriptions of the operations of these instructions.

An attempt to execute an FPACK16 or FPACKFIX instruction when rs1 ≠ 0 causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute any FPACK instruction causes an *fp_disabled* exception.

*Exceptions* *illegal_instruction* *fp_disabled*

*See Also* FEXPAND on page 185
FPMERGE on page 222

## 7.34.1 FPACK16

FPACK16 takes four 16-bit fixed values from the 64-bit floating-point register $F_D$[rs2], scales, truncates, and clips them into four 8-bit unsigned integers, and stores the results in the 32-bit destination register, $F_S$[rd]. FIGURE 7-17 illustrates the FPACK16 operation.

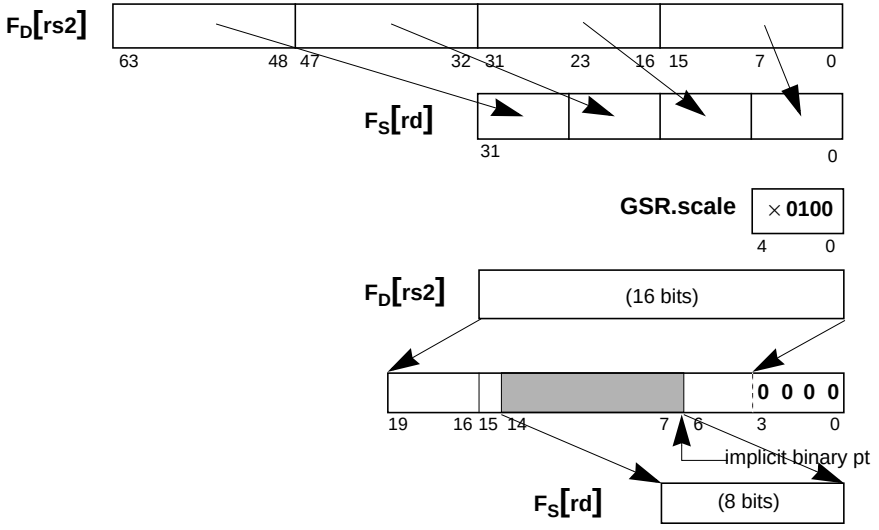



**FIGURE 7-17** FPACK16 Operation

> **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 $F_D$[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, $F_S$[rd].

For each 16-bit partition, the sequence of operations performed is shown in the following example pseudo-code:

```
tmp ← source_operand{15:0} << GSR.scale;
// Pick off the bits from bit position 15+GSR.scale to
```

```
// bit position 7 from the shifted result
trunc_signed_value ← tmp{(15+GSR.scale):7};
If (trunc_signed_value < 0)
   unsigned_8bit_result ← 0;
else if (trunc_signed_value > 255)
   unsigned_8bit_result ← 255;
else
   unsigned_8bit_result ← trunc_signed_value{14:7};
```

## 7.34.2 FPACK32

FPACK32 takes two 32-bit fixed values from the second source operand (64-bit floating-point register $F_D$[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 of each 32-bit word in the 64-bit floating-point register $F_D$[rs1], left-shifted by 8 bits. The 64-bit result is stored in $F_D$[rd]. Thus, successive FPACK32 instructions can assemble two pixels by using three or four pairs of 32-bit fixed values. FIGURE 7-18 illustrates the FPACK32 operation.

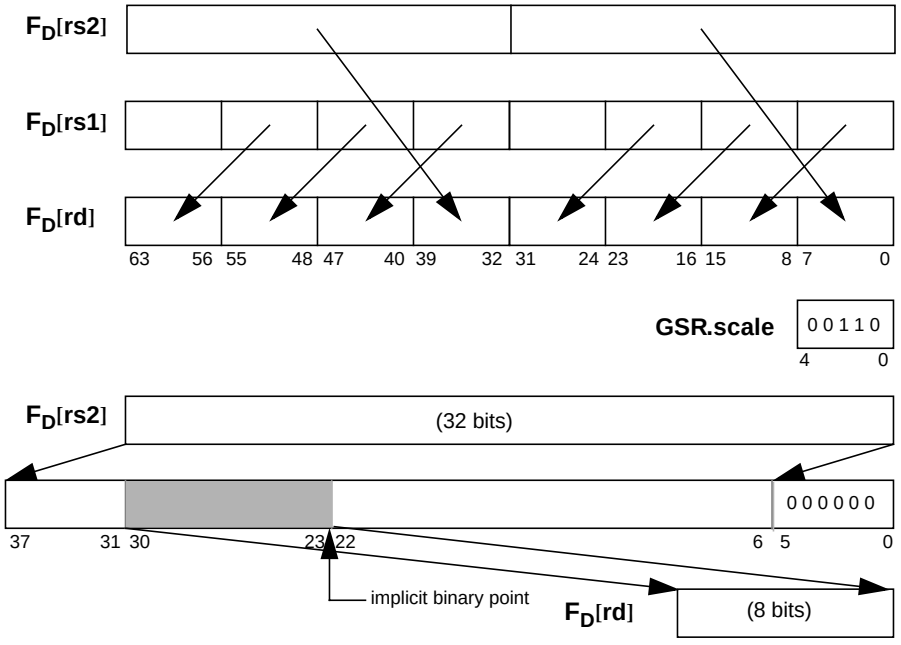



**FIGURE 7-18** FPACK32 Operation

This operation, illustrated in FIGURE 7-18, is carried out as follows:

1. Left-shift each 32-bit value in $F_D$[rs2] 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 is performed to convert the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative (that is, the most significant bit is 1), 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 $F_D$[rs1] by 8 bits.

4. Merge the two clipped 8-bit unsigned values into the corresponding least significant byte positions in the left-shifted $F_D$[rs2] value.

5. Store the result in the 64-bit destination register $F_D$[rd].

For each 32-bit partition, the sequence of operations performed is shown in the following pseudo-code:

```
tmp ← source_operand2{31:0} << GSR.scale;
// Pick off the bits from bit position 31+GSR.scale to
// bit position 23 from the shifted result
trunc_signed_value ← tmp{(31+GSR.scale):23};
if (trunc_signed_value < 0)
   unsigned_8bit_value ← 0;
else if (trunc_signed_value > 255)
   unsigned_8bit_value ← 255;
else
   unsigned_8bit_value ← trunc_signed_value{30:23};
Final_32bit_Result ← (source_operand1{31:0} << 8) |
         (unsigned_8bit_value{7:0});
```

## 7.34.3 FPACKFIX

FPACKFIX takes two 32-bit fixed values from the 64-bit floating-point register $F_D$[rs2], scales, truncates, and clips them into two 16-bit unsigned integers, and then stores the result in the 32-bit destination register $F_S$[rd]. FIGURE 7-19 illustrates the FPACKFIX operation.

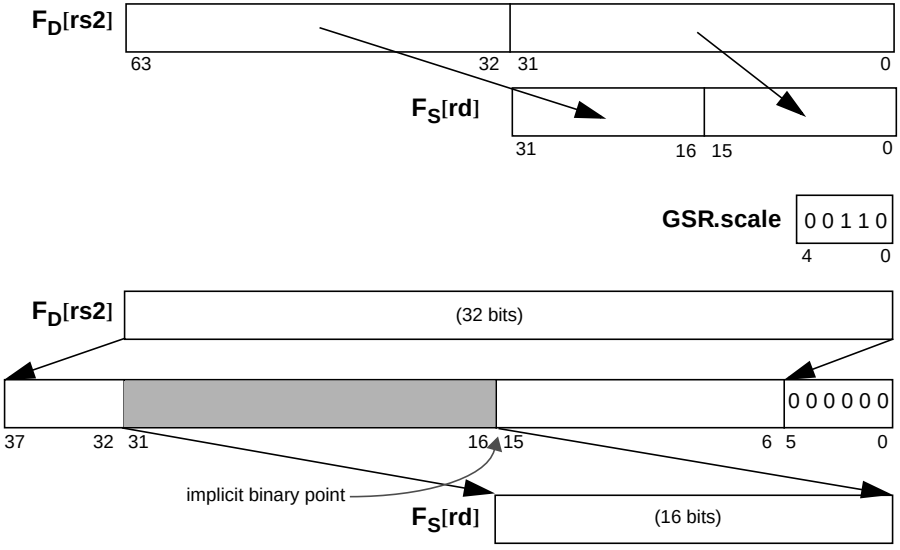



**FIGURE 7-19** FPACKFIX Operation

This operation is carried out as follows:

1. Left-shift each 32-bit value from $F_D$[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 is performed to convert 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 $F_S$[rd].

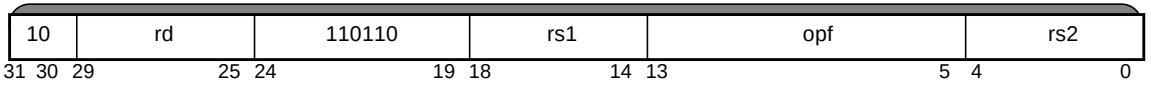
For each 32-bit partition, the sequence of operations performed is shown in the following pseudo-code:

```
tmp ← source_operand{31:0} << GSR.scale;
// Pick off the bits from bit position 31+GSR.scale to
// bit position 16 from the shifted result
```

```
trunc_signed_value ← tmp{(31+GSR.scale):16};
if (trunc_signed_value < -32768)
   signed_16bit_result ← -32768;
else if (trunc_signed_value > 32767)
   signed_16bit_result ← 32767;
else
   signed_16bit_result ← trunc_signed_value{31:16};
```

# 7.35 Fixed-point Partitioned Add VIS 1

| Instruction | opf | Operation | s1 | s2 | d | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|---|---|
| FPADD16 | 0 0101 0000 | Four 16-bit adds | f64 | f64 | f64 | `fpadd16` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FPADD16S | 0 0101 0001 | Two 16-bit adds | f32 | f32 | f32 | `fpadd16s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FPADD32 | 0 0101 0010 | Two 32-bit adds | f64 | f64 | f64 | `fpadd32` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FPADD32S | 0 0101 0011 | One 32-bit add | f32 | f32 | f32 | `fpadd32s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |




*Description* FPADD16 (FPADD32) performs four 16-bit (two 32-bit) partitioned additions between the corresponding fixed-point values contained in the source operands ($F_D$[rs1], $F_D$[rs2]). The result is placed in the destination register, $F_D$[rd].

The 32-bit versions of these instructions (FPADD16S and FPADD32S) perform two 16-bit or one 32-bit partitioned additions.

Any carry out from each addition is discarded and a 2′s-complement arithmetic result is produced.

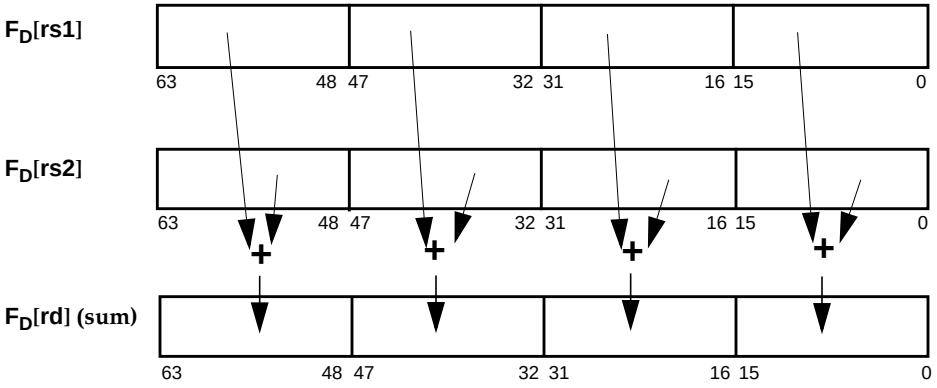



**FIGURE 7-20** FPADD16 Operation

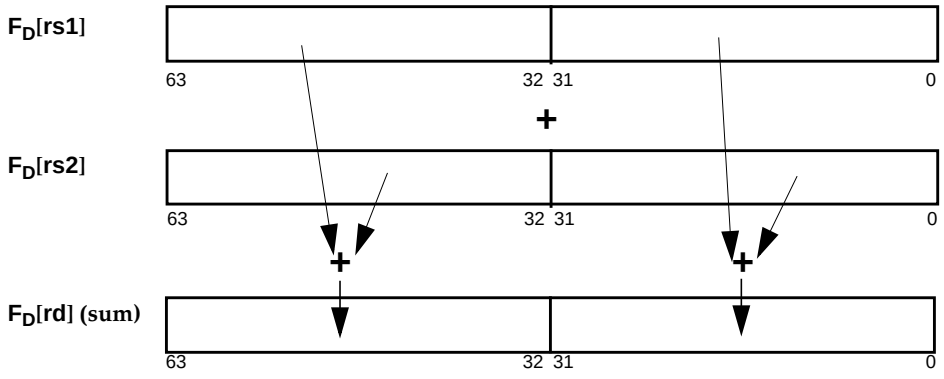



**FIGURE 7-21** FPADD32 Operation

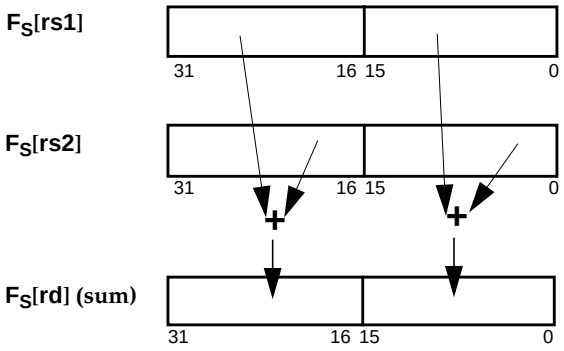



**FIGURE 7-22** FPADD16S Operation

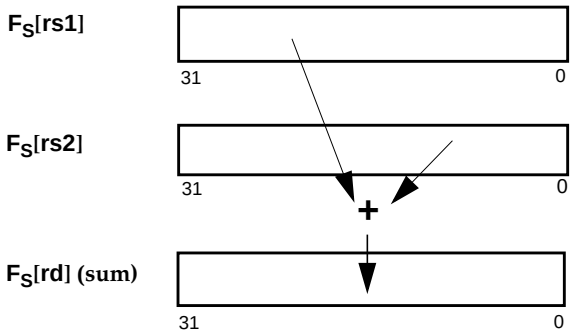


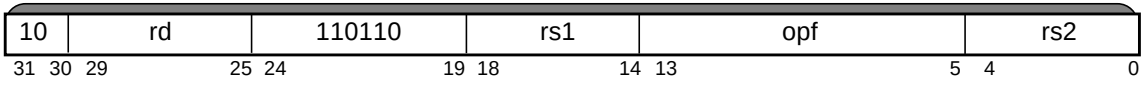

**FIGURE 7-23** FPADD32S Operation

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FPADD instruction causes an *fp_disabled* exception.

*Exceptions* *fp_disabled*

# 7.36 FPMERGE VIS 1

| Instruction | opf | Operation | s1 | s2 | d | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|---|
| FPMERGE | 0 0100 1011 | Two 32-bit merges | f32 | f32 | f64 | `fpmerge` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | **B1** |




*Description*

FPMERGE interleaves eight 8-bit unsigned values in $F_S$[rs1] and $F_S$[rs2] to produce a 64-bit value in the destination register $F_D$[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 7-24 illustrates the operation.

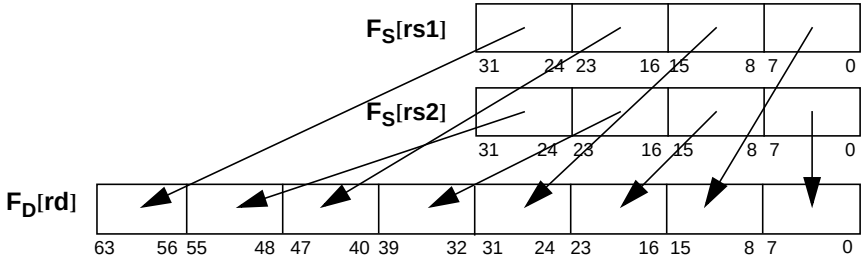



**FIGURE 7-24** FPMERGE Operation

```
                        %d0   R1 G1 B1 A1 R2 G2 B2 A2
                        %d2   R3 G3 B3 A3 R4 G4 B4 A4   } packed representation
-------------------------------------------------------
fpmerge %f0, %f2, %d4   !r1 R3 G1 G3 B1 B3 A1 A3
fpmerge %f1, %f3, %d6   !r2 R4 G2 G4 B2 B4 A2 A4        } intermediate
-------------------------------------------------------
fpmerge %f4, %f6, %d0   !r1 R2 R3 R4 G1 G2 G3 G4
fpmerge %f5, %f7, %d2   !B1 B2 B3 B4 A1 A2 A3 A4        } planar representation
-------------------------------------------------------
fpmerge %f0, %f2, %d4   !r1 B1 R2 B2 R3 B3 R4 B4
fpmerge %f1, %f3, %d6   !G1 A1 G2 A2 G3 A3 G4 A4        } intermediate
-------------------------------------------------------
fpmerge %f4, %f6, %d0   !R1 G1 B1 A1 R2 G2 B2 A2
fpmerge %f5, %f7, %d2   !R3 G3 B3 A3 R4 G4 B4 A4        } packed representation
```

# FPMERGE

**CODE EXAMPLE 7-3** FPMERGE

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FPMERGE instruction causes an *fp_disabled* exception.

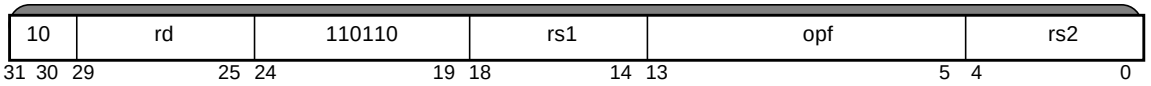
*Exceptions* *fp_disabled*

*See Also* FPACK on page 213
FEXPAND on page 185

# 7.37 Fixed-point Partitioned Subtract (64-bit)

**VIS 1**

| Instruction | opf | Operation | s1 | s2 | d | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|---|---|
| FPSUB16 | 0 0101 0100 | Four 16-bit subtracts | f64 | f64 | f64 | `fpsub16` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FPSUB16S | 0 0101 0101 | Two 16-bit subtracts | f32 | f32 | f32 | `fpsub16s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FPSUB32 | 0 0101 0110 | Two 32-bit subtracts | f64 | f64 | f64 | `fpsub32` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FPSUB32S | 0 0101 0111 | One 32-bit subtract | f32 | f32 | f32 | `fpsub32s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |




*Description*

FPSUB16 (FPSUB32) performs four 16-bit (two 32-bit) partitioned subtractions between the corresponding fixed-point values contained in the source operands ($F_D$[rs1], $F_D$[rs2]). The values in $F_D$[rs2] are subtracted from those in $F_D$[rs1], and the result is placed in the destination register, $F_D$[rd].

The 32-bit versions of these instructions (FPSUB16S and FPSUB32S) perform two 16-bit or one 32-bit partitioned subtractions.

Any carry out from each subtraction is discarded and a 2's-complement arithmetic result is produced.

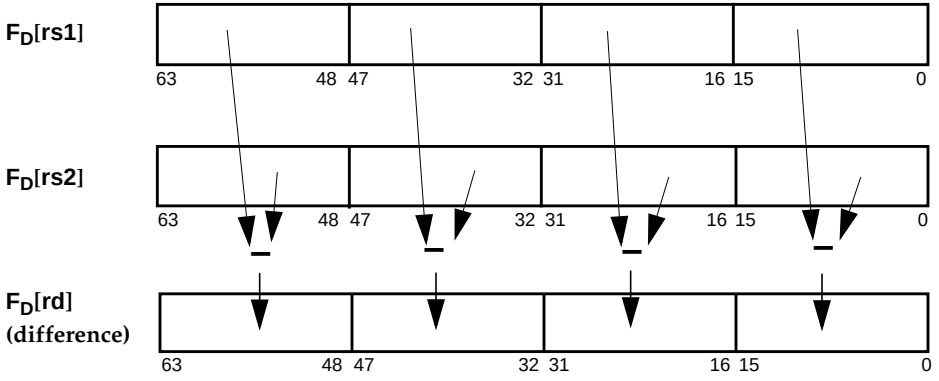



**FIGURE 7-25** FPSUB16 Operation

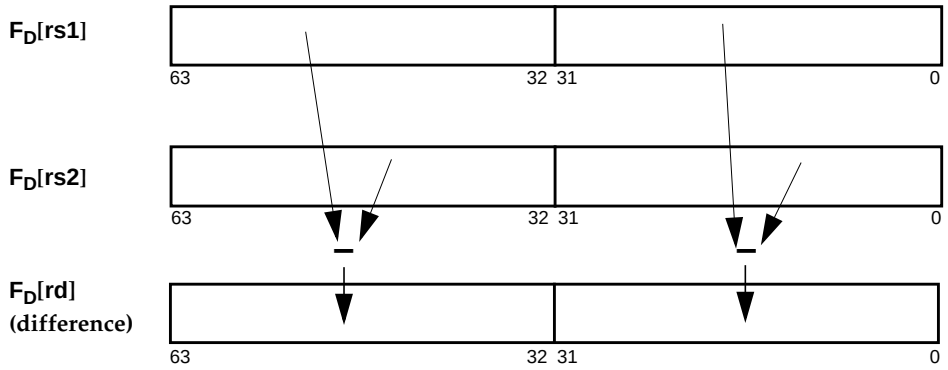



**FIGURE 7-26** FPSUB32 Operation

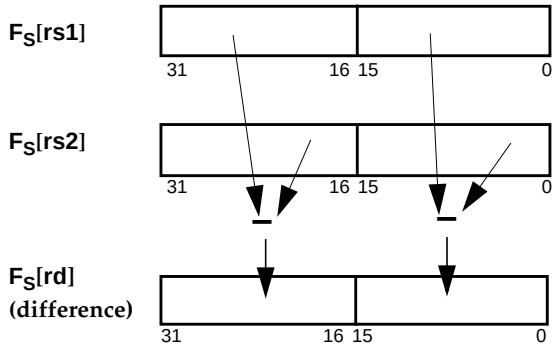



**FIGURE 7-27** FPSUB16S Operation

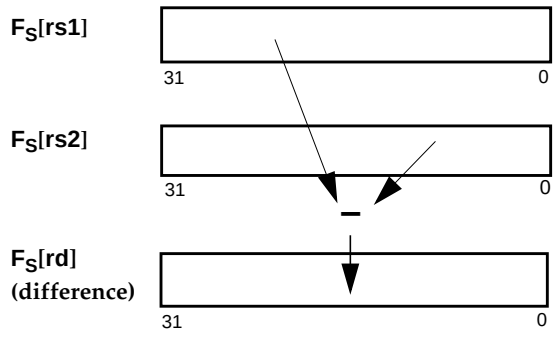


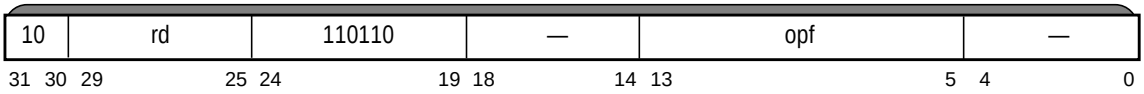

**FIGURE 7-28** FPSUB32S Operation

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FPSUB instruction causes an *fp_disabled* exception.

*Exceptions* *fp_disabled*

# 7.38 F Register Logical Operate (1 operand) VIS 1

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| FZERO | 0 0110 0000 | Zero fill | `fzero` $freg_{rd}$ | **A1** |
| FZEROs | 0 0110 0001 | Zero fill, 32-bit | `fzeros` $freg_{rd}$ | **A1** |
| FONE | 0 0111 1110 | One fill | `fone` $freg_{rd}$ | **A1** |
| FONEs | 0 0111 1111 | One fill, 32-bit | `fones` $freg_{rd}$ | **A1** |




*Description*

FZERO and FONE fill the 64-bit destination register, $F_D$[rd], with all '0' bits or all '1' bits (respectively).

FZEROs and FONEs fill the 32-bit destination register, $F_D$[rd], with all '0' bits or all '1' bits (respectively.

An attempt to execute an FZERO or FONE instruction when instruction bits 18:14 or bits 4:0 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FZERO[s] or FONE[s] instruction causes an *fp_disabled* exception.

*Exceptions*

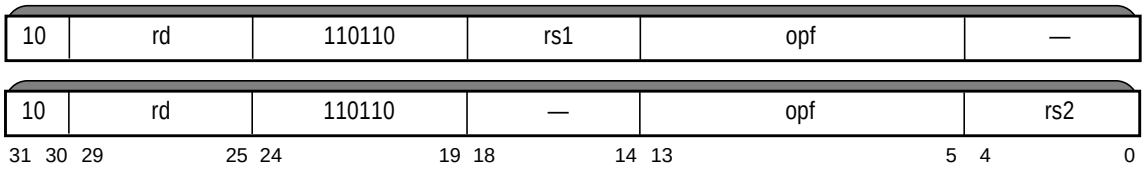
*illegal_instruction*
*fp_disabled*

*See Also*

F Register 2-operand Logical Operations on page 228
F Register 3-operand Logical Operations on page 230

# 7.39 F Register Logical Operate (2 operand) VIS 1

| Instruction | opf | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| FSRC1 | 0 0111 0100 | Copy $F_D$[rs1] to $F_D$[rd] | `fsrc1` | *freg*$_{rs1}$, *freg*$_{rd}$ | **A1** |
| FSRC1s | 0 0111 0101 | Copy $F_S$[rs1] to $F_S$[rd], 32-bit | `fsrc1s` | *freg*$_{rs1}$, *freg*$_{rd}$ | **A1** |
| FSRC2 | 0 0111 1000 | Copy $F_D$[rs2] to $F_D$[rd] | `fsrc2` | *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FSRC2s | 0 0111 1001 | Copy $F_S$[rs2] to $F_S$[rd], 32-bit | `fsrc2s` | *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FNOT1 | 0 0110 1010 | Negate (1's complement) $F_D$[rs1] | `fnot1` | *freg*$_{rs1}$, *freg*$_{rd}$ | **A1** |
| FNOT1s | 0 0110 1011 | Negate (1's complement) $F_S$[rs1], 32-bit | `fnot1s` | *freg*$_{rs1}$, *freg*$_{rd}$ | **A1** |
| FNOT2 | 0 0110 0110 | Negate (1's complement) $F_D$[rs2] | `fnot2` | *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FNOT2s | 0 0110 0111 | Negate (1's complement) $F_S$[rs2], 32-bit | `fnot2s` | *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |




*Description*

The standard 64-bit versions of these instructions perform one of four 64-bit logical operations on the 64-bit floating-point register $F_D$[rs1] (or $F_D$[rs2]) and store the result in the 64-bit floating-point destination register $F_D$[rd].

The 32-bit (single-precision) versions of these instructions perform 32-bit logical operations on $F_S$[rs1] (or $F_S$[rs2]) and store the result in $F_S$[rd].

An attempt to execute an FSRC1(s) or FNOT1(s) instruction when instruction bits 4:0 are nonzero causes an *illegal_instruction* exception. An attempt to execute an FSRC2(s) or FNOT2(s) instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FSRC1[s], FNOT1[s], FSRC1[s], or FNOT1[s] instruction causes an *fp_disabled* exception.

**Programming Note** FSRC1s (FSRC1) functions similarly to FMOVs (FMOVd), except that FSRC1s (FSRC1) does not modify the FSR register while FMOVs (FMOVd) update some fields of FSR (see *Floating-Point Move* on page 195). Programmers are encouraged to use FMOVs (FMOVd) instead of FSRC1s (FSRC1) whenever practical.

*Exceptions*

*illegal_instruction*
*fp_disabled*

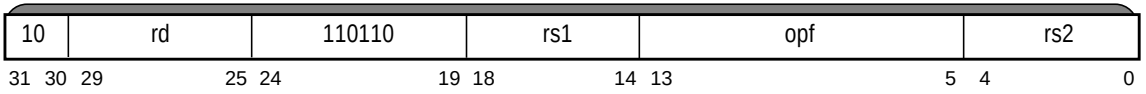
# F Register 2-operand Logical Ops

*See Also*

*Floating-Point Move* on page 195
F Register 1-operand Logical Operations on page 227
F Register 3-operand Logical Operations on page 230

# 7.40 F Register Logical Operate (3 operand) VIS 1

| Instruction | opf | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| FOR | 0 0111 1100 | Logical **or** | `for` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FORs | 0 0111 1101 | Logical **or**, 32-bit | `fors` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FNOR | 0 0110 0010 | Logical **nor** | `fnor` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FNORs | 0 0110 0011 | Logical **nor**, 32-bit | `fnors` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FAND | 0 0111 0000 | Logical **and** | `fand` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FANDs | 0 0111 0001 | Logical **and**, 32-bit | `fands` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FNAND | 0 0110 1110 | Logical **nand** | `fnand` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FNANDs | 0 0110 1111 | Logical **nand**, 32-bit | `fnands` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FXOR | 0 0110 1100 | Logical **xor** | `fxor` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FXORs | 0 0110 1101 | Logical **xor**, 32-bit | `fxors` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FXNOR | 0 0111 0010 | Logical **xnor** | `fxnor` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FXNORs | 0 0111 0011 | Logical **xnor**, 32-bit | `fxnors` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FORNOT1 | 0 0111 1010 | (**not** F[rs1]) **or** F[rs2] | `fornot1` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FORNOT1s | 0 0111 1011 | (**not** F[rs1]) **or** F[rs2], 32-bit | `fornot1s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FORNOT2 | 0 0111 0110 | F[rs1] **or** (**not** F[rs2]) | `fornot2` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FORNOT2s | 0 0111 0111 | F[rs1] **or** (**not** F[rs2]), 32-bit | `fornot2s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FANDNOT1 | 0 0110 1000 | (**not** F[rs1]) **and** F[rs2] | `fandnot1` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FANDNOT1s | 0 0110 1001 | (**not** F[rs1]) **and** F[rs2], 32-bit | `fandnot1s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FANDNOT2 | 0 0110 0100 | F[rs1] **and** (**not** F[rs2]) | `fandnot2` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FANDNOT2s | 0 0110 0101 | F[rs1] **and** (**not** F[rs2]), 32-bit | `fandnot2s` | $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |




*Description* The standard 64-bit versions of these instructions perform one of ten 64-bit logical operations between the 64-bit floating-point registers $F_D$[rs1] and $F_D$[rs2]. The result is stored in the 64-bit floating-point destination register $F_D$[rd].

The 32-bit (single-precision) versions of these instructions perform 32-bit logical operations between $F_S$[rs1] and $F_S$[rs2], storing the result in $F_S$[rd].

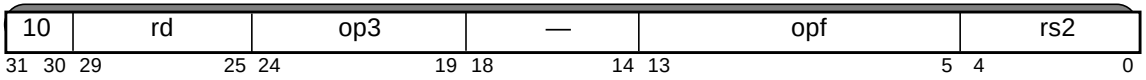
If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute any 3-operand F Register Logical Operate instruction causes an *fp_disabled* exception.

*Exceptions* *fp_disabled*

*See Also* F Register 1-operand Logical Operations on page 227
F Register 2-operand Logical Operations on page 228

# 7.41 Floating-Point Square Root

| Instruction | op3 | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| FSQRTs | 11 0100 | 0 0010 1001 | Square Root Single | `fsqrts` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FSQRTd | 11 0100 | 0 0010 1010 | Square Root Double | `fsqrtd` $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FSQRTq | 11 0100 | 0 0010 1011 | Square Root Quad | `fsqrtq` $freg_{rs2}$, $freg_{rd}$ | **C3** |




*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 FSR.rd.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FSQRTq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FSQRT instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FSQRT instruction causes an *fp_disabled* exception.

An attempt to execute an FSQRTq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

An *fp_exception_other* (with FSR.ftt = unfinished_FPop) can occur if the operand to the square root is positive and subnormal. See *FSR_floating-point_trap_type (ftt)* on page 55 for additional details.

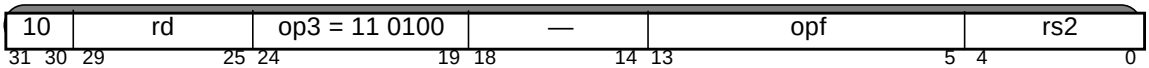
For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FSQRTq only))
*fp_exception_other* (FSR.ftt = unfinished_FPop)
*fp_exception_ieee_754* (IEEE_754_exception (NV, NX))

# 7.42 Convert Floating-Point to Integer

| Instruction | opf | Operation | s1 | s2 | d | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|---|
| FsTOx | 0 1000 0001 | Convert Single to 64-bit Integer | — | f32 | f64 | `fstox` *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FdTOx | 0 1000 0010 | Convert Double to 64-bit Integer | — | f64 | f64 | `fdtox` *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FqTOx | 0 1000 0011 | Convert Quad to 64-bit Integer | — | f128 | f64 | `fqtox` *freg*$_{rs2}$, *freg*$_{rd}$ | **C3** |
| FsTOi | 0 1101 0001 | Convert Single to 32-bit Integer | — | f32 | f32 | `fstoi` *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FdTOi | 0 1101 0010 | Convert Double to 32-bit Integer | — | f64 | f32 | `fdtoi` *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FqTOi | 0 1101 0011 | Convert Quad to 32-bit Integer | — | f128 | f32 | `fqtoi` *freg*$_{rs2}$, *freg*$_{rd}$ | **C3** |




*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 $F_D$[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 $F_S$[rd].

The result is always rounded toward zero; that is, the rounding direction (rd) field of the FSR register is ignored.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FqTOx or FqTOi instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an F<s | d | q>TO<i | x> instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an F<s | d | q>TO<i | x> instruction causes an *fp_disabled* exception.

An attempt to execute an FqTOi or FqTOx instruction when rs2{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

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 393.

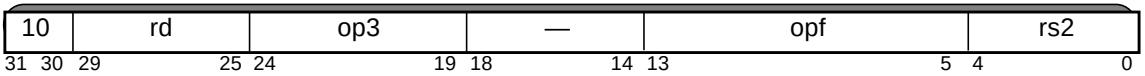
For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FqTOx and FqTOi only))
*fp_exception_other* (FSR.ftt = unfinished_FPop)
*fp_exception_ieee_754* (NV, NX)

# 7.43 Convert Between Floating-Point Formats

| Instruction | op3 | opf | Operation | s1 | s2 | d | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|---|---|---|
| FsTOd | 11 0100 | 0 1100 1001 | Convert Single to Double | — | f32 | f64 | `fstod` | $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FsTOq | 11 0100 | 0 1100 1101 | Convert Single to Quad | — | f32 | f128 | `fstoq` | $freg_{rs2}$, $freg_{rd}$ | **C3** |
| FdTOs | 11 0100 | 0 1100 0110 | Convert Double to Single | — | f64 | f32 | `fdtos` | $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FdTOq | 11 0100 | 0 1100 1110 | Convert Double to Quad | — | f64 | f128 | `fdtoq` | $freg_{rs2}$, $freg_{rd}$ | **C3** |
| FqTOs | 11 0100 | 0 1100 0111 | Convert Quad to Single | — | f128 | f32 | `fqtos` | $freg_{rs2}$, $freg_{rd}$ | **C3** |
| FqTOd | 11 0100 | 0 1100 1011 | Convert Quad to Double | — | f128 | f64 | `fqtod` | $freg_{rs2}$, $freg_{rd}$ | **C3** |




*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.

The value of FSR.rd determines how rounding is performed by these instructions.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FsTOq, FdTOq, FqTOs, or FqTOd instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an F<s | d | q>TO<s | d | q> instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an F<s | d | q>TO<s | d | q> instruction causes an *fp_disabled* exception.

An attempt to execute an FsTOq or FdTOq instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception. An attempt to execute an FqTOs orFqTOd instruction when rs2{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

FqTOd, FqTOs, and FdTOs (the "narrowing" conversion instructions) can cause *fp_exception_ieee_754* OF, UF, and NX exceptions. FdTOq, FsTOq, and FsTOd (the "widening" conversion instructions) cannot.

Any of these six instructions can trigger an *fp_exception_ieee_754* NV exception if the source operand is a signalling NaN.

**Note** | For FdTOs and FsTOd, an *fp_exception_other* with FSR.ftt = unfinished_FPop can occur if implementation-dependent conditions are detected during the conversion operation.

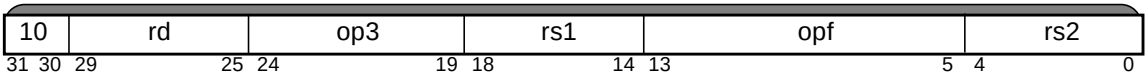
For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FsTOq, FqTOs, FdTOq, and FqTOd only))
*fp_exception_other* (FSR.ftt = unfinished_FPop)
*fp_exception_ieee_754* (NV)
*fp_exception_ieee_754* (OF, UF, NX (FqTOd, FqTOs, and FdTOs))

# 7.44 Floating-Point Subtract

| Instruction | op3 | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| FSUBs | 11 0100 | 0 0100 0101 | Subtract Single | `fsubs` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FSUBd | 11 0100 | 0 0100 0110 | Subtract Double | `fsubd` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **A1** |
| FSUBq | 11 0100 | 0 0100 0111 | Subtract Quad | `fsubq` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$ | **C3** |




*Description*

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 FSR.rd.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FSUBq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FSUB instruction causes an *fp_disabled* exception.

An attempt to execute an FSUBq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) or (rd{1:0} ≠ 0) causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

**Note** | An *fp_exception_other* with FSR.ftt = unfinished_FPop can occur if the operation detects unusual, implementation-specific conditions (for FSUBs or FSUBd).

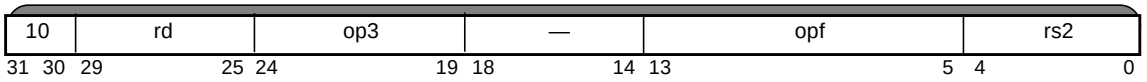
For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FSUBq only))
*fp_exception_other* (FSR.ftt = unfinished_FPop)
*fp_exception_ieee_754* (OF, UF, NX, NV)

# 7.45 Convert 64-bit Integer to Floating Point

| Instruction | op3 | opf | Operation | s1 | s2 | d | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|---|---|
| FxTOs | 11 0100 | 0 1000 0100 | Convert 64-bit Integer to Single | — | i64 | f32 | `fxtos` *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FxTOd | 11 0100 | 0 1000 1000 | Convert 64-bit Integer to Double | — | i64 | f64 | `fxtod` *freg*$_{rs2}$, *freg*$_{rd}$ | **A1** |
| FxTOq | 11 0100 | 0 1000 1100 | Convert 64-bit Integer to Quad | — | i64 | f128 | `fxtoq` *freg*$_{rs2}$, *freg*$_{rd}$ | **C3** |




*Description*

FxTOs, FxTOd, and FxTOq convert the 64-bit signed integer operand in the floating-point register $F_D$[rs2] into a floating-point number in the destination format.

All write their result into the floating-point register(s) specified by rd.

The value of FSR.rd determines how rounding is performed by FxTOs and FxTOd.

**Note** | UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FxTOq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FxTO<s | d | q> instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FxTO<s | d | q> instruction causes an *fp_disabled* exception.

An attempt to execute an FxTOq instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

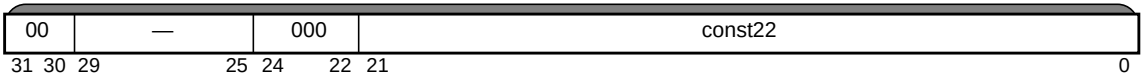
For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*fp_exception_other* (FSR.ftt = invalid_fp_register (FxTOq))
*fp_exception_ieee_754* (NX (FxTOs and FxTOd only))

# 7.46 Illegal Instruction Trap

| Instruction | op | op2 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| ILLTRAP | 00 | 000 | *illegal_instruction* trap | `illtrap` *const22* | **A1** |




*Description* The ILLTRAP instruction causes an *illegal_instruction* exception. The const22 value in the instruction is ignored by the virtual processor; 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.

An attempt to execute an ILLTRAP instruction when reserved instruction bits 29:25 are nonzero (also) causes an *illegal_instruction* exception. However, software should not rely on this behavior, because a future version of the architecture may use nonzero values of bits 29:25 to encode other functions.

*Exceptions* *illegal_instruction*

# 7.47 Implementation-Dependent Instructions

| Instruction | op3 | op4 | Operation | Class |
|---|---|---|---|---|
| IMPDEP1 | 11 0110 | (any) | Implementation-Dependent Instruction 1 | **N3** |
| IMPDEP2A | 11 0111 | 0 | Implementation-Dependent Instruction 2A | **N3** |
| IMPDEP2B | 11 0111 | 1, 2, 3 | Implementation-Dependent Instruction 2B | **N3** |

*Description*

**IMPL. DEP. #106-V9:** 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 opcodes are reserved; see *IMDEP2B Opcodes* on page 240.

See “Implementation-Dependent and Reserved Opcodes” in the "Extending the UltraSPARC Architecture" section of the separate document *UltraSPARC Architecture Application Notes*, for information about extending the instruction set by means of implementation-dependent instructions.

**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 the UltraSPARC Architecture, so it should not be used for implementation-dependent instructions.

*Exceptions*

implementation-dependent (IMPDEP2A, IMPDEP2B)

## 7.47.1 IMPDEP1 Opcodes **VIS 1, 2**

All operands of instructions using IMPDEP1 opcodes are in floating-point registers, unless otherwise specified. Pixel values are stored in single-precision floating point registers and fixed values are stored in double-precision floating point registers, unless otherwise specified.

### 7.47.1.1 Opcode Formats

Most of the VIS instruction set maps to the opcode space reserved for the Implementation-Dependent Instruction 1 (op3 = IMPDEP1 = $36_{16}$) instructions.

**Note** The only IMPDEP1 instructions that currently modify FSR.cexc and FSR.aexc are FP Multiply-Add/Subtract instructions. (See *Floating-Point Multiply-Add and Multiply-Subtract (fused)* on page 192.) The rest of the IMPDEP1 instructions, regardless of whether they use floating-point registers or integer registers, leave FSR.cexc and FSR.aexc unchanged.

## 7.47.2 IMDEP2B Opcodes

The following instructions, collectively known as the fused multiply-accumulate or "FMAf" instructions, are encoded in the op3 = IMPDEP2B = $37_{16}$ opcode space:

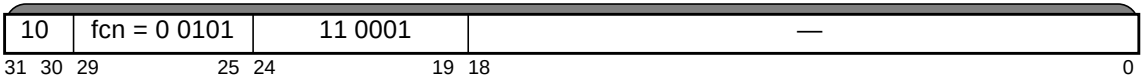
- FMADDs, FMADDd
- FMSUBs, FMSUBd
- FNMSUBs, FNMSUBd
- FNMADDs, FNMADDd

All source operand and destination registers for these instructions are floating-point registers.

For a full description of the instructions currently defined in the IMPDEP2B opcode space, see *Floating-Point Multiply-Add and Multiply-Subtract (fused)* on page 192.

# 7.48 Mark Register Window Sets as "Invalid"

| Instruction | Operation | Assembly Language Syntax | Class |
|---|---|---|---|
| INVALW[P] | Mark all register window sets as "invalid" | `invalw` | **A1** |




*Description* The INVALW instruction marks all register window sets as "invalid"; specifically, it atomically performs the following operations:

CANSAVE ← (*N_REG_WINDOWS* – 2)
CANRESTORE ← 0
OTHERWIN ← 0

**Programming Notes** INVALW marks all windows as invalid; after executing INVALW, *N_REG_WINDOWS*-2 SAVEs can be performed without generating a spill trap. This instruction allows window manipulations to be atomic, without the value of *N_REG_WINDOWS* being visible to privileged software and without an assumption that *N_REG_WINDOWS* is constant (since hyperprivileged software can migrate a thread among virtual processors, across which *N_REG_WINDOWS* may vary).

An attempt to execute an INVALW instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

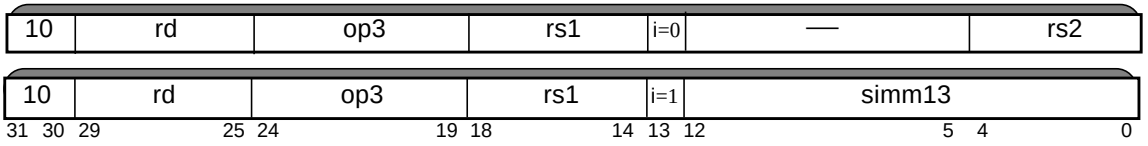
An attempt to execute an INVALW instruction in nonprivileged mode (PSTATE.priv = 0 and HSTATE.hpriv = 0) causes a *privileged_opcode* exception.

*Exceptions* *illegal_instruction*
*privileged_opcode*

*See Also* ALLCLEAN on page 147
NORMALW on page 291
OTHERW on page 293
RESTORED on page 313
SAVED on page 321

# 7.49 Jump and Link

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| JMPL | 11 1000 | Jump and Link | `jmpl` *address*, $reg_{rd}$ | **A1** |




*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].

An attempt to execute a JMPL instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If either of the low-order two bits of the jump address is nonzero, a *mem_address_not_aligned* exception occurs.

**Programming Notes**

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.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then JMPL generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the JMPL instruction) is stored in TPC[TL] and the value of NPC from before the JMPL was executed is stored in TNPC[TL].

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system or being written into R[rd] (or, if a *control_transfer_instruction* trap occurs, into TPC[TL]). (closed impl. dep. #125-V9-Cs10)

# JMPL

*Exceptions*

*illegal_instruction*
*mem_address_not_aligned*
*control_transfer_instruction* (impl. dep. #450-S20)

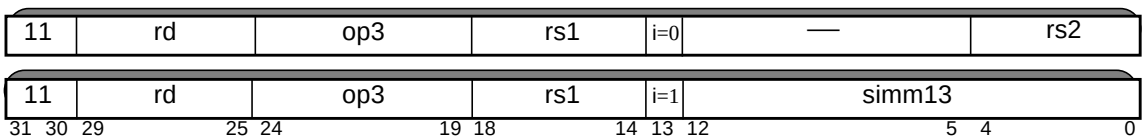
*See Also*

CALL on page 161
Bicc on page 153
BPCC on page 159

# 7.50 Load Integer

| Instruction | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| LDSB | 00 1001 | Load Signed Byte | `ldsb` | [*address*], $reg_{rd}$ | **A1** |
| LDSH | 00 1010 | Load Signed Halfword | `ldsh` | [*address*], $reg_{rd}$ | **A1** |
| LDSW | 00 1000 | Load Signed Word | `ldsw` | [*address*], $reg_{rd}$ | **A1** |
| LDUB | 00 0001 | Load Unsigned Byte | `ldub` | [*address*], $reg_{rd}$ | **A1** |
| LDUH | 00 0010 | Load Unsigned Halfword | `lduh` | [*address*], $reg_{rd}$ | **A1** |
| LDUW | 00 0000 | Load Unsigned Word | `lduw`† | [*address*], $reg_{rd}$ | **A1** |
| LDX | 00 1011 | Load Extended Word | `ldx` | [*address*], $reg_{rd}$ | **A1** |

† *synonym:* `ld`




*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 memory using the implicit ASI (see page 104). 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.

An attempt to execute a load integer instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the effective address is not halfword-aligned, an attempt to execute an LDUH or LDSH causes a *mem_address_not_aligned* exception. If the effective address is not word-aligned, an attempt to execute an LDUW or LDSW instruction causes a *mem_address_not_aligned* exception. If the effective address is not doubleword-aligned, an attempt to execute an LDX instruction causes a *mem_address_not_aligned* exception.

**V8 Compatibility Note** The SPARC V8 LD instruction was renamed LDUW in the SPARC V9 architecture. The LDSW instruction was new in the SPARC V9 architecture.

A load integer twin word (LDTW) instruction exists, but is deprecated; see *Load Integer Twin Word* on page 267 for details.

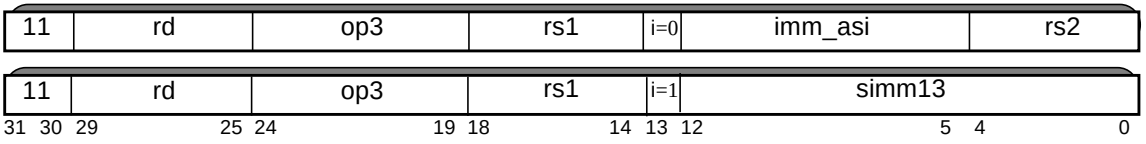
*Exceptions*

*illegal_instruction*
*mem_address_not_aligned* (all except LDSB, LDUB)
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

# 7.51 Load Integer from Alternate Space

| Instruction | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| LDSBA$^{P_{ASI}}$ | 01 1001 | Load Signed Byte from Alternate Space | `ldsba` [*regaddr*] *imm_asi*, *reg*$_{rd}$<br>`ldsba` [*reg_plus_imm*] `%asi`, *reg*$_{rd}$ | | **A1** |
| LDSHA$^{P_{ASI}}$ | 01 1010 | Load Signed Halfword from Alternate Space | `ldsha` [*regaddr*] *imm_asi*, *reg*$_{rd}$<br>`ldsha` [*reg_plus_imm*] `%asi`, *reg*$_{rd}$ | | **A1** |
| LDSWA$^{P_{ASI}}$ | 01 1000 | Load Signed Word from Alternate Space | `ldswa` [*regaddr*] *imm_asi*, *reg*$_{rd}$<br>`ldswa` [*reg_plus_imm*] `%asi`, *reg*$_{rd}$ | | **A1** |
| LDUBA$^{P_{ASI}}$ | 01 0001 | Load Unsigned Byte from Alternate Space | `lduba` [*regaddr*] *imm_asi*, *reg*$_{rd}$<br>`lduba` [*reg_plus_imm*] `%asi`, *reg*$_{rd}$ | | **A1** |
| LDUHA$^{P_{ASI}}$ | 01 0010 | Load Unsigned Halfword from Alternate Space | `lduha` [*regaddr*] *imm_asi*, *reg*$_{rd}$<br>`lduha` [*reg_plus_imm*] `%asi`, *reg*$_{rd}$ | | **A1** |
| LDUWA$^{P_{ASI}}$ | 01 0000 | Load Unsigned Word from Alternate Space | `lduwa`† [*regaddr*] *imm_asi*, *reg*$_{rd}$<br>`lduwa` [*reg_plus_imm*] `%asi`, *reg*$_{rd}$ | | **A1** |
| LDXA$^{P_{ASI}}$ | 01 1011 | Load Extended Word from Alternate Space | `ldxa` [*regaddr*] *imm_asi*, *reg*$_{rd}$<br>`ldxa` [*reg_plus_imm*] `%asi`, *reg*$_{rd}$ | | **A1** |

† *synonym:* `lda`




*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.

A load integer twin word from alternate space (LDTWA) instruction exists, but is deprecated; see *Load Integer Twin Word from Alternate Space* on page 269 for details.

An attempt to execute a load integer from alternate space instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the effective address is not halfword-aligned, an attempt to execute an LDUHA or LDSHA instruction causes a *mem_address_not_aligned* exception. If the effective address is not word-aligned, an attempt to execute an LDUWA or LDSWA instruction causes a *mem_address_not_aligned* exception. If the effective address is not doubleword-aligned, an attempt to execute an LDXA instruction causes a *mem_address_not_aligned* exception.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, these instructions cause a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, these instructions cause a *privileged_action* exception.

LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

| ASIs valid for LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA | |
|---|---|
| ASI_AS_IF_PRIV_PRIMARY | ASI_AS_IF_PRIV_PRIMARY_LITTLE |
| ASI_AS_IF_PRIV_SECONDARY | ASI_AS_IF_PRIV_SECONDARY_LITTLE |
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_REAL_IO | ASI_REAL_IO_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |
| ASI_PRIMARY_NO_FAULT | ASI_PRIMARY_NO_FAULT_LITTLE |
| ASI_SECONDARY_NO_FAULT | ASI_SECONDARY_NO_FAULT_LITTLE |

LDXA can be used with any ASI (including, but not limited to, the above list), unless it either (a) violates the privilege mode rules described for the *privileged_action* exception above or (b) is used with any of the following ASIs, which causes a *DAE_invalid_asi* exception.

| ASIs invalid for LDXA (cause *DAE_invalid_asi* exception) | |
|---|---|
| $22_{16}$ (ASI_TWINX_AIUP) | $2A_{16}$ (ASI_TWINX_AIUP_L) |
| $23_{16}$ (ASI_TWINX_AIUS) | $2B_{16}$ (ASI_TWINX_AIUS_L) |
| $26_{16}$ (ASI_TWINX_REAL) | $2E_{16}$ (ASI_TWINX_REAL_L) |
| $27_{16}$ (ASI_TWINX_N) | $2F_{16}$ (ASI_TWINX_NL) |
| ASI_BLOCK_AS_IF_USER_PRIMARY | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE |
| ASI_BLOCK_AS_IF_USER_SECONDARY | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE |
| ASI_PST8_PRIMARY | ASI_PST8_PRIMARY_LITTLE |
| ASI_PST8_SECONDARY | ASI_PST8_SECONDARY_LITTLE |
| ASI_PST16_PRIMARY | ASI_PST16_PRIMARY_LITTLE |
| ASI_PST16_SECONDARY | ASI_PST16_SECONDARY_LITTLE |

| ASIs invalid for LDXA (cause *DAE_invalid_asi* exception) | |
|---|---|
| ASI_PST32_PRIMARY | ASI_PST32_PRIMARY_LITTLE |
| ASI_PST32_SECONDARY | ASI_PST32_SECONDARY_LITTLE |
| ASI_FL8_PRIMARY | ASI_FL8_PRIMARY_LITTLE |
| ASI_FL8_SECONDARY | ASI_FL8_SECONDARY_LITTLE |
| ASI_FL16_PRIMARY | ASI_FL16_PRIMARY_LITTLE |
| ASI_FL16_SECONDARY | ASI_FL16_SECONDARY_LITTLE |
| ASI_BLOCK_COMMIT_PRIMARY | ASI_BLOCK_COMMIT_SECONDARY |
| $E2_{16}$ (ASI_TWINX_P) | $EA_{16}$ (ASI_TWINX_PL) |
| $E3_{16}$ (ASI_TWINX_S) | $EB_{16}$ (ASI_TWINX_SL) |
| ASI_BLOCK_PRIMARY | ASI_BLOCK_PRIMARY_LITTLE |
| ASI_BLOCK_SECONDARY | ASI_BLOCK_SECONDARY_LITTLE |

*Exceptions*

*mem_address_not_aligned* (all except LDSBA and LDUBA)
*privileged_action*
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nfo_page*
*DAE_side_effect_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

*See Also*

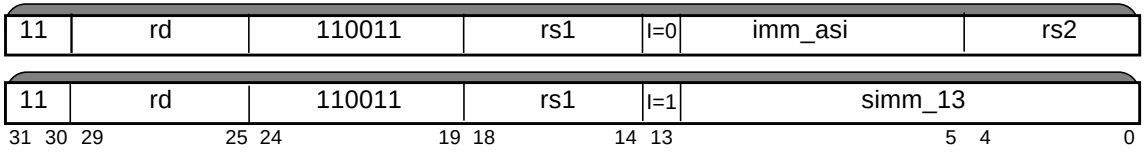
LD on page 244
STA on page 335

# 7.52 Block Load VIS 1

The LDBLOCKF instructions are deprecated and should not be used in new software. A sequence of LDX instructions should be used instead.

The LDBLOCKF instruction is intended to be a processor-specific instruction, which may or may not be implemented in future UltraSPARC Architecture implementations. Therefore, it should only be used in platform-specific dynamically-linked libraries, in hyperprivileged software, or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

| Instruc-tion | ASI Value | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| LDBLOCKF[D] | $16_{16}$ | 64-byte block load from primary address space, user privilege | `ldda` [*regaddr*] `#ASI_BLK_AIUP,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | D2 |
| LDBLOCKF[D] | $17_{16}$ | 64-byte block load from secondary address space, user privilege | `ldda` [*regaddr*] `#ASI_BLK_AIUS,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | D2 |
| LDBLOCKF[D] | $1E_{16}$ | 64-byte block load from primary address space, little-endian, user privilege | `ldda` [*regaddr*] `#ASI_BLK_AIUPL,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | D2 |
| LDBLOCKF[D] | $1F_{16}$ | 64-byte block load from secondary address space, little-endian, user privilege | `ldda` [*regaddr*] `#ASI_BLK_AIUSL,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | D2 |
| LDBLOCKF[D] | $F0_{16}$ | 64-byte block load from primary address space | `ldda` [*regaddr*] `#ASI_BLK_P,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | D2 |
| LDBLOCKF[D] | $F1_{16}$ | 64-byte block load from secondary address space | `ldda` [*regaddr*] `#ASI_BLK_S,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | D2 |
| LDBLOCKF[D] | $F8_{16}$ | 64-byte block load from primary address space, little-endian | `ldda` [*regaddr*] `#ASI_BLK_PL,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | D2 |
| LDBLOCKF[D] | $F9_{16}$ | 64-byte block load from secondary address space, little-endian | `ldda` [*regaddr*] `#ASI_BLK_SL,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | D2 |




*Description* A block load (LDBLOCKF) instruction uses one of several special block-transfer ASIs. Block transfer ASIs allow block loads to be performed accessing the same address space as normal loads. 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 64-bit (double-precision) F registers used by the instruction.

A block load instruction 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 64-bit (double-precision) destination F register.

A block load only guarantees atomicity for each 64-bit (8-byte) portion of the 64 bytes it accesses.

The block load instruction is intended to support fast block-copy operations.

**Programming Note** LDBLOCKF is intended to be a processor-specific instruction (see the warning at the top of page 249). If LDBLOCKF *must* be used in software intended to be portable across current and previous processor implementations, then it must be coded to work in the face of any implementation variation that is permitted by implementation dependency #410-S10, described below.

**IMPL. DEP. #410-S10**: The following aspects of the behavior of block load (LDBLOCKF) instructions are implementation dependent:

- What memory ordering model is used by LDBLOCKF (LDBLOCKF is not required to follow TSO memory ordering)
- Whether LDBLOCKF follows memory ordering with respect to stores (including block stores), including whether the virtual processor detects read-after-write and write-after-read hazards to overlapping addresses
- Whether LDBLOCKF appears to execute out of order, or follow LoadLoad ordering (with respect to older loads, younger loads, and other LDBLOCKFs)
- Whether LDBLOCKF follows register-dependency interlocks, as do ordinary load instructions
- Whether *VA_watchpoint* and *PA_watchpoint* exceptions are recognized on accesses to all 64 bytes of a LDBLOCKF (the recommended behavior), or only on the first eight bytes
- Whether the MMU ignores the side-effect bit (TTE.e) for LDBLOCKF accesses

**Programming Note** If ordering with respect to earlier stores is important (for example, a block load that overlaps a previous store) and read-after-write hazards are not detected, there must be a MEMBAR #StoreLoad instruction between earlier stores and a block load.

If ordering with respect to later stores is important, there must be a MEMBAR #LoadStore instruction between a block load and subsequent stores.

If LoadLoad ordering with respect to older or younger loads or other block load instructions is important and is not provided by an implementation, an intervening MEMBAR #LoadLoad is required.

For further restrictions on the behavior of the block load instruction, see implementation-specific processor documentation.

**Implementation Note** In all UltraSPARC Architecture implementations, the MMU ignores the side-effect bit (TTE.e) for LDBLOCKF accesses (impl. dep. #410-S10).

**Exceptions.** An *illegal_instruction* exception occurs if LDBLOCKF's floating-point destination registers are not aligned on an eight-double-precision register boundary.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an LDBLOCKF instruction causes an *fp_disabled* exception.

If the least significant 6 bits of the effective memory address in an LDBLOCKF instruction are nonzero, a *mem_address_not_aligned* exception occurs.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0 (ASIs $16_{16}$, $17_{16}$, $1E_{16}$, and $1F_{16}$), LDBLOCKF causes a *privileged_action* exception.

An access caused by LDBLOCKF may trigger a *VA_watchpoint* or *PA_watchpoint* exception (impl. dep. #410-S10).

An attempted access by an LDBLOCKF instruction to noncacheable memory causes an a *DAE_nc_page* exception.

**Implementation Note** LDBLOCKF shares an opcode with LDDFA and LDSHORTF; it is distinguished by the ASI used.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*mem_address_not_aligned*
*privileged_action*
*VA_watchpoint* (impl. dep. #410-S10)

*DAE_privilege_violation*
*DAE_nc_page*
*DAE_nfo_page* (attempted access to Non-Faulting-Only page of memory)
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint* (impl. dep. #410-S10)
*data_access_error*

*See Also* STBLOCKF on page 338

# 7.53 Load Floating-Point Register

| Instruction | op3 | rd | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|
| LDF | 10 0000 | 0–31 | Load Floating-Point Register | `ld` | [*address*], $freg_{rd}$ | **A1** |
| LDDF | 10 0011 | ‡ | Load Double Floating-Point Register | `ldd` | [*address*], $freg_{rd}$ | **A1** |
| LDQF | 10 0010 | ‡ | Load Quad Floating-Point Register | `ldq` | [*address*], $freg_{rd}$ | **C3** |

‡ Encoded floating-point register value, as described on page 51.

*Description*

The load single floating-point instruction (LDF) copies a word from memory into 32-bit floating-point destination register $F_S$[rd].

The load doubleword floating-point instruction (LDDF) copies a word-aligned doubleword from memory into a 64-bit floating-point destination register, $F_D$[rd]. The unit of atomicity for LDDF is 4 bytes (one word).

The load quad floating-point instruction (LDQF) copies a word-aligned quadword from memory into a 128-bit floating-point destination register, $F_Q$[rd]. The unit of atomicity for LDQF is 4 bytes (one word).

These load floating-point instructions access memory using the implicit ASI (see page 104).

If i = 0, the effective address for these instructions is "R[rs1] + R[rs2]" and if i = 0, the effective address is "R[rs1] + **sign_ext**(simm13)".

**Exceptions.** An attempt to execute an LDF, LDDF, or LDQF instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an LDF, LDDF, or LDQF instruction causes an *fp_disabled* exception.

If the effective address is not word-aligned, an attempt to execute an LDF instruction causes a *mem_address_not_aligned* exception.

LDDF requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute an LDDF instruction causes an *LDDF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the LDDF instruction and return (impl. dep. #109-V9-Cs10(a)).

LDQF requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an LDQF instruction causes an *LDQF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the LDQF instruction and return (impl. dep. #111-V9-Cs10(a)).

**Programming Note** Some compilers issued sequences of single-precision loads for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9 processors, 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.

An attempt to execute an LDQF instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

**Implementation Note** Since UltraSPARC Architecture 2007 processors do not implement in hardware instructions (including LDQF) that refer to quad-precision floating-point registers, the *LDQF_mem_address_not_aligned* and *fp_exception_other* (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an *illegal_instruction* exception and subsequent trap.

**Destination Register(s) when Exception Occurs.** If a load floating-point instruction generates an exception that causes a *precise* trap, the destination floating-point register(s) remain unchanged.

**IMPL. DEP. #44-V8-Cs10(a)(1):** If a load floating-point instruction generates an exception that causes a *non-precise* trap, the contents of the destination floating-point register(s) remain unchanged or are undefined.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*LDDF_mem_address_not_aligned*
*LDQF_mem_address_not_aligned* (not used in UltraSPARC Architecture 2007)
*mem_address_not_aligned*
*fp_exception_other* (FSR.ftt = invalid_fp_register (LDQF only))
*VA_watchpoint*
*DAE_privilege_violation*

*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

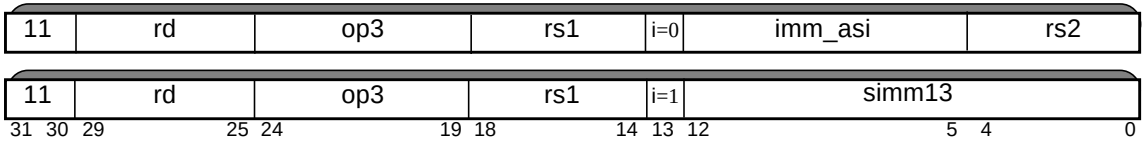
*See Also*

*Load Floating-Point from Alternate Space* on page 256
*Load Floating-Point State Register (Lower)* on page 260
*Store Floating-Point* on page 342

# 7.54 Load Floating-Point from Alternate Space

| Instruction | op3 | rd | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| LDFA$^{P_{ASI}}$ | 11 0000 | 0–31 | Load Floating-Point Register from Alternate Space | `lda` [*regaddr*] *imm_asi*, *freg*$_{rd}$<br>`lda` [*reg_plus_imm*] `%asi`, *freg*$_{rd}$ | **A1** |
| LDDFA$^{P_{ASI}}$ | 11 0011 | ‡ | Load Double Floating-Point Register from Alternate Space | `ldda` [*regaddr*] *imm_asi*, *freg*$_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi`, *freg*$_{rd}$ | **A1** |
| LDQFA$^{P_{ASI}}$ | 11 0010 | ‡ | Load Quad Floating-Point Register from Alternate Space | `ldqa` [*regaddr*] *imm_asi*, *freg*$_{rd}$<br>`ldqa` [*reg_plus_imm*] `%asi`, *freg*$_{rd}$ | **C3** |

‡ Encoded floating-point register value, as described in *Floating-Point Register Number Encoding* on page 51.




*Description*

The load single floating-point from alternate space instruction (LDFA) copies a word from memory into 32-bit floating-point destination register $F_S$[rd].

The load double floating-point from alternate space instruction (LDDFA) copies a word-aligned doubleword from memory into a 64-bit floating-point destination register, $F_D$[rd]. The unit of atomicity for LDDFA is 4 bytes (one word).

The load quad floating-point from alternate space instruction (LDQFA) copies a word-aligned quadword from memory into a 128-bit floating-point destination register, $F_Q$[rd]. The unit of atomicity for LDQFA is 4 bytes (one word).

If i = 0, these instructions contain the address space identifier (ASI) to be used for the load in the imm_asi field and the effective address for the instruction is "R[rs1] + R[rs2]". If i = 1, the ASI to be used is contained in the ASI register and the effective address for the instruction is "R[rs1] + **sign_ext(**simm13)".

**Exceptions.** If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an LDFA, LDDFA, or LDQFA instruction causes an *fp_disabled* exception.

LDFA causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

**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 causes an *LDDF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the LDDFA instruction and return (impl. dep. #109-V9-Cs10(b)).

LDQFA requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, LDQFA causes an *LDQF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the LDQFA instruction and return (impl. dep. #111-V9-Cs10(b)).

An attempt to execute an LDQFA instruction when rd{1} ≠ 0 causes an *fp_exception_other* (with FSR.ftt = invalid_fp_register) exception.

**Implementation Note** Since UltraSPARC Architecture 2007 processors do not implement in hardware instructions (including LDQFA) that refer to quad-precision floating-point registers, the *LDQF_mem_address_not_aligned* and *fp_exception_other* (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an *illegal_instruction* exception and subsequent trap.

**Programming Note** Some compilers issued sequences of single-precision loads for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9 processors, 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.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, this instruction causes a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, this instruction causes a *privileged_action* exception.

LDFA and LDQFA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

**ASIs valid for LDFA and LDQFA**

| | |
|---|---|
| ASI_AS_IF_PRIV_PRIMARY | ASI_AS_IF_PRIV_PRIMARY_LITTLE |
| ASI_AS_IF_PRIV | ASI_AS_IF_PRIV_SECONDARY_LITTLE |
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_REAL_IO | ASI_REAL_IO_LITTLE |

| ASIs valid for LDFA and LDQFA | |
|---|---|
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |
| ASI_PRIMARY_NO_FAULT | ASI_PRIMARY_NO_FAULT_LITTLE |
| ASI_SECONDARY_NO_FAULT | ASI_SECONDARY_NO_FAULT_LITTLE |

LDDFA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with the LDDFA instruction causes a *DAE_invalid_asi* exception.

| ASIs valid for LDDFA | |
|---|---|
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_REAL_IO | ASI_REAL_IO_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |
| ASI_PRIMARY_NO_FAULT | ASI_PRIMARY_NO_FAULT_LITTLE |
| ASI_SECONDARY_NO_FAULT | ASI_SECONDARY_NO_FAULT_LITTLE |

**Behavior with Block-Store-with-Commit ASIs.** ASIs $E0_{16}$ and $E1_{16}$ are only defined for use in Block Store with Commit operations (see page 338). Neither ASI $E0_{16}$ nor $E1_{16}$ should be used with LDDFA; however, if it *is* used, the LDDFA behaves as follows:

1. If an LDDFA opcode is used with an ASI of $E0_{16}$ or $E1_{16}$ and a destination register number rd is specified which is not a multiple of 8 ("misaligned" rd), an UltraSPARC Architecture 2007 virtual processor generates an *illegal_instruction* exception (impl. dep. #255-U3-Cs10).

2. **IMPL. DEP. #256-U3:** If an LDDFA opcode is used with an ASI of $E0_{16}$ or $E1_{16}$ and a memory address is specified with less than 64-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the exception generated is *DAE_invalid_asi*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned*.

3. If both rd and the memory address are correctly aligned, a *DAE_invalid_asi* exception occurs.

**Behavior with Partial Store ASIs.** ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–$CD_{16}$ are only defined for use in Partial Store operations (see page 351). None of them should be used with LDDFA; however, if any of those ASIs *is* used with LDDFA, the LDDFA behaves as follows:

1. **IMPL. DEP. #257-U3:** If an LDDFA opcode is used with an ASI of $C0_{16}$–$C5_{16}$ or $C8_{16}$–$CD_{16}$ (Partial Store ASIs, which are an illegal combination with LDDFA) and a memory address is specified with less than 8-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the generated exception is a *DAE_invalid_asi*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned* exception.

2. If the memory address is correctly aligned, the virtual processor generates a *DAE_invalid_asi*.

**Destination Register(s) when Exception Occurs.** If a load floating-point alternate instruction generates an exception that causes a precise trap, the destination floating-point register(s) remain unchanged.

**IMPL. DEP. #44-V8-Cs10(b):** If a load floating-point alternate instruction generates an exception that causes a non-precise trap, it is implementation dependent whether the contents of the destination floating-point register(s) are undefined or are guaranteed to remain unchanged.

**Implementation Note** | LDDFA shares an opcode with the LDBLOCKF and LDSHORTF instructions; it is distinguished by the ASI used.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*LDDF_mem_address_not_aligned*
*LDQF_mem_address_not_aligned* (not generated in UltraSPARC Architecture 2007)
*mem_address_not_aligned*
*fp_exception_other* (FSR.ftt = invalid_fp_register (LDQFA only))
*privileged_action*
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nfo_page*
*DAE_side_effect_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

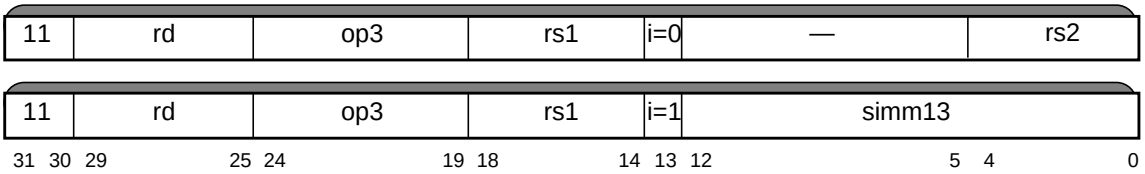
*See Also*

*Load Floating-Point Register* on page 253
*Block Load* on page 249
*Store Short Floating-Point* on page 355
*Store Floating-Point into Alternate Space* on page 345

# 7.55 Load Floating-Point State Register (Lower)

The LDFSR instruction is deprecated and should not be used in new software. The LDXFSR instruction should be used instead.

| Opcode | op3 | rd | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| LDFSR[D] | 10 0001 | 0 | Load Floating-Point State Register (Lower) | `ld` [*address*], `%fsr` | **D2** |
| | 10 0001 | 1-31 | (see page 275) | | |




*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 more-significant 32 bits of FSR are unaffected by LDFSR. LDFSR does not alter the ver, ftt, qne, reserved, or unimplemented (for example, ns) fields of FSR (see page 60).

**Programming Note** For future compatibility, software should only issue an LDFSR instruction with a zero value (or a value previously read from the same field) in any reserved field of FSR.

LDFSR accesses memory using the implicit ASI (see page 117).

An attempt to execute an LDFSR instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an LDFSR instruction causes an *fp_disabled* exception.

LDFSR causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

**V8 Compatibility Note** The SPARC V9 architecture supports two different instructions to load the FSR: the (deprecated) 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.

# LDFSR (Deprecated)

**Implementation Note** — LDFSR shares an opcode with the LDXFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = $11_2$, op3 = $10\ 0001_2$ opcode with an invalid rd value causes an *illegal_instruction* exception.

*Exceptions*

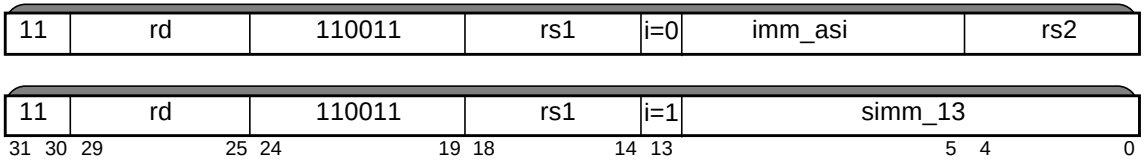
*illegal_instruction*
*fp_disabled*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

*See Also*

*Load Floating-Point Register* on page 253
*Load Floating-Point State Register* on page 275
*Store Floating-Point* on page 342

# 7.56 Short Floating-Point Load VIS 1

| Instruction | ASI Value | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| LDSHORTF | $D0_{16}$ | 8-bit load from primary address space | `ldda` [*regaddr*] `#ASI_FL8_P,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | **B1** |
| LDSHORTF | $D1_{16}$ | 8-bit load from secondary address space | `ldda` [*regaddr*] `#ASI_FL8_S,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | **B1** |
| LDSHORTF | $D8_{16}$ | 8-bit load from primary address space, little-endian | `ldda` [*regaddr*] `#ASI_FL8_PL,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | **B1** |
| LDSHORTF | $D9_{16}$ | 8-bit load from secondary address space, little-endian | `ldda` [*regaddr*] `#ASI_FL8_SL,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | **B1** |
| LDSHORTF | $D2_{16}$ | 16-bit load from primary address space | `ldda` [*regaddr*] `#ASI_FL16_P,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | **B1** |
| LDSHORTF | $D3_{16}$ | 16-bit load from secondary address space | `ldda` [*regaddr*] `#ASI_FL16_S,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | **B1** |
| LDSHORTF | $DA_{16}$ | 16-bit load from primary address space, little-endian | `ldda` [*regaddr*] `#ASI_FL16_PL,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | **B1** |
| LDSHORTF | $DB_{16}$ | 16-bit load from secondary address space, little-endian | `ldda` [*regaddr*] `#ASI_FL16_SL,` $freg_{rd}$<br>`ldda` [*reg_plus_imm*] `%asi,` $freg_{rd}$ | **B1** |




*Description* Short floating-point load instructions allow an 8- or 16-bit value to be loaded from memory into a 64-bit floating-point register.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an LDSHORTF instruction causes an *fp_disabled* exception.

An 8-bit load places the loaded value in the least significant byte of $F_D$[rd] and zeroes in the most-significant three bytes of $F_D$[rd]. An 8-bit LDSHORTF can be performed from an arbitrary byte address.

A 16-bit load places the loaded value in the least significant halfword of $F_D$[rd] and zeroes in the more-significant halfword of $F_D$[rd]. A 16-bit LDSHORTF from an address that is not halfword-aligned (an odd address) causes a *mem_address_not_aligned* exception.

# LDSHORTF

Little-endian ASIs transfer data in little-endian format from memory; otherwise, memory is assumed to be in big-endian byte order.

**Programming Note** | LDSHORTF is typically used with the FALIGNDATA instruction (see *Align Address* on page 146) to assemble or store 64 bits from noncontiguous components.

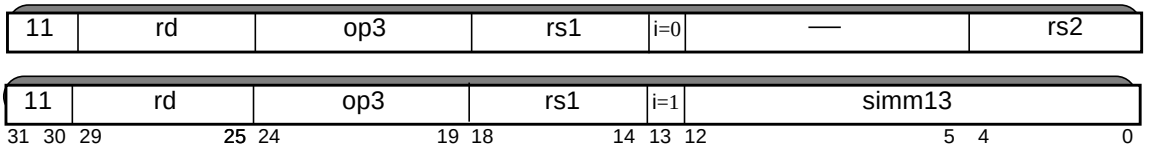
**Implementation Note** | LDSHORTF shares an opcode with the LDBLOCKF and LDDFA instructions; it is distinguished by the ASI used.

*Exceptions*

*fp_disabled*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

# 7.57 Load-Store Unsigned Byte

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| LDSTUB | 00 1101 | Load-Store Unsigned Byte | `ldstub` [*address*], $reg_{rd}$ | **A1** |




*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.

LDSTUB accesses memory using the implicit ASI (see page 104). The effective address for this instruction 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-V9).

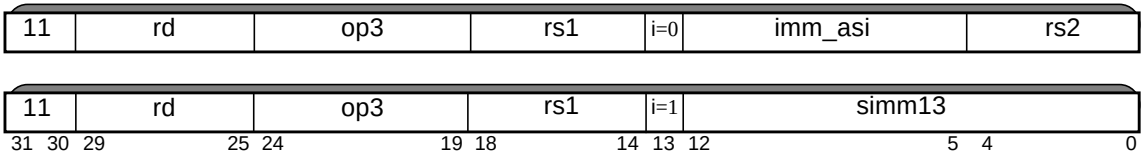
An attempt to execute an LDSTUB instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions*

*illegal_instruction*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nc_page*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

# 7.58 Load-Store Unsigned Byte to Alternate Space

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| LDSTUBA$^{P_{ASI}}$ | 01 1101 | Load-Store Unsigned Byte into Alternate Space | `ldstuba` [*regaddr*] *imm_asi*, $reg_{rd}$<br>`ldstuba` [*reg_plus_imm*] `%asi`, $reg_{rd}$ | **A1** |




*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.

If i = 0, LDSTUBA contains the address space identifier (ASI) to be used for the load in the imm_asi field. If i = 1, the ASI is found in the ASI register. In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, this instruction causes a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, this instruction causes a *privileged_action* exception.

LDSTUBA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *DAE_invalid_asi* exception.

**ASIs valid for LDSTUBA**

| | |
|---|---|
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |

*Exceptions*

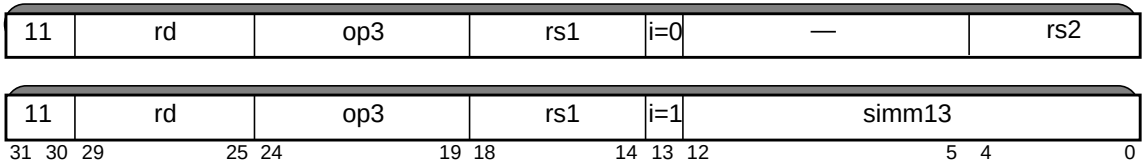
*privileged_action*
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nc_page*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

# 7.59 Load Integer Twin Word

The LDTW instruction is deprecated and should not be used in new software. It is provided only for compatibility with previous versions of the architecture.The LDX instruction should be used instead.

| Instruction | op3 | Operation | Assembly Language Syntax † | Class |
|---|---|---|---|---|
| LDTW$^{D}$ | 00 0011 | Load Integer Twin Word | `ldtw` [*address*], $reg_{rd}$ | D2 |

† The original assembly language syntax for this instruction used an "`ldd`" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "`ldtw`" mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original "`ldd`" mnemonic.




*Description*

The load integer twin word instruction (LDTW) copies two words (with doubleword alignment) from memory into a pair of R registers. The word at the effective memory address is copied into the least significant 32 bits of the even-numbered R register. The word at the effective memory address + 4 is copied into the least significant 32 bits of the following odd-numbered R register. The most significant 32 bits of both the even-numbered and odd-numbered R registers are zero-filled.

**Note** — Execution of an LDTW instruction with rd = 0 modifies only R[1].

Load integer twin word instructions access memory using the implicit ASI (see page 104). If i = 0, the effective address for these instructions is "R[rs1] + R[rs2]" and if i = 0, the effective address is "R[rs1] + **sign_ext**(simm13)".

With respect to little endian memory, an LDTW instruction behaves as if it comprises two 32-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

**IMPL. DEP. #107-V9a:** It is implementation dependent whether LDTW is implemented in hardware. If not, an attempt to execute an LDTW instruction will cause an *unimplemented_LDTW* exception.

**Programming Note** — LDTW is provided for compatibility with existing SPARC V8 software. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties.

# LDTW (Deprecated)

**SPARC V9 Compatibility Note** | LDTW was (inaccurately) named LDD in the SPARC V8 and SPARC V9 specifications. It does not load a doubleword; it loads two words (into two registers), and has been renamed accordingly.

The least significant bit of the rd field in an LDTW instruction is unused and should always be set to 0 by software. An attempt to execute an LDTW instruction that refers to a misaligned (odd-numbered) destination register causes an *illegal_instruction* exception.

An attempt to execute an LDTW instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the effective address is not doubleword-aligned, an attempt to execute an LDTW instruction causes a *mem_address_not_aligned* exception.

A successful LDTW instruction operates atomically.

*Exceptions*

*unimplemented_LDTW* (not used in UltraSPARC Architecture 2007)
*illegal_instruction*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

*See Also*

LDW/LDX on page 244
STTW on page 357

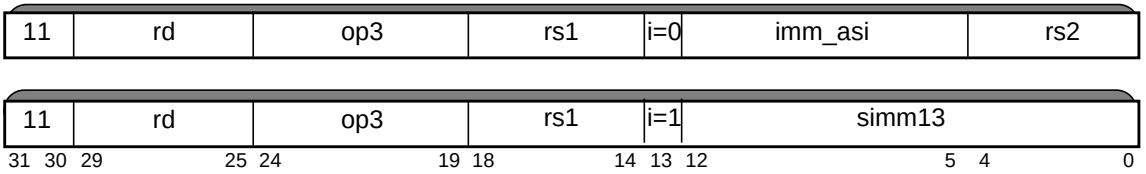
# 7.60 Load Integer Twin Word from Alternate Space

The LDTWA instruction is deprecated and should not be used in new software. The LDXA instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| LDTWA$^{D, P_{ASI}}$ | 01 0011 | Load Integer Twin Word from Alternate Space | `ldtwa` [*regaddr*] *imm_asi*, $reg_{rd}$<br>`ldtwa` [*reg_plus_imm*] `%asi`, $reg_{rd}$ | **D2, Y3**‡ |

† The original assembly language syntax for this instruction used an "`ldda`" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "`ldtwa`" mnemonic for this instruction. In the meantime, some assemblers may only recognize the original "`ldda`" mnemonic.

‡ **Y3** for restricted ASIs ($00_{16}$-$7F_{16}$); **D2** for unrestricted ASIs ($80_{16}$-$FF_{16}$)




*Description*

The load integer twin word from alternate space instruction (LDTWA) copies two 32-bit words from memory (with doubleword memory alignment) into a pair of R registers. The word at the effective memory address is copied into the least significant 32 bits of the even-numbered R register. The word at the effective memory address + 4 is copied into the least significant 32 bits of the following odd-numbered R register. The most significant 32 bits of both the even-numbered and odd-numbered R registers are zero-filled.

**Note** | Execution of an LDTWA instruction with rd = 0 modifies only R[1].

If i = 0, the LDTWA instruction contains the address space identifier (ASI) to be used for the load in its imm_asi field and the effective address for the instruction is "R[rs1] + R[rs2]". If i = 1, the ASI to be used is contained in the ASI register and the effective address for the instruction is "R[rs1] + **sign_ext(** simm13)".

With respect to little endian memory, an LDTWA 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.

# LDTWA (Deprecated)

**IMPL. DEP. #107-V9b:** It is implementation dependent whether LDTWA is implemented in hardware. If not, an attempt to execute an LDTWA instruction will cause an *unimplemented_LDTW* exception so that it can be emulated.

**Programming Note**

LDTWA is provided for compatibility with existing SPARC V8 software. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties.

If LDTWA is emulated in software, an LDXA instruction instruction should be used for the memory access in the emulation code in order to preserve atomicity.

**SPARC V9 Compatibility Note**

LDTWA was (inaccurately) named LDDA in the SPARC V8 and SPARC V9 specifications.

The least significant bit of the rd field in an LDTWA instruction is unused and should always be set to 0 by software. An attempt to execute an LDTWA instruction that refers to a misaligned (odd-numbered) destination register causes an *illegal_instruction* exception.

If the effective address is not doubleword-aligned, an attempt to execute an LDTWA instruction causes a *mem_address_not_aligned* exception.

A successful LDTWA instruction operates atomically.

LDTWA causes a *mem_address_not_aligned* exception if the address is not doubleword-aligned.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, these instructions cause a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, these instructions cause a *privileged_action* exception.

LDTWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *DAE_invalid_asi* exception (impl. dep. #300-U4-Cs10).

**ASIs valid for LDTWA**

| | |
|---|---|
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_REAL_IO | ASI_REAL_IO_LITTLE |
| $22_{16}$‡ (ASI_TWINX_AIUP) | $2A_{16}$‡ (ASI_TWINX_AIUP_L) |
| $23_{16}$‡ (ASI_TWINX_AIUS) | $2B_{16}$‡ (ASI_TWINX_AIUS_L) |
| $26_{16}$‡ (ASI_TWINX_REAL) | $2E_{16}$‡ (ASI_TWINX_REAL_L) |
| $27_{16}$‡ (ASI_TWINX_N) | $2F_{16}$‡ (ASI_TWINX_NL) |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |

# LDTWA (Deprecated)

**ASIs valid for LDTWA**

| | |
|---|---|
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |
| ASI_PRIMARY_NO_FAULT | ASI_PRIMARY_NO_FAULT_LITTLE |
| ASI_SECONDARY_NO_FAULT | ASI_SECONDARY_NO_FAULT_LITTLE |
| $E2_{16}$‡ (ASI_TWINX_P) | $EA_{16}$‡ (ASI_TWINX_PL) |
| $E3_{16}$‡ (ASI_TWINX_S) | $EB_{16}$‡ (ASI_TWINX_SL) |

‡ If this ASI is used with the opcode for LDTWA and i = 0, the LDTXA instruction is executed instead of LDTWA. For behavior of LDTXA, see *Load Integer Twin Extended Word from Alternate Space* on page 272. If this ASI is used with the opcode for LDTWA and i = 1, a *DAE_invalid_asi* exception occurs.

**Programming Note** Nontranslating ASIs (see page 429) should only be accessed using LDXA (not LDTWA) instructions. If an LDTWA referencing a nontranslating ASI is executed, per the above table, it generates a *DAE_invalid_asi* exception (impl. dep. #300-U4-Cs10).

**Implementation Note** The deprecated instruction LDTWA shares an opcode with LDTXA. LDTXA is *not* deprecated and has different address alignment requirements than LDTWA. See *Load Integer Twin Extended Word from Alternate Space* on page 272.

*Exceptions*

*unimplemented_LDTW* (not used in UltraSPARC Architecture 2007)
*illegal_instruction*
*mem_address_not_aligned*
*privileged_action*
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nfo_page*
*DAE_side_effect_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

*See Also*

LDWA/LDXA on page 246
LDTXA on page 272
STTWA on page 359

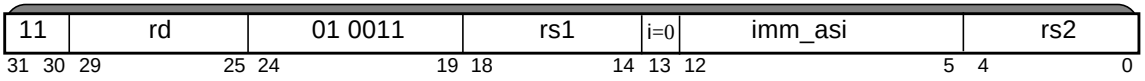
# 7.61 Load Integer Twin Extended Word from Alternate Space VIS 2+

The LDTXA instructions are not guaranteed to be implemented on all UltraSPARC Architecture implementations. Therefore, they should only be used in platform-specific dynamically-linked libraries, in hyperprivileged software, or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

| Instruction | ASI Value | Operation | Assembly Language Syntax † | Class |
|---|---|---|---|---|
| LDTXA$^N$ | $22_{16}$ | Load Integer Twin Extended Word, as if user (nonprivileged), Primary address space | `ldtxa` [*regaddr*] `#ASI_TWINX_AIUP,` $reg_{rd}$ | **N1** |
| | $23_{16}$ | Load Integer Twin Extended Word, as if user (nonprivileged), Secondary address space | `ldtxa` [*regaddr*] `#ASI_TWINX_AIUS,` $reg_{rd}$ | **N1** |
| | $26_{16}$ | Load Integer Twin Extended Word, real address | `ldtxa` [*regaddr*] `#ASI_TWINX_REAL,` $reg_{rd}$ | **N1** |
| | $27_{16}$ | Load Integer Twin Extended Word, nucleus context | `ldtxa` [*regaddr*] `#ASI_TWINX_N,` $reg_{rd}$ | **N1** |
| | $2A_{16}$ | Load Integer Twin Extended Word, as if user (nonprivileged), Primary address space, little endian | `ldtxa` [*regaddr*] `#ASI_TWINX_AIUP_L,` $reg_{rd}$ | **N1** |
| | $2B_{16}$ | Load Integer Twin Extended Word, as if user (nonprivileged), Secondary address space, little endian | `ldtxa` [*regaddr*] `#ASI_TWINX_AIUS_L,` $reg_{rd}$ | **N1** |
| | $2E_{16}$ | Load Integer Twin Extended Word, real address, little endian | `ldtxa` [*regaddr*] `#ASI_TWINX_REAL_L,` $reg_{rd}$ | **N1** |
| | $2F_{16}$ | Load Integer Twin Extended Word, nucleus context, little-endian | `ldtxa` [*regaddr*] `#ASI_TWINX_NL,` $reg_{rd}$ | **N1** |
| LDTXA$^N$ | $E2_{16}$ | Load Integer Twin Extended Word, Primary address space | `ldtxa` [*regaddr*] `#ASI_TWINX_P,` $reg_{rd}$ | **N1** |
| | $E3_{16}$ | Load Integer Twin Extended Word, Secondary address space | `ldtxa` [*regaddr*] `#ASI_TWINX_S,` $reg_{rd}$ | **N1** |
| | $EA_{16}$ | Load Integer Twin Extended Word, Primary address space, little endian | `ldtxa` [*regaddr*] `#ASI_TWINX_PL,` $reg_{rd}$ | **N1** |
| | $EB_{16}$ | Load Integer Twin Extended Word, Secondary address space, little-endian | `ldtxa` [*regaddr*] `#ASI_TWINX_SL,` $reg_{rd}$ | **N1** |

† The original assembly language syntax for these instructions used the "`ldda`" instruction mnemonic. That syntax is now deprecated. Over time, assemblers will support the new "`ldtxa`" mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original "`ldda`" mnemonic.

# LDTXA




*Description* ASIs $26_{16}$, $2E_{16}$, $E2_{16}$, $E3_{16}$, $F0_{16}$, and $F1_{16}$ are used with the LDTXA instruction to atomically read a 128-bit data item into a pair of 64-bit R registers (a "twin extended word"). The data are placed in an even/odd pair of 64-bit registers. The lowest-address 64 bits are placed in the even-numbered register; the highest-address 64 bits are placed in the odd-numbered register.

**Note** | Execution of an LDTXA instruction with rd = 0 modifies only R[1].

ASIs $E2_{16}$, $E3_{16}$, $F0_{16}$, and $F1_{16}$ perform an access using a virtual address, while ASIs $26_{16}$ and $2E_{16}$ use a real address.

An LDTXA instruction that performs a little-endian access behaves as if it comprises two 64-bit loads (performed atomically), each of which is byte-swapped independently before being written into its respective destination register.

**Exceptions.** An attempt to execute an LDTXA instruction with an odd-numbered destination register (rd{0} = 1) causes an *illegal_instruction* exception.

An attempt to execute an LDTXA instruction with an effective memory address that is not aligned on a 16-byte boundary causes a *mem_address_not_aligned* exception.

**IMPL. DEP. #413-S10**: It is implementation dependent whether *VA_watchpoint* and *PA_watchpoint* exceptions are recognized on accesses to all 16 bytes of a LDTXA instruction (the recommended behavior) or only on accesses to the first 8 bytes.

An attempted access by an LDTXA instruction to noncacheable memory causes an a *DAE_nc_page* exception (impl. dep. #306-U4-Cs10).

**Programming Note** | A key use for this instruction is to read a full TTE entry (128 bits, tag and data) in a TSB directly, without using software interlocks. The "real address" variants can perform the access using a real address, bypassing the VA-to-RA translation.

**Programming Note** | In hyperprivileged mode, an access to ASI $E2_{16}$, $E3_{16}$, $F0_{16}$, or $F1_{16}$ is performed using physical (not virtual) addressing.

The virtual processor MMU does not provide virtual-to-real translation for ASIs $26_{16}$ and $2E_{16}$; the effective address provided with either of those ASIs is interpreted directly as a real address.

**Compatibility Note** | ASIs $27_{16}$, $2F_{16}$, $26_{16}$, and $2E_{16}$ are now standard ASIs that replace (respectively) ASIs $24_{16}$, $2C_{16}$, $34_{16}$, and $3C_{16}$ that were supported in some previous UltraSPARC implementations.

A *mem_address_not_aligned* trap is taken if the access is not aligned on a 128-byte boundary.

**Implementation Note** LDTXA shares an opcode with the "i = 0" variant of the (deprecated) LDTWA instruction; they are differentiated by the combination of the value of "i" and the ASI used in the instruction. See *Load Integer Twin Word from Alternate Space* on page 269.

*Exceptions*

*illegal_instruction*
*mem_address_not_aligned*
*privileged_action*
*VA_watchpoint* (impl. dep. #413-S10)
*DAE_nc_page*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint* (impl. dep. #413-S10)
*data_access_error*

*See Also* LDTWA on page 269

# 7.62 Load Floating-Point State Register

| Instruction | op3 | rd | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| | 10 0001 | 0 | (see page 260) | | |
| LDXFSR | 10 0001 | 1 | Load Floating-Point State Register | `ldx` [*address*], `%fsr` | **A1** |
| — | 10 0001 | 2–31 | *Reserved* | | |

*Description* A 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.

LDXFSR does not alter the ver, ftt, qne, reserved, or unimplemented (for example, ns) fields of FSR (see page 60).

**Programming Note** For future compatibility, software should only issue an LDXFSR instruction with a zero value (or a value previously read from the same field) written into any reserved field of FSR.

LDXFSR accesses memory using the implicit ASI (see page 104).

If i = 0, the effective address for these instructions is "R[rs1] + R[rs2]" and if i = 0, the effective address is "R[rs1] + **sign_ext**( simm13)".

**Exceptions.** An attempt to execute an instruction encoded as op = 2 and op3 = $21_{16}$ when any of the following conditions exist causes an *illegal_instruction* exception:

- i = 0 and instruction bits 12:5 are nonzero
- (rd > 1)

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an LDXFSR instruction causes an *fp_disabled* exception.

If the effective address is not doubleword-aligned, an attempt to execute an LDXFSR instruction causes a *mem_address_not_aligned* exception.

**Destination Register(s) when Exception Occurs.** If a load floating-point state register instruction generates an exception that causes a *precise* trap, the destination register (FSR) remains unchanged.

**IMPL. DEP. #44-V8-Cs10(a)(2):** If an LDXFSR instruction generates an exception that causes a *non-precise* trap, it is implementation dependent whether the contents of the destination register (FSR) is undefined or is guaranteed to remain unchanged.

**Implementation Note** LDXFSR shares an opcode with the (deprecated) LDFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = $11_2$, op3 = $10\ 0001_2$ opcode with an invalid rd value causes an *illegal_instruction* exception.

*Exceptions*

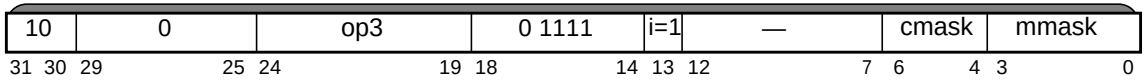
*illegal_instruction*
*fp_disabled*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*PA_watchpoint*
*data_access_error*

*See Also*

*Load Floating-Point Register* on page 253
*Load Floating-Point State Register (Lower)* on page 260
*Store Floating-Point State Register* on page 362

# 7.63 Memory Barrier

| Instruction | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| MEMBAR | 10 1000 | Memory Barrier | `membar` | *membar_mask* | **A1** |




*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 CASX[A] 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 7-7 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 7-7** MEMBAR mmask Encodings

| Mask Bit | Assembly Language 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. |
| 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 7-8, 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 7-8** MEMBAR cmask Encodings

| Mask Bit | Function | Assembly Language 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. |

A MEMBAR instruction with both mmask = 0 and cmask = 0 is functionally a NOP.

For information on the use of MEMBAR, see *Memory Ordering and Synchronization* on page 423 and *Programming with the Memory Models* contained in the separate volume *UltraSPARC Architecture Application Notes*. For additional information about the memory models themselves, see Chapter 9, *Memory*.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

**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.

An attempt to execute a MEMBAR instruction when instruction bits 12:7 are nonzero causes an *illegal_instruction* exception.

**Implementation Note** MEMBAR shares an opcode withRDasr; it is distinguished by rs1 = 15, rd = 0, i = 1, and bit 12 = 0.

## 7.63.1 Memory Synchronization

The UltraSPARC Architecture provides some level of software control over memory synchronization, through use of the MEMBAR and FLUSH instructions for explicit control of memory ordering in program execution.

**IMPL. DEP. #412-S10**: An UltraSPARC Architecture implementation may define the operation of each MEMBAR variant in any manner that provides the required semantics.

**Implementation Note** For an UltraSPARC Architecture virtual processor that only provides TSO memory ordering semantics, three of the ordering MEMBARs would normally be implemented as NOPs. TABLE 7-9 shows an acceptable implementation of MEMBAR for a TSO-only UltraSPARC Architecture implementation.

**TABLE 7-9** MEMBAR Semantics for TSO-only implementation

| MEMBAR variant | Preferred Implementation |
|---|---|
| `#StoreStore` | NOP |
| `#LoadStore` | NOP |
| `#StoreLoad` | `#Sync` |
| `#LoadLoad` | NOP |
| `#Sync` | `#Sync` |
| `#MemIssue` | `#Sync` |
| `#Lookaside` | `#Sync` |

If an UltraSPARC Architecture implementation provides a less restrictive memory model than TSO (for example, RMO), the implementation of the MEMBAR variants may be different. See implementation-specific documentation for details.

## 7.63.2 Synchronization of the Virtual Processor

*Synchronization* of a virtual processor forces all outstanding instructions to be completed and any associated hardware errors to be detected and reported before any instruction after the synchronizing instruction is issued.

Synchronization can be explicitly caused by executing a synchronizing MEMBAR instruction (MEMBAR #Sync) or by executing an LDXA/STXA/LDDFA/STDFA instruction with an ASI that forces synchronization.

During synchronization, if a disrupting trap condition due to a hardware error is detected and external interrupts are enabled, the disrupting trap will occur before the instruction after the synchronizing instruction is executed. In this case, the PC value saved in TPC during trap entry will be the address of the instruction after the synchronizing instruction.

**Programming Note** Completion of a MEMBAR #Sync instruction does *not* guarantee that data previously stored has been written all the way out to external memory (that is, that cache writebacks to external memory have completed). Software cannot rely on that behavior. There is no mechanism in the UltraSPARC Architecture that allows software to wait for all previous stores to be written to external memory (that is, for cache writebacks to completely drain).

## 7.63.3 TSO Ordering Rules affecting Use of MEMBAR

For detailed rules on use of MEMBAR to enable software to adhere to the ordering rules on a virtual processor running with the TSO memory model, refer to *TSO Ordering Rules* on page 421.

*Exceptions* *illegal_instruction*

# 7.64 Move Integer Register on Condition (MOVcc)

*For Integer Condition Codes*

| Instruction | op3 | cond | Operation | icc / xcc Test | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|---|
| MOVA | 10 1100 | 1000 | Move Always | 1 | `mova` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVN | 10 1100 | 0000 | Move Never | 0 | `movn` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVNE | 10 1100 | 1001 | Move if Not Equal | **not** Z | `movne`† | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVE | 10 1100 | 0001 | Move if Equal | Z | `move`‡ | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVG | 10 1100 | 1010 | Move if Greater | **not** (Z **or** N **xor** V)) | `movg` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVLE | 10 1100 | 0010 | Move if Less or Equal | Z **or** (N **xor** V) | `movle` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVGE | 10 1100 | 1011 | Move if Greater or Equal | **not** (N **xor** V) | `movge` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVL | 10 1100 | 0011 | Move if Less | N **xor** V | `movl` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVGU | 10 1100 | 1100 | Move if Greater, Unsigned | **not** (C **or** Z) | `movgu` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVLEU | 10 1100 | 0100 | Move if Less or Equal, Unsigned | (C **or** Z) | `movleu` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVCC | 10 1100 | 1101 | Move if Carry Clear (Greater or Equal, Unsigned) | **not** C | `movcc`◊ | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVCS | 10 1100 | 0101 | Move if Carry Set (Less than, Unsigned) | C | `movcs`▽ | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVPOS | 10 1100 | 1110 | Move if Positive | **not** N | `movpos` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVNEG | 10 1100 | 0110 | Move if Negative | N | `movneg` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVVC | 10 1100 | 1111 | Move if Overflow Clear | **not** V | `movvc` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVVS | 10 1100 | 0111 | Move if Overflow Set | V | `movvs` | *i_or_x_cc*, *reg_or_imm11*, $reg_{rd}$ | **A1** |

† *synonym:* `movnz` ‡ *synonym:* `movz` ◊ *synonym:* `movgeu` ▽ *synonym:* `movlu`

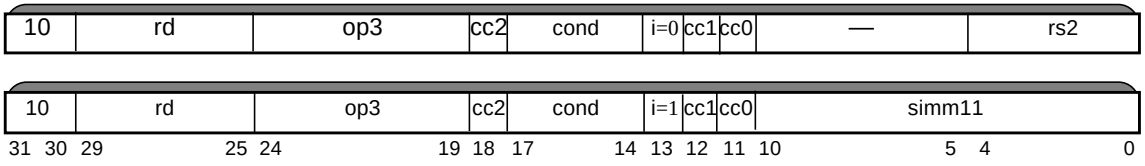
**Programming Note** | In assembly language, to select the appropriate condition code, include `%icc` or `%xcc` before the *reg_or_imm11* field.

*For Floating-Point Condition Codes*

| Instruction | op3 | cond | Operation | fcc Test | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|
| MOVFA | 10 1100 | 1000 | Move Always | 1 | `mova` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFN | 10 1100 | 0000 | Move Never | 0 | `movn` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFU | 10 1100 | 0111 | Move if Unordered | U | `movu` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFG | 10 1100 | 0110 | Move if Greater | G | `movg` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFUG | 10 1100 | 0101 | Move if Unordered or Greater | G **or** U | `movug` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFL | 10 1100 | 0100 | Move if Less | L | `movl` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFUL | 10 1100 | 0011 | Move if Unordered or Less | L **or** U | `movul` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFLG | 10 1100 | 0010 | Move if Less or Greater | L **or** G | `movlg` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFNE | 10 1100 | 0001 | Move if Not Equal | L **or** G **or** U | `movne`† `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFE | 10 1100 | 1001 | Move if Equal | E | `move`‡ `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFUE | 10 1100 | 1010 | Move if Unordered or Equal | E **or** U | `movue` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFGE | 10 1100 | 1011 | Move if Greater or Equal | E **or** G | `movge` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFUGE | 10 1100 | 1100 | Move if Unordered or Greater or Equal | E **or** G **or** U | `movuge` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFLE | 10 1100 | 1101 | Move if Less or Equal | E **or** L | `movle` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFULE | 10 1100 | 1110 | Move if Unordered or Less or Equal | E **or** L **or** U | `movule` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |
| MOVFO | 10 1100 | 1111 | Move if Ordered | E **or** L **or** G | `movo` `%fccn`, *reg_or_imm11*, $reg_{rd}$ | **A1** |

† *synonym:* `movnz`  ‡ *synonym:* `movz`

**Programming Note** | In assembly language, to select the appropriate condition code, include `%fcc0`, `%fcc1`, `%fcc2`, or `%fcc3` before the *reg_or_imm11* field.



| 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* (*illegal_instruction*) |
| 1 | 1 | 0 | xcc |
| 1 | 1 | 1 | *Reserved* (*illegal_instruction*) |

*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.

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:...
```

The above sequence requires 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.

An attempt to execute a MOVcc instruction when either instruction bits 10:5 are nonzero or (cc2 :: cc1 :: cc0) = $101_2$ or $111_2$ causes an *illegal_instruction* exception.

If cc2 = 0 (that is, a floating-point condition code is being referenced in the MOVcc instructions) and either the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a MOVcc instruction causes an *fp_disabled* exception.

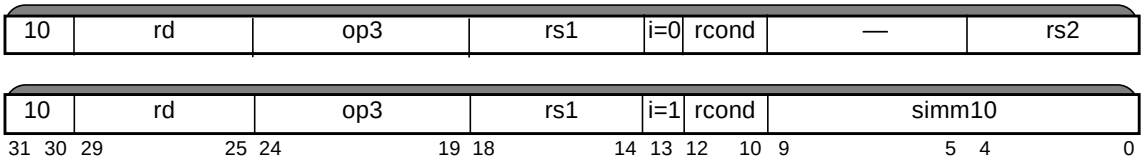
*Exceptions*

*illegal_instruction*
*fp_disabled*

# 7.65 Move Integer Register on Register Condition (MOVr)

| Instruction | op3 | rcond | Operation | Test | Assembly Language Syntax | Class |
|---|---|---|---|---|---|---|
| — | 10 1111 | 000 | *Reserved* (*illegal_instruction*) | | | — |
| MOVRZ | 10 1111 | 001 | Move if Register Zero | R[rs1] = 0 | `movrz`† *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | **A1** |
| MOVRLEZ | 10 1111 | 010 | Move if Register Less Than or Equal to Zero | R[rs1] ≤ 0 | `movrlez` *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | **A1** |
| MOVRLZ | 10 1111 | 011 | Move if Register Less Than Zero | R[rs1] < 0 | `movrlz` *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | **A1** |
| — | 10 1111 | 100 | *Reserved* (*illegal_instruction*) | | | — |
| MOVRNZ | 10 1111 | 101 | Move if Register Not Zero | R[rs1] ≠ 0 | `movrnz`‡ *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | **A1** |
| MOVRGZ | 10 1111 | 110 | Move if Register Greater Than Zero | R[rs1] > 0 | `movrgz` *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | **A1** |
| MOVRGEZ | 10 1111 | 111 | Move if Register Greater Than or Equal to Zero | R[rs1] ≥ 0 | `movrgez` *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$ | **A1** |

† *synonym:* `movre` ‡ *synonym:* `movrne`




*Description*

If the contents of integer register R[rs1] satisfy the condition specified in the rcond field, these instructions copy their second operand (if i = 0, R[rs2]; if i = 1, **sign_ext**(simm10)) 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.

**Programming Note** The MOVr instructions are "64-bit-only" instructions; there is no version of these instructions that operates on just the less-significant 32 bits of their source operands.

**Implementation 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 |

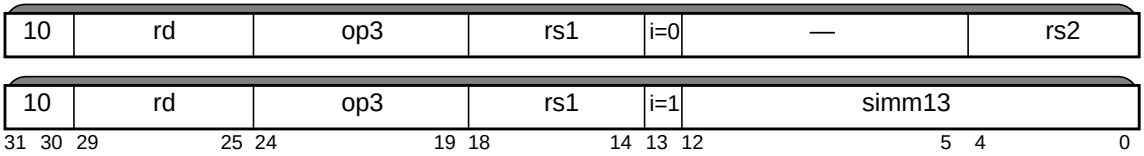
An attempt to execute a MOVr instruction when either instruction bits 9:5 are nonzero or rcond = $000_2$ or $100_2$ causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

# 7.66 Multiply Step

The MULScc instruction is deprecated and should not be used in new software. The MULX instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| MULScc[D] | 10 0100 | Multiply Step and modify cc's | `mulscc` $reg_{rs1}$, *reg_or_imm*, $reg_{rd}$ | **Y3** |




*Description*

MULScc treats the less-significant 32 bits of R[rs1] and the less-significant 32 bits of 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 performs an addition operation, 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** | In a standard MULScc instruction, rs1 = rd.

MULScc operates as follows:

1. If i = 0, the multiplicand is R[rs2]; if i = 1, the multiplicand is **sign_ext(**simm13).
2. A 32-bit value is computed by shifting the value from 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. MULScc writes the following result values:

| Register field | Value written by MULScc |
|---|---|
| CCR.icc | updated according to the result of the addition in step (3) above |
| R[rd]{63:33} | 0 |
| R[rd]{32} | CCR.icc.c |
| R[rd]{31:0} | the least-significant 32 bits of the sum from step (3) above |
| Y | the previous value of the Y register, shifted right by one bit, with Y{31} replaced by the value of R[rs1]{0} prior to shifting in step (2) |
| CCR.xcc.n | 0 |
| CCR.xcc.v | 0 |
| CCR.xcc.c | 0 |
| CCR.xcc.z | **if** (R[rd]{63:0} = 0) **then** 1 **else** 0 |

**SPARC V9 Compatibility Note** In SPARC V9, MULScc's effect on R[rd]{63:32} and CCR.xcc were explicitly left undefined.

5. The Y register is shifted right by one bit, with the least significant bit of the unshifted R[rs1] replacing bit 31 of Y.

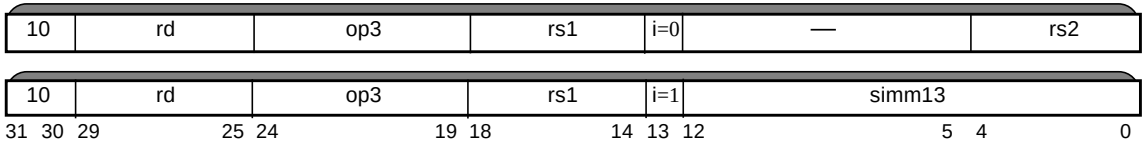
An attempt to execute a MULScc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

*See Also*
RDY on page 305
SDIV, SDIVcc on page 323
SMUL, SMULcc on page 331
UDIV, UDIVcc on page 377
UMUL, UMULcc on page 379

# 7.67 Multiply and Divide (64-bit)

| Instruction | op3 | Operation | Assembly Language | | Class |
|---|---|---|---|---|---|
| MULX | 00 1001 | Multiply (signed or unsigned) | `mulx` | *reg_or_imm*, $reg_{rd}$ | **A1** |
| SDIVX | 10 1101 | Signed Divide | `sdivx` | *reg_or_imm*, $reg_{rd}$ | **A1** |
| UDIVX | 00 1101 | Unsigned Divide | `udivx` | *reg_or_imm*, $reg_{rd}$ | **A1** |




*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.

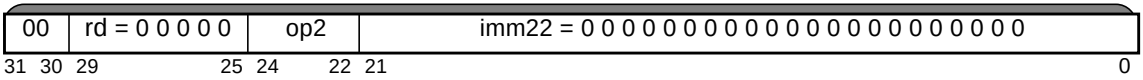
An attempt to execute a MULX, SDIVX, or UDIVX instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions*

*illegal_instruction*
*division_by_zero*

# 7.68 No Operation

| Instruction | op2 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| NOP | 100 | No Operation | nop | **A1** |




*Description* The NOP instruction changes no program-visible state (except that of the PC register).

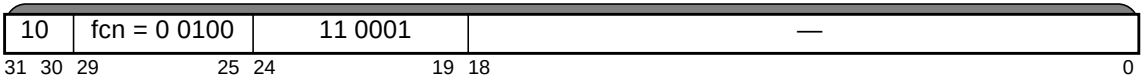
NOP is a special case of the SETHI instruction, with imm22 = 0 and rd = 0.

**Programming Note** There are many other opcodes that may execute as NOPs; however, this dedicated NOP instruction is the only one guaranteed to be implemented efficiently across all implementations.

*Exceptions* None

# 7.69 NORMALW

| Instruction | Operation | Assembly Language Syntax | Class |
|---|---|---|---|
| NORMALW$^{P}$ | "Other" register windows become "normal" register windows | `normalw` | **A1** |




*Description*

NORMALW$^{P}$ is a privileged instruction that copies the value of the OTHERWIN register to the CANRESTORE register, then sets the OTHERWIN register to zero.

**Programming Notes**

The NORMALW instruction is used when changing address spaces. NORMALW indicates the current "other" windows are now "normal" windows and should use the *spill_n_normal* and *fill_n_normal* traps when they generate a trap due to window spill or fill exceptions. The window state may become inconsistent if NORMALW is used when CANRESTORE is nonzero.

This instruction allows window manipulations to be atomic, without the value of *N_REG_WINDOWS* being visible to privileged software and without an assumption that *N_REG_WINDOWS* is constant (since hyperprivileged software can migrate a thread among virtual processors, across which *N_REG_WINDOWS* may vary).

An attempt to execute a NORMALW instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute an NORMALW instruction in nonprivileged mode (PSTATE.priv = 0 and HSTATE.hpriv = 0) causes a *privileged_opcode* exception.

*Exceptions*

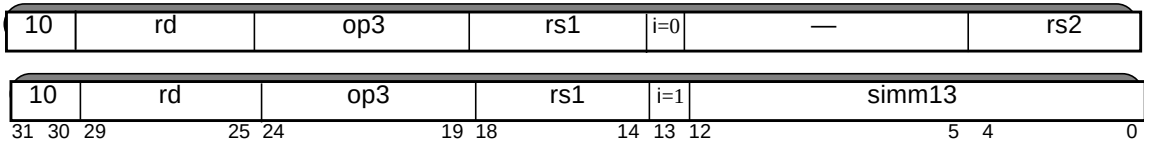
*illegal_instruction*
*privileged_opcode*

*See Also*

ALLCLEAN on page 147
INVALW on page 241
OTHERW on page 293
RESTORED on page 313
SAVED on page 321

# 7.70 OR Logical Operation

| Instruction | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| OR | 00 0010 | Inclusive **or** | `or` | *reg_rs1*, *reg_or_imm*, *reg_rd* | **A1** |
| ORcc | 01 0010 | Inclusive **or** and modify cc's | `orcc` | *reg_rs1*, *reg_or_imm*, *reg_rd* | **A1** |
| ORN | 00 0110 | Inclusive **or not** | `orn` | *reg_rs1*, *reg_or_imm*, *reg_rd* | **A1** |
| ORNcc | 01 0110 | Inclusive **or not** and modify cc's | `orncc` | *reg_rs1*, *reg_or_imm*, *reg_rd* | **A1** |




*Description* These instructions implement bitwise logical **or** 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].

ORcc and ORNcc 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)

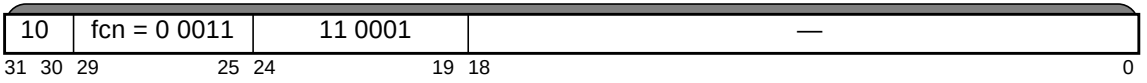
ORN and ORNcc logically negate their second operand before applying the main (**or**) operation.

An attempt to execute an OR[N][cc] instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

# 7.71 OTHERW

| Instruction | Operation | Assembly Language Syntax | Class |
|---|---|---|---|
| OTHERW$^{P}$ | "Normal" register windows become "other" register windows | `otherw` | **A1** |




*Description*

OTHERW$^{P}$ is a privileged instruction that copies the value of the CANRESTORE register to the OTHERWIN register, then sets the CANRESTORE register to zero.

**Programming Notes**

The OTHERW instruction is used when changing address spaces. OTHERW indicates the current "normal" register windows are now "other" register windows and should use the *spill_n_other* and *fill_n_other* traps when they generate a trap due to window spill or fill exceptions. The window state may become inconsistent if OTHERW is used when OTHERWIN is nonzero.

This instruction allows window manipulations to be atomic, without the value of *N_REG_WINDOWS* being visible to privileged software and without an assumption that *N_REG_WINDOWS* is constant (since hyperprivileged software can migrate a thread among virtual processors, across which *N_REG_WINDOWS* may vary).

An attempt to execute an OTHERW instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute an OTHERW instruction in nonprivileged mode (PSTATE.priv = 0 and HSTATE.hpriv = 0) causes a *privileged_opcode* exception.

*Exceptions*

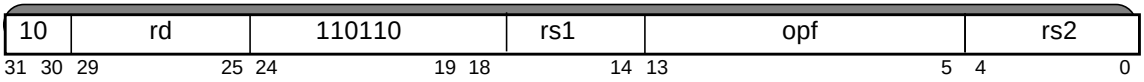
*illegal_instruction*
*privileged_opcode*

*See Also*

ALLCLEAN on page 147
INVALW on page 241
NORMALW on page 291
RESTORED on page 313
SAVED on page 321

# 7.72 Pixel Component Distance (with Accumulation) VIS 1

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| PDIST | 0 0011 1110 | Distance between eight 8-bit components, with accumulation | `pdist` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$ | C2 |




*Description* Eight unsigned 8-bit values are contained in the 64-bit floating-point source registers $F_D$[rs1] and $F_D$[rs2]. The corresponding 8-bit values in the source registers are subtracted (that is, each byte in $F_D$[rs2] is subtracted from the corresponding byte in $F_D$[rs1]). The sum of the absolute value of each difference is added to the integer in $F_D$[rd] and the resulting integer sum is stored in the destination register, $F_D$[rd].

**Programming Notes**

PDIST uses $F_D$[rd] as both a source and a destination register.

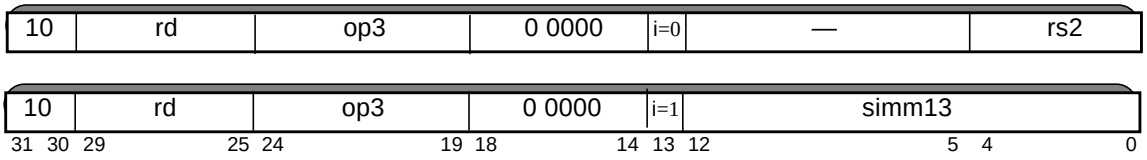
Typically, PDIST is used for motion estimation in video compression algorithms.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FPMERGE instruction causes an *fp_disabled* exception.

*Exceptions* *fp_disabled*

# 7.73 Population Count

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| POPC | 10 1110 | Population Count | `popc` *reg_or_imm*, $reg_{rd}$ | **C2** |




*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.

**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'-language 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 computation:

```
            IN = ...00101000 !1st '1' bit from right is
           -IN = ...11011000 !          bit 3 (4th bit)
         ~ -IN = ...00100111
     IN ^ ~ -IN = ...00001111
popc(IN ^ ~ -IN = 4
```

**Programming Note** POPC can be used to "centrifuge" all the '1' bits in a register to the least significant end of a destination register. Assembly-language code illustrating how POPC can be used for this purpose follows:

```
popc    %IN, %DEST
cmp     %IN, -1             ! Test for pattern of all 1's
mov     -1, %TEMP           ! Constant -1 -> temp register
sllx    %TEMP,%DEST,%DEST ! (shift count of 64 same as 0)
not     %DEST               !
movcc   %xcc, -1, %DEST     ! If src was -1, result is -1
```

where *IN*, *TEMP*, and *DEST* are integer registers.

**Programming Note** POPC is a "64-bit-only" instruction; there is no version of this instruction that operates on just the less-significant 32 bits of its source operand.

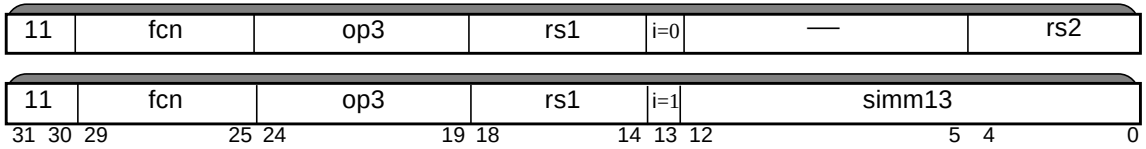
An attempt to execute a POPC instruction when either instruction bits 18:14 are nonzero, or i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

# 7.74 Prefetch

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| PREFETCH | 10 1101 | Prefetch Data | `prefetch` [*address*], *prefetch_fcn* | **A1** |
| PREFETCHA$^{P_{ASI}}$ | 11 1101 | Prefetch Data from Alternate Space | `prefetcha` [*regaddr*] *imm_asi*, *prefetch_fcn*<br>`prefetcha` [*reg_plus_imm*] `%asi`, *prefetch_fcn* | **A1** |

PREFETCH




PREFETCHA

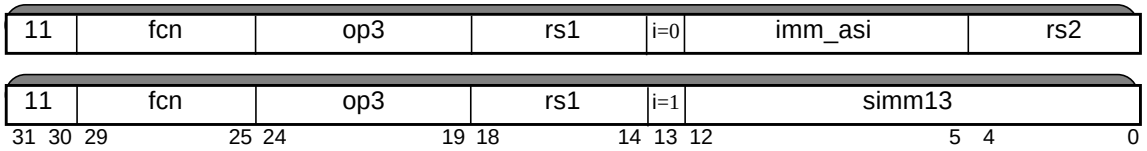



**TABLE 7-10** Prefetch Variants, by Function Code

| fcn | Prefetch Variant |
|---|---|
| 0 | (Weak) Prefetch for several reads |
| 1 | (Weak) Prefetch for one read |
| 2 | (Weak) Prefetch for several writes and possibly reads |
| 3 | (Weak) Prefetch for one write |
| 4 | Prefetch page |
| 5–15 ($05_{16}$–$0F_{16}$) | *Reserved* (*illegal_instruction*) |
| 16 ($10_{16}$) | Implementation dependent (NOP if not implemented) |
| 17 ($11_{16}$) | Prefetch to nearest unified cache |
| 18–19 ($12_{16}$–$13_{16}$) | Implementation dependent (NOP if not implemented) |
| 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 |

*Description* A PREFETCH[A] instruction provides a hint to the virtual processor that software expects to access a particular address in memory in the near future, so that the

virtual processor may take action to reduce the latency of accesses near that address. Typically, execution of a prefetch instruction initiates movement of a block of data containing the addressed byte from memory toward the virtual processor or creates an address mapping.

**Implementation Note** A PREFETCH[A] instruction may be used by software to:

- prefetch a cache line into a cache
- prefetch a valid address translation into a TLB
- invalidate a cache line that may have caused a correctable error during a load instruction.

If i = 0, the effective address operand for the PREFETCH instruction is "R[rs1] + R[rs2]"; if i = 1, it is "R[rs1] + **sign_ext** (simm13)".

PREFETCH instructions access the primary address space (ASI_PRIMARY[_LITTLE]).

PREFETCHA instructions access an alternate address space. If i = 0, the address space identifier (ASI) to be used for the instruction is in the imm_asi field. If i = 1, the ASI is found in the ASI register.

A prefetch operates much the same as a regular load operation (including a possible hardware tablewalk to load a TLB entry), but with certain important differences. In particular, a PREFETCH[A] instruction is non-blocking; subsequent instructions can continue to execute while the prefetch is in progress.

**Implementation Note** A PREFETCH[A] instruction is "released" by hardware after the TLB access, allowing subsequent instructions to continue to execute while the virtual processor performs the hardware tablewalk (in the case of a TLB miss for a Strong prefetch) or the cache access in the background.

When executed in nonprivileged or privileged mode, PREFETCH[A] has the same observable effect as a NOP. A prefetch instruction will not cause a trap if applied to an illegal or nonexistent memory address. (impl. dep. #103-V9-Ms10(e))

Whether a PREFETCH[A] instruction always succeeds when the MMU is disabled is implementation dependent (impl. dep. # 117-V9).

**IMPL. DEP. #103-V9-Ms10(a):** The size and alignment in memory of the data block prefetched 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
```

Variants of the prefetch instruction can be used to prepare the memory system for different types of accesses.

**IMPL. DEP. #103-V9-Ms10(b):** An implementation may implement none, some, or all of the defined PREFETCH[A] variants. It is implementation-dependent whether each variant is (1) not implemented and executes as a NOP, (2) is implemented and supports the full semantics for that variant, or (3) is implemented and only supports the simple common-case prefetching semantics for that variant.

## 7.74.1 Exceptions

Prefetch instructions PREFETCH and PREFETCHA generate exceptions under the conditions detailed in TABLE 7-11. Only the implementation-dependent prefetch variants (see TABLE 7-10) may generate an exception under conditions not listed in this table; the predefined variants only generate the exceptions listed here.

**TABLE 7-11** Behavior of PREFETCH[A] Instructions Under Exceptional Conditions *(1 of 2)*

| fcn | Instruction | Condition | Result |
|---|---|---|---|
| any | PREFETCH | i = 0 and instruction bits 12:5 are nonzero | *illegal_instruction* |
| any | PREFETCHA | reference to an ASI in the range $0_{16}$-$7F_{16}$, while in nonprivileged mode (*privileged_action* condition) | executes as NOP |
| any | PREFETCHA | reference to an ASI in range $30_{16}$..$7F_{16}$, while in privileged mode (*privileged_action* condition) | executes as NOP |
| 0-3 (weak) | PREFETCH[A] | condition detected for MMU miss (*data_access_MMU_miss* or *fast_data_access_MMU_miss* ) | executes as NOP |
| 0-3 (weak) | PREFETCH[A] | condition detected for *data_access_MMU_error* | executes as NOP |
| 0-4 | PREFETCH[A] | variant unimplemented | executes as NOP |
| 0-4 | PREFETCHA | reference to an invalid ASI (ASI not listed in following table) | executes as NOP |
| 0-4, 17, 20-23 | PREFETCH[A] | condition detected for *DAE_invalid_asi* (see following table), *DAE_privilege_violation*, *DAE_nc_page* ((TTE.cp = 0) **or** ((fcn = 0) **and** TTE.cv = 0)), *DAE_nfo_page*, or *DAE_side_effect_page* (TTE.e = 1) | executes as NOP |
| 4, 20-23 (strong) | PREFETCH[A] | prefetching the requested data would be a very time-consuming operation (condition detected for *data_access_MMU_miss* ) | executes as NOP |

**TABLE 7-11** Behavior of PREFETCH[A] Instructions Under Exceptional Conditions *(2 of 2)*

| fcn | Instruction | Condition | Result |
|---|---|---|---|
| 4, 20-23 (strong) | PREFETCH[A] | prefetching the requested data would be a time-consuming operation (condition detected for *fast_data_access_MMU_miss )* | *fast_data_access_MMU_miss* |
| 4, 20-23 (strong) | PREFETCH[A] | condition detected for *data_access_MMU_error*, *hw_corrected_error,* or *sw_recoverable_error* | *data_access_MMU_error*, *hw_corrected_error,* or *sw_recoverable_error* |
| 5–15 ($05_{16}$–$0F_{16}$) | PREFETCH[A] | (always) | *illegal_instruction* |

| ASIs valid for PREFETCHA (all others are invalid) | |
|---|---|
| ASI_AS_IF_PRIV_PRIMARY | ASI_AS_IF_PRIV_PRIMARY_LITTLE |
| ASI_AS_IF_PRIV_SECONDARY | ASI_AS_IF_PRIV_SECONDARY_LITTLE |
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |
| ASI_PRIMARY_NO_FAULT | ASI_PRIMARY_NO_FAULT_LITTLE |
| ASI_SECONDARY_NO_FAULT | ASI_SECONDARY_NO_FAULT_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |

## 7.74.2 Weak versus Strong Prefetches

Some prefetch variants are available in two versions, “Weak” and “Strong”.

From software’s perspective, the difference between the two is the degree of certainty that the data being prefetched will subsequently be accessed. That, in turn, affects the amount of effort (time) it’s willing for the underlying hardware to invest to perform the prefetch. If the prefetch is speculative (software believes the data will probably be needed, but isn’t sure), a Weak prefetch will initiate data movement if the operation can be performed quickly, but abort the prefetch and behave like a NOP if it turns out that performing the full prefetch will be time-consuming. If software has very high confidence that data being prefetched will subsequently be accessed, then a Strong prefetch will ensure that the prefetch operation will continue, even if the prefetch operation does become time-consuming.

From the virtual processor’s perspective, the difference between a Weak and a Strong prefetch is whether the prefetch is allowed to perform a time-consuming operation[1] in order to complete. If a time-consuming operation is required, a Weak

prefetch will abandon the operation and behave like a NOP while a Strong prefetch will pay the cost of performing the time-consuming operation so it can finish initiating the requested data movement. Behavioral differences among loads, strong prefetches, and weak prefetches are compared in TABLE 7-12.

**TABLE 7-12** Comparative Behavior of Load and Weak Prefetch Operations

| | Behavior | |
|---|---|---|
| **Condition** | **Load** | **Prefetch** |
| On a μTLB miss, is an MMU access performed? | Yes | Yes |
| On an MMU miss, is a hardware tablewalk performed? | Yes | No |
| Upon detection of *data_access_MMU_miss* exception (which only occurs during a hardware tablewalk) … | Traps | — (does not occur) |
| Upon detection of *fast_data_access_MMU_miss* exception... | Traps | NOP‡ |
| Upon detection of *privileged_action*, *DAE_**, *data_access_protection*, *PA_watchpoint*, or *VA_watchpoint* exception… | Traps | NOP‡ |
| If page table entry has cp = 0, e = 1, and cv = 0 for Prefetch for Several Reads | Traps | NOP‡ |
| If page table entry has nfo = 1 for a non-NoFault access… | Traps | NOP‡ |
| If page table entry has w = 0 for any prefetch for write access (fcn = 2, 3, 22, or 23)… | Traps | NOP‡ |
| Upon detection of fatal error or disrupting error conditions… | Traps | Traps |
| Instruction blocks until cache line filled? | Yes | No |

## 7.74.3 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 UltraSPARC Architecture 2007.

Each prefetch variant reflects an intent on the part of the compiler or programmer, a "hint" to the underlying virtual processor. This is different from other instructions (except BPN), all of which cause specific actions to occur. An UltraSPARC Architecture implementation may implement a prefetch variant by any technique, as long as the intent of the variant is achieved (impl. dep. #103-V9-Ms10(b)).

The prefetch instruction is designed to treat common cases 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

1. such as a hardware tablewalk or (if hardware tablewalk is disabled) a *fast_data_access_MMU_miss* trap, plus subsequently filling the cache line at the requested address

recommended overloading of the unimplemented variants is provided in the SPARC V9 specification. An implementation must treat any unimplemented prefetch fcn values as NOPs (impl. dep. #103-V9-Ms10).

### 7.74.3.1 Prefetch for Several Reads (fcn = 0, $20(14_{16})$)

The intent of these variants is to cause movement of data into the cache nearest the virtual processor.

There are Weak and Strong versions of this prefetch variant; fcn = 0 is Weak and fcn = 20 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

**Programming Note** The intended use of this variant is for streaming relatively small amounts of data into the primary data cache of the virtual processor.

### 7.74.3.2 Prefetch for One Read (fcn = 1, $21(15_{16})$)

The data to be read from the given address are 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.

There are Weak and Strong versions of this prefetch variant; fcn = 1 is Weak and fcn = 21 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

**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.

### 7.74.3.3 Prefetch for Several Writes (and Possibly Reads) (fcn = 2, $22(16_{16})$)

The intent of this variant is to cause movement of data in preparation for multiple writes.

There are Weak and Strong versions of this prefetch variant; fcn = 2 is Weak and fcn = 22 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

**Programming Note** An example use of this variant is to initialize a cache line, in preparation for a partial write.

**Implementation Note** On a multiprocessor system, this variant indicates that exclusive ownership of the addressed data is needed. Therefore, it may have the additional effect of obtaining exclusive ownership of the addressed cache line.

### 7.74.3.4 Prefetch for One Write (fcn = 3, 23($17_{16}$))

The intent of this variant is to initiate movement of data in preparation for a single write. 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.

There are Weak and Strong versions of this prefetch variant; fcn = 3 is Weak and fcn = 23 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

### 7.74.3.5 Prefetch Page (fcn = 4)

In a virtual memory system, the intended action of this variant is for hardware (or privileged or hyperprivileged software) to initiate asynchronous mapping of the referenced virtual address (assuming that it is legal to do so).

**Programming Note** Prefetch Page is used 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.

**Implementation Note** The mapping required by Prefetch Page may be performed by privileged software, hyperprivileged software, or hardware.

### 7.74.3.6 Prefetch to Nearest Unified Cache (fcn = 17($11_{16}$))

The intent of this variant is to cause movement of data into the nearest unified (combined instruction and data) cache. At the successful completion of this variant, the selected line from memory will be in the unified cache in the shared state, and in caches (if any) below it in the cache hierarchy.

Prefetch to Nearest Unified Cache is a Strong prefetch variant.

## 7.74.4 Implementation-Dependent Prefetch Variants (fcn = 16, 18, 19, and 24–31)

**IMPL. DEP. #103-V9-Ms10(c):** Whether and how PREFETCH fcns 16, 18, 19 and 24-31 are implemented are implementation dependent. If a variant is not implemented, it must execute as a NOP.

## 7.74.5 Additional Notes

**Programming Note** Prefetch instructions do have some "cost to execute". As long as the cost of executing a prefetch instruction is well less than the cost of a cache miss, use of prefetching provides a net gain in performance.

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 compiler that generates PREFETCH instructions should generate each of the variants where its use is most appropriate. That will help portable software be reasonably efficient across a range of hardware configurations.

**Implementation Note** Any effects of a data prefetch operation in privileged or hyperprivileged 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.

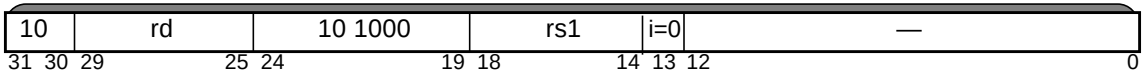
**Implementation Note** A prefetch from a nonprefetchable location has no effect. It is up to memory management hardware to determine how locations are identified as not prefetchable.

*Exceptions* *illegal_instruction*
*fast_data_access_MMU_miss*
*data_access_MMU_error*

# 7.75 Read Ancillary State Register

| Instruction | rs1 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| RDY[D] | 0 | Read Y register *(deprecated)* | `rd %y,` $reg_{rd}$ | **D2** |
| — | 1 | *Reserved* | | |
| RDCCR | 2 | Read Condition Codes register (CCR) | `rd %ccr,` $reg_{rd}$ | **A1** |
| RDASI | 3 | Read ASI register | `rd %asi,` $reg_{rd}$ | **A1** |
| RDTICK[Pnpt] | 4 | Read TICK register | `rd %tick,` $reg_{rd}$ | **A1** |
| RDPC | 5 | Read Program Counter (PC) | `rd %pc,` $reg_{rd}$ | **A2** |
| RDFPRS | 6 | Read Floating-Point Registers Status (FPRS) register | `rd %fprs,` $reg_{rd}$ | **A1** |
| — | 7–14 | *Reserved* | | |
| See text | 15 | MEMBAR or *Reserved*; see text | | |
| — | 16-18 | *Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | | |
| RDGSR | 19 | Read General Status register (GSR) | `rd %gsr,` $reg_{rd}$ | **A1** |
| — | 20–21 | *Reserved* (impl. dep. #8-V8-Cs20, #9-V8-Cs20) | | |
| RDSOFTINT[P] | 22 | Read per-virtual processor Soft Interrupt register (SOFTINT) | `rd %softint,` $reg_{rd}$ | **N2** |
| RDTICK_CMPR[P] | 23 | Read Tick Compare register (TICK_CMPR) | `rd %tick_cmpr,` $reg_{rd}$ | **N2** |
| RDSTICK[Pnpt] | 24 | Read System Tick Register (STICK) | `rd %stick†,` $reg_{rd}$ | **N2** |
| RDSTICK_CMPR[P] | 25 | Read System Tick Compare register (STICK_CMPR) | `rd %stick_cmpr†,` $reg_{rd}$ | **N2** |
| — | 26-27 | *Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | | |
| — | 28 | Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | | |
| — | 29-31 | Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | | |

† The original assembly language names for `%stick` and `%stick_cmpr` were, respectively, `%sys_tick` and `%sys_tick_cmpr`, which are now deprecated. Over time, assemblers will support the new `%stick` and `%stick_cmpr` names for these registers (which are consistent with `%tick` and `%tick_cmpr`). In the meantime, some existing assemblers may only recognize the original names.



# RDasr

*Description*

The Read Ancillary State Register (RDasr) instructions copy the contents of the state register specified by rs1 into R[rd].

An RDasr instruction with rs1 = 0 is a (deprecated) RDY instruction (which should not be used in new software).

> The RDY instruction is deprecated. It is recommended that all instructions that reference the Y register be avoided.

RDPC copies the contents of the PC register into R[rd]. If PSTATE.am = 0, the full 64-bit address is copied into R[rd]. If PSTATE.am = 1, only a 32-bit address is saved; PC{31:0} is copied to R[rd]{31:0} and R[rd]{63:32} is set to 0. (closed impl. dep. #125-V9-Cs10)

RDFPRS waits for all pending FPops and loads of floating-point registers to complete before reading the FPRS register.

The following values of rs1 are reserved for future versions of the architecture: 1, 7–14, 16-18, 20-21, and 26-27.

**IMPL. DEP. #47-V8-Cs20:** RDasr instructions with rd in the range 28–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For an RDasr instruction with rs1 in the range 28–31, the following are implementation dependent:

- the interpretation of bits 13:0 and 29:25 in the instruction
- whether the instruction is nonprivileged or privileged or hyperprivileged (impl. dep. #9-V8-Cs20), and
- whether an attempt to execute the instruction causes an *illegal_instruction* exception.

**Implementation Note** | See the section "Read/Write Ancillary State Registers (ASRs)" in *Extending the UltraSPARC Architecture*, contained in the separate volume *UltraSPARC Architecture Application Notes*, for a discussion of extending the SPARC V9 instruction set using read/write ASR instructions.

**Note** | Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers.

**SPARC V8 Compatibility Note** | The SPARC V8 RDPSR, RDWIM, and RDTBR instructions do not exist in the UltraSPARC Architecture, since the PSR, WIM, and TBR registers do not exist.

See *Ancillary State Registers* on page 69 for more detailed information regarding ASR registers.

**Exceptions.** An attempt to execute a RDasr instruction when any of the following conditions are true causes an *illegal_instruction* exception:

- rs1 = 15 and rd ≠ 0 (reserved for future versions of the architecture)
- rs1 = 1, 7–14, 16-18, 20-21, or 26-27 (reserved for future versions of the architecture)
- instruction bits 13:0 are nonzero

An attempt to execute a RDSOFTINT, RDTICK_CMPR, RDSTICK, or RDSTICK_CMPR instruction in nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0) causes a *privileged_opcode* exception (impl. dep. #250-U3-Cs10).

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a RDGSR instruction causes an *fp_disabled* exception.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), the following cause a *privileged_action* exception:

- execution of RDTICK when nonprivileged access to TICK is disabled (TICK.npt = 1)
- execution of RDSTICK when nonprivileged access to STICK is disabled (STICK.npt = 1)

**Implementation Note** | RDasr shares an opcode withMEMBAR; it is distinguished by rs1 = 15 or rd = 0 or (i = 0, and bit 12 = 0).

*Exceptions*

*illegal_instruction*
*privileged_opcode*
*fp_disabled*
*privileged_action*

*See Also*

RDHPR on page 308
RDPR on page 309
WRasr on page 381

# 7.76 Read Hyperprivileged Register

| Instruction | op3 | Operation | rs1 | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| RDHPR[H] | 10 1001 | Read hyperprivileged register | | | N2 |
| | | HPSTATE | 0 | `rdhpr %hpstate,` $reg_{rd}$ | |
| | | HTSTATE | 1 | `rdhpr %htstate,` $reg_{rd}$ | |
| | | *Reserved* | 2 | | |
| | | HINTP | 3 | `rdhpr %hintp,` $reg_{rd}$ | |
| | | *Reserved* | 4 | | |
| | | HTBA | 5 | `rdhpr %htba,` $reg_{rd}$ | |
| | | HVER | 6 | `rdhpr %hver,` $reg_{rd}$ | |
| | | *Reserved* | 7–30 | | |
| | | HSTICK_CMPR | 31 | `rdhpr %hstick_cmpr,` $reg_{rd}$ | |

*Description*

This instruction reads the contents of the specified hyperprivileged state register into the destination register, R[rd]. The rs1 field in the RDHPR instruction determines which hyperprivileged register is read.

There are *MAXTL* copies of the HTSTATE register. A read from HTSTATE returns the value in the copy of HTSTATE indexed by the current value in the trap level register (TL).

An attempt to execute a RDHPR instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- instruction bits 13:0 are nonzero
- rs1 = 2, rs1 = 4, or 7 ≤ rs1 ≤ 30 (reserved rs1 values)
- HPSTATE.hpriv = 0 (the processor is not in hyperprivileged mode)
- rs1 = 1 (attempt to read the HTSTATE register) while TL = 0 (current trap level is zero)

*Exceptions*

*illegal_instruction*

*See Also*

RDasr on page 305
RDPR on page 309
WRHPR on page 384

# 7.77 Read Privileged Register

| Instruction | op3 | Operation | rs1 | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|
| RDPR[P] | 10 1010 | Read Privileged register | | | | **N2** |
| | | TPC | 0 | `rdpr` | `%tpc,` $reg_{rd}$ | |
| | | TNPC | 1 | `rdpr` | `%tnpc,` $reg_{rd}$ | |
| | | TSTATE | 2 | `rdpr` | `%tstate,` $reg_{rd}$ | |
| | | TT | 3 | `rdpr` | `%tt,` $reg_{rd}$ | |
| | | TICK | 4 | `rdpr` | `%tick,` $reg_{rd}$ | |
| | | TBA | 5 | `rdpr` | `%tba,` $reg_{rd}$ | |
| | | PSTATE | 6 | `rdpr` | `%pstate,` $reg_{rd}$ | |
| | | TL | 7 | `rdpr` | `%tl,` $reg_{rd}$ | |
| | | PIL | 8 | `rdpr` | `%pil,` $reg_{rd}$ | |
| | | CWP | 9 | `rdpr` | `%cwp,` $reg_{rd}$ | |
| | | CANSAVE | 10 | `rdpr` | `%cansave,` $reg_{rd}$ | |
| | | CANRESTORE | 11 | `rdpr` | `%canrestore,` $reg_{rd}$ | |
| | | CLEANWIN | 12 | `rdpr` | `%cleanwin,` $reg_{rd}$ | |
| | | OTHERWIN | 13 | `rdpr` | `%otherwin,` $reg_{rd}$ | |
| | | WSTATE | 14 | `rdpr` | `%wstate,` $reg_{rd}$ | |
| | | *Reserved* | 15 | | | |
| | | GL | 16 | `rdpr` | `%gl,` $reg_{rd}$ | |
| | | *Reserved* | 17–31 | | | |

*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.

An attempt to execute a RDPR instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- instruction bits 13:0 are nonzero
- rs1 = 15, or 17 ≤ rs1 ≤ 31 (reserved rs1 values)
- 0 ≤ rs1 ≤ 3 (attempt to read TPC, TNPC,TSTATE, or TT register) while TL = 0 (current trap level is zero) and the virtual processor is in privileged or hyperprivileged mode.

**Implementation Note** In nonprivileged mode, *illegal_instruction* exception due to 0 ≤ rs1 ≤ 3 and TL = 0 does not occur; the *privileged_opcode* exception occurs instead.

# RDPR

An attempt to execute a RDPR instruction in nonprivileged mode (PSTATE.priv = 0 and HSTATE.hpriv = 0) causes a *privileged_opcode* exception.

**Historical Note** On some early SPARC implementations, floating-point exceptions could cause deferred traps. To ensure that execution could be correctly resumed after handling a deferred trap, hardware provided a floating-point queue (FQ), from which the address of the trapping instruction could be obtained by the trap handler. The front of the FQ was accessed by executing a RDPR instruction with rs1 = 15.

On UltraSPARC Architecture implementations, all floating-point traps are precise. When one occurs, the address of a trapping instruction can be found by the trap handler in the TPC[TL], so no floating-point queue (FQ) is needed or implemented (impl. dep. #25-V8) and RDPR with rs1 = 15 generates an *illegal_instruction* exception.

*Exceptions*

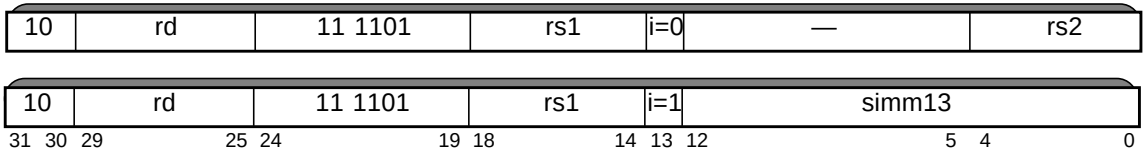
*illegal_instruction*
*privileged_opcode*

*See Also*

RDasr on page 305
RDHPR on page 308
WRPR on page 386

# 7.78 RESTORE

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| RESTORE | 11 1101 | Restore Caller's Window | `restore` *reg*$_{rs1}$`,` *reg_or_imm*`,` *reg*$_{rd}$ | **A1** |




*Description*

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 fill trap is not generated, RESTORE behaves like a normal ADD instruction, 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, *N_REG_WINDOWS*.

**Programming Notes** | Typically, if a RESTORE instruction traps, the 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.

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 a RESTORE instruction does not trap, it decrements the CWP (**mod** *N_REG_WINDOWS*) 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 Spi ll/Fill Traps* on page 496. 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 fill traps can be controlled by setting the value of the OTHERWIN and WSTATE registers appropriately. For details, see the section "Splitting the Register Windows" in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes.*

The fill handler normally will end with a RESTORED instruction followed by a RETRY instruction.

An attempt to execute a RESTORE instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions*

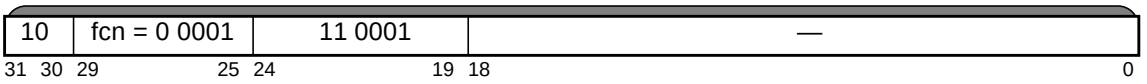
*illegal_instruction*
*fill_n_normal* (*n* = 0–7)
*fill_n_other* (*n* = 0–7)

*See Also*

SAVE on page 319

# 7.79 RESTORED

| Instruction | Operation | Assembly Language Syntax | Class |
|---|---|---|---|
| RESTORED[P] | Window has been restored | `restored` | **A1** |




*Description* RESTORED adjusts the state of the register-windows control registers.

RESTORED increments CANRESTORE.

If CLEANWIN < (*N_REG_WINDOWS*–1), then RESTORED increments CLEANWIN.

If OTHERWIN = 0, RESTORED decrements CANSAVE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

**Programming Notes**

Trap handler software for register window fills use the RESTORED instruction to indicate that a window has been filled successfully. For details, see the section "Example Code for Spill Handler" in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

Normal privileged software would probably not execute a RESTORED instruction from trap level zero (TL = 0). However, it is not illegal to do so and doing so does not cause a trap.

Executing a RESTORED instruction outside of a window 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.

If CANSAVE = 0 or CANRESTORE ≥ (*N_REG_WINDOWS* – 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 85) and consistent with the state prior to the RESTORED.

An attempt to execute a RESTORED instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute a RESTORED instruction in nonprivileged mode (PSTATE.priv = 0 and HSTATE.hpriv = 0) causes a *privileged_opcode* exception.

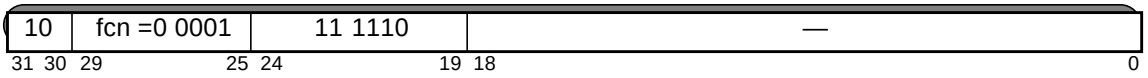
# RESTORED

*Exceptions* *illegal_instruction*
*privileged_opcode*

*See Also* ALLCLEAN on page 147
INVALW on page 241
NORMALW on page 291
OTHERW on page 293
SAVED on page 321

# 7.80 RETRY

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| RETRY[P] | 11 1110 | Return from Trap (retry trapped instruction) | `retry` | **A1** |




*Description* The RETRY instruction restores the saved state from TSTATE[TL] (GL, CCR, ASI, PSTATE, and CWP), HTSTATE[TL] (HPSTATE), sets PC and NPC, and decrements TL. RETRY sets PC←TPC[TL] and NPC←TNPC[TL] (normally, the values of PC and NPC saved at the time of the original trap).

**Programming Note** The DONE and RETRY instructions are used to return from privileged trap handlers.

If the saved TPC[TL] and TNPC[TL] were not altered by trap handler software, RETRY causes execution to resume at the instruction that originally caused the trap ("retrying" it).

Execution of a RETRY instruction in the delay slot of a control-transfer instruction produces undefined results.

When a RETRY instruction is executed and HTSTATE[TL].hpstate.hpriv = 0 (which will cause the RETRY to return the virtual processor to nonprivileged or privileged mode), the value of GL restored from TSTATE[TL] saturates at *MAXPGL*. That is, if the value in TSTATE[TL].gl is greater than *MAXPGL*, then *MAXPGL* is substituted and written to GL. This protects against non-hyperprivileged software executing with GL > *MAXPGL*.

If software writes invalid or inconsistent state to TSTATE or HTSTATE before executing RETRY, virtual processor behavior during and after execution of the RETRY instruction is undefined.

The RETRY instruction does not provide an error barrier, as MEMBAR #Sync does (impl. dep. #215-U3).

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system.

**IMPL. DEP. #417-S10**: If *(1)* TSTATE[TL].pstate.am = 1 and *(2)* a RETRY instruction is executed (which sets PSTATE.am to '1' by restoring the value from TSTATE[TL].pstate.am to PSTATE.am), it is implementation dependent whether the RETRY instruction masks (zeroes) the more-significant 32 bits of the values it places into PC and NPC.

**Exceptions.** An attempt to execute the RETRY instruction when either of the following conditions is true causes an *illegal_instruction* exception:

- instruction bits 18:0 are nonzero
- TL = 0 and the virtual processor is in privileged mode or hyperprivileged mode (PSTATE.priv = 1 or HPSTATE.hpriv = 1)

An attempt to execute a RETRY instruction in nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0) causes a *privileged_opcode* exception.

**Implementation Note** In nonprivileged mode, *illegal_instruction* exception due to TL = 0 does not occur. The *privileged_opcode* exception occurs instead, regardless of the current trap level (TL).

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then RETRY generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the RETRY instruction) is stored in TPC[TL] and the value of NPC from before the RETRY was executed is stored in TNPC[TL]. The full 64-bit (nonmasked) PC and NPC values are stored in TPC[TL] and TNPC[TL], regardless of the value of PSTATE.am. Note that since PSTATE.tct is automatically set to 0 during entry to a trap handler, a *control_transfer_instruction* will not occur when RETRY executes unless trap handler software has explicitly set PSTATE.tct to 1.

Note that since PSTATE.tct is automatically set to 0 during entry to a trap handler, the execution of a RETRY instruction at the end of a trap handler will not cause a *control_transfer_instruction* exception unless trap handler software has explicitly set PSTATE.tct to 1. During execution of the RETRY instruction, the value of PSTATE.tct is restored from TSTATE.

**Programming Note** RETRY should *not* normally be used to return from the trap handler for the *control_transfer_instruction* exception itself.

See the DONE instruction on page 165 and *Trap on Control Transfer (tct)* on page 92.

A *trap_level_zero* disrupting trap can occur upon the *completion* of a RETRY instruction, if the following three conditions are true after RETRY has executed:

- *trap_level_zero* exceptions are enabled (HPSTATE.tlz = 1),
- the virtual processor is in nonprivileged or privileged mode (HPSTATE.hpriv = 0), and
- the trap level (TL) register's value is zero (TL = 0)

*Exceptions*

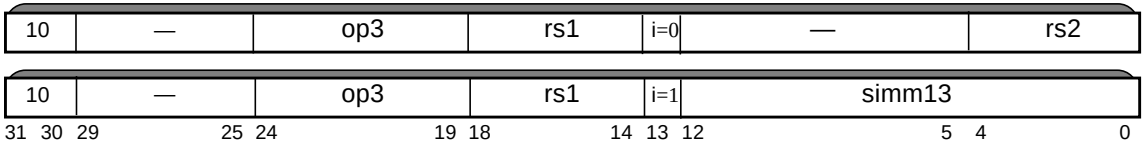
*illegal_instruction*
*privileged_opcode*
*control_transfer_instruction* (impl. dep. #450-S20)
*trap_level_zero*

*See Also*

DONE on page 165

# 7.81 RETURN

| Instruction | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| RETURN | 11 1001 | Return | `return` | *address* | **A1** |




*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.

Like other DCTIs, all effects of RETURN (including modification of CWP) are visible prior to execution of the delay slot instruction.

**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"
```

An attempt to execute a RETURN instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

A RETURN instruction may cause a *window_fill* exception as part of its RESTORE semantics.

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system. However, if a *control_transfer_instruction* trap occurs, the full 64-bit (nonmasked) address of the RETURN instruction is written into TPC[TL].

A RETURN instruction causes a *mem_address_not_aligned* exception if either of the two least-significant bits of the target address is nonzero.

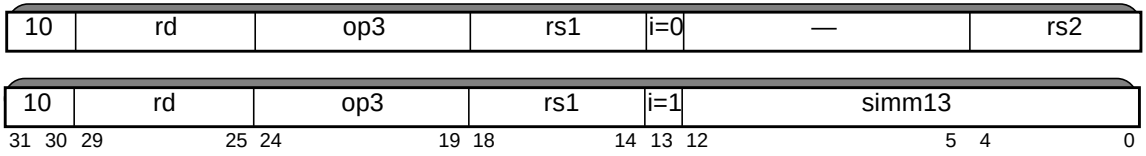
If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then RETURN generates a *control_transfer_instruction* exception instead of causing a control transfer.

*Exceptions*

*illegal_instruction*
*fill_n_normal* ($n$ = 0–7)
*fill_n_other* ($n$ = 0–7)
*mem_address_not_aligned*
*control_transfer_instruction* (impl. dep. #450-S20)

# 7.82 SAVE

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| SAVE | 11 1100 | Save Caller's Window | `save` *reg$_{rs1}$*, *reg_or_imm*, *reg$_{rd}$* | **A1** |




*Description*

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.

Furthermore, if and only if a spill trap is not generated, SAVE behaves like a normal ADD instruction, 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, *N_REG_WINDOWS*.

**Programming Notes**

Typically, if a SAVE instruction traps, the spill 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. For details, see the section "Leaf-Procedure Optimization" in Software Considerations, contained in the separate volume *UltraSPARC Architecture Application Notes*.

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 a SAVE instruction does not trap, it increments the CWP (**mod** *N_REG_WINDOWS*) 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).

An attempt to execute a SAVE instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

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).

**Programming Note**

The vectoring of spill traps can be controlled by setting the value of the OTHERWIN and WSTATE registers appropriately. For details, see the section "Splitting the Register Windows" in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

The spill handler normally will end with a SAVED instruction followed by a RETRY instruction.

*Exceptions*

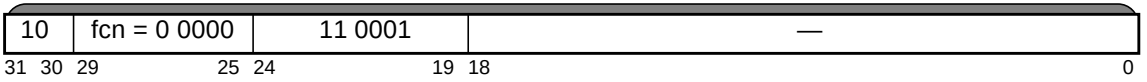
*illegal_instruction*
*spill_n_normal* ($n$ = 0–7)
*spill_n_other* ($n$ = 0–7)
*clean_window*

*See Also*

RESTORE on page 311

# 7.83 SAVED

| Instruction | Operation | Assembly Language Syntax | Class |
|---|---|---|---|
| SAVED[P] | Window has been saved | `saved` | **A1** |




*Description* SAVED adjusts the state of the register-windows control registers.

SAVED increments CANSAVE. If OTHERWIN = 0, SAVED decrements CANRESTORE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

**Programming Notes**

Trap handler software for register window spills uses the SAVED instruction to indicate that a window has been spilled successfully. For details, see the section "Example Code for Spill Handler" in Software Considerations, contained in the separate volume *UltraSPARC Architecture Application Notes*.

Normal privileged software would probably not execute a SAVED instruction 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 instruction outside of a window spill 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.

If CANSAVE ≥ (*N_REG_WINDOWS* – 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 85) and consistent with the state prior to the SAVED.

An attempt to execute a SAVED instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute a SAVED instruction in nonprivileged mode (PSTATE.priv = 0 and HSTATE.hpriv = 0) causes a *privileged_opcode* exception.

*Exceptions* *illegal_instruction*
*privileged_opcode*

# SAVED

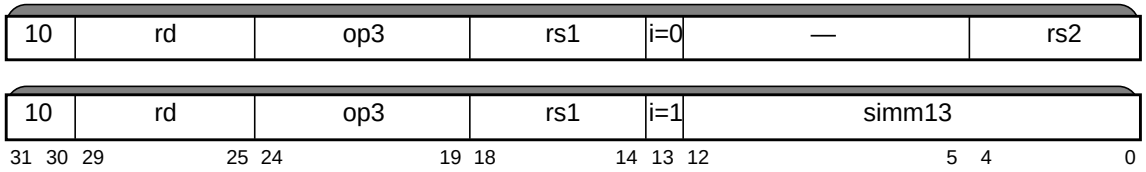
*See Also*

ALLCLEAN on page 147
INVALW on page 241
NORMALW on page 291
OTHERW on page 293
RESTORED on page 313

# 7.84 Signed Divide (64-bit ÷ 32-bit)

The SDIV and SDIVcc instructions are deprecated and should not be used in new software. The SDIVX instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| SDIV[D] | 00 1111 | Signed Integer Divide | `sdiv` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **D2** |
| SDIVcc[D] | 01 1111 | Signed Integer Divide and modify cc's | `sdivcc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **D2** |




*Description*

The signed 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.

*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 7-13.

**TABLE 7-13** SDIV / SDIVcc Overflow Detection and Value Returned

| Condition Under Which Overflow Occurs | Value Returned in R[rd] |
|---|---|
| Rational quotient ≥ $2^{31}$ | $2^{31} - 1$ (0000 0000 7FFF $FFFF_{16}$) |
| Rational quotient ≤ $-2^{31} - 1$ | $-2^{31}$ (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 | Effect on bit of SDIVcc instruction |
|---|---|
| icc.n | Set to 1 if R[rd]{31} = 1; otherwise, set to 0 |
| icc.z | Set to 1 if R[rd]{31:0} = 0; otherwise, set to 0 |
| icc.v | Set to 1 if overflow (*per* TABLE 7-12); otherwise set to 0 |
| icc.c | Set to 0 |
| xcc.n | Set to 1 if R[rd]{63} = 1; otherwise, set to 0 |
| xcc.z | Set to 1 if R[rd]{63:0} = 0; otherwise, set to 0 |
| xcc.v | Set to 0 |
| xcc.c | Set to 0 |

An attempt to execute an SDIV or SDIVcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions*

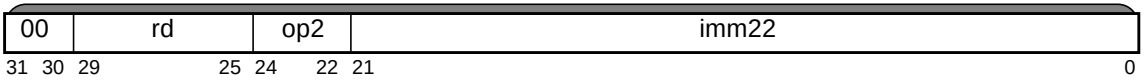
*illegal_instruction*
*division_by_zero*

*See Also*

MULScc on page 287
RDY on page 305
UDIV[cc] on page 377

# 7.85 SETHI

| Instruction | op2 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| SETHI | 100 | Set High 22 Bits of Low Word | `sethi` *const22*`,` $reg_{rd}$<br>`sethi` `%hi` (*value*)`,` $reg_{rd}$ | **A1** |




*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 *No Operation*)
- rd = 0 and imm22 ≠ 0 may be used to trigger hardware performance counters in some UltraSPARC Architecture implementations (for details, see implementation-specific documentation).

**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 place 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
```

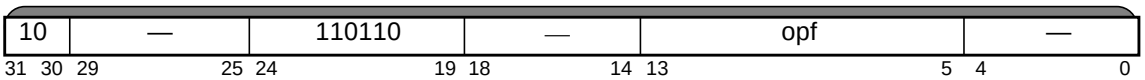
*Exceptions*

None

# 7.86 SHUTDOWN VIS 1

The SHUTDOWN instruction is deprecated and should not be used in new software.

| Instruction | opf | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| SHUTDOWN$^{D,P}$ | 0 1000 0000 | Enter low-power mode | `shutdown` | **D3** |




*Description*

SHUTDOWN is a deprecated, privileged instruction that was used in early UltraSPARC implementations to bring the virtual processor or its containing system into a low-power state in an orderly manner. It had no effect on software-visible virtual processor state.

On an UltraSPARC Architecture implementation operating in privileged or hyperprivileged mode, SHUTDOWN behaves like a NOP (impl. dep. #206-U3-Cs10).

An attempt to execute a SHUTDOWN instruction when instruction bits 29:25, 18:14, or 4:0 are nonzero causes an *illegal_instruction* exception (and a *privileged_opcode* exception does not occur).

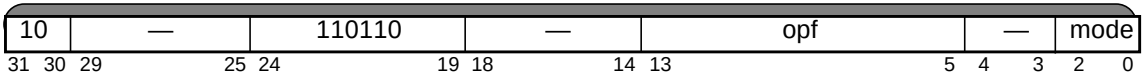
Because SHUTDOWN is a privileged instruction, an attempt to execute it while in nonprivileged mode causes a *privileged_opcode* trap.

*Exceptions*

*illegal_instruction* (reserved bits nonzero)
*privileged_opcode*

# 7.87 Set Interval Arithmetic Mode VIS 2

| Instruction | opf | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| SIAM | 0 1000 0001 | Set the interval arithmetic mode fields in the GSR | `siam` | *siam_mode* | **B1** |




*Description* The SIAM instruction sets the GSR.im and GSR.irnd fields as follows:

GSR.im ← mode{2}

GSR.irnd ← mode{1:0}

**Note** | When GSR.im is set to 1, all subsequent floating-point instructions requiring round mode settings derive rounding-mode information from the General Status Register (GSR.irnd) instead of the Floating-Point State Register (FSR.rd).

**Note** | When GSR.im = 1, the processor operates in standard floating-point mode regardless of the setting of FSR.ns.

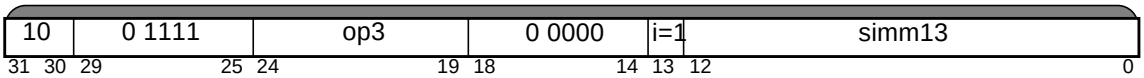
An attempt to execute a SIAM instruction when instruction bits 29:25, 18:14, or 4:3 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a SIAM instruction causes an *fp_disabled* exception.

*Exceptions* *illegal_instruction*
*fp_disabled*

# 7.88 Software-Initiated Reset

| Instruction | op3 | rd | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|
| SIR[H] | 11 0000 | 15 | Software-Initiated Reset | `sir` | *simm13* | **A2** |




*Description*

SIR is a hyperprivileged instruction, 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 506 and *Software-Initiated Reset (SIR)* on page 630 for more information about software-initiated resets.

When executed in nonprivileged or privileged mode (HPSTATE.hpriv = 0), SIR causes an *illegal_instruction* exception (impl. dep. #116-V9).

**Implementation Notes**

The SIR instruction shares an opcode with WRasr; they are distinguished by the rd, rs1, and i fields (rd = 15,rs1 = 0, and i = 1 for SIR).

An instruction that uses the WRasr opcode (op1 = $10_2$, op3 = 11 $0000_2$) with i = 1 is not an SIR instruction; see *Write Ancillary State Register* on page 381 for specification of its behavior.

*Exceptions*

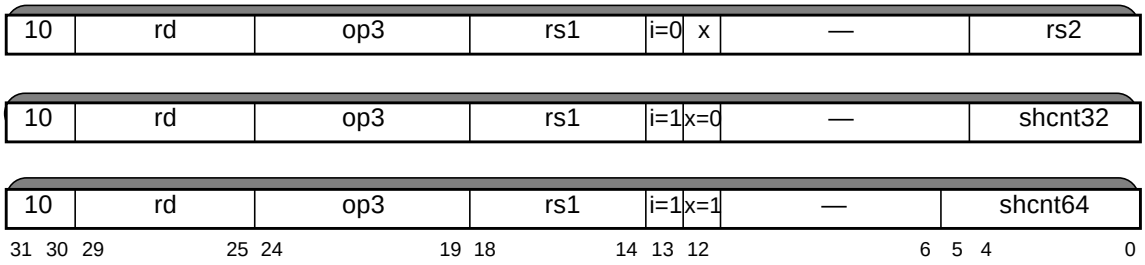
*software_initiated_reset*
*illegal_instruction*

*See Also*

WRasr on page 381

# 7.89 Shift

| Instruction | op3 | x | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| SLL | 10 0101 | 0 | Shift Left Logical – 32 bits | `sll` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ | **A1** |
| SRL | 10 0110 | 0 | Shift Right Logical – 32 bits | `srl` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ | **A1** |
| SRA | 10 0111 | 0 | Shift Right Arithmetic– 32 bits | `sra` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ | **A1** |
| SLLX | 10 0101 | 1 | Shift Left Logical – 64 bits | `sllx` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ | **A1** |
| SRLX | 10 0110 | 1 | Shift Right Logical – 64 bits | `srlx` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ | **A1** |
| SRAX | 10 0111 | 1 | Shift Right Arithmetic – 64 bits | `srax` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$ | **A1** |




*Description* These instructions perform logical or arithmetic shift operations.

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 7-14 shows the shift count encodings for all values of i and x.

**TABLE 7-14** 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].

# SLL / SRL / SRA

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` *reg*$_{rs1}$`,0,` *reg*$_{rd}$" can be used to convert a 32-bit value to 64 bits, with sign extension into the upper word. "`srl` *reg*$_{rs1}$`,0,` *reg*$_{rd}$" can be used to clear the upper 32 bits of R[rd].

An attempt to execute a SLL, SRL, or SRA instruction when instruction bits 11:5 are nonzero causes an *illegal_instruction* exception.

An attempt to execute a SLLX, SRLX, or SRAX instruction when either of the following conditions exist causes an *illegal_instruction* exception:

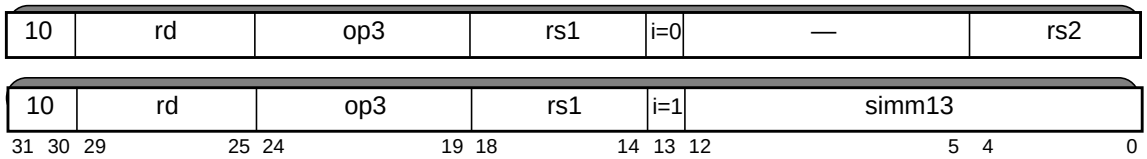
- i = 0 or x = 0 and instruction bits 11:5 are nonzero
- x = 1 and instruction bits 11:6 are nonzero

*Exceptions* *illegal_instruction*

# 7.90 Signed Multiply (32-bit)

The SMUL and SMULcc instructions are deprecated and should not be used in new software. The MULX instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| SMUL[D] | 00 1011 | Signed Integer Multiply | `smul` *reg$_{rs1}$, reg_or_imm, reg$_{rd}$* | D2 |
| SMULcc[D] | 01 1011 | Signed Integer Multiply and modify cc's | `smulcc` *reg$_{rs1}$, reg_or_imm, reg$_{rd}$* | D2 |




*Description*

The signed 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].

Signed multiply instructions (SMUL, SMULcc) operate on signed integer word operands and compute a signed integer doubleword product.

SMUL does not affect the condition code bits. SMULcc writes the integer condition code bits, icc and xcc, as shown below.

| Bit | Effect on bit by execution of SMULcc |
|---|---|
| icc.n | Set to 1 if product{31} = 1; otherwise, set to 0 |
| icc.z | Set to 1 if product{31:0}= 0; otherwise, set to 0 |
| icc.v | Set to 0 |
| icc.c | Set to 0 |
| xcc.n | Set to 1 if product{63} = 1; otherwise, set to 0 |
| xcc.z | Set to 1 if product{63:0} = 0; otherwise, set to 0 |
| xcc.v | Set to 0 |
| xcc.c | Set to 0 |

**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.

# SMUL, SMULcc (Deprecated)

**Programming Notes**

32-bit overflow after SMUL or SMULcc is indicated by Y ≠ (R[rd] >> 31), where ">>" indicates 32-bit arithmetic right-shift.

An attempt to execute a SMUL or SMULcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

*See Also* UMUL[cc] on page 379

# 7.91 Store Integer

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| STB | 00 0101 | Store Byte | `stb`† *reg*$_{rd}$, [*address*] | **A1** |
| STH | 00 0110 | Store Halfword | `sth`‡ *reg*$_{rd}$, [*address*] | **A1** |
| STW | 00 0100 | Store Word | `stw`◊ *reg*$_{rd}$, [*address*] | **A1** |
| STX | 00 1110 | Store Extended Word | `stx` *reg*$_{rd}$, [*address*] | **A1** |

† *synonyms:* `stub`, `stsb` ‡ *synonyms:* `stuh`, `stsh` ◊ *synonyms:* `st`, `stuw`, `stsw`

*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.

These instructions access memory using the implicit ASI (see page 104). 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, STX) integer instruction operates atomically.

An attempt to execute a store integer instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

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.

*Exceptions*

*illegal_instruction*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*

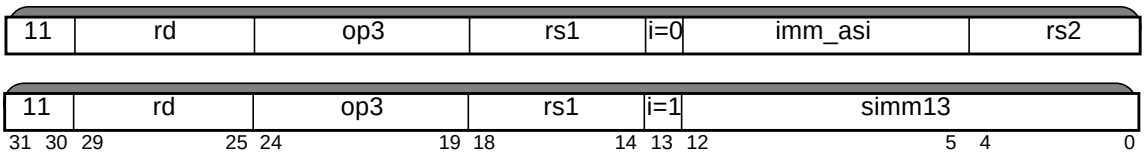
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

*See Also* STTW on page 357

# 7.92 Store Integer into Alternate Space

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| STBA$^{P_{ASI}}$ | 01 0101 | Store Byte into Alternate Space | `stba`† *reg$_{rd}$*, [*regaddr*] *imm_asi*<br>`stba` *reg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STHA$^{P_{ASI}}$ | 01 0110 | Store Halfword into Alternate Space | `stha`‡ *reg$_{rd}$*, [*regaddr*] *imm_asi*<br>`stha` *reg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STWA$^{P_{ASI}}$ | 01 0100 | Store Word into Alternate Space | `stwa`◊ *reg$_{rd}$*, [*regaddr*] *imm_asi*<br>`stwa` *reg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STXA$^{P_{ASI}}$ | 01 1110 | Store Extended Word into Alternate Space | `stxa` *reg$_{rd}$*, [*regaddr*] *imm_asi*<br>`stxa` *reg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |

† *synonyms:* `stuba`, `stsba` ‡ *synonyms:* `stuha`, `stsha` ◊ *synonyms:* `sta`, `stuwa`, `stswa`




*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 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, STXA) instruction operates atomically.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, these instructions cause a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, these instructions cause a *privileged_action* exception.

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.

# STBA / STHA / STWA / STXA

STBA, STHA, and STWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

| ASIs valid for STBA, STHA, and STWA | |
|---|---|
| ASI_AS_IF_PRIV_PRIMARY | ASI_AS_IF_PRIV_PRIMARY_LITTLE |
| ASI_AS_IF_PRIV_SECONDARY | ASI_AS_IF_PRIV_SECONDARY_LITTLE |
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_REAL_IO | ASI_REAL_IO_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |

STXA can be used with any ASI (including, but not limited to, the above list), unless it either (a) violates the privilege mode rules described for the *privileged_action* exception above or (b) is used with any of the following ASIs, which causes a *DAE_invalid_asi* exception.

| ASIs invalid for STXA (cause *DAE_invalid_asi* exception) | |
|---|---|
| ASI_BLOCK_AS_IF_PRIV_PRIMARY | ASI_BLOCK_AS_IF_PRIV_PRIMARY_LITTLE |
| ASI_BLOCK_AS_IF_PRIV_SECONDARY | ASI_BLOCK_AS_IF_PRIV_SECONDARY_LITTLE |
| ASI_BLOCK_AS_IF_USER_PRIMARY | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE |
| ASI_BLOCK_AS_IF_USER_SECONDARY | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE |
| ASI_BLOCK_AS_IF_USER_PRIMARY | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE |
| ASI_BLOCK_AS_IF_USER_SECONDARY | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE |
| ASI_PST8_PRIMARY | ASI_PST8_PRIMARY_LITTLE |
| ASI_PST8_SECONDARY | ASI_PST8_SECONDARY_LITTLE |
| ASI_PRIMARY_NO_FAULT | ASI_PRIMARY_NO_FAULT_LITTLE |
| ASI_SECONDARY_NO_FAULT | ASI_SECONDARY_NO_FAULT_LITTLE |
| ASI_PST16_PRIMARY | ASI_PST16_PRIMARY_LITTLE |
| ASI_PST16_SECONDARY | ASI_PST16_SECONDARY_LITTLE |
| ASI_PST32_PRIMARY | ASI_PST32_PRIMARY_LITTLE |
| ASI_PST32_SECONDARY | ASI_PST32_SECONDARY_LITTLE |
| ASI_FL8_PRIMARY | ASI_FL8_PRIMARY_LITTLE |
| ASI_FL8_SECONDARY | ASI_FL8_SECONDARY_LITTLE |
| ASI_FL16_PRIMARY | ASI_FL16_PRIMARY_LITTLE |
| ASI_FL16_SECONDARY | ASI_FL16_SECONDARY_LITTLE |
| ASI_BLOCK_COMMIT_PRIMARY | ASI_BLOCK_COMMIT_SECONDARY |
| ASI_BLOCK_PRIMARY | ASI_BLOCK_PRIMARY_LITTLE |
| ASI_BLOCK_SECONDARY | ASI_BLOCK_SECONDARY_LITTLE |

# STBA / STHA / STWA / STXA

**V8 Compatibility Note** | The SPARC V8 STA instruction was renamed STWA in the SPARC V9 architecture.

*Exceptions*

*mem_address_not_aligned* (all except STBA)
*privileged_action*
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

*See Also*

LDA on page 246
STTWA on page 359

# 7.93 Block Store VIS 1

> The STBLOCKF instruction is intended to be a processor-specific instruction, which may or may not be implemented in future UltraSPARC Architecture implementations. Therefore, it should only be used in platform-specific dynamically-linked libraries, in hyperprivileged software, or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

| Instruction | ASI Value | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| STBLOCKF | $16_{16}$ | 64-byte block store to primary address space, user privilege | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_AIUP`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STBLOCKF | $17_{16}$ | 64-byte block store to secondary address space, user privilege | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_AIUS`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STBLOCKF | $1E_{16}$ | 64-byte block store to primary address space, little-endian, user privilege | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_AIUPL`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STBLOCKF | $1F_{16}$ | 64-byte block store to secondary address space, little-endian, user privilege | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_AIUSL`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STBLOCKF | $F0_{16}$ | 64-byte block store to primary address space | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_P`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STBLOCKF | $F1_{16}$ | 64-byte block store to secondary address space | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_S`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STBLOCKF | $F8_{16}$ | 64-byte block store to primary address space, little-endian | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_PL`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STBLOCKF | $F9_{16}$ | 64-byte block store to secondary address space, little-endian | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_SL`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STBLOCKF | $E0_{16}$ | 64-byte block commit store to primary address space | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_COMMIT_P`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |
| STBLOCKF | $E1_{16}$ | 64-byte block commit store to secondary address space | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_BLK_COMMIT_S`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |

*Description* A block store instruction references one of several special block-transfer ASIs. Block-transfer ASIs allow block stores to be performed accessing the same address space as normal 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 accessed by the instruction.

**Programming Note** The block store instruction, STBLOCKF, and its companion, LDBLOCKF, were originally defined to provide a fast mechanism for block-copy operations.

STBLOCKF stores data from the eight double-precision floating-point registers specified by rd to a 64-byte-aligned memory area. The lowest-addressed eight bytes in memory are stored from the lowest-numbered double-precision rd.

While a STBLOCKF operation is in progress, any of the following values may be observed in a destination doubleword memory locations: (1) the old data value, (2) zero, or (3) the new data value. When the operation is complete, only the new data values will be seen.

**Compatibility Note** Software written for older UltraSPARC implementations that reads data being written by STBLOCKF instructions may or may not allow for case (2) above. Such software should be checked to verify that either it always waits for STBLOCKF to complete before reading the values written, or that it will operate correctly if an intermediate value of zero (not the "old" or "new" data values) is observed while the STBLOCKF operation is in progress.

A Block Store only guarantees atomicity for each 64-bit (8-byte) portion of the 64 bytes that it stores.

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[1]. 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 426 for more information.)

ASIs $E0_{16}$ and $E1_{16}$ are only used for block store-with-commit operations; they are not available for use by block load operations. See *Block Load and Store ASIs* on page 451 for more information.

Software should assume the following (where "load operation" includes load, load-store, and LDBLOCKF instructions and "store operation" includes store, load-store, and STBLOCKF instructions):

- A STBLOCKF does not follow memory ordering with respect to earlier or later load operations. If there is overlap between the addresses of destination memory locations of a STBLOCKF and the source address of a later load operation, the

[1] Even if all data stores on a given implementation coherently update the instruction cache (see page 389), stores (other than Block Store with Commit) on SPARC V9 implementations in general do *not* maintain coherency between instruction and data caches.

load operation may receive incorrect data. Therefore, if ordering with respect to later load operations is important, a MEMBAR #StoreLoad instruction must be executed between the STBLOCKF and subsequent load operations.

- A STBLOCKF does not follow memory ordering with respect to earlier or later store operations. Those instructions' data may commit to memory in a different order from the one in which those instructions were issued. Therefore, if ordering with respect to later store operations is important, a MEMBAR #StoreStore instruction must be executed between the STBLOCKF and subsequent store operations.
- STBLOCKFs do not follow register dependency interlocks, as do ordinary stores.

**Programming Note** STBLOCKF is intended to be a processor-specific instruction (see the warning at the top of page 338). If STBLOCKF *must* be used in software intended to be portable across current and previous processor implementations, then it must be coded to work in the face of any implementation variation that is permitted by implementation dependency #411-S10, described below.

**IMPL. DEP. #411-S10**: The following aspects of the behavior of the block store (STBLOCKF) instruction are implementation dependent:

- The memory ordering model that STBLOCKF follows (other than as constrained by the rules outlined above).
- Whether *VA_watchpoint* and *PA_watchpoint* exceptions are recognized on accesses to all 64 bytes of the STBLOCKF (the recommended behavior), or only on accesses to the first eight bytes.
- Whether STBLOCKFs to non-cacheable (TTE.cp = 0) pages execute in strict program order or not. If not, a STBLOCKF to a non-cacheable page causes an *illegal_instruction* exception.
- Whether STBLOCKF follows register dependency interlocks (as ordinary stores do).
- Whether a non-Commit STBLOCKF forces the data to be written to memory and invalidates copies in all caches present (as the Commit variants of STBLOCKF do).
- Any other restrictions on the behavior of STBLOCKF, as described in implementation-specific documentation.

**Exceptions.** An *illegal_instruction* exception occurs if the source floating-point registers are not aligned on an eight-register boundary.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a STBLOCKF instruction causes an *fp_disabled* exception.

If the least significant 6 bits of the memory address are not all zero, a *mem_address_not_aligned* exception occurs.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0 (ASIs $16_{16}$, $17_{16}$, $1E_{16}$, and $1F_{16}$), STBLOCKF causes a *privileged_action* exception.

An access caused by STBLOCKF may trigger a *VA_watchpoint* or *PA_watchpoint* exception (impl. dep. #411-S10).

**Implementation Note** | STBLOCKF shares an opcode with the STDFA, STPARTIALF, and STSHORTF instructions; it is distinguished by the ASI used.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*mem_address_not_aligned*
*privileged_action*
*VA_watchpoint* (impl. dep. #411-S10)
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint* (impl. dep. #411-S10)
*data_access_error*

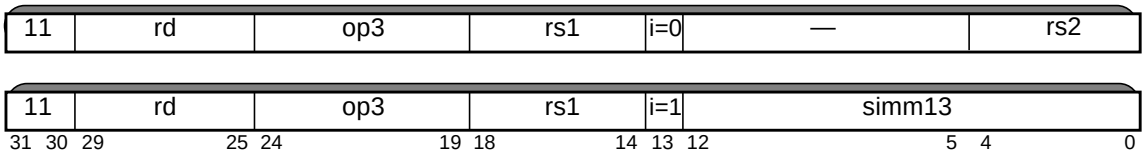
*See Also*

LDBLOCKF on page 249

# 7.94 Store Floating-Point

| Instruction | op3 | rd | Operation | Assembly Language | | Class |
|---|---|---|---|---|---|---|
| STF | 10 0100 | 0–31 | Store Floating-Point register | `st` | *freg*$_{rd}$, [*address*] | **A1** |
| STDF | 10 0111 | † | Store Double Floating-Point register | `std` | *freg*$_{rd}$, [*address*] | **A1** |
| STQF | 10 0110 | † | Store Quad Floating-Point register | `stq` | *freg*$_{rd}$, [*address*] | **C3** |

† Encoded floating-point register value, as described on page 51.




*Description* The store single floating-point instruction (STF) copies the contents of the 32-bit floating-point register $F_S$[rd] into memory.

The store double floating-point instruction (STDF) copies the contents of 64-bit floating-point register $F_D$[rd] into a word-aligned doubleword in memory. The unit of atomicity for STDF is 4 bytes (one word).

The store quad floating-point instruction (STQF) copies the contents of 128-bit floating-point register $F_Q$[rd] into a word-aligned quadword in memory. The unit of atomicity for STQF is 4 bytes (one word).

These instruction access memory using the implicit ASI (see page 104). The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

**Exceptions.** An attempt to execute a STF or STDF instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STF or STDF instruction causes an *fp_disabled* exception.

STF causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

STDF requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute an STDF instruction causes an *STDF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the STDF instruction and return (impl. dep. #110-V9-Cs10(a)).

STQF requires only word alignment in memory. If the effective address is word-aligned but not quadword-aligned, an attempt to execute an STQF instruction causes an *STQF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the STQF instruction and return (impl. dep. #112-V9-Cs10(a)).

**Programming Note** Some compilers issued sequences of single-precision stores for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned stores is expected to be fast, compilers should issue sets of single-precision stores only when they can determine that double- or quadword operands are *not* properly aligned.

An attempt to execute an STQF instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

**Implementation Note** Since UltraSPARC Architecture 2007 processors do not implement in hardware instructions (including STQF) that refer to quad-precision floating-point registers, the *STQF_mem_address_not_aligned* and *fp_exception_other* (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an *illegal_instruction* exception and subsequent trap.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*STDF_mem_address_not_aligned*
*STQF_mem_address_not_aligned* (not used in UltraSPARC Architecture 2007)
*mem_address_not_aligned*
*fp_exception_other* (FSR.ftt = invalid_fp_register (STQF only))
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

*See Also*

*Load Floating-Point Register* on page 253
*Block Store* on page 338
*Store Floating-Point into Alternate Space* on page 345
*Store Floating-Point State Register (Lower)* on page 349
*Store Short Floating-Point* on page 355

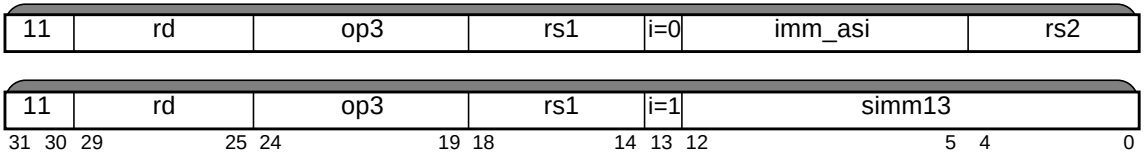
# STF / STDF / STQF / STXFSR

*Store Partial Floating-Point* on page 351
*Store Floating-Point State Register* on page 362

# 7.95 Store Floating-Point into Alternate Space

| Instruction | op3 | rd | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| STFA$^{P_{ASI}}$ | 11 0100 | 0–31 | Store Floating-Point Register to Alternate Space | `sta` *freg$_{rd}$*, [*regaddr*] *imm_asi*<br>`sta` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STDFA$^{P_{ASI}}$ | 11 0111 | † | Store Double Floating-Point Register to Alternate Space | `stda` *freg$_{rd}$*, [*regaddr*] *imm_asi*<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **A1** |
| STQFA$^{P_{ASI}}$ | 11 0110 | † | Store Quad Floating-Point Register to Alternate Space | `stqa` *freg$_{rd}$*, [*regaddr*] *imm_asi*<br>`stqa` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **C3** |

† Encoded floating-point register value, as described on page 51.




*Description*

The store single floating-point into alternate space instruction (STFA) copies the contents of the 32-bit floating-point register $F_S$[rd] into memory.

The store double floating-point into alternate space instruction (STDFA) copies the contents of 64-bit floating-point register $F_D$[rd] into a word-aligned doubleword in memory. The unit of atomicity for STDFA is 4 bytes (one word).

The store quad floating-point into alternate space instruction (STQFA) copies the contents of 128-bit floating-point register $F_Q$[rd] into a word-aligned quadword in memory. The unit of atomicity for STQFA is 4 bytes (one word).

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.

**Programming Note** Some compilers issued sequences of single-precision stores for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned stores is expected to be fast, compilers should issue sets of single-precision stores only when they can determine that double- or quadword operands are *not* properly aligned.

**Exceptions.** STFA causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

STDFA requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute an STDFA instruction causes an *STDF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the STDFA instruction and return (impl. dep. #110-V9-Cs10(b)).

STQFA requires only word alignment in memory. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an STQFA instruction may cause an *STQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STQFA instruction and return (impl. dep. #112-V9-Cs10(b)).

**Implementation Note** | STDFA shares an opcode with the STBLOCKF, STPARTIALF, and STSHORTF instructions; it is distinguished by the ASI used.

An attempt to execute an STQFA instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

**Implementation Note** | Since UltraSPARC Architecture 2007 processors do not implement in hardware instructions (including STQFA) that refer to quad-precision floating-point registers, the *STQF_mem_address_not_aligned* and *fp_exception_other* (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an *illegal_instruction* exception and subsequent trap.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, this instruction causes a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, this instruction causes a *privileged_action* exception.

STFA and STQFA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

**ASIs valid for STFA and STQFA**

| | |
|---|---|
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_REAL_IO | ASI_REAL_IO_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |

STDFA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with the STDFA instruction causes a *DAE_invalid_asi* exception.

**ASIs valid for STDFA**

| | |
|---|---|
| ASI_AS_IF_PRIV_PRIMARY | ASI_AS_IF_PRIV_PRIMARY_LITTLE |
| ASI_AS_IF_PRIV_SECONDARY | ASI_AS_IF_PRIV_SECONDARY_LITTLE |
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_REAL_IO | ASI_REAL_IO_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |
| ASI_BLOCK_AS_IF_USER_PRIMARY † | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE † |
| ASI_BLOCK_AS_IF_USER_SECONDARY † | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE † |
| ASI_BLOCK_PRIMARY † | ASI_BLOCK_PRIMARY_LITTLE † |
| ASI_BLOCK_SECONDARY † | ASI_BLOCK_SECONDARY_LITTLE † |
| ASI_BLOCK_COMMIT_PRIMARY † | |
| ASI_BLOCK_COMMIT_SECONDARY † | |
| ASI_FL8_PRIMARY ‡ | ASI_FL8_PRIMARY_LITTLE ‡ |
| ASI_FL8_SECONDARY ‡ | ASI_FL8_SECONDARY_LITTLE ‡ |
| ASI_FL16_PRIMARY ‡ | ASI_FL16_PRIMARY_LITTLE ‡ |
| ASI_FL16_SECONDARY ‡ | ASI_FL16_SECONDARY_LITTLE ‡ |
| ASI_PST8_PRIMARY * | ASI_PST8_PRIMARY_LITTLE * |
| ASI_PST8_SECONDARY * | ASI_PST8_SECONDARY_LITTLE * |
| ASI_PST16_PRIMARY * | ASI_PST16_PRIMARY_LITTLE * |
| ASI_PST16_SECONDARY * | ASI_PST16_SECONDARY_LITTLE * |
| ASI_PST32_PRIMARY * | ASI_PST32_PRIMARY_LITTLE * |
| ASI_PST32_SECONDARY * | ASI_PST32_SECONDARY_LITTLE * |

† If this ASI is used with the opcode for STDFA, the STBLOCKF instruction is executed instead of STFA. For behavior of STBLOCKF, see *Block Store* on page 338.

‡ If this ASI is used with the opcode for STDFA, the STSHORTF instruction is executed instead of STDFA. For behavior of STSHORTF, see *Store Short Floating-Point* on page 355.

* If this ASI is used with the opcode for STDFA, the STPARTIALF instruction is executed instead of STDFA. For behavior of STPARTIALF, see *Store Partial Floating-Point* on page 351.

*Exceptions*

*fp_disabled*
*STDF_mem_address_not_aligned*
*STQF_mem_address_not_aligned* (STQFA only) (not used in UA-2007)
*mem_address_not_aligned*
*fp_exception_other* (FSR.ftt = invalid_fp_register (STQFA only))

*privileged_action*
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

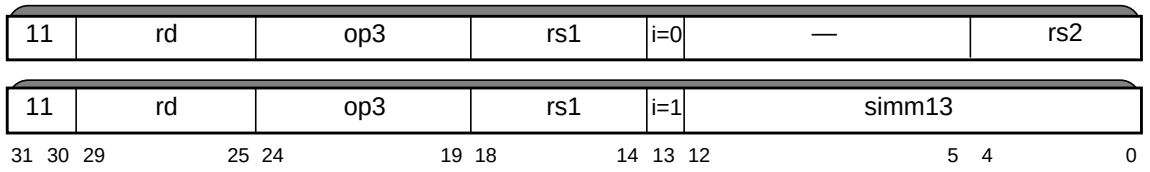
*See Also*

*Load Floating-Point from Alternate Space* on page 256
*Block Store* on page 338
*Store Floating-Point* on page 342
*Store Short Floating-Point* on page 355
*Store Partial Floating-Point* on page 351

# 7.96 Store Floating-Point State Register (Lower)

The STFSR instruction is deprecated and should not be used in new software. The STXFSR instruction should be used instead.

| Opcode | op3 | rd | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| STFSR[D] | 10 0101 | 0 | Store Floating-Point State Register (Lower) | `st %fsr, [`*address*`]` | **D2** |
| | 10 0101 | 1-31 | (see page 362) | | |




*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 FSR into memory.

After writing the FSR to memory, STFSR zeroes FSR.ftt

**V9 Compatibility Note** | FSR.ftt should not be zeroed until it is known that the store will not cause a precise trap.

STFSR accesses memory using the implicit ASI (see page 104). The effective address for this instruction is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext(** simm13)" if i = 1.

An attempt to execute a STFSR instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STFSR instruction causes an *fp_disabled* exception.

# STFSR (Deprecated)

STFSR causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

**V9 Compatibility Note** — Although STFSR is deprecated, UltraSPARC Architecture implementations continue to support it for compatibility with existing SPARC V8 software. The STFSR instruction is defined to store only the less-significant 32 bits of the FSR into memory, while STXFSR allows SPARC V9 software to store all 64 bits of the FSR.

**Implementation Note** — STFSR shares an opcode with the STXFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = $10_2$, op3 = 10 $0101_2$ opcode with an invalid rd value causes an *illegal_instruction* exception.

*Exceptions*

*illegal_instruction*
*fp_disabled*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

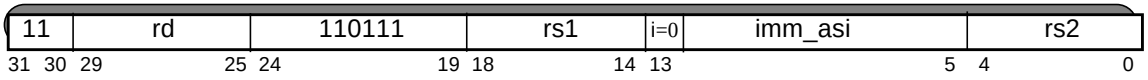
*See Also*

*Store Floating-Point* on page 342
*Store Floating-Point State Register* on page 362

# 7.97 Store Partial Floating-Point VIS 1

| Instruction | ASI Value | Operation | Assembly Language Syntax † | Class |
|---|---|---|---|---|
| STPARTIALF | $C0_{16}$ | Eight 8-bit conditional stores to primary address space | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST8_P` | **B1** |
| STPARTIALF | $C1_{16}$ | Eight 8-bit conditional stores to secondary address space | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST8_S` | **B1** |
| STPARTIALF | $C8_{16}$ | Eight 8-bit conditional stores to primary address space, little-endian | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST8_PL` | **B1** |
| STPARTIALF | $C9_{16}$ | Eight 8-bit conditional stores to secondary address space, little-endian | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST8_SL` | **B1** |
| STPARTIALF | $C2_{16}$ | Four 16-bit conditional stores to primary address space | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST16_P` | **B1** |
| STPARTIALF | $C3_{16}$ | Four 16-bit conditional stores to secondary address space | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST16_S` | **B1** |
| STPARTIALF | $CA_{16}$ | Four 16-bit conditional stores to primary address space, little-endian | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST16_PL` | **B1** |
| STPARTIALF | $CB_{16}$ | Four 16-bit conditional stores to secondary address space, little-endian | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST16_SL` | **B1** |
| STPARTIALF | $C4_{16}$ | Two 32-bit conditional stores to primary address space | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST32_P` | **B1** |
| STPARTIALF | $C5_{16}$ | Two 32-bit conditional stores to secondary address space | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST32_S` | **B1** |
| STPARTIALF | $CC_{16}$ | Two 32-bit conditional stores to primary address space, little-endian | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST32_PL` | **B1** |
| STPARTIALF | $CD_{16}$ | Two 32-bit conditional stores to secondary address space, little-endian | `stda` *freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] `#ASI_PST32_SL` | **B1** |

† 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 existing assemblers may only recognize the original syntax.




*Description* The partial store instructions are selected by one of the partial store ASIs with the STDFA instruction.

# STPARTIALF

Two 32-bit, four 16-bit, or eight 8-bit values from the 64-bit floating-point register $F_D$[rd] are conditionally stored at the address specified by R[rs1], using the mask specified in R[rs2]. STPARTIALF has the effect of merging selected data from its source register, $F_D$[rd], into the existing data at the corresponding destination locations.

The mask value in R[rs2] has the same format as the result specified by the pixel compare instructions (see *SIMD Signed Compare* on page 179). The most significant bit of the mask (not of the entire register) corresponds to the most significant part of $F_D$[rd]. The data is stored in little-endian form in memory if the ASI name has an "L" (or "_LITTLE") suffix; otherwise, it is stored in big-endian format.

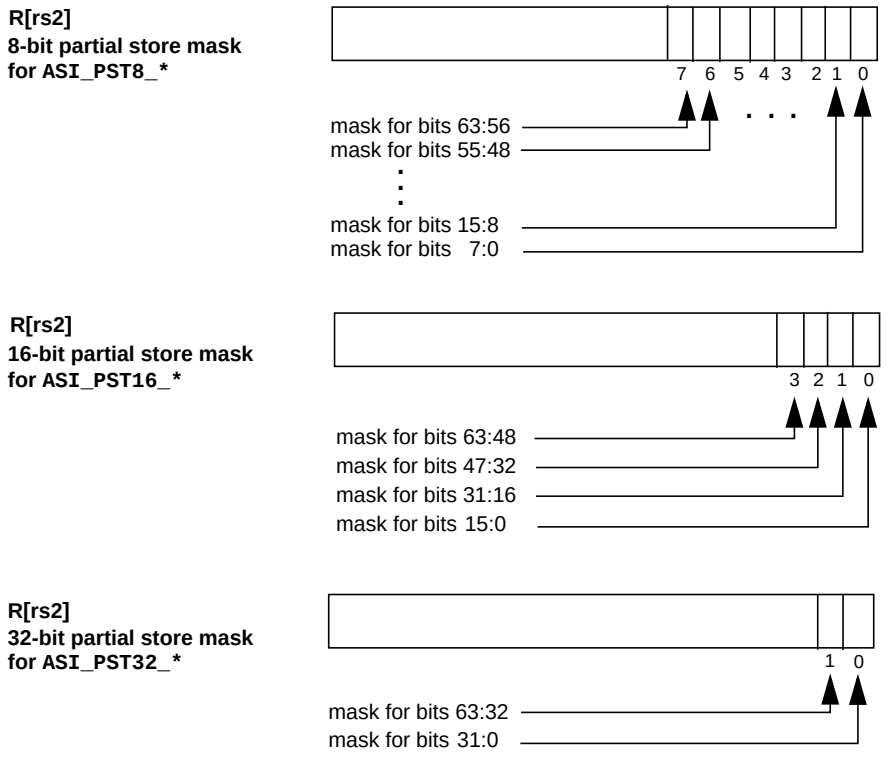



**FIGURE 7-29** Mask Format for Partial Store

**Exceptions.** 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, halfword or word is to be stored.

- The Virtual (Physical) Data Watchpoint Mask in ASI_DCU_WATCHPOINT_CONTROL_REG indicates that one or more of the bytes to be stored at the watched address is being watched.

For data watchpoints of partial stores in UltraSPARC Architecture 2007, the byte store mask (R[rs2]) in the Partial Store instruction is ignored, and a watchpoint exception can occur even if the mask is zero (that is, no store will take place). The ASI_DCU_WATCHPOINT_CONTROL_REG Data Watchpoint masks are only checked for nonzero value (watchpoint enabled) (impl. dep. #249).

An attempt to execute a STPARTIALF instruction when i = 1 causes an *illegal_instruction* exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STPARTIALF instruction causes an *fp_disabled* exception.

STPARTIALF causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

STPARTIALF requires only word alignment in memory for eight byte stores. If the effective address is word-aligned but not doubleword-aligned, it generates an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the STDFA instruction and return.

**IMPL. DEP. #249-U3-Cs10**: For an STPARTIAL 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 the LSU Control register 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; however, if any of them *is* used, the resulting behavior is specified in the LDDFA instruction description on page 258.

**Implementation Note** | STPARTIALF shares an opcode with the STBLOCKF, STDFA, and STSHORTF instructions; it is distinguished by the ASI used.

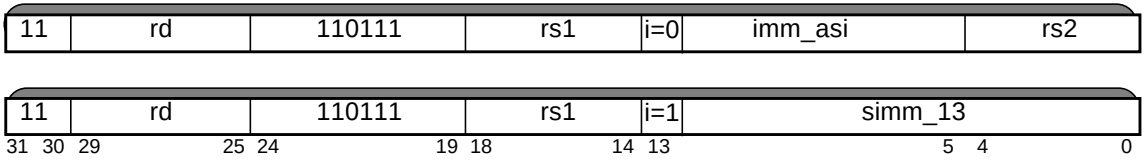
*Exceptions*

*illegal_instruction*
*fp_disabled*
*mem_address_not_aligned*
*VA_watchpoint* (see text)
*DAE_privilege_violation*
*DAE_nc_page*
*DAE_nfo_page*

*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint* (see text)
*data_access_error*

# 7.98 Store Short Floating-Point VIS 1

| Instruction | ASI Value | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| STSHORTF | $D0_{16}$ | 8-bit store to primary address space | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_FL8_P`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |
| STSHORTF | $D1_{16}$ | 8-bit store to secondary address space | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_FL8_S`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |
| STSHORTF | $D8_{16}$ | 8-bit store to primary address space, little-endian | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_FL8_PL`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |
| STSHORTF | $D9_{16}$ | 8-bit store to secondary address space, little-endian | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_FL8_SL`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |
| STSHORTF | $D2_{16}$ | 16-bit store to primary address space | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_FL16_P`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |
| STSHORTF | $D3_{16}$ | 16-bit store to secondary address space | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_FL16_S`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |
| STSHORTF | $DA_{16}$ | 16-bit store to primary address space, little-endian | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_FL16_PL`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |
| STSHORTF | $DB_{16}$ | 16-bit store to secondary address space, little-endian | `stda` *freg$_{rd}$*, [*regaddr*] `#ASI_FL16_SL`<br>`stda` *freg$_{rd}$*, [*reg_plus_imm*] `%asi` | **B1** |




*Description*

The short floating-point store instruction allows 8- and 16-bit stores to be performed from the floating-point registers. Short stores access the low-order 8 or 16 bits of the register.

Little-endian ASIs transfer data in little-endian format from memory; otherwise, memory is assumed to be big-endian. Short stores are typically used with the FALIGNDATA instruction (see *Align Data* on page 173) to assemble or store 64 bits on noncontiguous components.

**Implementation Note** STSHORTF shares an opcode with the STBLOCKF, STDFA, and STPARTIALF instructions; it is distinguished by the ASI used.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STSHORTF instruction causes an *fp_disabled* exception.

# STSHORTF

STSHORTF causes a *mem_address_not_aligned* exception if the effective memory address is not halfword-aligned.

An 8-bit STSHORTF (using ASI $D0_{16}$, $D1_{16}$, $D8_{16}$, or $D9_{16}$) can be performed to an arbitrary memory address (no alignment requirement).

A 16-bit STSHORTF (using ASI $D2_{16}$, $D3_{16}$, $DA_{16}$, or $DB_{16}$) to an address that is not halfword-aligned (an odd address) causes a *mem_address_not_aligned* exception.

*Exceptions*

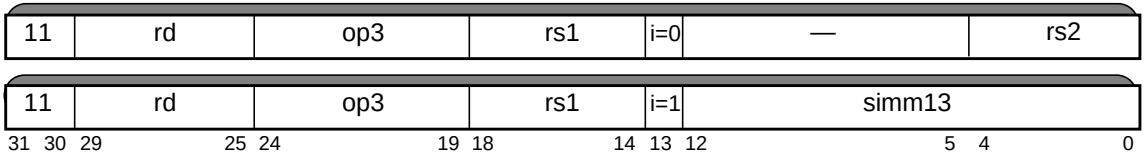
*fp_disabled*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

# 7.99 Store Integer Twin Word

The STTW instruction is deprecated and should not be used in new software. The STX instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax † | | Class |
|---|---|---|---|---|---|
| STTW$^{D}$ | 00 0111 | Store Integer Twin Word | `sttw` | *reg$_{rd}$*, [*address*] | **D2** |

† The original assembly language syntax for this instruction used an "`std`" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "`sttw`" mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original "`std`" mnemonic.




*Description*

The store integer twin word instruction (STTW) 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 twin word instruction is unused and should always be set to 0 by software.

STTW accesses memory using the implicit ASI (see page 104). 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 twin word instruction operates atomically.

**IMPL. DEP. #108-V9a:** It is implementation dependent whether STTW is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_STTW* exception. (STTW is implemented in hardware in all UltraSPARC Architecture 2007 implementations.)

An attempt to execute an STTW instruction when either of the following conditions exist causes an *illegal_instruction* exception:

- destination register number rd is an odd number (is misaligned)
- i = 0 and instruction bits 12:5 are nonzero

# STTW (Deprecated)

STTW causes a *mem_address_not_aligned* exception if the effective address is not doubleword-aligned.

With respect to little-endian memory, an STTW 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**

STTW 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 STTW.

If STTW is emulated in software, STX instruction should be used for the memory access in the emulation code to preserve atomicity.

*Exceptions*

*unimplemented_STTW* (not used in UltraSPARC Architecture 2007)
*illegal_instruction*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

*See Also*

STW/STX on page 333
STTWA on page 359

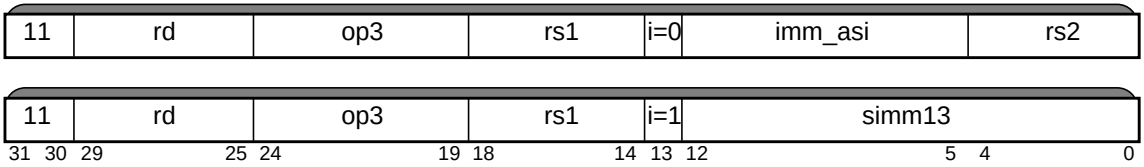
# 7.100 Store Integer Twin Word into Alternate Space

The STTWA instruction is deprecated and should not be used in new software. The STXA instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| STTWA$^{D, PASI}$ | 01 0111 | Store Twin Word into Alternate Space | `sttwa` *reg$_{rd}$* [*regaddr*] *imm_asi*<br>`sttwa` *reg$_{rd}$* [*reg_plus_imm*] `%asi` | **D2, Y3**‡ |

† The original assembly language syntax for this instruction used an "`stda`" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "`sttwa`" mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original "`stda`" mnemonic.

‡ **Y3** for restricted ASIs ($00_{16}$-$7F_{16}$); **D2** for unrestricted ASIs ($80_{16}$-$FF_{16}$)




*Description*

The store twin word integer into alternate space instruction (STTWA) 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 an STTWA instruction is unused and should always be set to 0 by software.

Store integer twin word 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 twin word instruction operates atomically.

With respect to little-endian memory, an STTWA 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.

# STTWA (Deprecated)

**IMPL. DEP. #108-V9b:** It is implementation dependent whether STTWA is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_STTW* exception. (STTWA is implemented in hardware in all UltraSPARC Architecture 2007 implementations.)

An attempt to execute an STTWA instruction with a misaligned (odd) destination register number rd causes an *illegal_instruction* exception.

STTWA causes a *mem_address_not_aligned* exception if the effective address is not doubleword-aligned.

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, this instruction causes a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, this instruction causes a *privileged_action* exception.

STTWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *DAE_invalid_asi* exception (impl. dep. #300-U4-Cs10).

**ASIs valid for STTWA**

| | |
|---|---|
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |
| ASI_REAL_IO | ASI_REAL_IO_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |

**Programming Note** Nontranslating ASIs (see page 429) may only be accessed using STXA (not STTWA) instructions. If an STTWA referencing a nontranslating ASI is executed, per the above table, it generates a *DAE_invalid_asi* exception (impl. dep. #300-U4-Cs10).

**Programming Note** STTWA is provided for compatibility with existing SPARC V8 software. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties. Therefore, software should avoid using STTWA.

If STTWA is emulated in software, the STXA instruction should be used for the memory access in the emulation code to preserve atomicity.

*Exceptions*

*unimplemented_STTW*
*illegal_instruction*
*mem_address_not_aligned*
*privileged_action*

# STTWA (Deprecated)

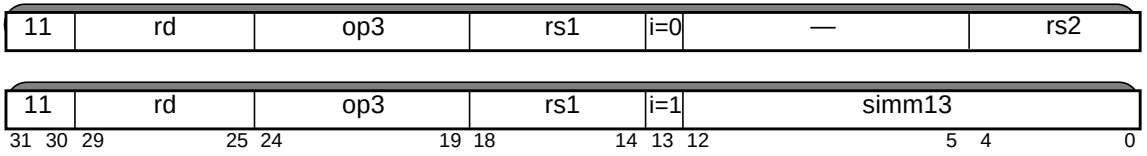
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

*See Also*

STWA/STXA on page 335
STTW on page 357

# 7.101 Store Floating-Point State Register

| Instruction | op3 | rd | Operation | Assembly Language | Class |
|---|---|---|---|---|---|
| | 10 0101 | 0 | (see page 349) | | |
| STXFSR | 10 0101 | 1 | Store Floating-Point State register | `stx %fsr, [`*address*`]` | **A1** |
| — | 10 0101 | 2–31 | *Reserved* | | |




*Description* 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.

**Implementation Note** | FSR.ftt should not be zeroed by STXFSR until it is known that the store will not cause a precise trap.

STXFSR accesses memory using the implicit ASI (see page 104). The effective address for this instruction is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

**Exceptions.** An attempt to execute a STXFSR instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STXFSR instruction causes an *fp_disabled* exception.

If the effective address is not doubleword-aligned, an attempt to execute an STXFSRinstruction causes a *mem_address_not_aligned* exception.

**Implementation Note** | STXFSR shares an opcode with the (deprecated) STFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = $10_2$, op3 = 10 $0101_2$ opcode with an invalid rd value causes an *illegal_instruction* exception.

# STXFSR

*Exceptions*

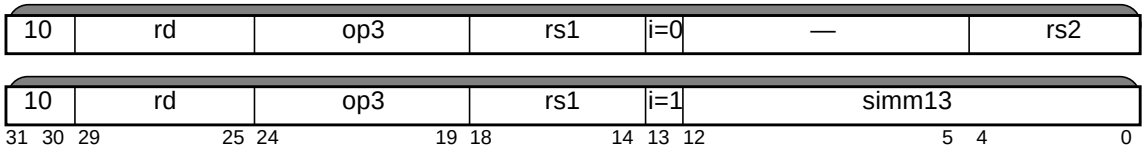
*illegal_instruction*
*fp_disabled*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

*See Also*

*Load Floating-Point State Register* on page 275
*Store Floating-Point* on page 342
*Store Floating-Point State Register (Lower)* on page 349

# 7.102 Subtract

| Instruction | op3 | Operation | Assembly Language Syntax | | | | Class |
|---|---|---|---|---|---|---|---|
| SUB | 00 0100 | Subtract | `sub` | $reg_{rs1}$, | *reg_or_imm*, | $reg_{rd}$ | **A1** |
| SUBcc | 01 0100 | Subtract and modify cc's | `subcc` | $reg_{rs1}$, | *reg_or_imm*, | $reg_{rd}$ | **A1** |
| SUBC | 00 1100 | Subtract with Carry | `subc` | $reg_{rs1}$, | *reg_or_imm*, | $reg_{rd}$ | **A1** |
| SUBCcc | 01 1100 | Subtract with Carry and modify cc's | `subccc` | $reg_{rs1}$, | *reg_or_imm*, | $reg_{rd}$ | **A1** |




*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 instruction with rd = 0 can be used to effect a signed or unsigned integer comparison. See the `cmp` synthetic instruction in Appendix C, *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).

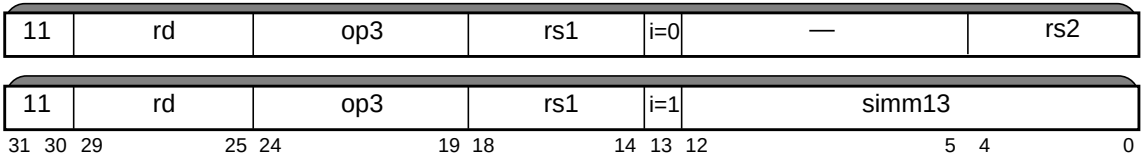
An attempt to execute a SUB instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

# 7.103 Swap Register with Memory

The SWAP instruction is deprecated and should not be used in new software. The CASA or CASXA instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| SWAP$^{D}$ | 00 1111 | Swap Register with Memory | `swap` [*address*], $reg_{rd}$ | **D2** |




*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.

SWAP accesses memory using the implicit ASI (see page 104). The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext(**simm13)" if i = 1.

An attempt to execute a SWAP instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the effective address is not word-aligned, an attempt to execute a SWAP instruction causes a *mem_address_not_aligned* exception.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

*Exceptions*

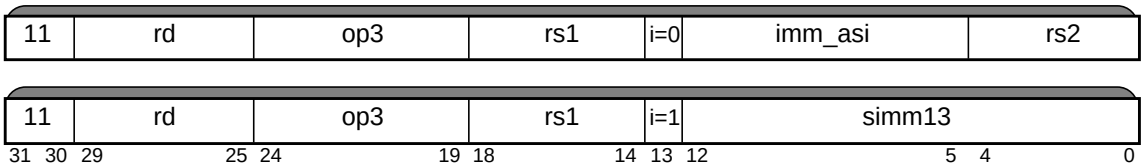
*illegal_instruction*
*mem_address_not_aligned*
*VA_watchpoint*
*DAE_privilege_violation*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

# 7.104 Swap Register with Alternate Space Memory

The SWAPA instruction is deprecated and should not be used in new software. The CASXA instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| SWAPA$^{D,\ P_{ASI}}$ | 01 1111 | Swap register with Alternate Space Memory | `swapa` [*regaddr*] *imm_asi*, $reg_{rd}$<br>`swapa` [*reg_plus_imm*] `%asi`, $reg_{rd}$ | | **D2, Y3**‡ |

‡ **Y3** for restricted ASIs ($00_{16}$-$7F_{16}$); **D2** for unrestricted ASIs ($80_{16}$-$FF_{16}$)




*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-V9).

If the effective address is not word-aligned, an attempt to execute a SWAPA instruction causes a *mem_address_not_aligned* exception.

# SWAPA (Deprecated)

In nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), if bit 7 of the ASI is 0, this instruction causes a *privileged_action* exception. In privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), if the ASI is in the range $30_{16}$ to $7F_{16}$, this instruction causes a *privileged_action* exception.

SWAPA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *DAE_invalid_asi* exception.

**ASIs valid for SWAPA**

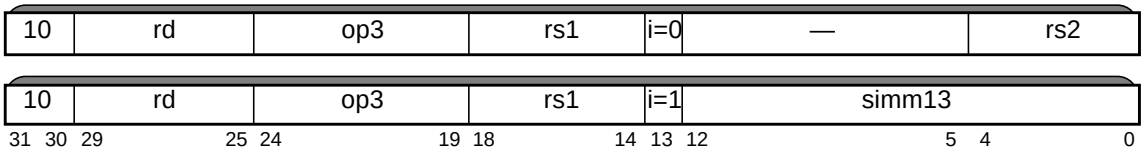
| | |
|---|---|
| ASI_NUCLEUS | ASI_NUCLEUS_LITTLE |
| ASI_AS_IF_USER_PRIMARY | ASI_AS_IF_USER_PRIMARY_LITTLE |
| ASI_AS_IF_USER_SECONDARY | ASI_AS_IF_USER_SECONDARY_LITTLE |
| ASI_PRIMARY | ASI_PRIMARY_LITTLE |
| ASI_SECONDARY | ASI_SECONDARY_LITTLE |
| ASI_REAL | ASI_REAL_LITTLE |

*Exceptions*

*mem_address_not_aligned*
*privileged_action*
*VA_watchpoint*
*DAE_invalid_asi*
*DAE_privilege_violation*
*DAE_nc_page*
*DAE_nfo_page*
*fast_data_access_MMU_miss*
*data_access_MMU_miss*
*data_access_MMU_error*
*fast_data_access_protection*
*PA_watchpoint*
*data_access_error*

# 7.105 Tagged Add

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| TADDcc | 10 0000 | Tagged Add and modify cc's | `taddcc` *reg$_{rs1}$*, *reg_or_imm*, *reg$_{rd}$* | **A1** |




*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.

An attempt to execute a TADDcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

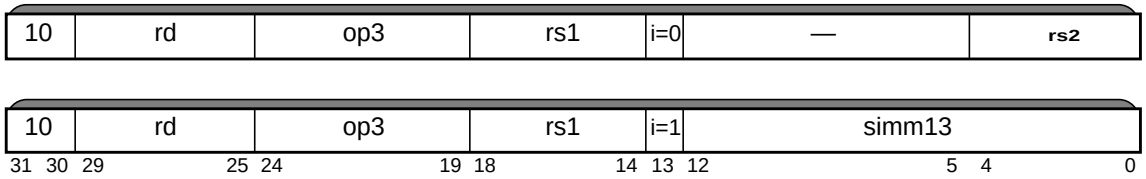
*Exceptions* *illegal_instruction*

*See Also* TADDccTV$^{D}$ on page 369
TSUBcc on page 374

# 7.106 Tagged Add and Trap on Overflow

The TADDccTV instruction is deprecated and should not be used in new software. The TADDcc instruction followed by the BPVS instruction (with instructions to save the pre-TADDcc integer condition codes if necessary) should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| TADDccTV[D] | 10 0010 | Tagged Add and modify cc's or Trap on Overflow | `taddcctv` *reg$_{rs1}$, reg_or_imm, reg$_{rd}$* | **D2** |




*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.

An attempt to execute a TADDccTV instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

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 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 only on the basis of the normal 64-bit arithmetic overflow condition, like a normal 64-bit add.

# TADDccTV (Deprecated)

**SPARC V8 Compatibility Note**

TADDccTV traps based on the 32-bit overflow condition, just as in the SPARC V8 architecture. Although the tagged add instructions set the 64-bit condition codes CCR.xcc, there is no form of the instruction that traps on the 64-bit overflow condition.

*Exceptions*

*illegal_instruction*
*tag_overflow*

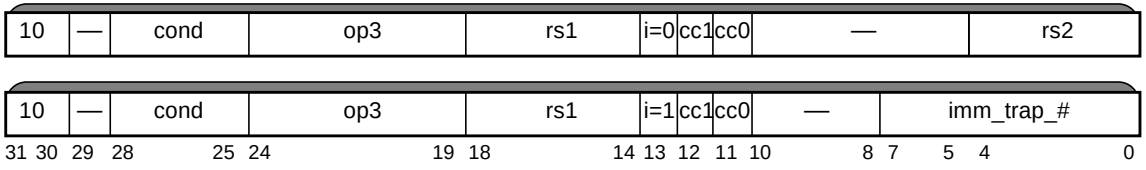
*See Also*

TADDcc on page 368
TSUBccTV$^{D}$ on page 375

# 7.107 Trap on Integer Condition Codes (Tcc)

| Instruction | op3 | cond | Operation | cc Test | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|---|
| TA | 11 1010 | 1000 | Trap Always | 1 | `ta` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TN | 11 1010 | 0000 | Trap Never | 0 | `tn` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TNE | 11 1010 | 1001 | Trap on Not Equal | **not** Z | `tne`† | *i_or_x_cc*, *software_trap_number* | **A1** |
| TE | 11 1010 | 0001 | Trap on Equal | Z | `te`‡ | *i_or_x_cc*, *software_trap_number* | **A1** |
| TG | 11 1010 | 1010 | Trap on Greater | **not** (Z **or** (N **xor** V)) | `tg` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TLE | 11 1010 | 0010 | Trap on Less or Equal | Z **or** (N **xor** V) | `tle` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TGE | 11 1010 | 1011 | Trap on Greater or Equal | **not** (N **xor** V) | `tge` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TL | 11 1010 | 0011 | Trap on Less | N **xor** V | `tl` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TGU | 11 1010 | 1100 | Trap on Greater, Unsigned | **not** (C **or** Z) | `tgu` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TLEU | 11 1010 | 0100 | Trap on Less or Equal, Unsigned | (C **or** Z) | `tleu` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TCC | 11 1010 | 1101 | Trap on Carry Clear (Greater than or Equal, Unsigned) | **not** C | `tcc`◊ | *i_or_x_cc*, *software_trap_number* | **A1** |
| TCS | 11 1010 | 0101 | Trap on Carry Set (Less Than, Unsigned) | C | `tcs`▽ | *i_or_x_cc*, *software_trap_number* | **A1** |
| TPOS | 11 1010 | 1110 | Trap on Positive or zero | **not** N | `tpos` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TNEG | 11 1010 | 0110 | Trap on Negative | N | `tneg` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TVC | 11 1010 | 1111 | Trap on Overflow Clear | **not** V | `tvc` | *i_or_x_cc*, *software_trap_number* | **A1** |
| TVS | 11 1010 | 0111 | Trap on Overflow Set | V | `tvs` | *i_or_x_cc*, *software_trap_number* | **A1** |

† *synonym:* `tnz` ‡ *synonym:* `tz` ◊ *synonym:* `tgeu` ▽ *synonym:* `tlu`



| cc1 :: cc0 | Condition Codes Evaluated |
|---|---|
| 00 | CCR.icc |
| 01 | — (*illegal_instruction*) |
| 10 | CCR.xcc |
| 11 | — (*illegal_instruction*) |

*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* or *htrap_instruction* exception is generated. If FALSE, the *trap_instruction* (or *htrap_instruction*) exception does not occur and the instruction behaves like a NOP.

For brevity, in the remainder of this section the value of the "software trap number" used by Tcc will be referred to as "SWTN".

In nonprivileged mode, if i = 0 the SWTN is specified by the least significant seven bits of "R[rs1] + R[rs2]". If i = 1, the SWTN is provided by the least significant seven bits of "R[rs1] + imm_trap_#". Therefore, the valid range of values for SWTN in nonprivileged mode is 0 to 127. The most significant 57 bits of SWTN are unused and should be supplied as zeroes by software.

In privileged and hyperprivileged modes, if i = 0 the SWTN is specified by the least significant eight bits of "R[rs1] + R[rs2]". If i = 1, the SWTN is provided by the least significant eight bits of "R[rs1] + *imm_trap_#*". Therefore, the valid range of values for SWTN in privileged and hyperprivileged modes is 0 to 255. The most significant 56 bits of SWTN are unused an should be supplied as zeroes by software.

Generally, values of $0 \le SWTN \le 127$ are used to trap to privileged-mode software and values of $128 \le SWTN \le 255$ are used to trap to hyperprivileged-mode software. The behavior of Tcc, based on the privilege mode in effect when it is executed and the value of the supplied SWTN, is as follows:

| Privilege Mode in effect when Tcc is executed | Behavior of Tcc instruction: 0 ≤ SWTN ≤ 127 | Behavior of Tcc instruction: 128 ≤ SWTN ≤ 255 |
|---|---|---|
| Nonprivileged (PSTATE.priv = 0 **and** HSTATE.hpriv = 0) | *trap_instruction* exception (to privileged mode) (256 ≤ TT ≤ 383) | — (not possible, because SWTN is a 7-bit value in nonprivileged mode) |
| Privileged (PSTATE.priv = 1 **and** HSTATE.hpriv = 0) | *trap_instruction* exception (to privileged mode) (256 ≤ TT ≤ 383) | *htrap_instruction* exception (to hyperprivileged mode) (384 ≤ TT ≤ 511) |
| Hyperprivileged (**and** HSTATE.hpriv = 1) | *htrap_instruction* exception (to hyperprivileged mode) (256 ≤ TT ≤ 383) | *htrap_instruction* exception (to hyperprivileged mode) (384 ≤ TT ≤ 511) |

**Programming Note** Tcc can be used to implement breakpointing, tracing, and calls to privileged and hyperprivileged software. It can also be used for runtime checks, such as for out-of-range array indexes and integer overflow.

**Exceptions.** An attempt to execute a Tcc instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- instruction bit 29 is nonzero
- i = 0 and instruction bits 12:5 are nonzero
- i = 1 and instruction bits 10:8 are nonzero
- cc0 = 1

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then Tcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the Tcc instruction) is stored in TPC[TL] and the value of NPC from before the Tcc was executed is stored in TNPC[TL]. The full 64-bit (nonmasked) PC and NPC values are stored in TPC[TL] and TNPC[TL], regardless of the value of PSTATE.am.

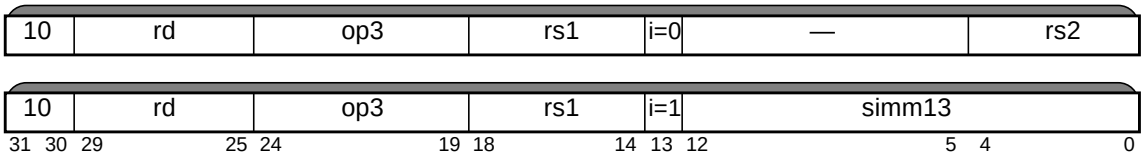
If a Tcc instruction causes a *trap_instruction* or *htrap_instruction* trap, 256 plus the SWTN value is written into TT[TL]. Then the trap is taken and the virtual processor performs the normal trap entry procedure, as described in *Trap Processing* on page 497.

*Exceptions*

*illegal_instruction*
*control_transfer_instruction* (impl. dep. #450-S20)
*trap_instruction* (0 ≤ SWTN ≤ 127)
*htrap_instruction* (128 ≤ SWTN ≤ 255)

# 7.108 Tagged Subtract

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| TSUBcc | 10 0001 | Tagged Subtract and modify cc's | `tsubcc` *reg$_{rs1}$*, *reg_or_imm*, *reg$_{rd}$* | **A1** |




*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.

An attempt to execute a TSUBcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

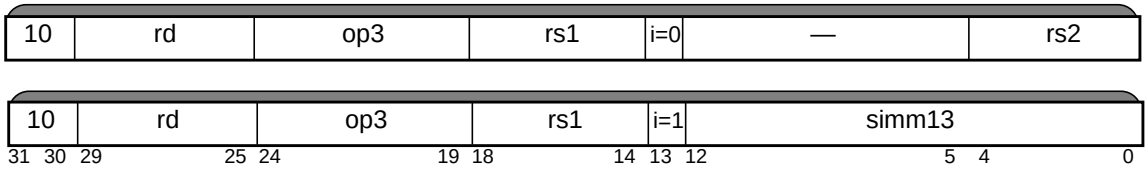
*Exceptions* *illegal_instruction*

*See Also* TADDcc on page 368
TSUBccTV$^{D}$ on page 375

# 7.109 Tagged Subtract and Trap on Overflow

The TSUBccTV instruction is deprecated and should not be used in new software. The TSUBcc instruction followed by BPVS instead (with instructions to save the pre-TSUBcc integer condition codes if necessary) should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| TSUBccTV[D] | 10 0011 | Tagged Subtract and modify cc's or Trap on Overflow | `tsubcctv` *reg$_{rs1}$, reg_or_imm, reg$_{rd}$* | **D2** |




*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 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].

An attempt to execute a TSUBccTV instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

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 only on the basis of the normal 64-bit arithmetic overflow condition, like a normal 64-bit subtract.

# TSUBccTV (Deprecated)

**SPARC V8 Compatibility Note** TSUBccTV traps based on the 32-bit overflow condition, just as in the SPARC V8 architecture. Although the tagged add instructions set the 64-bit condition codes CCR.xcc, there is no form of the instruction that traps on the 64-bit overflow condition.

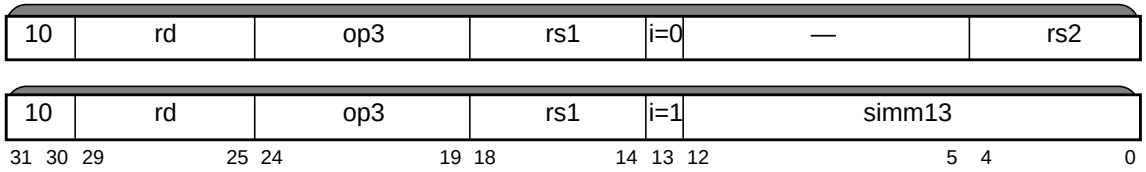
*Exceptions* *illegal_instruction*
*tag_overflow*

*See Also* TADDccTV$^{D}$ on page 369
TSUBcc on page 374

# 7.110 Unsigned Divide (64-bit ÷ 32-bit)

The UDIV and UDIVcc instructions are deprecated and should not be used in new software. The UDIVX instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|
| UDIV[D] | 00 1110 | Unsigned Integer Divide | `udiv` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **D2** |
| UDIVcc[D] | 01 1110 | Unsigned Integer Divide and modify cc's | `udivcc` | *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **D2** |




*Description*

The unsigned 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).

# UDIV, UDIVcc (Deprecated)

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 7-15.

**TABLE 7-15** UDIV / UDIVcc Overflow Detection and Value Returned

| Condition Under Which Overflow Occurs | Value Returned in R[rd] |
|---|---|
| Rational quotient $\geq 2^{32}$ | $2^{32} - 1$ ($0000\ 0000\ FFFF\ FFFF_{16}$) |

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 | Effect on bit of UDIVcc instruction |
|---|---|
| icc.n | Set if R[rd]{31} = 1 |
| icc.z | Set if R[rd]{31:0} = 0 |
| icc.v | Set if overflow (*per* TABLE 7-15) |
| 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 |

An attempt to execute a UDIV or UDIVcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions*

*illegal_instruction*
*division_by_zero*

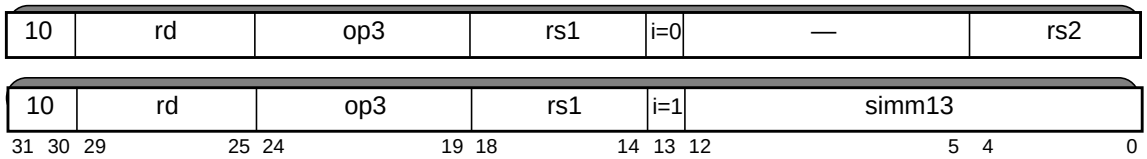
*See Also*

RDY on page 305
SDIV[cc] on page 323,
UMUL[cc] on page 379

# 7.111 Unsigned Multiply (32-bit)

The UMUL and UMULcc instructions are deprecated and should not be used in new software. The MULX instruction should be used instead.

| Opcode | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| UMUL[D] | 00 1010 | Unsigned Integer Multiply | `umul` *reg$_{rs1}$, reg_or_imm, reg$_{rd}$* | **D2** |
| UMULcc[D] | 01 1010 | Unsigned Integer Multiply and modify cc's | `umulcc` *reg$_{rs1}$, reg_or_imm, reg$_{rd}$* | **D2** |




*Description*

The unsigned 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.

UMUL does not affect the condition code bits. UMULcc writes the integer condition code bits, icc and xcc, as shown below.

| Bit | Effect on bit by execution of UMULcc |
|---|---|
| icc.n | Set to 1 if product{31} = 1; otherwise, set to 0 |
| icc.z | Set to 1 if product{31:0}= 0; otherwise, set to 0 |
| icc.v | Set to 0 |
| icc.c | Set to 0 |
| xcc.n | Set to 1 if product{63} = 1; otherwise, set to 0 |
| xcc.z | Set to 1 if product{63:0} = 0; otherwise, set to 0 |
| xcc.v | Set to 0 |
| xcc.c | Set to 0 |

**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.

# UMUL, UMULcc (Deprecated)

**Programming Notes** | 32-bit overflow after UMUL or UMULcc is indicated by Y ≠ 0.

An attempt to execute a UMUL or UMULcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions* *illegal_instruction*

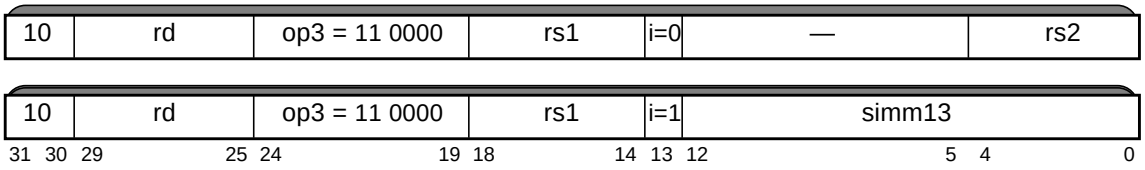
*See Also*
MULScc on page 287
RDY on page 305
SMUL[cc] on page 331,
UDIV[cc] on page 377

# 7.112 Write Ancillary State Register

| Instruction | rd | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| WRY[D] | 0 | Write Y register *(deprecated)* | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%y` | **D1** |
| — | 1 | *Reserved* | | |
| WRCCR | 2 | Write Condition Codes register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%ccr` | **A1** |
| WRASI | 3 | Write ASI register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%asi` | **A1** |
| — | 4 | *Reserved* (read-only ASR (TICK)) | | |
| — | 5 | *Reserved* (read-only ASR (PC)) | | |
| WRFPRS | 6 | Write Floating-Point Registers Status register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%fprs` | **A1** |
| — | 7–14 | *Reserved* | | |
| — | 15 | Software-initiated reset (see *Software-Initiated Reset* on page 328) | | |
| — | 16-18 | *Reserved* (impl. dep. #8-V8-Cs20, #9-V8-Cs20) | | |
| WRGSR | 19 | Write General Status register (GSR) | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%gsr` | **A1** |
| WRSOFTINT_SET[P] | 20 | Set bits of per-virtual processor Soft Interrupt register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`, %softint_set` | **N1** |
| WRSOFTINT_CLR[P] | 21 | Clear bits of per-virtual processor Soft Interrupt register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`, %softint_clr` | **N1** |
| WRSOFTINT[P] | 22 | Write per-virtual processor Soft Interrupt register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%softint` | **N1** |
| WRTICK_CMPR[P] | 23 | Write Tick Compare register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%tick_cmpr` | **N1** |
| WRSTICK[H] | 24 | Write System Tick register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%stick`† | **N1** |
| WRSTICK_CMPR[P] | 25 | Write System Tick Compare register | `wr` *reg$_{rs1}$*`,` *reg_or_imm*`,%stick_cmpr`† | **N1** |
| — | 26 | *Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | | |
| — | 27 | *Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | | |
| — | 28 | Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | | |
| — | 29–31 | Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20) | | |

† The original assembly language names for `%stick` and `%stick_cmpr` were, respectively, `%sys_tick` and `%sys_tick_cmpr`, which are now deprecated. Over time, assemblers will support the new `%stick` and `%stick_cmpr` names for these registers (which are consistent with `%tick` and `%tick_cmpr`). In the meantime, some existing assemblers may only recognize the original names.

# WRasr




*Description*

The WRasr instructions each store a value to the writable fields of the ancillary state register (ASR) specified by rd.

The value stored by these instructions (other than the implementation-dependent variants) is as follows: if i = 0, store the value "R[rs1] **xor** R[rs2]"; if i = 1, store "R[rs1] **xor sign_ext**(simm13)".

**Note** | The operation is **exclusive-or**.

The WRasr instruction with rs1 = 0 is a (deprecated) WRY instruction (which should not be used in new software). WRY is *not* a delayed-write instruction; the instruction immediately following a WRY observes the new value of the Y register.

The WRY instruction is deprecated. It is recommended that all instructions that reference the Y register be avoided.

WRCCR, WRFPRS, and WRASI are *not* delayed-write instructions. The instruction immediately following a WRCCR, WRFPRS, or WRASI 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.

**IMPL. DEP. #48-V8-Cs20**: WRasr instructions with rd in the range 16-18 and 26–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). 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 nonprivileged or privileged or hyperprivileged (impl. dep. #9-V8-Cs20), and
- whether an attempt to execute the instruction causes an *illegal_instruction* exception.

**Note** | See the section "Read/Write Ancillary State Registers (ASRs)" in *Extending the UltraSPARC Architecture*, contained in the separate volume *UltraSPARC Architecture Application Notes*, for a discussion of extending the SPARC V9 instruction set by means of read/write ASR instructions.

**V9 Compatibility Notes**

Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers.

The SPARC V8 WRIER, WRPSR, WRWIM, and WRTBR instructions do not exist in the UltraSPARC Architecture because the IER, PSR, TBR, and WIM registers do not exist in the UltraSPARC Architecture.

See *Ancillary State Registers* on page 69 for more detailed information regarding ASR registers.

**Exceptions.** An attempt to execute a WRasr instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- i = 0 **and** instruction bits 12:5 are nonzero
- rd = 1, 4, 5, 7–14, 18, or 26-31
- rd = 15 **and** ((rs1 ≠ 0) or (i = 0))
- the instruction is WRSTICK and the virtual processor is not in hyperprivileged mode (HPSTATE.hpriv = 0)

An attempt to execute a WRSOFTINT_SET, WRSOFTINT_CLR, WRSOFTINT, WRTICK_CMPR, or WRSTICK_CMPR instruction in nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0) causes a *privileged_opcode* exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a WRGSR instruction causes an *fp_disabled* exception.

**Implementation Note**

The SIR instruction shares an opcode with WRasr; they are distinguished by the rd, rs1, and i fields (rd = 15,rs1 = 0, and i = 1 for SIR). See *Software-Initiated Reset* on page 328.

*Exceptions*

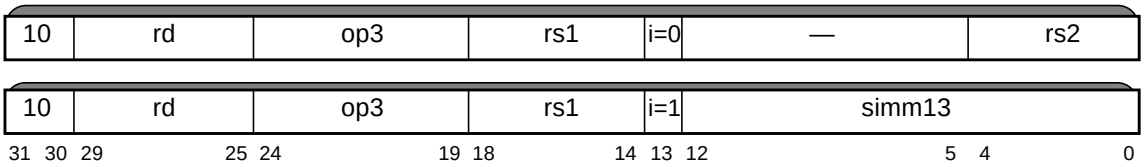
*illegal_instruction*
*privileged_opcode*
*fp_disabled*

*See Also*

RDasr on page 305
WRHPR on page 384
WRPR on page 386

# 7.113 Write Hyperprivileged Register

| Instruction | op3 | Operation | rd | Assembly Language Syntax | Class |
|---|---|---|---|---|---|
| WRHPR[H] | 11 0011 | Write hyperprivileged register | | | **N1** |
| | | HPSTATE | 0 | `wrhpr` *reg$_{rs1}$*, *reg_or_imm*, `%hpstate` | |
| | | HTSTATE | 1 | `wrhpr` *reg$_{rs1}$*, *reg_or_imm*, `%htstate` | |
| | | *Reserved* | 2 | | |
| | | HINTP | 3 | `wrhpr` *reg$_{rs1}$*, *reg_or_imm*, `%hintp` | |
| | | *Reserved* | 4 | | |
| | | HTBA | 5 | `wrhpr` *reg$_{rs1}$*, *reg_or_imm*, `%htba` | |
| | | *Reserved* | 6–29 | | |
| | | *Reserved* | 30 | | |
| | | HSTICK_CMPR | 31 | `wrhpr` *reg$_{rs1}$*, *reg_or_imm*, `%hsys_tick_cmpr` | |




*Description* A WRHPR 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 hyperprivileged state register.

**Note** | The operation is **exclusive-or**.

The rd field in the instruction determines the hyperprivileged register that is written. There are *MAXTL* copies of the HTSTATE register, one for each trap level. A write to one of these registers sets the copy of HTSTATE indexed by the current value in the trap-level register (TL).

The WRHPR instruction is a *non*-delayed-write instruction. The instruction immediately following the WRHPR observes any changes made to virtual processor state made by the WRHPR.

An attempt to execute a WRHPR instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- i = 0 and instruction bits 12:5 are nonzero
- rd = 2, 4, or 6-30 (reserved for future versions of the architecture)
- rd = 1 and TL = 0 (write to HTSTATE when the trap level is zero)
- virtual processor is in nonprivileged or privileged mode (HPSTATE.hpriv = 0)

A *trap_level_zero* disrupting trap can occur upon the *completion* of a WRHPR instruction to HPSTATE, if the following three conditions are true after WRHPR has executed:

- *trap_level_zero* exceptions are enabled (HPSTATE.tlz = 1),
- the virtual processor is in nonprivileged or privileged mode (HPSTATE.hpriv = 0), and
- the trap level (TL) register's value is zero (TL = 0)

**Programming Note** Execution of a WRHPR instruction that causes the value of HPSTATE.hpriv to change from 1 to 0 is not guaranteed to work if the WRHPR is in the delay slot of a DCTI instruction. Therefore, it is recommended that WRHPR not be executed in a delay slot, especially if it will toggle the value of HPSTATE.hpriv to 0.

**Programming Note** For historical reasons, the WRPR instruction, not WRHPR, is used to write to the hyperprivileged TICK register. See *Write Privileged Register* on page 386.

*Exceptions*

*illegal_instruction*
*trap_level_zero*

*See Also*

RDHPR on page 308
WRasr on page 381
WRPR on page 386

# 7.114 Write Privileged Register

| Instruction | op3 | Operation | rd | Assembly Language Syntax | | Class |
|---|---|---|---|---|---|---|
| WRPR[P] | 11 0010 | Write Privileged register | | | | **A1** |
| | | TPC | 0 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%tpc` | |
| | | TNPC | 1 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%tnpc` | |
| | | TSTATE | 2 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%tstate` | |
| | | TT | 3 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%tt` | |
| | | TICK | 4 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%tick` | |
| | | TBA | 5 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%tba` | |
| | | PSTATE | 6 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%pstate` | |
| | | TL | 7 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%tl` | |
| | | PIL | 8 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%pil` | |
| | | CWP | 9 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%cwp` | |
| | | CANSAVE | 10 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%cansave` | |
| | | CANRESTORE | 11 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%canrestore` | |
| | | CLEANWIN | 12 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%cleanwin` | |
| | | OTHERWIN | 13 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%otherwin` | |
| | | WSTATE | 14 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%wstate` | |
| | | *Reserved* | 15 | | | |
| | | GL | 16 | `wrpr` | *reg*$_{rs1}$, *reg_or_imm*, `%gl` | |
| | | *Reserved* | 17–31 | | | |

*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 *MAXTL* 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 WRPR to TL only stores a value to TL; it does not cause a trap, cause a return from a trap, or alter any machine state other than TL and state (such as PC, NPC, TICK, etc.) that is indirectly modified by every instruction.

**Programming Note** | A WRPR of TL can be used to read the values of TPC, TNPC, and TSTATE for any trap level; however, software must 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.

In privileged mode, *MAXPTL* is the maximum value that may be written by a WRPR to TL; an attempt to write a larger value results in *MAXPTL* being written to TL. In hyperprivileged mode, *MAXTL* is the maximum value that may be written by a WRPR to TL; an attempt to write a larger value results in *MAXTL* being written to TL. For details, see TABLE 5-21 on page 98.

In privileged mode, *MAXPGL* is the maximum value that may be written by a WRPR to GL; an attempt to write a larger value results in *MAXPGL* being written to GL. In hyperprivileged mode, *MAXGL* is the maximum value that may be written by a WRPR to GL; an attempt to write a larger value results in *MAXGL* being written to GL. For details, see TABLE 5-22 on page 101.

**Programming Note** | For historical reasons, the WRPR instruction, not WRHPR, is used to write to the hyperprivileged TICK register.

**Exceptions.** An attempt to execute a WRPR instruction in nonprivileged mode (PSTATE.priv = 0 and HSTATE.hpriv = 0) causes a *privileged_opcode* exception.

An attempt to execute a WRPR instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- i = 0 and instruction bits 12:5 are nonzero
- (rd = 4) and (PSTATE.priv = 1 and HSTATE.hpriv = 0)
  (an attempt to write to hyperprivileged register TICK while in privileged mode)
- rd = 15, or 17-31 (reserved for future versions of the architecture)
- 0 ≤ rd ≤ 3 (attempt to write TPC, TNPC,TSTATE, or TT register) while TL = 0 (current trap level is zero) and the virtual processor is in privileged or hyperprivileged mode.

**Implementation Note** | In nonprivileged mode, *illegal_instruction* exception due to 0 ≤ rd ≤ 3 and TL = 0 does not occur; the *privileged_opcode* exception occurs instead.

A *trap_level_zero* disrupting trap can occur upon the *completion* of a WRPR instruction to TL, if the following three conditions are true after WRPR has executed:

- *trap_level_zero* exceptions are enabled (HPSTATE.tlz = 1)
- the virtual processor is in nonprivileged or privileged mode (HPSTATE.hpriv = 0), and

- the trap level (TL) register's value is zero (TL = 0)

*Exceptions*

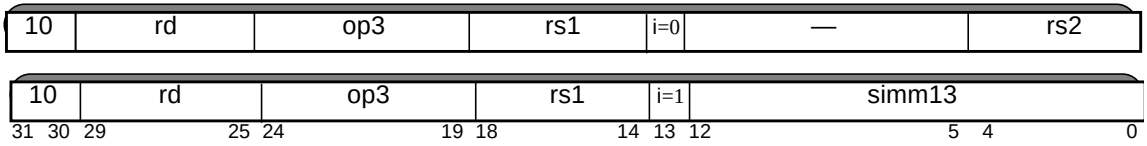
*privileged_opcode*
*illegal_instruction*
*trap_level_zero*

*See Also*

RDPR on page 309
WRasr on page 381
WRHPR on page 384

# 7.115 XOR Logical Operation

| Instruction | op3 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| XOR | 00 0011 | Exclusive **or** | `xor` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| XORcc | 01 0011 | Exclusive **or** and modify cc's | `xorcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| XNOR | 00 0111 | Exclusive **nor** | `xnor` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |
| XNORcc | 01 0111 | Exclusive **nor** and modify cc's | `xnorcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ | **A1** |




*Description*

These instructions implement bitwise logical **xor** 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].

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)

**Programming Note** | XNOR (and XNORcc) is identical to the **xor_not** (and set condition codes) **xor_not_cc** logical operation, respectively.

An attempt to execute an XOR, XORcc, XNOR, or XNORcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

*Exceptions*

*illegal_instruction*

# XOR / XNOR

CHAPTER 8

# IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007

The IEEE Std 754-1985 floating-point standard contains a number of implementation dependencies. This chapter specifies choices for these implementation dependencies, to ensure that SPARC V9 implementations are as consistent as possible.

The chapter contains these major sections:


- **Traps Inhibiting Results** on page 391.
- **Underflow Behavior** on page 392.
- **Integer Overflow Definition** on page 393.
- **Floating-Point Nonstandard Mode** on page 394.
- **Arithmetic Result Tables** on page 395.


Exceptions are discussed in this chapter 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.

## 8.1 Traps Inhibiting Results

As described in *Floating-Point State Register (FSR)* on page 60 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_exception*s; 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 FPops execute as if by hardware; that is, such a trap is undetectable by application software, except that timing may be affected.

**Programming Note**

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 privileged software handling of an *fp_exception_other* trap with FSR.ftt = unfinished_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, FSR.qne, and reserved fields of FSR are zero.

## 8.2 Underflow Behavior

An UltraSPARC Architecture virtual processor detects tininess before rounding occurs. (impl. dep. #55-V8-Cs10)

TABLE 8-1 summarizes what happens when an exact *unrounded* value *u* satisfying

$$0 \le |u| \le \textit{smallest normalized number}$$

would round, if no trap intervened, to a *rounded* value *r* which might be zero, subnormal, or the smallest normalized value.

**TABLE 8-1** Floating-Point Underflow Behavior (Tininess Detected Before Rounding)

| | Underflow trap: / Inexact trap: | ufm = 1<br>nxm = x | ufm = 0<br>nxm = 1 | ufm = 0<br>nxm = 0 |
|---|---|---|---|---|
| $u = r$ | *r* is minimum normal | None | None | None |
| | *r* is subnormal | UF | None | None |
| | *r* is zero | None | None | None |
| $u \neq r$ | *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<br>NX = *fp_exception_ieee_754* trap with cexc.nxc = 1<br>uf = cexc.ufc = 1, aexc.ufa = 1, no *fp_exception_ieee_754* trap<br>nx = cexc.nxc = 1, aexc.nxa = 1, no *fp_exception_ieee_754* trap | | | | |

### 8.2.1 Trapped Underflow Definition (ufm = 1)

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 the UltraSPARC Architecture at the hardware, hyperprivileged, and privileged software levels. If they are created at all, it would be by user software in a nonprivileged-mode trap handler.

### 8.2.2 Untrapped Underflow Definition (ufm = 0)

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.

## 8.3 Integer Overflow Definition

- **F<sdq>TOi** — When a NaN, infinity, large positive argument $\geq 2^{31}$ or large negative argument $\leq -(2^{31} + 1)$ is converted to an integer, the invalid_current (nvc) bit of FSR.cexc is set to 1, and if the floating-point invalid trap is enabled (FSR.tem.nvm = 1), the *fp_exception_IEEE_754* exception is 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 $2^{31} - 1$; if the sign bit of the operand is 1, the result is $-2^{31}$.

- **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 is set to 1, and if the floating-point invalid trap is enabled (FSR.tem.nvm = 1), the *fp_exception_IEEE_754* exception is 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 $-2^{63}$.

# 8.4 Floating-Point Nonstandard Mode

If implemented, floating-point nonstandard mode is enabled by setting FSR.ns = 1 (see *Nonstandard Floating-Point (ns)* on page 62).

An UltraSPARC Architecture 2007 processor may choose to implement nonstandard floating-point mode in order to obtain higher performance in certain circumstances. For example, when FSR.ns = 1 an implementation that processes fully normalized operands more efficiently than subnormal operands may convert a subnormal floating-point operand or result to zero.

**Implementation Note** | UltraSPARC Architecture virtual processors are strongly discouraged from implementing a nonstandard floating-point mode.

Implementations are encouraged to support standard IEEE 754 floating-point arithmetic with reasonable performance in all cases, even if some cases are slower than others.

Assuming that nonstandard floating-point mode is implemented, the effects of FSR.ns = 1 are as follows:

- **IMPL. DEP. #18-V8-Ms10(a):** When FSR.ns = 1 and a floating-point *source operand* is subnormal, an implementation may treat the subnormal operand as if it were a floating-point zero value of the same sign.
  The cases in which this replacement is performed are implementation dependent. However, if it occurs,
  (1) it should *not* apply to FABS, FMOV, or FNEG instructions and
  (2) FADD, FSUB, and FCMP should give identical treatment to subnormal source operands.
  Treating a subnormal source operand as zero may generate an IEEE 754 floating-point "inexact", "division by zero", or "invalid" condition (see *Current Exception (cexc)* on page 66). Whether the generated condition(s) trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem.

- **IMPL. DEP. #18-V8-Ms10(b):** When a floating-point operation generates a subnormal *result* value, an UltraSPARC Architecture 2007 implementation may either write the result as a subnormal value or replace the subnormal result by a floating-point zero value of the same sign and generate IEEE 754 floating-point "inexact" and "underflow" conditions. Whether these generated conditions trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem.
- **IMPL. DEP. #18-V8-Ms10(c):** If an FPop generates an *intermediate* result value, the intermediate value is subnormal, and FSR.ns = 1, it is implementation dependent whether (1) the operation continues, using the subnormal value (possibly with some loss of accuracy), or (2) the virtual processor replaces the subnormal intermediate value with a floating-point zero value of the same sign, generates IEEE 754 floating-point "inexact" and "underflow" conditions, completes the instruction, and writes a final result (possibly with some loss of accuracy). Whether generated IEEE conditions trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem.

If GSR.im = 1, then the value of FSR.ns is ignored and the processor operates as if FSR.ns = 0 (see page 77).

## 8.5 Arithmetic Result Tables

This section contains detailed tables, showing the results produced by various floating-point operations, depending on their source operands.

Notes on source types:

- N*n* is a number in F[rs*n*], which may be normal or subnormal.
- QNaN*n* and SNaN*n* are Quiet and Signaling Not-a-Number values in F[rs*n*], respectively.

Notes on result types:

- R: (rounded) result of operation, which may be normal, subnormal, zero, or infinity. May also cause OF, UF, NX, unfinished.
- dQNaN is the generated default Quiet NaN (sign = 0, exponent = all 1s, fraction = all 1s). The sign of the default Quiet NaN is zero to distinguish it from storage initialized to all ones.
- QSNaN*n* is the Signalling NaN operand from F[rs*n*] with the Quiet bit asserted

## 8.5.1 Floating-Point Add (FADD)

**TABLE 8-2** Floating-Point Add operation (F[rs1] + F[rs2])

| F[rs1] \ F[rs2] | −∞ | −N2 | −0 | +0 | +N2 | +∞ | QNaN2 | SNaN2 |
|---|---|---|---|---|---|---|---|---|
| −∞ | -∞ | | | | | dQNaN, NV | QNaN2 | QSNaN2, NV |
| −N1 | | −R | −N1 | | ±R* | | | |
| −0 | | −N2 | −0 | ±0** | +N2 | | | |
| +0 | | | ±0** | +0 | | | | |
| +N1 | | ±R* | +N1 | | +R | | | |
| +∞ | dQNaN, NV | | | | | +∞ | | |
| QNaN1 | QNaN1 | | | | | | | |
| SNaN1 | QSNaN1, NV | | | | | | | |

* if N1 = -N2, then **

** result is +0 unless rounding mode is round to −∞, in which case the result is −0

For the FADD instructions, R may be any number; its generation may cause OF, UF, and/or NX.

Floating-point add is not commutative when both operands are NaN.

## 8.5.2 Floating-Point Subtract (FSUB)

**TABLE 8-3** Floating-Point Subtract operation (F[rs1] – F[rs2])

| F[rs1] \ F[rs2] | −∞ | −N2 | −0 | +0 | +N2 | +∞ | QNaN2 | SNaN2 |
|---|---|---|---|---|---|---|---|---|
| −∞ | dQNaN, NV | | | | | -∞ | QNaN2 | QSNaN2, NV |
| −N1 | | ±R* | −N1 | | −R | | | |
| −0 | | +N2 | ±0** | −0 | −N2 | | | |
| +0 | | | +0 | ±0** | | | | |
| +N1 | | +R | +N1 | | ±R* | | | |
| +∞ | +∞ | | | | | dQNaN, NV | | |
| QNaN1 | QNaN1 | | | | | | | |
| SNaN1 | QSNaN1, NV | | | | | | | |

* if N1 = N2, then **

** result is +0 unless rounding mode is round to –∞, in which case the result is –0

For the FSUB instructions, R may be any number; its generation may cause OF, UF, and/or NX.

Note that $-x \neq 0-x$ when $x$ is zero or NaN.

## 8.5.3 Floating-Point Multiply

**TABLE 8-4** Floating-Point Multiply operation (F[rs1] × F[rs2])

| F[rs1] \ F[rs2] | −∞ | −N2 | −0 | +0 | +N2 | +∞ | QNaN2 | SNaN2 |
|---|---|---|---|---|---|---|---|---|
| -∞ | +∞ | | dQNaN, NV | | | -∞ | QNaN2 | QSNaN2, NV |
| −N1 | | +R | | | −R | | | |
| − 0 | dQNaN, NV | + 0 | | −0 | | dQNaN, NV | | |
| +0 | | −0 | | +0 | | | | |
| + N1 | | −R | | | +R | | | |
| +∞ | -∞ | | dQNaN, NV | | | +∞ | | |
| QNaN1 | QNaN1 | | | | | | | |
| SNaN1 | QSNaN1, NV | | | | | | | |

R may be any number; its generation may cause OF, UF, and/or NX.

Floating-point multiply is not commutative when both operands are NaN.

FsMULd (FdMULq) never causes OF, UF, or NX.

A NaN input operand to FsMULd (FdMULq) must be widened to produce a double-precision (quad-precision) NaN output, by filling the least-significant bits of the NaN result with zeros.

## 8.5.4 Floating-Point Multiply-Add (FMADD)

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 397) to 0000000e0lect a row in the table below.

**TABLE 8-5** Floating-Point Multiply-Add ((F[rs1] × F[rs2]) + F[rs3])

| | | **F[rs3]** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **−∞** | **−N3** | **−0** | **+0** | **+N3** | **+∞** | **QNaN3** | **SNaN3** |
| **F[rs1] × F[rs2]** | **−∞** | −∞ | | | | | dQNaN, NV | QNaN3 | QSNaN3, NV |
| | **−N** | | −R | −N | | ±R* | | | |
| | **−0** | | −N3 | −0 | ±0** | +N3 | | | |
| | **+0** | | | ±0** | +0 | | | | |
| | **+N** | | ±R* | +N | | +R | | | |
| | **+∞** | dQNaN, NV | | | | | +∞ | | |
| | **QNaN1** | QNaN1 | | | | | | | |
| | **QNaN2** | QNaN2 | | | | | | | |
| | **QNaN** (±0 × ±∞) | dQNaN, NV*** | | | | | | QNaN3, NV*** | |
| | **QSNaN1** | QSNaN1, NV*** | | | | | | | |
| | **QSNaN2** | QSNaN2, NV*** | | | | | | | |

* if N = −N3, then **

** result is +0 unless rounding mode is round to −∞, in which case the result is −0

*** if FSR.nvm = 1, FSR.nvc ← 1, the trap occurs, and FSR.aexc is left unchanged; otherwise, FSR.nvm = 0 so FSR.nva ← 1 and for FMADD FSR.nvc ← 1.

In the above table, R may be any number; its generation may cause OF, UF, and/or NX

The multiply operation in fused floating-point multiply-add (FMADD) instructions cannot cause inexact, underflow, or overflow exceptions.

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

## 8.5.5 Floating-Point Negative Multiply-Add (FNMADD)

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 397) to select a row in the table below.

**TABLE 8-6** Floating-Point Negative Multiply-Add (–(F[rs1] × F[rs2])– F[rs3])

| F[rs1] × F[rs2] | F[rs3] = –∞ | F[rs3] = –N3 | F[rs3] = –0 | F[rs3] = +0 | F[rs3] = +N3 | F[rs3] = +∞ | F[rs3] = QNaN3 | F[rs3] = SNaN3 |
|---|---|---|---|---|---|---|---|---|
| –∞ | +∞ | | | | | dQNaN, NV | QNaN3 | QSNaN3, NV |
| –N | | +R | +N | | ±R* | | | |
| –0 | | +N3 | +0 | ±0** | –N3 | | | |
| +0 | | | ±0** | –0 | | | | |
| +N | | ±R* | –N | | –R | | | |
| +∞ | dQNaN, NV | | | | | –∞ | | |
| QNaN1 | QNaN1 | | | | | | | |
| QNaN2 | QNaN2 | | | | | | | |
| QNaN (±0 × ±∞) | dQNaN, NV*** | | | | | | QNaN3 NV*** | |
| QSNaN1 | QSNaN1, NV*** | | | | | | | |
| QSNaN2 | QSNaN2, NV*** | | | | | | | |

* if N = –N3, then **

** result is +0 unless rounding mode is round to –∞, in which case the result is –0

*** if FSR.nvm = 1, FSR.nvc ← 1, the trap occurs, and FSR.aexc is left unchanged; otherwise, FSR.nvm = 0 so FSR.nva ← 1 and for FMADD FSR.nvc ← 1.

R may be any number; its generation may cause OF, UF, and/or NX.

The multiply operation in fused floating-point negative multiply-add (FNMADD) instructions cannot cause inexact, underflow, or overflow exceptions.

Note that rounding occurs after the negation. Thus, when the rounding mode is towards ±∞, FNMADD is not equivalent to FMADD followed by FNEG.

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

## 8.5.6 Floating-Point Multiply-Subtract (FMSUB)

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 397) to select a row in the table below.

**TABLE 8-7** Floating-Point Multiply-Subtract ((F[rs1] × F[rs2])– F[rs3])

| F[rs1] × F[rs2] | F[rs3] −∞ | −N3 | −0 | +0 | +N3 | +∞ | QNaN3 | SNaN3 |
|---|---|---|---|---|---|---|---|---|
| −∞ | dQNaN, NV | | | | | − ∞ | QNaN3 | QSNaN3, NV |
| −N | +∞ | ±R* | −N | | −R | | | |
| −0 | | +N3 | ±0** | −0 | −N3 | | | |
| +0 | | | +0 | ±0** | | | | |
| +N | | +R | +N | | ±R* | | | |
| +∞ | | | | | | dQNaN, NV | | |
| QNaN1 | QNaN1 | | | | | | | |
| QNaN2 | QNaN2 | | | | | | | |
| QNaN (±0 × ±∞) | dQNaN, NV*** | | | | | | QNaN3, NV*** | |
| QSNaN1 | QSNaN1, NV*** | | | | | | | |
| QSNaN2 | QSNaN2, NV*** | | | | | | | |

* if N = N3, then **

** result is +0 unless rounding mode is round to −∞, in which case the result is −0

*** if FSR.nvm = 1, FSR.nvc ← 1, the trap occurs, and FSR.aexc is left unchanged; otherwise, FSR.nvm = 0 so FSR.nva ← 1 and for FMSUB FSR.nvc ← 1.

R may be any number; its generation may cause OF, UF, and/or NX..

The multiply operation in fused floating-point multiply-subtract (FMSUB) instructions cannot cause inexact, underflow, or overflow exceptions.

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

## 8.5.7 Floating-Point Negative Multiply-Subtract (FNMSUB)

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 397) to select a row in the table below.

**TABLE 8-8** Floating-Point Negative Multiply-Subtract ( – (F[rs1] × F[rs2]) + F[rs3] )

| F[rs1] × F[rs2] | F[rs3] = –∞ | –N3 | –0 | +0 | +N3 | +∞ | QNaN3 | SNaN3 |
|---|---|---|---|---|---|---|---|---|
| –∞ | dQNaN, NV | | | | | +∞ | QNaN3 | QSNaN3, NV |
| –N | – ∞ | ±R* | +N | | +R | | | |
| –0 | | –N3 | ±0** | +0 | +N3 | | | |
| +0 | | | –0 | ±0** | | | | |
| +N | | –R | –N | | ±R* | | | |
| +∞ | | | | | | dQNaN, NV | | |
| QNaN1 | QNaN1 | | | | | | | |
| QNaN2 | QNaN2 | | | | | | | |
| QNaN (±0 × ±∞) | dQNaN, NV*** | | | | | | QNaN3, NV*** | |
| QSNaN1 | QSNaN1, NV*** | | | | | | | |
| QSNaN2 | QSNaN2, NV*** | | | | | | | |

* if N = N3, then **

** result is +0 unless rounding mode is round to –∞, in which case the result is –0

*** if FSR.nvm = 1, FSR.nvc ← 1, the trap occurs, and FSR.aexc is left unchanged; otherwise, FSR.nvm = 0 so FSR.nva ← 1 and for FNMSUB FSR.nvc ← 1.

R may be any number; its generation may cause OF, UF, and/or NX...

The multiply operation in fused floating-point negative multiply-subtract (FNMSUB) instructions cannot cause inexact, underflow, or overflow exceptions.

Note that rounding occurs after the negation. Thus, FNMSUB is not equivalent to FMSUB followed by FNEG when the rounding mode is towards ±∞.

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

## 8.5.8 Floating-Point Divide (FDIV)

**TABLE 8-9** Floating-Point Divide operation (F[rs1] ÷ F[rs2])

| F[rs1] \ F[rs2] | −∞ | −N2 | −0 | +0 | +N2 | +∞ | QNaN2 | SNaN2 |
|---|---|---|---|---|---|---|---|---|
| −∞ | dQNaN, NV | +∞ | | −∞ | | dQNaN, NV | QNaN2 | QSNaN2, NV |
| −N1 | +R | | +∞, DZ | −∞, DZ | −R | | | |
| −0 | +0 | | dQNaN, NV | | | −0 | | |
| +0 | −0 | | | | | +0 | | |
| +N1 | −R | | −∞, DZ | +∞, DZ | +R | | | |
| +∞ | dQNaN, NV | −∞ | | +∞ | | dQNaN, NV | | |
| QNaN1 | QNaN1 | | | | | | | |
| SNaN1 | QSNaN1, NV | | | | | | | |

R may be any number; its generation may cause OF, UF, and/or NX.

## 8.5.9 Floating-Point Square Root (FSQRT)

**TABLE 8-10** Floating-Point Square Root operation ($\sqrt{F[rs2]}$)

| F[rs2] | | | | | | | |
|---|---|---|---|---|---|---|---|
| −∞ | −N2 | −0 | +0 | +N2 | +∞ | QNaN2 | SNaN2 |
| dQNaN, NV | | −0 | +0 | +R | +∞ | QNaN2 | QSNaN2, NV |

R may be any number; its generation may cause NX.

Square root cannot cause DZ, OF, or UF.

## 8.5.10 Floating-Point Compare (FCMP, FCMPE)

**TABLE 8-11** Floating-Point Compare (FCMP, FCMPE) operation (F[rs1] ? F[rs2])

| F[rs1] \ F[rs2] | −∞ | −N2 | −0 | +0 | +N2 | +∞ | QNaN2 | SNaN2 |
|---|---|---|---|---|---|---|---|---|
| −∞ | 0 | | | | | | | |
| −N1 | | 0, 1, 2 | | | 1 | | | |
| −0 | | | 0 | | | | | |
| +0 | | | | | | | | |
| +N1 | | 2 | | | 0,1,2 | | | |
| +∞ | | | | | | 0 | | |
| QNaN1 | | | | | | | 3, NV* | |
| SNaN1 | | | | | | | | 3, NV |

* NV for FCMPE, but not for FCMP.

**TABLE 8-12** FSR.fcc Encoding for Result of FCMP, FCMPE

| fcc result | meaning |
|---|---|
| 0 | = |
| 1 | < |
| 2 | > |
| 3 | unordered |

NaN is considered to be unequal to anything else, even the identical NaN bit pattern.

FCMP/FCMPE cannot cause DZ, OF, UF, NX.

## 8.5.11 Floating-Point to Floating-Point Conversions (F<s | d | q>TO<s | d | q>)

**TABLE 8-13** Floating-Point to Float-Point Conversions (convert(F[rs2]))

| | F[rs2] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | −SNaN2 | −QNaN2 | −∞ | −N2 | −0 | +0 | +N2 | +∞ | +QNaN2 | +SNaN2 |
| | −QSNaN2, NV | −QNaN2 | −∞ | −R | −0 | +0 | +R | +∞ | +QNaN2 | +QSNaN2, NV |

For FsTOd:

- the least-significant fraction bits of a normal number are filled with zero to fit in double-precision format
- the least-significant bits of a NaN result operand are filled with zero to fit in double-precision format

For FsTOq and FdTOq:

- the least-significant fraction bits of a normal number are filled with zero to fit in quad-precision format
- the least-significant bits of a NaN result operand are filled with zero to fit in quad-precision format

For FqTOs and FdTOs:

- the fraction is rounded according to the current rounding mode
- the lower-order bits of a NaN source are discarded to fit in single-precision format; this discarding is not considered a rounding operation, and will not cause an NX exception

For FqTOd:

- the fraction is rounded according to the current rounding mode
- the least-significant bits of a NaN source are discarded to fit in double-precision format; this discarding is not considered a rounding operation, and will not cause an NX exception

**TABLE 8-14** Floating-Point to Float-Point Conversion Exception Conditions

| | |
|---|---|
| NV | • SNaN operand |
| OF | • FdTOs, FqTOs: the input is larger than can be expressed in single precision<br>• FqTOd: the input is larger than can be expressed in double precision<br>• does not occur during other conversion operations |
| UF | • FdTOs, FqTOs: the input is smaller than can be expressed in single precision<br>• FqTOd: the input is smaller than can be expressed in double precision<br>• does not occur during other conversion operations |
| NX | • FdTOs, FqTOs: the input fraction has more significant bits than can be held in a single precision fraction<br>• FqTOd: the input fraction has more significant bits than can be held in a double precision fraction<br>• does not occur during other conversion operations |

## 8.5.12 Floating-Point to Integer Conversions (F<s | d | q>TO<i | x>)

**TABLE 8-15** Floating-Point to Integer Conversions (convert(F[rs2]))

| | F[rs2] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **−SNaN2** | **−QNaN2** | **−∞** | **−N2** | **−0** | **+0** | **+N2** | **+∞** | **+QNaN2** | **+SNaN2** |
| **FdTOx FsTOx FqTOx** | $-2^{63}$, NV | | $-2^{63}$, NV | −R | 0 | | +R | $2^{63}-1$, NV | $2^{63}-1$, NV | |
| **FdTOi FsTOi FqTOi** | $-2^{31}$, NV | | $-2^{31}$, NV | | | | | $2^{31}-1$, NV | $2^{31}-1$, NV | |

R may be any integer, and may cause NV, NX.

Float-to-Integer conversions are always treated as round-toward-zero (truncated).

These operations are invalid (due to integer overflow) under the conditions described in *Integer Overflow Definition* on page 393.

**TABLE 8-16** Floating-point to Integer Conversion Exception Conditions

| | |
|---|---|
| NV | • SNaN operand<br>• QNaN operand<br>• ±∞ operand<br>• integer overflow |
| NX | • non-integer source (truncation occurred) |

## 8.5.13 Integer to Floating-Point Conversions (F<i | x>TO<s | d | q>)

**TABLE 8-17** Integer to Floating-Point Conversions (convert(F[rs2]))

| F[rs2] | | |
|---|---|---|
| **–int** | **0** | **+int** |
| –R | +0 | +R |

R may be any number; its generation may cause NX.

**TABLE 8-18** Floating-Point Conversion Exception Conditions

| | |
|---|---|
| NX | • FxTOd, FxTOs, FiTOs (possible loss of precision)<br>• not applicable to FiTOd, FxTOq, or FiTOq (FSR.cexc will always be cleared) |

CHAPTER 9

# Memory

The UltraSPARC Architecture *memory models* define the semantics of memory operations. The instruction set semantics require that loads and stores behave *as if* they are 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. For additional information on the use of the models in programming real systems, see *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

This chapter contains a great deal of theoretical information so that the discussion of the UltraSPARC Architecture TSO memory model has sufficient background.

This chapter describes memory models in these sections:


- **Memory Location Identification** on page 408.
- **Memory Accesses and Cacheability** on page 408.
- **Memory Addressing and Alternate Address Spaces** on page 411.
- **SPARC V9 Memory Model** on page 415.
- **The UltraSPARC Architecture Memory Model — TSO** on page 418.
- **Nonfaulting Load** on page 427.
- **Store Coalescing** on page 428.

# 9.1 Memory Location Identification

A memory location is identified by an 8-bit address space identifier (ASI) and a 64-bit memory address. The 8-bit ASI can be obtained from an ASI register or included in a memory access instruction. The ASI used for an access can distinguish among different 64-bit address spaces, such as Primary memory space, Secondary memory space, and internal control registers. It can also apply attributes to the access, such as whether the access should be performed in big- or little-endian byte order, or whether the address should be taken as a virtual, real, or physical address.

# 9.2 Memory Accesses and Cacheability

Memory is logically divided into real memory (cached) and I/O memory (noncached with and without side effects) spaces.

*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 (or, in the case of a load or load-store, on the destination register).

*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.

## 9.2.1 Coherence Domains

Two types of memory operations are supported in the UltraSPARC Architecture: cacheable and noncacheable accesses. The manner in which addresses are differentiated is implementation dependent. In some implementations, it is indicated in the page translation entry (TTE.cp), while in other implementations, it is indicated by a bit in the physical address.

Although SPARC V9 does not specify memory ordering between cacheable and noncacheable accesses, the UltraSPARC Architecture maintains TSO ordering between memory references regardless of their cacheability.

The UltraSPARC Architecture obeys the Sun-5 Ordering rules as documented in the "Sun-4u/Sun-5 Ordering with TSO" specification.

#### 9.2.1.1 Cacheable Accesses

Accesses within the coherence domain are called cacheable accesses. They have these properties:

- Data reside in real memory locations.
- Accesses observe supported cache coherency protocol(s).
- The cache line size is $2^n$ bytes (where $n \geq 4$), and can be different for each cache.

#### 9.2.1.2 Noncacheable Accesses

Noncacheable accesses are outside of the coherence domain. They have the following properties:

- Data might not reside in real memory locations. Accesses may result in programmer-visible side effects. An example is memory-mapped I/O control registers.
- Accesses do not observe supported cache coherency protocol(s).
- The smallest unit in each transaction is a single byte.

The UltraSPARC Architecture MMU optionally includes an attribute bit in each page translation, TTE.e, which when set signifies that this page has side effects.

Noncacheable accesses without side effects (TTE.e = 0) are processor-consistent and obey TSO memory ordering. In particular, processor consistency ensures that a noncacheable load that references the same location as a previous noncacheable store will load the data from the previous store.

Noncacheable accesses with side effects (TTE.e = 1) are processor consistent and are strongly ordered. These accesses are described in more detail in the following section.

#### 9.2.1.3 Noncacheable Accesses with Side-Effect

Loads, stores, and load-stores to I/O locations might not behave with memory semantics. Loads and stores could have side effects; for example, a read access could clear a register or pop an entry off a FIFO. A write access could set a register address port so that the next access to that address will read or write a particular internal register. Such devices are considered order sensitive. Also, such devices may only allow accesses of a fixed size, so store merging of adjacent stores or stores within a 16-byte region would cause an error (see *Store Coalescing* on page 428).

Noncacheable accesses (other than block loads and block stores) to pages with side effects (TTE.e = 1) exhibit the following behavior:

- Noncacheable accesses are strongly ordered with respect to each other. Bus protocol should guarantee that IO transactions to the same device are delivered in the order that they are received.
- Noncacheable loads with the TTE.e bit = 1 will not be issued to the system until all previous instructions have completed, and the store queue is empty.
- Noncacheable store coalescing is disabled for accesses with TTE.e = 1.
- A MEMBAR may be needed between side-effect and non-side-effect accesses. See TABLE 9-3 on page 425.

Whether block loads and block stores adhere to the above behavior or ignore TTE.e and always behave as if TTE.e = 0 is implementation-dependent (impl. dep. #410-S10, #411-S10).

On UltraSPARC Architecture virtual processors, noncacheable and side-effect accesses do not observe supported cache coherency protocols (impl. dep. #120).

Non-faulting loads (using ASI_PRIMARY_NO_FAULT[_LITTLE] or ASI_SECONDARY_NO_FAULT[_LITTLE]) with the TTE.e bit = 1 cause a *DAE_side_effect_page* trap.

Prefetches to noncacheable addresses result in nops.

The processor does speculative instruction memory accesses and follows branches that it predicts are taken. Instruction addresses mapped by the MMU can be accessed even though they are not actually executed by the program. Normally, locations with side effects or that generate timeouts or bus errors are not mapped as instruction addresses by the MMU, so these speculative accesses will not cause problems.

**IMPL. DEP. #118-V9:** The manner in which I/O locations are identified is implementation dependent.

**IMPL. DEP. #120-V9:** 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.

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-V9**: An implementation may choose to identify certain addresses and use an implementation-dependent memory model for references to them.

## 9.3 Memory 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 within 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 112, 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.

**Notes**

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.

### 9.3.1 Memory Addressing Types

The UltraSPARC Architecture supports the following types of memory addressing:

**Virtual Addresses (VA).** Virtual addresses are addresses produced by a virtual processor that maps all systemwide, program-visible memory. Virtual addresses are translated by the MMU in order to locate data in physical memory. Virtual addresses can be presented in nonprivileged mode and privileged mode, or in hyperprivileged mode using the `ASI_AS_IF_USER*` ASI variants.

1. Two exceptions to this are the special `ASI_TWIN_DW_NUCLEUS[_L]` and `ASI_TWINX_REAL[_L]` which provide hardware support for an atomic quad load to be used for TTE loads from TSBs.

**Real addresses (RA).** A real address is provided to privileged software to describe the underlying physical memory allocated to it. Translation storage buffers (TSBs) maintained by privileged software are used to translate privileged or nonprivileged mode virtual addresses into real addresses. MMU bypass addresses in privileged mode are also real addresses.

**Physical addresses (PA).** A physical address is one that appears on the system bus and is the same as the physical addresses in legacy architectures. Hyperprivileged software is responsible for managing the translation of real addresses into physical addresses.

Nonprivileged software only uses virtual addresses. Privileged software uses virtual and real addresses. Hyperprivileged software uses physical addresses, except when the explicit `ASI_AS_IF_USER*` or `ASI_*REAL*` ASI variants are used for load and store alternate instructions.

## 9.3.2 Memory Address Spaces

The UltraSPARC Architecture supports accessing memory using virtual, real, or physical addresses. Multiple virtual address spaces within the same real address space are distinguished by a *context identifier* (context ID). Multiple real address spaces within the same physical address space are distinguished by a *partition identifier* (partition ID).

Privileged software can create multiple virtual address spaces, using the primary and secondary context registers to associate a context ID with every virtual address. Privileged software manages the allocation of context IDs.

Hyperprivileged software can create multiple real address spaces, using the partition register to associate a partition ID with every real address. Hyperprivileged software manages the allocation of partition IDs.

**IMPL. DEP. #___** The number of bits in the partition register is implementation dependent.

The full representation of each type of address is as follows:

real_address = context_ID **::** virtual_address

physical_address = partition ID **::** real_address
or
physical_address = partition ID **::** context ID **::** virtual_address

## 9.3.3 Address Space Identifiers

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 specify the endianness of the reference
- To specify the address of an internal control register in the virtual processor, cache, or memory management hardware

Memory management hardware can associate an independent $2^{64}$-byte memory address space with each ASI. In practice, the three independent memory address spaces (contexts) created by the MMU are Primary, Secondary, and Nucleus.

**Programming Note** Independent address spaces, accessible through ASIs, make it possible for system software to easily access the address space of faulting software when processing exceptions or to implement access to a client program's memory space by a server program.

Alternate-space load, store, load-store and prefetch instructions specify an *explicit* ASI to use for their data access. The behavior of the access depends on the current privilege mode.

Non-alternate space load, store, load-store, and prefetch instructions use an *implicit* ASI value that is determined by current virtual processor state (the current privilege mode, trap level (TL), and the value of the PSTATE.cle). Instruction fetches use an implicit ASI that depends only on the current mode and trap level.

The architecturally specified ASIs are listed in Chapter 10, *Address Space Identifiers (ASIs)*. The operation of each ASI in nonprivileged, privileged and hyperprivileged modes is indicated in TABLE 10-1 on page 431.

Attempts by nonprivileged software (PSTATE.priv = 0 and HPSTATE.hpriv = 0) to access restricted ASIs (ASI bit 7 = 0) cause a *privileged_action* exception. Attempts by privileged software (PSTATE.priv = 1 and HPSTATE.hpriv = 0) to access ASIs $30_{16}$–$7F_{16}$ cause a *privileged_action* exception.

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 PSTATE.cle.

When TL = 1 or 2 (> 0 but ≤ *MAXPTL*), the implicit ASI in privileged mode is:

- for instruction fetches, ASI_NUCLEUS
- for loads and stores, ASI_NUCLEUS if PSTATE.cle = 0 or ASI_NUCLEUS_LITTLE if PSTATE.cle = 1 (impl. dep. #124-V9).

In hyperprivileged mode, all instruction fetches and loads and stores with implicit ASIs use a physical address, regardless of the value of TL.

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, restricted-to-privileged, or restricted-to-hyperprivileged:

- A nonrestricted ASI (ASI range $80_{16}$ – $FF_{16}$) is one that may be used independently of the privilege level (PSTATE.privand HPSTATE.hpriv) at which the virtual processor is running.
- A restricted-to-privileged ASI (ASI range $00_{16}$ – $2F_{16}$) requires that the virtual processor be in privileged or hyperprivileged mode for a legal access to occur.
- A restricted-to-hyperprivileged ASI (ASI range $30_{16}$ – $7F_{16}$) requires that the virtual processor be in hyperprivileged mode for a legal access to occur.

The relationship between virtual processor state and ASI restriction is shown in TABLE 9-1.

**TABLE 9-1** Allowed Accesses to ASIs

| ASI Value | Type | Result of ASI Access in NP Mode | Result of ASI Access in P Mode | Result of ASI Access in HP Mode |
|---|---|---|---|---|
| $00_{16}$ –- $2F_{16}$ | Restricted-to-privileged | *privileged_action* exception | Valid Access | Valid Access |
| $30_{16}$ – $7F_{16}$ | Restricted-to-hyperprivileged | *privileged_action* exception | *privileged_action* exception | Valid Access |
| $80_{16}$ – $FF_{16}$ | Nonrestricted | Valid Access | Valid Access | Valid Access |

Some 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 privileged software efficient, yet secure access to the memory space of nonprivileged software.

The normal address space is *primary address space*, which is accessed by the unrestricted ASI_PRIMARY[_LITTLE] ASIs. The *secondary address space*, which is accessed by the unrestricted ASI_SECONDARY[_LITTLE] ASIs, is provided to allow server software to access client software's address space.

ASI_PRIMARY_NOFAULT[_LITTLE] and ASI_SECONDARY_NOFAULT[_LITTLE] support *nonfaulting loads*. These ASIs 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.

## 9.4 SPARC V9 Memory Model

The SPARC V9 processor architecture specified the organization and structure of a central processing unit but did not specify a memory system architecture. This section summarizes the MMU support required by an UltraSPARC Architecture processor.

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.

### 9.4.1 SPARC V9 Program Execution Model

The SPARC V9 strand model of a virtual processor consists of three units: an Issue Unit, a Reorder Unit, and an Execute Unit, as shown in FIGURE 9-1.

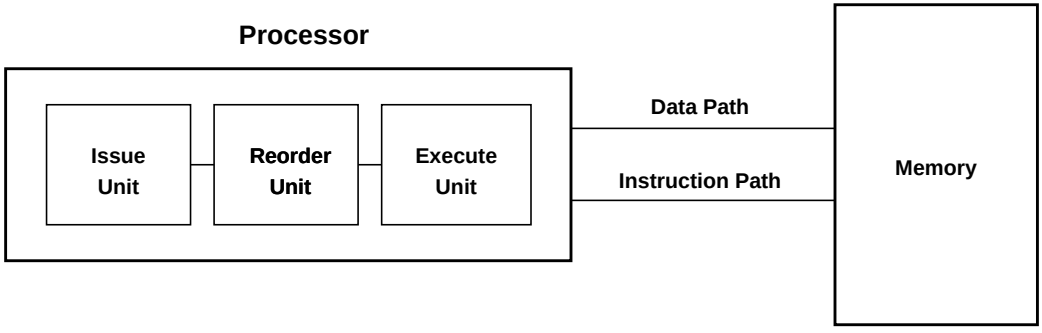



**FIGURE 9-1** Processor Model: Uniprocessor System

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 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 9-1, 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 that reads from or writes to a register cannot be performed until all earlier instructions that write to that register have been performed (read-after-write hazard; write-after-write hazard).

2. An instruction cannot be performed that writes to a register until all earlier instructions that read that register 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 uses (loads or stores) the value at a location cannot be performed until all earlier memory-reference instructions that set (store to) that location have been performed (read-after-write hazard, write-after-write hazard).

2. A memory-reference instruction that writes (stores to) a location cannot be performed until all previous instructions that read (load from) that location have been performed (write-after-read hazard).

Memory-barrier instruction (MEMBAR) and the TSO memory model also constrain the issue of memory-reference instructions. See *Memory Ordering and Synchronization* on page 423 and *The UltraSPARC Architecture Memory Model — TSO* on page 418 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.

## 9.4.2 Virtual Processor/Memory Interface Model

Each UltraSPARC Architecture virtual processor in a multiprocessor system is modeled as shown in FIGURE 9-2; that is, having two independent paths to memory: one for instructions and one for data.

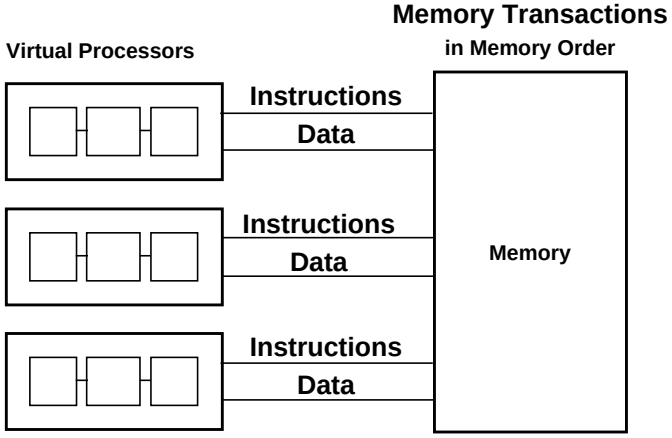



**FIGURE 9-2** Data Memory Paths: Multiprocessor System

Data caches are maintained by hardware so their contents always appear to be consistent (coherent). Instruction caches are *not* required to be kept consistent with data caches and therefore require explicit program (software) action to ensure consistency when a program modifies an executing instruction stream. See *Synchronizing Instruction and Data Memory* on page 426 for details. 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[1].

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 memory model. The ASI-address couples it generates are translated by a memory 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 implementation-specific MMU documentation for detailed information about how this is accomplished in each UltraSPARC Architecture implementation.

1. The model described here is only a model; implementations of UltraSPARC Architecture systems are unconstrained as long as their observable behaviors match those of the model.

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 orderings. 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.

## 9.5 The UltraSPARC Architecture Memory Model — TSO

The UltraSPARC Architecture is a *model* that specifies the behavior observable by software on UltraSPARC Architecture 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.

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 compatibility for SPARC V8 application software.

All UltraSPARC Architecture virtual processors implement TSO ordering. The PSO and RMO models from SPARC V9 are not described in this UltraSPARC Architecture specification. UltraSPARC Architecture 2007 processors do not implement the PSO memory model directly, but all software written to run under PSO will execute correctly on an UltraSPARC Architecture 2007 processor (using the TSO model).

Whether memory models represented by PSTATE.mm = $10_2$ or $11_2$ are supported in an UltraSPARC Architecture processor is implementation dependent (impl. dep. #113-V9-Ms10). If the $10_2$ model is supported, then when PSTATE.mm = $10_2$ the

implementation must correctly execute software that adheres to the RMO model described in *The SPARC Architecture Manual-Version 9*. If the $11_2$ model is supported, its definition is implementation dependent and will be described in implementation-specific documentation.

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 when run on hardware directly supporting that model, since the model exposes more scheduling opportunities, but use of that model 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. 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 TSO order. Although particular SPARC V9 implementations may support sequential consistency, portable software must not rely on the sequential consistency memory model.

## 9.5.1 Memory Model Selection

The active memory model is specified by the 2-bit value in PSTATE.mm,. The value $00_2$ represents the TSO memory model; increasing values of PSTATE.mm indicate increasingly weaker (less strongly ordered) memory models.

Writing a new value into PSTATE.mm causes subsequent memory reference instructions to be performed with the order constraints of the specified memory model.

**IMPL. DEP. #119-Ms10**: The effect of an attempt to write an unsupported memory model designation into PSTATE.mm is implementation dependent; however, it should never result in a value of PSTATE.mm value greater than the one that was written. In the case of an UltraSPARC Architecture implementation that only supports the TSO memory model, PSTATE.mm always reads as zero and attempts to write to it are ignored.

## 9.5.2 Programmer-Visible Properties of the UltraSPARC Architecture TSO Model

*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.

The rules for TSO, in addition to those required for self-consistency (see page 416), are:

- 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.
- Stores cannot bypass earlier loads.

**Programming Note** | Loads *can* bypass earlier stores to other addresses, which maintains processor self-consistency.

Atomic load-stores are treated as both a load and a store and can only be applied to cacheable address spaces.

Thus, TSO ensures the following behavior:

- Each load instruction behaves as if it were followed by a MEMBAR #`LoadLoad` and #LoadStore.
- Each store instruction behaves as if it were followed by a MEMBAR #`StoreStore`.
- Each atomic load-store behaves as if it were followed by a MEMBAR #`LoadLoad`, #`LoadStore`, and #`StoreStore`.

In addition to the above TSO rules, the following rules apply to UltraSPARC Architecture memory models:

- A MEMBAR #`StoreLoad` must be used to prevent a load from bypassing a prior store, if Strong Sequential Order (as defined in *The UltraSPARC Architecture Memory Model — TSO* on page 418) is desired.
- Accesses that have side effects are all strongly ordered with respect to each other.
- A MEMBAR #`Lookaside` is not needed between a store and a subsequent load to the same noncacheable address.
- Load (LDXA) and store (STXA) instructions that reference certain internal ASIs perform both an intra-virtual processor synchronization (i.e. an implicit MEMBAR #`Sync` operation before the load or store is executed) and an inter-virtual processor synchronization (that is, all active virtual processors are brought to a point where synchronization is possible, the load or store is executed, and all

virtual processors then resume instruction fetch and execution). The model-specific PRM should indicate which ASIs require intra-virtual processor synchronization, inter-virtual processor synchronization, or both.

## 9.5.3 TSO Ordering Rules

TABLE 9-2 summarizes the cases where a MEMBAR must be inserted between two memory operations on an UltraSPARC Architecture virtual processor running in TSO mode, to ensure that the operations appear to complete in a particular order. Memory operation *ordering* is not to be confused with processor consistency or deterministic operation; MEMBARs are required for deterministic operation of certain ASI register updates.

**Programming Note** | To ensure software portability across systems, the MEMBAR rules in this section should be followed (which may be stronger than the rules in SPARC V9).

TABLE 9-2 is to be read as follows: Reading from row to column, the first memory operation in program order in a row is followed by the memory operation found in the column. Symbols used as table entries:

- # — No intervening operation is required.
- M — an intervening MEMBAR `#StoreLoad` or MEMBAR `#Sync` or MEMBAR `#MemIssue` is required
- S — an intervening MEMBAR `#Sync` or MEMBAR `#MemIssue` is required
- nc — Noncacheable
- e — Side effect
- ne — No side effect

**TABLE 9-2** Summary of UltraSPARC Architecture Ordering Rules (TSO Memory Model)

| | To Memory Operation C (column): | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **From Memory Operation R (row):** | **load** | **store** | **atomic** | **bload** | **bstore** | **load_nc_e** | **store_nc_e** | **load_nc_ne** | **store_nc_ne** | **bload_nc** | **bstore_nc** |
| **load** | # | # | # | S | S | # | # | # | # | S | S |
| **store** | M[2] | # | # | M | S | M | # | M | # | M | S |
| **atomic** | # | # | # | M | S | # | # | # | # | M | S |
| **bload** | S | S | S | S | S | S | S | S | S | S | S |
| **bstore** | M | S | M | M | S | M | S | M | S | M | S |
| **load_nc_e** | # | # | # | S | S | #[1] | #[1] | #[1] | #[1] | S | S |
| **store_nc_e** | S | # | # | S | S | #[1] | #[1] | M[2] | #[1] | M | S |
| **load_nc_ne** | # | # | # | S | S | #[1] | #[1] | #[1] | #[1] | S | S |
| **store_nc_ne** | S | # | # | S | S | M[2] | #[1] | M[2] | #[1] | M | S |
| **bload_nc** | S | S | S | S | S | S | S | S | S | S | S |
| **bstore_nc** | S | S | S | S | S | M | S | M | S | M | S |

1. This table assumes that both noncacheable operations access the same device.

2. When the store and subsequent load access the *same* location, no intervening MEMBAR is required.

## 9.5.4 Hardware Primitives for Mutual Exclusion

In addition to providing memory-ordering primitives that allow programmers to construct mutual-exclusion mechanisms in software, the UltraSPARC Architecture 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 TSO memory model and may require barrier instructions to ensure proper data visibility.

Atomic load-store instructions can be used only in the cacheable domains (not in noncacheable I/O addresses). An attempt to use an atomic load-store instruction to access a noncacheable page results in a *DAE_nc_page* exception.

The atomic load-store alternate instructions can use a limited set of the ASIs. See the specific instruction descriptions for a list of the valid ASIs. An attempt to execute an atomic load-store alternate instruction with an invalid ASI results in a *DAE_invalid_asi* exception.

#### 9.5.4.1 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. For examples, see *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

#### 9.5.4.2 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.

#### 9.5.4.3 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.

### 9.5.5 Memory Ordering and Synchronization

The UltraSPARC Architecture provides some level of programmer control over memory ordering and synchronization through the MEMBAR and FLUSH instructions.

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 MEMBARs are needed only when a program requires that the effect of an operation becomes globally visible rather than simply being scheduled.[1] Because both forms are bit-encoded into the instruction, a single MEMBAR can function both as an ordering MEMBAR and as a sequencing MEMBAR.

The SPARC V9 instruction set architecture does not guarantee consistency between instruction and data spaces. A problem arises when instruction space is dynamically modified by a program writing to memory locations containing instructions (Self-Modifying Code). Examples are Lisp, debuggers, and dynamic linking. The FLUSH instruction synchronizes instruction and data memory after instruction space has been modified.

### 9.5.5.1 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 9-3. 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.

[1] Sequencing MEMBARs 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 *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*, for examples of the use of sequencing MEMBARs.

In TABLE 9-3 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 9-3** Ordering Relationships Selected by Mask

| Ordering Relation, Earlier <*m* Later | Assembly Language Constant Mnemonic | Effective Behavior in TSO model | Mask Value | nmask Bit # |
|---|---|---|---|---|
| Load <*m* Load | `#LoadLoad` | nop | $01_{16}$ | 0 |
| Store <*m* Load | `#StoreLoad` | `#StoreLoad` | $02_{16}$ | 1 |
| Load <*m* Store | `#LoadStore` | nop | $04_{16}$ | 2 |
| Store <*m* Store | `#StoreStore` | nop | $08_{16}$ | 3 |

**Implementation Note** An UltraSPARC Architecture 2007 implementation that only implements the TSO memory model may implement MEMBAR `#LoadLoad`, MEMBAR `#LoadStore`, and MEMBAR `#StoreStore` as nops and MEMBAR `#Storeload` as a MEMBAR `#Sync`.

### 9.5.5.2 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" in *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*.
- **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" in *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*.
- **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 9-4 shows the encoding of these functions in the MEMBAR instruction.

**TABLE 9-4** 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 |

**Implementation Note** | In UltraSPARC Architecture 2007 implementations, MEMBAR `#Lookaside` and MEMBAR `#MemIssue` are typically implemented as a MEMBAR `#Sync`.

For more details, see the MEMBAR instruction on page 277 of Chapter 7, *Instructions*.

### 9.5.5.3 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 (software) 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 modified.

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.

**IMPL. DEP. #122-V9:** 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 an UltraSPARC Architecture virtual processor:

- A FLUSH instruction causes a synchronization with the virtual processor, which flushes the instruction pipeline in the virtual processor on which the FLUSH instruction is executed.
- Coherency between instruction and data memories may or may not be maintained by hardware. If it is, an UltraSPARC Architecture implementation may ignore the address in the operands of a FLUSH instruction.

**Programming Note** UltraSPARC Architecture virtual processors are not required to maintain coherency between instruction and data caches in hardware. Therefore, portable software must do the following:

(1) must always assume that store instructions (except Block Store with Commit) do not coherently update instruction cache(s);

(2) must, in every FLUSH instruction, supply the address of the instruction or instructions that were modified.

For more details, see the FLUSH instruction on page 187 of Chapter 7, *Instructions*.

## 9.6 Nonfaulting Load

A nonfaulting load behaves like a normal load, with the following exceptions:

- A nonfaulting load from a location with side effects (TTE.e = 1) causes a *DAE_side_effect_page* exception.
- A nonfaulting load from a page marked for nonfault access only (TTE.nfo = 1) is allowed; other types of accesses to such a page cause a *DAE_nfo_page* exception.
- These loads are issued with `ASI_PRIMARY_NO_FAULT[_LITTLE]` or `ASI_SECONDARY_NO_FAULT[_LITTLE]`. A store with a `NO_FAULT` ASI causes a *DAE_invalid_asi* exception.

Typically, optimizers use nonfaulting loads to move loads across conditional control structures that guard their use. This technique potentially increases the distance between a load of data and the first use of that data, in order to hide latency. The technique allows more flexibility in instruction scheduling and improves performance in certain algorithms by removing address checking from the critical code path.

For example, when following a linked list, nonfaulting loads allow the null pointer to be accessed safely in a speculative, read-ahead fashion; the page at virtual address $0_{16}$ can safely be accessed with no penalty[1]. The TTE.nfo bit marks pages that are mapped for safe access by nonfaulting loads but that can still cause a trap by other, normal accesses.

Thus, programmers can trap on "wild" pointer references—many programmers count on an exception being generated when accessing address $0_{16}$ to debug software—while benefiting from the acceleration of nonfaulting access in debugged library routines.

## 9.7 Store Coalescing

Cacheable stores may be coalesced with adjacent cacheable stores within an 8 byte boundary offset in the store buffer to improve store bandwidth. Similarly non-side-effect-noncacheable stores may be coalesced with adjacent non-side-effect noncacheable stores within an 8-byte boundary offset in the store buffer.

In order to maintain strong ordering for I/O accesses, stores with side-effect attribute (e bit set) will not be combined with any other stores.

Stores that are separated by an intervening MEMBAR #Sync will not be coalesced.

[1] Other than the impact of occupying TLB entries.

CHAPTER 10

# Address Space Identifiers (ASIs)

This appendix describes address space identifiers (ASIs) in the following sections:


- **Address Space Identifiers and Address Spaces** on page 429.
- **ASI Values** on page 429.
- **ASI Assignments** on page 430.
- **Special Memory Access ASIs** on page 444.


## 10.1 Address Space Identifiers and Address Spaces

An UltraSPARC Architecture 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 unit uses a 64-bit virtual address and an 8-bit ASI to generate a memory, I/O, or internal register address. This physical address space can be accessed through virtual-to-physical address mapping or through the MMU bypass mode.

## 10.2 ASI Values

The range of address space identifiers (ASIs) is $00_{16}$-$FF_{16}$. That range is divided into restricted and unrestricted portions. ASIs in the range $80_{16}$–$FF_{16}$ are unrestricted; they may be accessed by software running in any privilege mode.

ASIs in the range $00_{16}$–$7F_{16}$ are restricted; they may only be accessed by software running in a mode with sufficient privilege for the particular ASI. ASIs in the range $00_{16}$–$2F_{16}$ may only be accessed by software running in privileged or hyperprivileged mode and ASIs in the range $30_{16}$–$7F_{16}$ may only be accessed by software running in hyperprivileged mode.

**SPARC V9 Compatibility Note** | In SPARC V9, the range of ASIs was evenly divided into restricted ($00_{16}$-$7F_{16}$) and unrestricted ($80_{16}$-$FF_{16}$) halves.

An attempt by nonprivileged software to access a restricted (privileged or hyperprivileged) ASI ($00_{16}$–$7F_{16}$) causes a *privileged_action* trap.

An attempt by privileged software to access a hyperprivileged ASI ($30_{16}$–$7F_{16}$) also causes a *privileged_action* trap.

An ASI can be categorized based on how it affects the MMU's treatment of the accompanying address, into one of three categories:

- A *Virtual-Translating* ASI (the most common type) causes the accompanying address to be treated as a virtual address (which is translated by the MMU into a physical address).
- A *Non-translating* ASI is not translated by the MMU; instead the address is passed through unchanged. Nontranslating ASIs are typically used for accessing internal registers.
- A *Real-Translating* ASI causes the accompanying address to be treated as a real address (which is translated by the MMU into a physical address). An access using a Real-Translating ASI can cause exception(s) only visible in hyperprivileged mode (such as a *PA_watchpoint* exception). Real-Translating ASIs are typically used by privileged or hyperprivileged software for directly accessing memory using real or physical (as opposed to virtual) addresses.

Implementation-dependent ASIs may or may not be translated by the MMU. See implementation-specific documentation for detailed information about implementation-dependent ASIs.

## 10.3 ASI Assignments

Every load or store address in an UltraSPARC Architecture 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, stores, and load-stores that do not use the load or store alternate instructions, the ASI is an implicit ASI generated by the virtual processor.

If a load alternate, store alternate, or load-store alternate instruction is used, the value of the ASI (an "explicit 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.

## 10.3.1 Supported ASIs

TABLE 10-1 lists architecturally-defined ASIs; some are in all UltraSPARC Architecture implementations and some are only present in some implementations.

An ASI marked with a closed bullet (●) is required to be implemented on all UltraSPARC Architecture 2007 processors.

An ASI marked with an open bullet (❍) is defined by the UltraSPARC Architecture 2007 but is not necessarily implemented in all UltraSPARC Architecture 2007 processors; its implemention is optional. Across all implementations on which it is implemented, it appears to software to behave identically.

Some ASIs may only be used with certain load or store instructions; see table footnotes for details.

The word “decoded” in the Virtual Address column of TABLE 10-1 indicates that the the supplied virtual address is decoded by the virtual processor.

The “TVP / non-T / TRP” column of the table indicates whether each ASI is a Virtual-Translating ASI(translates Virtual-to-Physical), non-Translating ASI, or-Translating (translates Real-to-Physical) ASI, respectively.

ASIs marked "Reserved" are set aside for use in future revisions to the architecture and are not to be used by implemenations. ASIs marked "implementation dependent" may be used for implementation-specific purposes.

Attempting to access an address space described as “Implementation dependent” in TABLE 10-1 produces implementation-dependent results.

**TABLE 10-1** UltraSPARC Architecture ASIs *(1 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $00_{16}$–$03_{16}$ | ❍ | — | —[2,12] | — | — | — | Implementation dependent[1] |
| $04_{16}$ | ● | ASI_NUCLEUS (ASI_N) | RW[2,4] | (decoded) | TVP | — | Implicit address space, nucleus context, TL > 0 |
| $05_{16}$–$0B_{16}$ | ❍ | — | —[2,12] | — | — | — | Implementation dependent[1] |

**TABLE 10-1** UltraSPARC Architecture ASIs *(2 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $0C_{16}$ | ● | ASI_NUCLEUS_LITTLE (ASI_NL) | RW[2,4] | (decoded) | TVP | — | Implicit address space, nucleus context, TL > 0, little-endian |
| $0D_{16}$–$0F_{16}$ | ❍ | — | —[2,12] | — | — | — | Implementation dependent[1] |
| $10_{16}$ | ● | ASI_AS_IF_USER_PRIMARY (ASI_AIUP) | RW[2,4,18] | (decoded) | TVP | — | Primary address space, as if user (nonprivileged) |
| $11_{16}$ | ● | ASI_AS_IF_USER_SECONDARY (ASI_AIUS) | RW[2,4,18] | (decoded) | TVP | — | Secondary address space, as if user (nonprivileged) |
| $12_{16}$–$13_{16}$ | ❍ | — | —[2,12] | — | — | — | Implementation dependent[1] |
| $14_{16}$ | ❍ | ASI_REAL | RW[2,4] | (decoded) | TRP | — | Real address |
| $15_{16}$ | ❍ | ASI_REAL_IO[D] | RW[2,5] | (decoded) | TRP | — | Real address, noncacheable, with side effect (deprecated) |
| $16_{16}$ | ❍ | ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP) | RW[2,8,14,18] | (decoded) | TVP | — | Primary address space, block load/store, as if user (nonprivileged) |
| $17_{16}$ | ❍ | ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS) | RW[2,8,14,18] | (decoded) | TVP | — | Secondary address space, block load/store, as if user (nonprivileged) |
| $18_{16}$ | ● | ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL) | RW[2,4,18] | (decoded) | TVP | — | Primary address space, as if user (nonprivileged), little-endian |
| $19_{16}$ | ● | ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL) | RW[2,4,18] | (decoded) | TVP | — | Secondary address space, as if user (nonprivileged), little-endian |
| $1A_{16}$–$1B_{16}$ | ❍ | — | —[2,12] | — | — | — | Implementation dependent[1] |
| $1C_{16}$ | ❍ | ASI_REAL_LITTLE (ASI_REAL_L) | RW [2,4] | (decoded) | TRP | — | Real address, little-endian |
| $1D_{16}$ | ❍ | ASI_REAL_IO_LITTLE[D] (ASI_REAL_IO_L[D]) | RW [2,5] | (decoded) | TRP | — | Real address, noncacheable, with side effect, little-endian (deprecated) |
| $1E_{16}$ | ❍ | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUPL) | RW[2,8,14,18] | (decoded) | TVP | — | Primary address space, block load/store, as if user (nonprivileged), little-endian |
| $1F_{16}$ | ❍ | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUS_L) | RW[2,8,14,18] | (decoded) | TVP | — | Secondary address space, block load/store, as if user (nonprivileged), little-endian |

**TABLE 10-1** UltraSPARC Architecture ASIs *(3 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $20_{16}$ | ❍ | ASI_SCRATCHPAD | RW[2,6] | (decoded; see below) | non-T | per strand | Privileged Scratchpad registers; implementation dependent[1] |
| | ❍ | | " | $0_{16}$ | " | " | Scratchpad Register 0[1] |
| | ❍ | | " | $8_{16}$ | " | " | Scratchpad Register 1[1] |
| | ❍ | | " | $10_{16}$ | " | " | Scratchpad Register 2[1] |
| | ❍ | | " | $18_{16}$ | " | " | Scratchpad Register 3[1] |
| | ❍ | | | $20_{16}$ | " | " | Scratchpad Register 4[1] |
| | ❍ | | " | $28_{16}$ | " | " | Scratchpad Register 5[1] |
| | ❍ | | " | $30_{16}$ | " | " | Scratchpad Register 6[1] |
| | ❍ | | " | $38_{16}$ | " | " | Scratchpad Register 7[1] |
| $21_{16}$ | ❍ | ASI_MMU_CONTEXTID | RW[2,6] | (decoded; see below) | non-T | per strand | MMU context registers |
| | ❍ | | " | $8_{16}$ | " | " | I/D MMU Primary Context ID register 0 |
| | ❍ | | " | $10_{16}$ | " | " | I/D MMU Secondary Context ID register_0 |
| | ❍ | | " | $108_{16}$ | " | " | I/D Primary Context ID register 1 |
| | ❍ | | " | $110_{16}$ | " | " | I/D MMU Secondary Context ID register 1 |
| $22_{16}$ | ❍ | ASI_TWINX_AS_IF_USER_PRIMARY (ASI_TWINX_AIUP) | R[2,7,11] | (decoded) | TVP | — | Primary address space, 128-bit atomic load twin extended word, as if user (nonprivileged) |
| $23_{16}$ | ❍ | ASI_TWINX_AS_IF_USER_SECONDARY (ASI_TWINX_AIUS) | R[2,7,11] | (decoded) | TVP | — | Secondary address space, 128-bit atomic load twin extended word, as if user (nonprivileged) |
| $24_{16}$ | ❍ | — | — | — | — | — | Implementation dependent[1] |

**TABLE 10-1** UltraSPARC Architecture ASIs *(4 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $25_{16}$ | ❍ | ASI_QUEUE | (see below) | (decoded; see below) | non-T | per strand | |
| | ❍ | | RW[2,6] | $3C0_{16}$ | " | " | CPU Mondo Queue Head Pointer |
| | ❍ | | RW[2,6,17] | $3C8_{16}$ | " | " | CPU Mondo Queue Tail Pointer |
| | ❍ | | RW[2,6] | $3D0_{16}$ | " | " | Device Mondo Queue Head Pointer |
| | ❍ | | RW[2,6,17] | $3D8_{16}$ | " | " | Device Mondo Queue Tail Pointer |
| | ❍ | | RW[2,6] | $3E0_{16}$ | " | " | Resumable Error Queue Head Pointer |
| | ❍ | | RW[2,6,17] | $3E8_{16}$ | " | " | Resumable Error Queue Tail Pointer |
| | ❍ | | RW[2,6] | $3F0_{16}$ | " | " | Nonresumable Error Queue Head Pointer |
| | ❍ | | RW[2,6,17] | $3F8_{16}$ | " | " | Nonresumable Error Queue Tail Pointer |
| $26_{16}$ | ❍ | ASI_TWINX_REAL (ASI_TWINX_R) ASI_QUAD_LDD_REAL[D†] | R[2,7,11] | (decoded) | TRP | — | 128-bit atomic twin extended-word load from real address |
| $27_{16}$ | ❍ | ASI_TWINX_NUCLEUS (ASI_TWINX_N) | R[2,7,11] | (decoded) | TVP | — | Nucleus context, 128-bit atomic load twin extended-word |
| $28_{16}$–$29_{16}$ | ❍ | — | —[2,12] | — | — | — | Implementation dependent[1] |
| $2A_{16}$ | ❍ | ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE (ASI_TWINXAIUPL) | R[2,7,11] | (decoded) | TVP | — | Primary address space, 128-bit atomic load twin extended-word, as if user (nonprivileged), little-endian |
| $2B_{16}$ | ❍ | ASI_TWINX_AS_IF_USER_SECONDARY_LITTLE (ASI_TWINX_AIUS_L) | R[2,7,11] | (decoded) | TVP | — | Secondary address space, 128-bit atomic load twin extended-word, as if user (nonprivileged), little-endian |
| $2C_{16}$ | ❍ | — | —[2] | — | — | — | Implementation dependent[1] |
| $2D_{16}$ | ❍ | — | —[2,12] | — | — | — | Implementation dependent[1] |
| $2E_{16}$ | ❍ | ASI_TWINX_REAL_LITTLE (ASI_TWINX_REAL_L) ASI_QUAD_LDD_REAL_LITTLE[D†] | R[2,7,11] | (decoded) | TRP | — | 128-bit atomic twin-extended-word load from real address, little-endian |

**TABLE 10-1** UltraSPARC Architecture ASIs *(5 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $2F_{16}$ | ❍ | ASI_TWINX_NUCLEUS_LITTLE (ASI_TWINX_NL) | R[2,7,11] | (decoded) | TVP | — | Nucleus context, 128-bit atomic load twin extended-word, little-endian |
| $30_{16}$ | ❍ | ASI_AS_IF_PRIV_PRIMARY (ASI_AIPP) | RW[3,4] | (decoded) | TVP | — | Primary address space, as if privileged |
| $31_{16}$ | ❍ | ASI_AS_IF_PRIV_SECONDARY (ASI_AIPS) | RW[3,4] | (decoded) | TVP | — | Secondary address space, as if privileged |
| $32_{16}$–$35_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $36_{16}$ | ❍ | ASI_AS_IF_PRIV_NUCLEUS (ASI_AIPN) | RW[3,4] | (decoded) | TVP | — | Implicit address space, nucleus context, as if privileged |
| $37_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $38_{16}$ | ❍ | ASI_AS_IF_PRIV_PRIMARY_LITTLE (ASI_AIPP_L) | RW[3,4] | (decoded) | TVP | — | Primary address space, as if privileged, little-endian |
| $39_{16}$ | ❍ | ASI_AS_IF_PRIV_SECONDARY_LITTLE (ASI_AIPS_L) | RW[3,4] | (decoded) | TVP | — | Secondary address space, as if privileged, little-endian |
| $3A_{16}$–$3C_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $3D_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $3E_{16}$ | ❍ | ASI_AS_IF_PRIV_NUCLEUS_LITTLE (ASI_AIPN_L) | RW[3,4] | (decoded) | TVP | — | Implicit address space, nucleus context, as if privileged, little-endian |
| $3F_{16}$–$40_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $41_{16}$ | ❍ | ASI_CMT_SHARED | (see below) | (decoded; see below) | non-T | shared | CMT control/status (shared) |
| | ❍ | | R[3,6,11] | $00_{16}$ | " | " | Virtual Processor (strand) Available Register |
| | ❍ | | R[3,6,11] | $10_{16}$ | " | " | Virtual Processor (strand) Enable Status Register |
| | ❍ | | RW[3,6] | $20_{16}$ | " | " | Virtual Processor (strand) Enable Register |
| | ❍ | | RW[1,3,6] | $30_{16}$ | " | " | XIR Steering Register Implementation dependent[1] (impl. dep. #1105) |
| | ❍ | | RW[3,6] | $50_{16}$ | " | " | Virtual Processor (strand) Running Register, general access |

**TABLE 10-1** UltraSPARC Architecture ASIs *(6 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| | ❍ | | R[3,6,11] | $58_{16}$ | " | " | Virtual Processor (strand) Running Status Register |
| | ❍ | | W[3,6,10] | $60_{16}$ | " | " | Virtual Processor (strand) Running Register, general access. Write '1' to set bit |
| | ❍ | | W[3,6,10] | $68_{16}$ | " | " | Virtual Processor (strand) Running Register, general access. Write '1' to clear bit |
| $42_{16}$–$44_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $45_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $46_{16}$–$48_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $49_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $4A_{16}$–$4B_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $4C_{16}$ | ❍ | Error Status and Enable Registers | | | | | Implementation dependent[1] |
| $4D_{16}$–$4E_{16}$ | ❍ | — | —[3,13] | — | | | Implementation dependent[1] |
| $4F_{16}$ | ❍ | ASI_HYP_SCRATCHPAD | RW[3,6] | (decoded; see below) | non-T | per strand | Hyperprivileged Scratchpad registers; implementation dependent[1] |
| | ❍ | | | $0_{16}$ | | | Hyperprivileged Scratchpad Register 0[1] |
| | ❍ | | | $8_{16}$ | | | Hyperprivileged Scratchpad Register 1[1] |
| | ❍ | | | $10_{16}$ | | | Hyperprivileged Scratchpad Register 2[1] |
| | ❍ | | | $18_{16}$ | | | Hyperprivileged Scratchpad Register 3[1] |
| | ❍ | | | $20_{16}$ | | | Hyperprivileged Scratchpad Register 4[1] |
| | ❍ | | | $28_{16}$ | | | Hyperprivileged Scratchpad Register 5[1] |
| | ❍ | | | $30_{16}$ | | | Hyperprivileged Scratchpad Register 6[1] |
| | ❍ | | | $38_{16}$ | | | Hyperprivileged Scratchpad Register 7[1] |

**TABLE 10-1** UltraSPARC Architecture ASIs *(7 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $50_{16}$ | ❍ | ASI_IMMU | — | (decoded; see below) | non-T | per strand | IMMU registers |
| | ❍ | | R[3,6,11] | $0_{16}$ | non-T | per strand | IMMU tag target register |
| | ❍ | | RW[3,6] | $18_{16}$ | non-T | per strand | Instruction fault status register |
| | ❍ | | RW[3,6] | $30_{16}$ | non-T | per strand | I TLB tag access register |
| $52_{16}$ | ❍ | ASI_MMU_REAL | RW[3,6] | (see below) | non-T | per strand | MMU registers |
| | ❍ | | | $108_{16}$ | " | " | MMU Real Range |
| | ❍ | | | $110_{16}$ | " | " | MMU Real Range |
| | ❍ | | | $118_{16}$ | " | " | MMU Real Range |
| | ❍ | | | $120_{16}$ | " | " | MMU Real Range |
| | ❍ | | | $208_{16}$ | " | " | MMU Physical Address Offset Registers |
| | ❍ | | | $210_{16}$ | " | " | MMU Physical Address Offset Registers |
| | ❍ | | | $218_{16}$ | " | " | MMU Physical Address Offset Registers |
| | ❍ | | | $220_{16}$ | " | " | MMU Physical Address Offset Registers |
| $53_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |

**TABLE 10-1** UltraSPARC Architecture ASIs *(8 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $54_{16}$ | ❍ | ASI_MMU | (see below) | (decoded; see below) | non-T | per strand | (more) MMU registers |
| | ❍ | | W[3,6,10] | $0_{16}$ | " | " | I TLB data in register |
| | ❍ | | RW[3,6] | $10_{16}$ | " | " | Context Zero TSB Configuration register 0 |
| | ❍ | | RW[3,6] | $18_{16}$ | " | " | Context Zero TSB Configuration register 1 |
| | ❍ | | RW[3,6] | $20_{16}$ | " | " | Context Zero TSB Configuration register 2 |
| | ❍ | | RW[3,6] | $28_{16}$ | " | " | Context Zero TSB Configuration register 3 |
| | ❍ | | RW[3,6] | $30_{16}$ | " | " | Context Nonzero TSB Configuration register 0 |
| | ❍ | | RW[3,6] | $38_{16}$ | " | " | Context Nonzero TSB Configuration register 1 |
| | ❍ | | RW[3,6] | $40_{16}$ | " | " | Context Nonzero TSB Configuration register 2 |
| | ❍ | | RW[3,6] | $48_{16}$ | " | " | Context Nonzero TSB Configuration register 3 |
| | ❍ | | RW[3,6] | $50_{16}$ | " | " | Instruction TSB Pointer register 0 |
| | ❍ | | RW[3,6] | $58_{16}$ | " | " | Instruction TSB Pointer register 1 |
| | ❍ | | RW[3,6] | $60_{16}$ | " | " | Instruction TSB Pointer register 2 |
| | ❍ | | RW[3,6] | $68_{16}$ | " | " | Instruction TSB Pointer register 3 |
| | ❍ | | RW[3,6] | $70_{16}$ | " | " | Data/Unified TSB Pointer register 0 |
| | ❍ | | RW[3,6] | $78_{16}$ | " | " | Data/Unified TSB Pointer register 1 |
| | ❍ | | RW[3,6] | $80_{16}$ | " | " | Data/Unified TSB Pointer register 2 |
| | ❍ | | RW[3,6] | $88_{16}$ | " | " | Data/Unified TSB Pointer register 3 |
| | ❍ | | RW[3,6] | $90_{16}$ | " | " | Tablewalk Pending Control register |
| | ❍ | | RW[3,6] | $98_{16}$ | " | " | Tablewalk Pending Status register |

**TABLE 10-1** UltraSPARC Architecture ASIs *(9 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $55_{16}$ | ❍ | ASI_ITLB_DATA_ACCESS_REG | RW[3,6] | $0_{16}$–$3F8_{16}$, $800_{16}$–$7FFF8_{16}$ | non-T | per strand | IMMU TLB data access register |
| $56_{16}$ | ❍ | ASI_ITLB_TAG_READ_REG | R[3,6,11] | $0_{16}$–$FFFF8_{16}$ | non-T | per strand | IMMU TLB tag read register |
| $57_{16}$ | ❍ | ASI_IMMU_DEMAP | W[3,6,10] | $0_{16}$ | non-T | per strand | IMMU TLB demap |
| $58_{16}$ | ❍ | ASI_DMMU /ASI_UMMU | (see below) | (decoded; see below) | non-T | — | Data or Unified MMU registers |
| | ❍ | | R[3,6,11] | $0_{16}$ | " | per strand | D/U TSB tag target register |
| | ❍ | | RW[3,6] | $18_{16}$ | " | per strand | Data error status register |
| | ❍ | | R[3,6,11] | $20_{16}$ | " | /core | Data error address register (DSFAR) |
| | ❍ | | RW[3,6] | $30_{16}$ | " | /core | D/U TLB tag access register |
| | ❍ | | RW[3,6] | $38_{16}$ | " | per strand | VA instruction, and PA/VA data watchpoint register |
| | ❍ | | RW[3,6] | $40_{16}$ | " | per strand | I/D/U MMU hardware tablewalk configuration register |
| | ❍ | | RW[3,6] | $80_{16}$ | " | per strand | I/D/U MMU partition ID register |
| $59_{16}$-$5B_{16}$ | ❍ | — | —[3,13] | — | — | — | Reserved |
| $5C_{16}$ | ❍ | ASI_DTLB_DATA_IN_REG | W[3,6,10] | $0_{16}$ | non-T | per strand | D/U TLB data in register |
| $5D_{16}$ | ❍ | ASI_DTLB_DATA_ACCESS_REG | RW[3,6] | $0_{16}$–$3F8_{16}$, $800_{16}$–$7FFF8_{16}$ | non-T | per strand | D/U TLB data access register |
| $5E_{16}$ | ❍ | ASI_DTLB_TAG_READ_REG | R[3,6,11] | $0_{16}$–$FFFF8_{16}$ | non-T | per strand | D/U TLB tag read register |
| $5F_{16}$ | ❍ | ASI_DMMU_DEMAP | W[3,6,10] | $0_{16}$ | non-T | per strand | D/U TLB demap |
| $60_{16}$–$62_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $61_{16}$–$62_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |

**TABLE 10-1** UltraSPARC Architecture ASIs *(10 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $63_{16}$ | ❍ | ASI_CMT_PER_STRAND, ASI_CMT_PER_CORE[†] | (see below) | (decoded; see below) | non-T | per strand | CMT control/status (per strand) |
| | ❍ | | RW[3,6] | $00_{16}$ | " | " | Virtual Processor (strand) Interrupt ID |
| | ❍ | | R[3,6,11] | $10_{16}$ | " | " | Virtual Processor (strand) ID |
| $64_{16}$–$67_{16}$ | ❍ | — | —[3,13] | — | — | — | Implementation dependent[1] |
| $68_{16}$–$7F_{16}$ | ● | — | —[3,13] | — | — | — | *Reserved* |
| $80_{16}$ | ● | ASI_PRIMARY (ASI_P) | RW[4] | (decoded) | TVP | — | Implicit primary address space |
| $81_{16}$ | ● | ASI_SECONDARY (ASI_S) | RW[4] | (decoded) | TVP | — | Secondary address space |
| $82_{16}$ | ● | ASI_PRIMARY_NO_FAULT (ASI_PNF) | R[9,11] | (decoded) | TVP | — | Primary address space, no fault |
| $83_{16}$ | ● | ASI_SECONDARY_NO_FAULT (ASI_SNF) | R[9,11] | (decoded) | TVP | — | Secondary address space, no fault |
| $84_{16}$–$87_{16}$ | ● | — | —[16] | — | — | — | *Reserved* |
| $88_{16}$ | ● | ASI_PRIMARY_LITTLE (ASI_PL) | RW[4] | (decoded) | TVP | — | Implicit primary address space, little-endian |
| $89_{16}$ | ● | ASI_SECONDARY_LITTLE (ASI_SL) | RW[4] | (decoded) | TVP | — | Secondary address space, little-endian |
| $8A_{16}$ | ● | ASI_PRIMARY_NO_FAULT_LITTLE (ASI_PNFL) | R[9,11] | (decoded) | TVP | — | Primary address space, no fault, little-endian |
| $8B_{16}$ | ● | ASI_SECONDARY_NO_FAULT_LITTLE (ASI_SNFL) | R[9,11] | (decoded) | TVP | — | Seondary address space, no fault, little-endian |
| $8C_{16}$–$BF_{16}$ | ● | — | —[16] | — | — | — | *Reserved* |
| $C0_{16}$ | ❍ | ASI_PST8_PRIMARY (ASI_PST8_P) | W[8,10,14] | (decoded) | TVP | — | Primary address space, 8×8-bit partial store |
| $C1_{16}$ | ❍ | ASI_PST8_SECONDARY (ASI_PST8_S) | W[8,10,14] | (decoded) | TVP | — | Secondary address space, 8x8-bit partial store |
| $C2_{16}$ | ❍ | ASI_PST16_PRIMARY (ASI_PST16_P) | W[8,10,14] | (decoded) | TVP | — | Primary address space, 4×16-bit partial store |
| $C3_{16}$ | ❍ | ASI_PST16_SECONDARY (ASI_PST16_S) | W[8,10,14] | (decoded) | TVP | — | Secondary address space, 4×16-bit partial store |
| $C4_{16}$ | ❍ | ASI_PST32_PRIMARY (ASI_PST32_P) | W[8,10,14] | (decoded) | TVP | — | Primary address space, 2x32-bit partial store |
| $C5_{16}$ | ❍ | ASI_PST32_SECONDARY (ASI_PST32_S) | W[8,10,14] | (decoded) | TVP | — | Secondary address space, 2×32-bit partial store |

**TABLE 10-1** UltraSPARC Architecture ASIs *(11 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $C6_{16}$–$C7_{16}$ | ● | — | —[15] | — | — | — | Implementation dependent[1] |
| $C8_{16}$ | ❍ | ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL) | W[8,10,14] | (decoded) | TVP | — | Primary address space, 8x8-bit partial store, little-endian |
| $C9_{16}$ | ❍ | ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL) | W[8,10,14] | (decoded) | TVP | — | Secondary address space, 8×8-bit partial store, little-endian |
| $CA_{16}$ | ❍ | ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL) | W[8,10,14] | (decoded) | TVP | — | Primary address space, 4x16-bit partial store, little-endian |
| $CB_{16}$ | ❍ | ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL) | W[8,10,14] | (decoded) | TVP | — | Secondary address space, 4×16-bit partial store, little-endian |
| $CC_{16}$ | ❍ | ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL) | W[8,10,14] | (decoded) | TVP | — | Primary address space, 2×32-bit partial store, little-endian |
| $CD_{16}$ | ❍ | ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL) | W[8,10,14] | (decoded) | TVP | — | Second address space, 2×32-bit partial store, little-endian |
| $CE_{16}$–$CF_{16}$ | ● | — | —[15] | — | — | — | Implementation dependent[1] |
| $D0_{16}$ | ❍ | ASI_FL8_PRIMARY (ASI_FL8_P) | RW[8,14] | (decoded) | TVP | — | Primary address space, one 8-bit floating-point load/ store |
| $D1_{16}$ | ❍ | ASI_FL8_SECONDARY (ASI_FL8_S) | RW[8,14] | (decoded) | TVP | — | Second address space, one 8-bit floating-point load/store |
| $D2_{16}$ | ❍ | ASI_FL16_PRIMARY (ASI_FL16_P) | RW[8,14] | (decoded) | TVP | — | Primary address space, one 16-bit floating-point load/ store |
| $D3_{16}$ | ❍ | ASI_FL16_SECONDARY (ASI_FL16_S) | RW[8,14] | (decoded) | TVP | — | Second address space, one 16-bit floating-point load/ store |
| $D4_{16}$–$D7_{16}$ | ● | — | —[15] | — | — | — | Implementation dependent[1] |
| $D8_{16}$ | ❍ | ASI_FL8_PRIMARY_LITTLE (ASI_FL8_PL) | RW[8,14] | (decoded) | TVP | — | Primary address space, one 8-bit floating point load/ store, little-endian |
| $D9_{16}$ | ❍ | ASI_FL8_SECONDARY_LITTLE (ASI_FL8_SL) | RW[8,14] | (decoded) | TVP | — | Second address space, one 8-bit floating point load/store, little-endian |
| $DA_{16}$ | ❍ | ASI_FL16_PRIMARY_LITTLE (ASI_FL16_PL) | RW[8,14] | (decoded) | TVP | — | Primary address space, one 16-bit floating-point load/ store, little-endian |

**TABLE 10-1** UltraSPARC Architecture ASIs *(12 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $DB_{16}$ | ❍ | ASI_FL16_SECONDARY_LITTLE (ASI_FL16_SL) | RW[8,14] | (decoded) | TVP | — | Second address space, one 16-bit floating point load/ store, little-endian |
| $DC_{16}$ –$DF_{16}$ | ● | — | —[15] | — | — | — | Implementation dependent[1] |
| $E0_{16}$ | ❍ | ASI_BLOCK_COMMIT_PRIMARY (ASI_BLK_COMMIT_P) | W[8,11,14] | (decoded) | TVP | — | Primary address space, 8x8-byte block store commit operation |
| $E1_{16}$ | ❍ | ASI_BLOCK_COMMIT_SECONDARY (ASI_BLK_COMMIT_S) | W[8,11,14] | (decoded) | TVP | — | Secondary address space, 8x8-byte block store commit operation |
| $E2_{16}$ | ❍ | ASI_TWINX_PRIMARY (ASI_TWINX_P) | R[19] | (decoded) | TVP | — | Primary address space, 128-bit atomic load twin extended word |
| $E3_{16}$ | ❍ | ASI_TWINX_SECONDARY (ASI_TWINX_S) | R[19] | (decoded) | TVP | — | Secondary address space, 128-bit atomic load twin extended-word |
| $E4_{16}$– $E9_{16}$ | ● | — | —[15] | — | — | — | Implementation dependent[1] |
| $EA_{16}$ | ❍ | ASI_TWINX_PRIMARY_LITTLE (ASI_TWINX_PL) | R[19] | (decoded) | TVP | — | Primary address space, 128-bit atomic load twin extended word, little endian |
| $EB_{16}$ | ❍ | ASI_TWINX_SECONDARY_LITTLE (ASI_TWINX_SL) | R[19] | (decoded) | TVP | — | Secondary address space, 128-bit atomic load twin extended word, little endian |
| $EC_{16}$– $EF_{16}$ | ❍ | — | —[15] | — | — | — | Implementation dependent[1] |
| $F0_{16}$ | ❍ | ASI_BLOCK_PRIMARY (ASI_BLK_P) | RW[8,14] | (decoded) | TVP | — | Primary address space, 8x8-byte block load/store |
| $F1_{16}$ | ❍ | ASI_BLOCK_SECONDARY (ASI_BLK_S) | RW[8,14] | (decoded) | TVP | — | Secondary address space, 8x8- byte block load/store |
| $F2_{16}$– $F5_{16}$ | ● | — | —[15] | — | — | — | Implementation dependent[1] |
| $F6_{16}$– $F7_{16}$ | ● | — | — | — | — | — | Implementation dependent[1] |
| $F8_{16}$ | ❍ | ASI_BLOCK_PRIMARY_LITTLE (ASI_BLK_PL) | RW[8,14] | (decoded) | TVP | — | Primary address space, 8x8-byte block load/store, little endian |

**TABLE 10-1** UltraSPARC Architecture ASIs *(13 of 13)*

| ASI Value | req'd(●) opt'l (❍) | ASI Name (and Abbreviation) | Access Type(s) | Virtual Address (VA) | TVP/ non-T/ TRP | Shared /per strand | Description |
|---|---|---|---|---|---|---|---|
| $F9_{16}$ | ❍ | `ASI_BLOCK_SECONDARY_LITTLE` (`ASI_BLK_SL`) | RW[8,14] | (decoded) | TVP | — | Secondary address space, 8x8- byte block load/store, little endian |
| $FA_{16}$–$FD_{16}$ | ● | — | —[15] | — | — | — | Implementation dependent[1] |
| $FE_{16}$–$FF_{16}$ | ● | — | —[15] | — | — | — | 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 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in nonprivileged mode causes a *privileged_action* exception.

3 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in nonprivileged mode or privileged mode causes a *privileged_action* exception.

4 May be used with all load alternate, store alternate, atomic alternate and prefetch alternate instructions (CASA, CASXA, LDSTUBA, LDTWA, LDDFA, LDFA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDXA, PREFETCHA, STBA, STTWA, STDFA, STFA, STHA, STWA, STXA, SWAPA).

5 May be used with all of the following load alternate and store alternate instructions: LDTWA, LDDFA, LDFA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDXA, STBA, STTWA, STDFA, STFA, STHA, STWA, STXA. Use with an atomic alternate or prefetch alternate instruction (CASA, CASXA, LDSTUBA, SWAPA or PREFETCHA) causes a *DAE_invalid_asi* exception.

6 May only be used in a LDXA or STXA instruction for RW ASIs, LDXA for read-only ASIs and STXA for write-only ASIs. Use of LDXA for write-only ASIs, STXA for read-only ASIs, or any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a *DAE_invalid_asi* exception.

7 May only be used in an LDTXA instruction. Use of this ASI in any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a *DAE_invalid_asi* exception.

8 May only be used in a LDDFA or STDFA instruction for RW ASIs, LDDFA for read-only ASIs and STDFA for write-only ASIs. Use of LDDFA for write-only ASIs, STDFA for read-only ASIs, or any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a *DAE_invalid_asi* exception.

9 May be used with all of the following load and prefetch alternate instructions: LDTWA, LDDFA, LDFA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDXA, PREFETCHA. Use with an atomic alternate or store alternate instruction causes a *DAE_invalid_asi* exception.

10 Write(store)-only ASI; an attempted load alternate, atomic alternate, or prefetch alternate instruction to this ASI causes a *DAE_invalid_asi* exception.

11 Read(load)-only ASI; an attempted store alternate or atomic alternate instruction to this ASI causes a *DAE_invalid_asi* exception.

12 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in privileged mode or hyperprivileged mode causes a *DAE_invalid_asi* exception.

13 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in hyperprivileged mode causes a *DAE_invalid_asi* exception if this ASI is not implemented by the specific implementation.

14 An attempted access to this ASI may cause an exception (see *Special Memory Access ASIs* on page 444 for details).

15 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in any mode causes a *DAE_invalid_asi* exception if this ASI is not implemented by the model dependent implementation.

16 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to a reserved ASI in any mode causes a *DAE_invalid_asi* exception.

17 The Queue Tail Registers (ASI $25_{16}$) are read-only by privileged software and read-write by hyperprivileged software. An attempted write to the Queue Tail Registers by privileged software causes a *DAE_invalid_asi* exception

18 An access to a privileged page (TTE.p = 1) using an ASI_*AS_IF_USER* ASI causes a *DAE_privilege_violation* exception.

19 May only be used in an LDTXA (load twin-extended-word) instruction (which shares an opcode with LDTWA). Use of this ASI in any other load instruction causes a *DAE_invalid_asi* exception.

## 10.4 Special Memory Access ASIs

This section describes special memory access ASIs that are not described in other sections.

### 10.4.1 ASIs $10_{16}$, $11_{16}$, $16_{16}$, $17_{16}$ and $18_{16}$ (ASI_*AS_IF_USER_*)

These ASI are intended to be used in accesses from privileged and hyperprivileged mode, but are processed as if they were issued from nonprivileged mode. Therefore, they are subject to privilege-related exceptions. They are distinguished from each other by the context from which the access is made, as described in TABLE 10-2.

When one of these ASIs is specified in a load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- In any other privilege mode:
  - If U/DMMU TTE.p = 1, a *DAE_privilege_violation* exception occurs
  - Otherwise, the access occurs and its endianness is determined by the current privileged mode and the U/DMMU TTE.ie bit. In hyperprivileged mode, the access is always made in big-endian byte order. In privileged mode, if U/DMMU TTE.ie = 0, the access is big-endian; otherwise, it is little-endian.

**TABLE 10-2** Privileged ASI_*AS_IF_USER_* ASIs

| ASI | Names | Addressing (Context) | Endianness of Access |
|---|---|---|---|
| $10_{16}$ | ASI_AS_IF_USER_PRIMARY (ASI_AIUP) | Virtual (Primary) | In nonprivileged or privileged mode: Big-endian when U/DMMU TTE.ie = 0; little-endian when U/DMMU TTE.ie = 1<br><br>In nyperprivileged mode: always big-endian. |
| $11_{16}$ | ASI_AS_IF_USER_SECONDARY (ASI_AIUS) | Virtual (Secondary) | |
| $16_{16}$ | ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP) | Virtual (Primary) | |
| $17_{16}$ | ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS) | Virtual (Secondary) | |

### 10.4.2 ASIs $18_{16}$, $19_{16}$, $1E_{16}$, and $1F_{16}$ (ASI_*AS_IF_USER_*_LITTLE)

These ASIs are little-endian versions of ASIs $10_{16}$, $11_{16}$, $16_{16}$, and $17_{16}$ (ASI_AS_IF_USER_*), described in section 10.4.1. Each operates identically to the corresponding non-little-endian ASI, except that if an access occurs its endianness is the opposite of that for the corresponding non-little-endian ASI.

These ASI are intended to be used in accesses from privileged and hyperprivileged mode, but are processed as if they were issued from nonprivileged mode. Therefore, they are subject to privilege-related exceptions. They are distinguished from each other by the context from which the access is made, as described in TABLE 10-3.

When one of these ASIs is specified in a load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- In any other privilege mode:
  - If U/DMMU TTE.p = 1, a *DAE_privilege_violation* exception occurs
  - Otherwise, the access occurs and its endianness is determined by the U/DMMU TTE.ie bit. If U/DMMU TTE.ie = 0, the access is little-endian; otherwise, it is big-endian.

**TABLE 10-3** Privileged ASI_*AS_IF_USER_*_LITTLE ASIs

| ASI | Names | Addressing (Context) | Endianness of Access |
|---|---|---|---|
| $18_{16}$ | ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL) | Virtual (Primary) | Little-endian when U/DMMU TTE.ie = 0; big-endian when U/DMMU TTE.ie = 1 |
| $19_{16}$ | ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL) | Virtual (Secondary) | |
| $1E_{16}$ | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUP) | Virtual (Primary) | |
| $1F_{16}$ | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL) | Virtual (Secondary) | |

## 10.4.3 ASI $14_{16}$ (ASI_REAL)

When ASI_REAL is specified in any load alternate, store alternate or prefetch alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- In any other privilege mode:
  - VA is passed through to RA , but the number of bits passed through is implementation dependent (impl. dep. #224-U3)
  - During the address translation, context values are disregarded.
  - The endianness of the access is dertermined by the U/DMMU TTE.ie bit; if U/DMMU TTE.ie = 0, the access is big-endian, otherwise it is little-endian.

Even if data address translation is disabled, an access with this ASI is still a cacheable access.

### 10.4.4 ASI $15_{16}$ (ASI_REAL_IO)

Accesses with ASI_REAL_IO bypass the external cache and behave as if the side effect bit (TTE.e bit) is set. When this ASI is specified in any load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- If used with a CASA, CASXA, LDSTUBA, SWAPA, or PREFETCHA instruction, a *DAE_invalid_asi* exception occurs
- Used with any other load alternate or store alternate instuction, in privileged mode or hyperprivileged mode:
  - VA is passed through to RA , but the number of bits passed through is implementation dependent (impl. dep. #224-U3)
  - During the address translation, context values are disregarded.
  - The endianness of the access is dertermined by the U/DMMU TTE.ie bit; if U/DMMU TTE.ie = 0, the access is big-endian, otherwise it is little-endian.

### 10.4.5 ASI $1C_{16}$ (ASI_REAL_LITTLE)

ASI_REAL_LITTLE is a little-endian version of ASI $14_{16}$ (ASI_REAL). It operates identically to ASI_REAL, except if an access occurs, its endianness the opposite of that for ASI_REAL.

### 10.4.6 ASI $1D_{16}$ (ASI_REAL_IO_LITTLE)

ASI_REAL_IO_LITTLE is a little-endian version of ASI $15_{16}$ (ASI_REAL_IO). It operates identically to ASI_REAL_IO, except if an access occurs, its endianness the opposite of that for ASI_REAL_IO.

### 10.4.7 ASIs $22_{16}$, $23_{16}$, $27_{16}$, $2A_{16}$, $2B_{16}$, $2F_{16}$ (Privileged Load Integer Twin Extended Word)

ASIs $22_{16}$, $23_{16}$, $27_{16}$, $2A_{16}$, $2B_{16}$ and $2F_{16}$ exist for use with the (nonportable) LDTXA instruction as atomic Load Integer Twin Extended Word operations (see *Load Integer Twin Extended Word from Alternate Space* on page 272). These ASIs are distinguished by the context from which the access is made and the endianness of the access, as described in TABLE 10-4.

**TABLE 10-4** Privileged Load Integer Twin Extended Word / Block Store Init ASIs

| ASI | Names | Addressing (Context) | Endianness of Access |
|---|---|---|---|
| $22_{16}$ | ASI_TWINX_AS_IF_USER_PRIMARY (ASI_TWINX_AIUP) | Virtual (Primary) | Big-endian when U/ DMMU TTE.ie = 0; little-endian when U/ DMMU TTE.ie = 1 |
| $23_{16}$ | ASI_TWINX_AS_IF_USER_SECONDARY (ASI_TWINX_AIUS) | Virtual (Secondary) | |
| $27_{16}$ | ASI_TWINX_NUCLEUS (ASI_TWINX_N) | Virtual‡ (Nucleus) | |
| $2A_{16}$ | ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE (ASI_TWINX_AIUP_L) | Virtual (Primary) | Little-endian when U/ DMMU TTE.ie = 0; big-endian when U/ DMMU TTE.ie = 1 |
| $2B_{16}$ | ASI_TWINX_AS_IF_USER_SECONDARY_ LITTLE (ASI_TWINX_AIUS_L) | Virtual (Secondary) | |
| $2F_{16}$ | ASI_TWINX_NUCLEUS_LITTLE (ASI_TWINX_NL) | Virtual‡ (Nucleus) | |

‡ In hyperprivileged mode, this ASI uses Physical addressing

When these ASIs are used with LDTXA, 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, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

**Compatibility Note** These ASIs replaced ASIs $24_{16}$ and $2C_{16}$ used in earlier UltraSPARC implementations; see the detailed Compatibility Note on page 455 for details.

## 10.4.8 ASIs $26_{16}$ and $2E_{16}$ (Privileged Load Integer Twin Extended Word, Real Addressing)

ASIs $26_{16}$ and $2E_{16}$ exist for use with the LDTXA instruction as atomic Load Integer Twin Extended Word operations using Real addressing (see *Load Integer Twin Extended Word from Alternate Space* on page 272). These two ASIs are distinguished by the endianness of the access, as described in TABLE 10-5.

**TABLE 10-5** Load Integer Twin Extended Word (Real) ASIs

| ASI | Name | Addressing (Context) | Endianness of Access |
|---|---|---|---|
| $26_{16}$ | ASI_TWINX_REAL (ASI_TWINX_R) | Real (—) | Big-endian when U/DMMU TTE.ie = 0; little-endian when U/DMMU TTE.ie = 1 |
| $2E_{16}$ | ASI_TWINX_REAL_LITTLE (ASI_TWINX_REAL_L) | Real (—) | Little-endian when U/DMMU TTE.ie = 0; big-endian when U/DMMU TTE.ie = 1 |

When these ASIs are used with LDTXA, 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, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

**Compatibility Note** These ASIs replaced ASIs $34_{16}$ and $3C_{16}$ used in earlier UltraSPARC implementations; see the Compatibility Note on page 455 for details.

## 10.4.9 ASIs $30_{16}$, $31_{16}$, $36_{16}$, $38_{16}$, $39_{16}$, $3E_{16}$ (ASI_AS_IF_PRIV_*)

These ASI are intended to be used in accesses from hyperprivileged mode, but are processed as if they were issued from privileged mode These ASIs are distinguished by the context from which the access is made and the endianness of the access, as described in TABLE 10-6.

When one of these ASIs is specified in a load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged or privileged mode, a *privileged_action* exception occurs
- In hyperprivileged mode:
  - The endianness of the access is determined by the U/DMMU TTE.ie bit; if U/DMMU TTE.ie = 0, the access is big-endian; otherwise, it is little-endian.

**TABLE 10-6** Hyperprivileged AS_IF_PRIV_* ASIs

| ASI | Names | Addressing (Context) | Endianness of Access |
|---|---|---|---|
| $30_{16}$ | ASI_AS_IF_PRIV_PRIMARY (ASI_AIPP) | Virtual (Primary) | Big-endian when U/DMMU TTE.ie = 0; little-endian when U/DMMU TTE.ie = 1 |
| $31_{16}$ | ASI_AS_IF_PRIV_SECONDARY (ASI_AIPS) | Virtual (Secondary) | |
| $36_{16}$ | ASI_AS_IF_PRIV_NUCLEUS (ASI_AIPN) | Virtual (Nucleus) | |
| $38_{16}$ | ASI_AS_IF_PRIV_PRIMARY_LITTLE (ASI_AIPP_L) | Virtual (Primary) | Little-endian when U/DMMU TTE.ie = 0; big-endian when U/DMMU TTE.ie = 1 |
| $39_{16}$ | ASI_AS_IF_PRIV_SECONDARY_LITTLE (ASI_AIPS_L) | Virtual (Secondary) | |
| $3E_{16}$ | ASI_AS_IF_PRIV_NUCLEUS_LITTLE (ASI_AIPN_L) | Virtual (Nucleus) | |

## 10.4.10 ASIs $E2_{16}$, $E3_{16}$, $EA_{16}$, $EB_{16}$ (Nonprivileged Load Integer Twin Extended Word)

ASIs $E2_{16}$, $E3_{16}$, $EA_{16}$, and $EB_{16}$ exist for use with the (nonportable) LDTXA instruction as atomic Load Integer Twin Extended Word operations (see *Load Integer Twin Extended Word from Alternate Space* on page 272). These ASIs are distinguished by the address space accessed (Primary or Secondary) and the endianness of the access, as described in TABLE 10-7.

**TABLE 10-7** Load Integer Twin Extended Word ASIs

| ASI | Names | Addressing (Context) | Endianness of Access |
|---|---|---|---|
| $E2_{16}$ | ASI_TWINX_PRIMARY (ASI_TWINX_P) | Virtual (Primary) | Big-endian when U/DMMU TTE.ie = 0, little-endian when U/DMMU TTE.ie = 1 |
| $E3_{16}$ | ASI_TWINX_SECONDARY (ASI_TWINX_S) | Virtual (Secondary) | |
| $EA_{16}$ | ASI_TWINX_PRIMARY_LITTLE (ASI_TWINX_PL) | Virtual (Primary) | Little-endian when U/DMMU TTE.ie = 0, big-endian when U/DMMU TTE.ie = 1 |
| $EB_{16}$ | ASI_TWINX_SECONDARY_LITTLE (ASI_TWINX_SL) | Virtual (Secondary) | |

When these ASIs are used with LDTXA, 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, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

## 10.4.11 Block Load and Store ASIs

ASIs $16_{16}$, $17_{16}$, $1E_{16}$, $1F_{16}$, $E0_{16}$, $E1_{16}$, $F0_{16}$, $F1_{16}$, $F8_{16}$, and $F9_{16}$ exist for use with LDDFA and STDFA instructions as Block Load (LDBLOCKF) and Block Store (STBLOCKF) operations (see *Block Load* on page 249 and *Block Store* on page 338).

When these ASIs are used with the LDDFA (STDFA) opcode 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 (see page 338). Neither ASI $E0_{16}$ nor $E1_{16}$ should be used with the LDDFA opcode; however, if either *is* used, the resulting behavior is specified in the LDDFA instruction description on page 258.

If a Block Load or Block Store ASI is used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

### 10.4.12 Partial Store ASIs

ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–$CD_{16}$ exist for use with the STDFA instruction as Partial Store (STPARTIALF) operations (see *Store Partial Floating-Point* on page 351).

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 and the ASI register contains one of the Partial Store ASIs.

If one of these ASIs is used with a Store Alternate instruction other than STDFA, a Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is generated and *mem_address_not_aligned*, *LDDF_mem_address_not_aligned*, and *illegal_instruction* (for i = 1) are not generated.

ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–$CD_{16}$ are only defined for use in Partial Store operations (see page 351). None of them should be used with LDDFA; however, if any of those ASIs *is* used with LDDFA, the resulting behavior is specified in the LDDFA instruction description on page 258.

### 10.4.13 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 *Load Floating-Point Register* on page 253 and *Store Floating-Point* on page 342).

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 Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

## 10.5 ASI-Accessible Registers

In this section the Data Watchpoint registers, scratchpad registers, and CMT registers are described.

A list of UltraSPARC Architecture 2007 ASIs is shown in TABLE 10-1 on page 431.

## 10.5.1 Privileged Scratchpad Registers (ASI_SCRATCHPAD) D1

An UltraSPARC Architecture virtual processor includes eight Scratchpad registers (64 bits each, read/write accessible) (impl.dep. #302-U4-Cs10). The use of the Scratchpad registers is completely defined by software.

For conventional uses of Scratchpad registers, see "Scratchpad Register Usage" in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

The Scratchpad registers are intended to be used by performance-critical trap handler code.

The addresses of the privileged scratchpad registers are defined in TABLE 10-8.

**TABLE 10-8** Scratchpad Registers

| Assembly Language ASI Name | ASI # | Virtual Address | Privileged Scratchpad Register # |
|---|---|---|---|
| ASI_SCRATCHPAD | $20_{16}$ | $00_{16}$ | 0 |
| | | $08_{16}$ | 1 |
| | | $10_{16}$ | 2 |
| | | $18_{16}$ | 3 |
| | | $20_{16}$ | 4 |
| | | $28_{16}$ | 5 |
| | | $30_{16}$ | 6 |
| | | $38_{16}$ | 7 |

**IMPL. DEP. #404-S10:** The degree to which Scratchpad registers 4–7 are accessible to privileged software is implementation dependent. Each may be
(1) fully accessible,
(2) accessible, with access much slower than to scratchpad registers 0–3 (emulated by *DAE_invalid_asi* trap to hyperprivileged software), or
(3) inaccessible (cause a *DAE_invalid_asi* exception).

**V9 Compatibility Note** Privileged scratchpad registers are an UltraSPARC Architecture extension to SPARC V9.

## 10.5.2 Hyperprivileged Scratchpad Registers (ASI_HYP_SCRATCHPAD) D2

An UltraSPARC Architecture virtual processor includes eight hyperprivileged Scratchpad registers (64 bits each, read/write accessible). The use of the hyperprivileged Scratchpad registers is completely defined by software.

The hyperprivileged Scratchpad registers are intended to be used in hyperprivileged trap handler code.

The hyperprivileged Scratchpad registers are accessed with Load Alternate and Store Alternate instructions, using the ASIs and addresses listed in TABLE 10-9.

**IMPL. DEP. #407-S10:** It is implementation dependent whether any of the hyperprivileged Scratchpad registers are aliased to the corresponding privileged Scratchpad register or is an independent register.

**TABLE 10-9** Hyperprivileged Scratchpad Registers

| Assembly Language ASI Name | ASI # | Virtual Address | Hyperprivileged Scratchpad Register # |
|---|---|---|---|
| ASI_HYP_SCRATCHPAD | $4F_{16}$ | $00_{16}$ | 0 |
| | | $08_{16}$ | 1 |
| | | $10_{16}$ | 2 |
| | | $18_{16}$ | 3 |
| | | $20_{16}$ | 4 |
| | | $28_{16}$ | 5 |
| | | $30_{16}$ | 6 |
| | | $38_{16}$ | 7 |

**V9 Compatibility Note** Hyperprivileged Scratchpad registers are an UltraSPARC Architecture extension to SPARC V9.

## 10.5.3 CMT Registers Accessed Through ASIs D2

All chip-level multithreading (CMT) registers are accessed through ASIs. See *Accessing CMT Registers* on page 597, for descriptions of ASI registers used to control CMT functions.

## 10.5.4 ASI Changes in the UltraSPARC Architecture

The following Compatibility Notes summarize the UltraSPARC ASI changes in UltraSPARC Architecture.

**Compatibility Note**

The names of several ASIs used in earlier UltraSPARC implementations have changed in UltraSPARC Architecture. Their functions have not changed; just their names have changed.

| ASI# | Previous UltraSPARC | UltraSPARC Architecture |
|---|---|---|
| $14_{16}$ | ASI_PHYS_USE_EC | ASI_REAL |
| $15_{16}$ | ASI_PHYS_BYPASS_EC_WITH_EBIT | ASI_REAL_IO |
| $1C_{16}$ | ASI_PHYS_USE_EC_LITTLE (ASI_PHYS_USE_EC_L) | ASI_REAL_LITTLE |
| $1D_{16}$ | ASI_PHYS_BYPASS_EC_WITH_ EBIT_LITTLE (ASI_PHY_BYPASS_EC_WITH_EBIT_L) | ASI_REAL_IO_LITTLE |

**Compatibility Note**

The names *and* ASI assignments (but not functions) changed between earlier UltraSPARC implementations and UltraSPARC Architecture, for the following ASIs:

| Previous UltraSPARC | | UltraSPARC Architecture | |
|---|---|---|---|
| ASI# | Name | ASI# | Name |
| $24_{16}$ | ASI_NUCLEUS_QUAD_LDD$^D$ | $27_{16}$ | ASI_TWINX_NUCLEUS (ASI_TWINX_N) |
| $2C_{16}$ | ASI_NUCLEUS_QUAD_LDD_ LITTLE$^D$ (ASI_NUCLEUS_QUAD_LDD_L$^D$) | $2F_{16}$ | ASI_TWINX_NUCLEUS_ LITTLE (ASI_TWINX_NL) |
| $34_{16}$ | ASI_QUAD_LDD_PHYS$^D$ | $26_{16}$ | ASI_TWINX_REAL (ASI_TWINX_R) |
| $3C_{16}$ | ASI_QUAD_LDD_LITTLE$^D$ (ASI_QUAD_LDD_L$^D$) | $2E_{16}$ | ASI_TWINX_REAL_LITTLE (ASI_TWINX_REAL_L) |

CHAPTER 11

# Performance Instrumentation

This chapter describes the architecture for performance monitoring hardware on UltraSPARC Architecture processors. The architecture is based on the design of performance instrumentation counters in previous UltraSPARC Architecture processors, with an extension for the selective sampling of instructions.

## 11.1 High-Level Requirements

### 11.1.1 Usage Scenarios

The performance monitoring hardware on UltraSPARC Architecture processors addresses the needs of various kinds of users. There are four scenarios envisioned:

- *System-wide performance monitoring.* In this scenario, someone skilled in system performance analysis (e.g, a Systems Engineer) is using analysis tools to evaluate the performance of the entire system. An example of such a tool is cpustat. The objective is to obtain performance data relating to the configuration and behavior of the system, e.g., the utilization of the memory system.
- *Self-monitoring of performance by the operating system.* In this scenario the OS is gathering performance data in order to tune the operation of the system. Some examples might be:
  - (a) determining whether the processors in the system should be running in single- or multi-stranded mode.
  - (b) determining the affinity of a process to a processor by examining that process's memory behavior.
- *Performance analysis of an application by a developer.* In this scenario a developer is trying to optimize the performance of a specific application, by altering the source code of the application or the compilation options. The developer needs to know the performance characteristics of the components of the application at a coarse

grain, and where these are problematic, to be able to determine fine-grained performance information. Using this information, the developer will alter the source or compilation parameters, re-run the application, and observe the new performance characteristics. This process is repeated until performance is acceptable, or no further improvements can be found.

An example might be that a loop nest is measured to be not performing well. Upon closer inspection, the developer determines that the loop has poor cache behavior, and upon more detailed inspection finds a specific operation which repeatedly misses the cache. Reorganizing the code and/or data may improve the cache behavior.

- *Monitoring of an application's performance, e.g., by a Java Virtual Machine*. In this scenario the application is not executing directly on the hardware, but its execution is being mediated by a piece of system software, which for the purposes of this document is called a Virtual Machine. This may be a Java VM, or a binary translation system running software compiled for another architecture, or for an earlier version of the UltraSPARC Architecture. One goal of the VM is to optimize the behavior of the application by monitoring its performance and dynamically reorganizing the execution of the application (e.g., by selective recompilation of the application).

  This scenario differs from the previous one principally in the time allowed to gather performance data. Because the data are being gathered during the execution of the program, the measurements must not adversely affect the performance of the application by more than, say, a few percent, and must yield insight into the performance of the application in a relatively short time (otherwise, optimization opportunities are deferred for too long). This implies an observation mechanism which is of very low overhead, so that many observations can be made in a short time.

  In contrast, a developer optimizing an application has the luxury of running or re-running the application for a considerable period of time (minutes or even hours) to gather data. However, the developer will also expect a level of precision and detail in the data which would overwhelm a virtual machine, so the accuracy of the data required by a virtual machine need not be as high as that supplied to the developer.

Scenarios 1 and 2 are adequately dealt with by a suitable set of performance counters capable of counting a variety of performance-related events. Counters are ideal for these situations because they provide low-overhead statistics without any intrusion into the behavior of the system or disruption to the code being monitored. However, counters may not adequately address the latter two scenarios, in which detailed and timely information is required at the level of individual instructions. Therefore, UltraSPARC Architecture processors may also implement an instruction sampling mechanism.

## 11.1.2 Metrics

There are two classes of data reported by a performance instrumentation mechanism:

- *Architectural performance metrics.* These are metrics related to the observable execution of code at the architectural level (UltraSPARC Architecture). Examples include:
  - The number of instructions executed
  - The number of floating point instructions executed
  - The number of conditional branch instructions executed
- *Implementation performance metrics.* These describe the behavior of the microprocessor in terms of its implementation, and would not necessarily apply to another implementation of the architecture.

In optimizing the performance of an application or system, attention will first be paid to the first class of metrics, and so these are more important. Only in performance-critical cases would the second class receive attention, since using these metrics requires a fairly extensive understanding of the specific implementation of the UltraSPARC Architecture.

## 11.1.3 Accuracy Requirements

Accuracy requirements for performance instrumentation vary depending on the scenario. The requirements are complicated by the possibly speculative nature of UltraSPARC Architecture processor implementations. For example, an implementation may include in its cache miss statistics the misses induced by speculative executions which were subsequently flushed, or provide two separate statistics, one for the misses induced by flushed instructions and one for misses induced by retired instructions. Although the latter would be desirable, the additional implementation complexity of associating events with specific instructions is significant, and so all events may be counted without distinction. The instruction sampling mechanism may distinguish between instructions that retired and those that were flushed, in which case sampling can be used to obtain statistical estimates of the frequencies of operations induced by mis-speculation.

For critical performance measurements, architectural event counts must be accurate to a high degree (1 part in $10^5$). Which counters are considered performance-critical (and therefore accurate to 1 part in $10^5$) are specified in implementation-specific documentation.

Implementation event counts must be accurate to 1 part in $10^3$, not including the speculative effects mentioned above. An upper bound on counter skew must be stated in implementation-specific documentation.

**Programming Note** Increasing the time between counter reads will mitigate the inaccurcies that could be introduced by counter skew (due to speculative effects).

## 11.2 Performance Counters and Controls

The performance instrumentation hardware provides performance instrumentation counters (PICs). The number and size of performance counters is implementation dependent, but each performance counter register contains at least one 32-bit counter. It is implementation dependent whether the performance counter registers are accessed as ASRs or are accessed through ASIs.

There are one or more performance counter control registers (PCRs) associated with the counter registers. It is implementation dependent whether the PCRs are accessed as ASRs or are accessed through ASIs.

Each counter in a counter register can count one kind of event at a time. The number of the kinds of events that can be counted is implementation dependent. For each performance counter register, the corresponding control register is used to select the event type being counted. A counter is incremented whenever an event of the matching type occurs. A counter may be incremented by an event caused by an instruction which is subsequently flushed (for example, due to mis-speculation). Counting of events may be controlled based on privilege mode or on the strand in which they occur. Masking may be provided to allow counting of subgroups of events (for example, various occurrences of different opcode groups).

A field that indicates when a counter has overflowed must be present in either each performance instrumentation counter or in a separate performance counter control register.

Performance counters are usually provided on a per-strand basis.

### 11.2.1 Counter Overflow

Overflow of a counter must cause a *pic_overflow* disrupting trap to be generated, when enabled by a Trap Overflow Enable bit (in an implementation-specific location). There must be a separate toe bit for each performance counter, so that overflow traps can be enabled on a per-counter basis. Overflow of a counter is recorded in the overflow-indication field of either a performance counter register or a separate performance counter control register.

**Programming Note** Counter overflow traps can also be used for sampling, by setting the initial counter value so that an interrupt occurs *n* counts later.

Counter overflow traps are provided so that large counts can be maintained in software, beyond the range directly supported in hardware. The counters continue to count after an overflow, and software can utilize the overflow traps to maintain additional high-order bits.

CHAPTER 12

# Traps

A *trap* is a vectored transfer of control to software running in a privilege mode (see page 464) with (typically) greater privileges. A trap in nonprivileged mode can be delivered to privileged mode or hyperprivileged mode. A trap that occurs while executing in privileged mode can be delivered to privileged mode or hyperprivileged mode. A trap that occurs while executing in hyperprivileged mode can only be delivered to hyperprivileged mode.

The actual transfer of control occurs through a trap table that contains the first eight instructions (32 instructions for *clean_window, fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, *fast_data_access_protection*, window spill, and window fill, traps) of each trap handler. The virtual base address of the trap table for traps to be delivered in privileged mode is specified in the Trap Base Address (TBA) register. The physical base address of the trap table for traps to be delivered in hyperprivileged mode is specified in the Hyperprivileged Trap Base Address (HTBA) register. The displacement within either table is determined by the trap type and the current trap level (TL). One-half of each table is reserved for hardware traps; the other half is reserved for software traps generated by Tcc instructions.

A trap behaves like an unexpected procedure call. It causes the hardware to do the following:

1. Save certain virtual processor state (such as program counters, CWP, ASI, CCR, PSTATE, and the trap type) on a hardware register stack.
2. Enter privileged execution mode with a predefined PSTATE, or enter hyperprivileged mode with a predefined PSTATE and HPSTATE.
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.

**V9 Compatibility Note** | Exceptions referred to as "catastrophic error exceptions" in the SPARC V9 specification do not exist in the UltraSPARC Architecture; they are handled using normal error-reporting exceptions. (impl. dep. #31-V8-Cs10)

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

Traps are described in these sections:


- **Virtual Processor Privilege Modes** on page 464.
- **Virtual Processor States, Normal Traps, and `RED_state` Traps** on page 466.
- **Trap Categories** on page 471.
- **Trap Control** on page 477.
- **Trap-Table Entry Addresses** on page 479.
- **Trap Processing** on page 497.
- **Exception and Interrupt Descriptions** on page 508.
- **Register Window Traps** on page 522.


# 12.1 Virtual Processor Privilege Modes

An UltraSPARC Architecture virtual processor is always operating in a discrete privilege mode. The privilege modes are listed below in order of increasing privilege:

- Nonprivileged mode (also known as "user mode")
- Privileged mode, in which supervisor (operating system) software primarily operates
- Hyperprivileged mode, in which hypervisor software operates, serving as a layer between the supervisor software and the underlying virtual processor

The virtual processor's operating mode is determined by the state of two mode bits, as shown in TABLE 12-1.

**TABLE 12-1** Virtual Processor Privilege Modes

| HPSTATE.hpriv | PSTATE.priv | Virtual Processor Privilege Mode |
|---|---|---|
| 0 | 0 | Nonprivileged |
| 0 | 1 | Privileged |
| 1 | — | Hyperprivileged |

A trap is delivered to the virtual processor in either privileged mode or hyperprivileged mode; in which mode the trap is delivered depends on:

- Its trap type
- The trap level (TL) at the time the trap is taken
- The privilege mode at the time the trap is taken

Traps detected in nonprivileged and privileged mode can be delivered to the virtual processor in privileged mode or hyperprivileged mode. Traps detected in hyperprivileged mode are either delivered to the virtual processor in hyperprivileged mode or may be masked. If masked, they are held pending.

TABLE 12-4 on page 485 indicates in which mode each trap is processed, based on the privilege mode at which it was detected.

A trap delivered to privileged mode uses the privileged-mode trap vector, based upon the TBA register. See *Trap-Table Entry Address to Privileged Mode* on page 479 for details. A trap delivered to hyperprivileged mode uses the hyperprivileged mode trap vector address, based upon the HTBA register. See *Trap-Table Entry Address to Hyperprivileged Mode* on page 480 for details.

The maximum trap level at which privileged software may execute is *MAXPTL* (which, on an UltraSPARC Architecture 2007 virtual processor, is 2). Therefore, if TL ≥ *MAXPTL* and a trap occurs that would normally be delivered in privileged mode, it is instead delivered in hyperprivileged mode; the trap table offset for *watchdog_reset* ($40_{16}$) is used, and the priority and trap type of the original exception is retained. This is referred to as a "*guest_watchdog*" trap (so named because it uses *watchdog_reset*'s trap table offset).

**Notes**

Execution in nonprivileged or privileged mode with TL > *MAXPTL* is an invalid condition that hyperprivileged software should never allow to occur.

Execution in nonprivileged mode with TL > 0 is an invalid condition that privileged and hyperprivileged software should never allow to occur.

FIGURE 12-1 shows how a virtual processor transitions between privilege modes, excluding transitions that can occur due to direct software writes to PSTATE.priv or HPSTATE.hpriv. In this figure, PT indicates a "trap destined for privileged mode" and HT indicates a "trap destined for hyperprivileged mode".

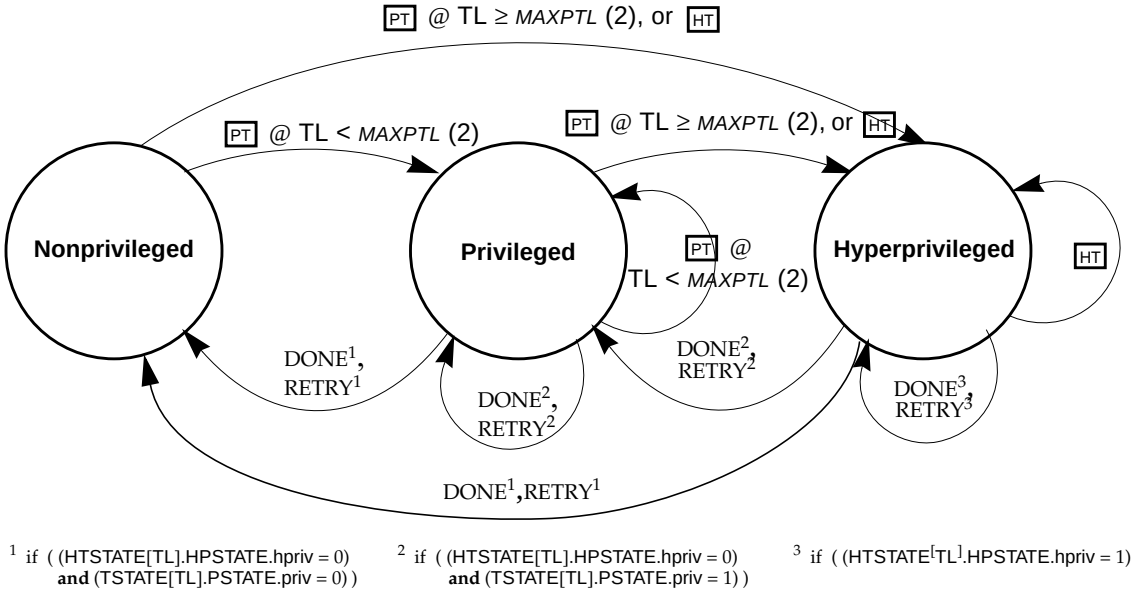



**FIGURE 12-1** Virtual Processor Privilege Mode Transition Diagram

## 12.2 Virtual Processor States, Normal Traps, and RED_state Traps

An UltraSPARC Architecture 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 transient state that is entered as a result of a non-reset trap, SIR, or XIR when TL = *MAXTL*

The values of TL and HPSTATE.red affect the generated trap vector address. TL also determines where (that is, into which element of the TSTATE and HTSTATE arrays) the states are saved..

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 the SPARC V9 specification. The name was changed to clarify the terminology.

FIGURE 12-2 shows the virtual processor state transition diagram.

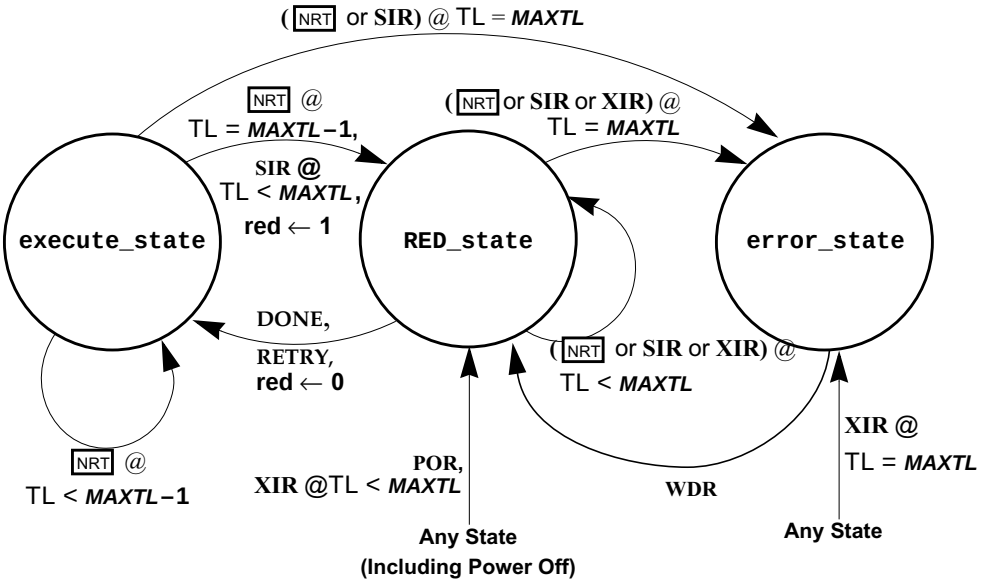



**FIGURE 12-2** Virtual Processor State Diagram ("NRT" = "non-reset trap")

## 12.2.1 `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 non-reset trap is taken when TL = *MAXTL* −1.
- A POR or WDR reset occurs.
- An SIR reset occurs when TL < *MAXTL*.
- An XIR reset occurs when TL < *MAXTL*.
- System software sets HPSTATE.red = 1. For this condition, no other machine state or operation is modified as a side-effect of the write to HPSTATE; software must set the appropriate machine state.

`RED_state` serves two purposes:

- During trap processing, it indicates that no more trap levels are available; that is, while executing in `RED_state` with TL = *MAXTL*, if another nested non-reset trap, SIR, or XIR is taken, the virtual processor will enter `error_state`. `RED_state` provides system software with a restricted execution environment.

- It provides the execution environment for all reset processing.

RED_state is indicated by HPSTATE.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 HPSTATE register, which zeroes the HPSTATE.red flag if it was zero in the stacked copy. System software can also directly write 1 or 0 to HPSTATE.red with a WRHPR instruction, which forces the virtual processor to enter or exit RED_state, respectively. In this case, the WRHPR instruction should be placed in the delay slot of a jump instruction so that the PC can be changed in concert with the state change.

When RED_state is entered due to a reset or a trap, the execution environment is altered in four ways:

- Address translation is disabled in the MMU, for both instruction and data references.
- Watchpoints are disabled.
- The trap vector for the traps occurring in RED_state is based on the RED_state Trap Table.
- The virtual processor enters hyperprivileged mode (HPSTATE.hpriv ← 1).

**Programming Note** Setting TL ← *MAXTL* with a WRHPR instruction does not also set HPSTATE.red ← 1, nor does it alter any other machine state. The values of HPSTATE.red and TL are independent.

Setting HPSTATE.red with a WRHPR instruction causes the virtual processor to execute in RED_state. This results in the execution environment defined in ***RED_state** Execution Environment* on page 468. 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).

The trap table organization for RED_state traps is described in ***RED_state** Trap Table Organization* on page 482.

### 12.2.1.1 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.

Some of the characteristics of RED_state include:

- Memory accesses in RED_state are by default noncacheable, so there must be noncacheable scratch memory somewhere in the system.
- The D-cache watchpoints and DMMU/UMMU can be enabled by software in RED_state, but any trap will disable them again.

- The caches continue to snoop and maintain coherence in RED_state if DMA or other virtual processors are still issuing cacheable accesses.

**IMPL. DEP. #115-V9**: A processor's behavior in RED_state is implementation dependent.

**Programming Note** When RED_state is entered because of component failures, trap handler software should attempt to recover from potentially fatal 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.

#### 12.2.1.2 RED_state Entry Traps

The following reset traps are processed in RED_state:

- **Power-on reset (POR)** — POR causes the virtual processor to start execution at this trap table entry.
- **Watchdog reset (WDR)** — While in error_state, the virtual processor automatically invokes a watchdog reset to enter RED_state (impl. dep. #254-U3-Cs10).
- **Externally initiated reset (XIR)** — This trap is typically used as a nonmaskable interrupt for debugging purposes. If TL < *MAXTL* when an XIR occurs, the reset trap is processed in RED_state; if TL = *MAXTL* when an XIR occurs, the virtual processor transitions directly to error_state.
- **Software-initiated reset (SIR**) If TL < *MAXTL* when an SIR occurs, the reset trap is processed in RED_state; if TL = *MAXTL* when an SIR occurs, the virtual processor transitions directly to error_state.

Non-reset traps that occur when TL = *MAXTL* – 1 also set HPSTATE.red = 1; that is, any non-reset trap handler entered with TL = *MAXTL* runs in RED_state.

Any non-reset trap that sets HPSTATE.red = 1 or that occurs when HPSTATE.red = 1 branches to a special entry in the RED_state trap vector at $RSTV_{ADDR} + A0_{16}$. Reset traps are described in *Reset Traps* on page 476.

#### 12.2.1.3 RED_state Software Considerations

In effect, RED_state reserves one level of the trap stack for recovery and reset processing. Hyperprivileged 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.

**Programming 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 hyperprivileged scratchpad registers for use in RED_state.

#### 12.2.1.4 Usage of Trap Levels

If *MAXPTL* = 2 and *MAXTL* = 5 in an UltraSPARC Architecture implementation, the trap levels might be used as shown in TABLE 12-2.

**TABLE 12-2** Typical Usage for Trap Levels

| TL | Corresponding Execution Mode | Usage |
|---|---|---|
| 0 | Nonprivileged | Normal execution |
| 1 | Privileged | System calls; interrupt handlers; instruction emulation |
| 2 | Privileged | Window spill/fill handler |
| 3 | Hyperprivileged | Real address TLB miss handler |
| 4 | Hyperprivileged | Reserved for error handling |
| 5 | Hyperprivileged | RED_state handler |

### 12.2.2 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, it enters error_state when a trap occurs while TL = *MAXTL*. No other conditions cause entry into error_state on an UltraSPARC Architecture virtual processor. (impl. dep. #39-V8-Cs10**)**

**IMPL. DEP. #40-V8**: 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, an UltraSPARC Architecture virtual processor may have other error_state entry traps that are implementation dependent.

Upon entering error_state, a virtual processor automatically generates a *watchdog_reset* (WDR) (impl. dep. #254-U3-Cs10), which causes entry into RED_state.

## 12.3 Trap Categories

An exception, error, or interrupt request can cause any of the following trap types:

- Precise trap
- Deferred trap
- Disrupting trap
- Reset trap

### 12.3.1 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 that trap handler software might take when processing a precise trap are:

- 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.

### 12.3.2 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.

There are two classes of deferred traps:

- *Termination deferred traps* — The instruction (usually a store) that caused the trap has passed the retirement point of execution (the TPC has been updated to point to an instruction beyond the one that caused the trap). The trap condition is an

error that prevents the instruction from completing and its results becoming globally visible. A termination deferred trap has high trap priority, second only to the priority of resets.

**Programming Note** Not enough state is saved for execution of the instruction stream to resume with the instruction that caused the trap. Therefore, the trap handler must terminate the process containing the instruction that caused the trap.

- *Restartable deferred traps* — The program-visible state has been changed by the trap-inducing instruction or by one or more other instructions after the trap-inducing instruction.

**SPARC V9 Compatibility Note** A *restartable* deferred trap is the "deferred trap" defined in the SPARC V9 specification.

The fundamental characteristic of a *restartable* deferred trap is that the state of the virtual processor on which the trap occurred may not be consistent with any precise point in the instruction sequence being executed on that virtual processor. When a restartable deferred trap occurs, TPC[TL] and TNPC[TL] contain a PC value and an NPC value, respectively, corresponding to a point in the instruction sequence being executed on the virtual processor. This PC may correspond to the trap-inducing instruction or it may correspond to an instruction following the trap-inducing instruction. With a restartable deferred trap, program-visible updates may be missing from instructions prior to the instruction to which TPC[TL] refers. The missing updates are limited to instructions in the range from (and including) the actual trap-inducing instruction up to (but not including) the instruction to which TPC[TL] refers. By definition, the instruction to which TPC[TL] refers has not yet executed, therefore it cannot have any updates, missing or otherwise.

With a restartable deferred trap there must exist sufficient information to report the error that caused the deferred trap. If system software can recover from the error that caused the deferred trap, then there must be sufficient information to generate a consistent state within the processor so that execution can resume. Included in that information must be an indication of the mode (nonprivileged, privileged, or hyperprivileged) in which the trap-inducing instruction was issued.

How the information necessary for repairing the state to make it consistent state is maintained and how the state is repaired to a consistent state are implementation dependent. It is also implementation dependent whether execution resumes at the point of the trap-inducing instruction or at an arbitrary point between the trap-inducing instruction and the instruction pointed to by the TPC[TL], inclusively.

Associated with a particular restartable deferred trap implementation, the following must exist:

- An instruction that causes a potentially outstanding restartable deferred trap exception to be taken as a trap

- Instructions with sufficient privilege to access the state information needed by software to emulate the restartable 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 restartable deferred trap, that is, those instructions in the deferred-trap queue.

Software should resume execution with the instruction starting at the instruction to which TPC[TL] refers. Hardware should provide enough information for software to recreate virtual processor state and update it to the point just before execution of the instruction to which TPC[TL] refers. After software has updated virtual processor state up to that point, it can then resume execution by issuing a RETRY instruction.

**IMPL. DEP. #32-V8-Ms10:** Whether any restartable deferred traps (and, possibly, associated deferred-trap queues) are present is implementation dependent.

Among the actions software can take after a restartable 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 restartable 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 restartable deferred trap.

A deferred trap (of either of the two classes) is always delivered to the virtual processor in hyperprivileged mode.

## 12.3.3 Disrupting Traps

### 12.3.3.1 Disrupting versus Precise and Deferred Traps

A *disrupting trap* is caused by a condition (for example, an interrupt) rather than directly by a particular instruction. This 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. This distinguishes disrupting traps from *reset* traps, since a reset trap vectors to a unique reset address and execution of the program that was running when the reset occurred is generally not expected to resume.

When a disrupting trap occurs, the following conditions are true:

1. The PC saved in TPC[TL] points to an instruction in the disrupted program stream and the NPC value saved in TNPC[TL] points to the instruction that was to be executed after that one.

2. All instructions issued before the instruction indicated by TPC[TL] have retired.

3. The instruction to which TPC[TL] refers and any instruction(s) that were issued after it remain unexecuted.

A disrupting trap may be due to an interrupt request directly related to a previously-executed instruction; for example, when a previous instruction sets a bit in the SOFTINT register.

### 12.3.3.2 Causes of Disrupting Traps

A disrupting trap may occur due to either an interrupt request or an error not directly related to instruction processing. The source of an interrupt request 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 condition that causes a disrupting trap persists until the condition is cleared.

### 12.3.3.3 Conditioning of Disrupting Traps

How disrupting traps are conditioned is affected by:

- The privilege mode in effect when the trap is outstanding, just before the trap is actually taken (regardless of the privilege mode that was in effect when the exception was detected).
- The privilege mode for which delivery of the trap is destined

**Outstanding in Nonprivileged or Privileged mode, destined for delivery in Privileged mode.** An outstanding disrupting trap condition in either nonprivileged mode or privileged mode and destined for delivery to privileged mode is held pending while the Interrupt Enable (ie) field of PSTATE is zero (PSTATE.ie = 0). *interrupt_level_n* interrupts are further conditioned by the Processor Interrupt Level (PIL) register. An interrupt is held pending while either PSTATE.ie = 0 or the condition's interrupt level is less than or equal to the level specified in PIL. When delivery of this disrupting trap is enabled by PSTATE.ie = 1, it is delivered to the virtual processor in privileged mode if TL < *MAXPTL* (2, in UltraSPARC Architecture 2007 implementations) or in hyperprivileged mode if TL ≥ *MAXPTL*.

**Outstanding in Hyperprivileged mode, destined for delivery in Privileged mode.** An outstanding disrupting trap condition detected while in hyperprivileged mode and destined for delivery in privileged mode is held pending while in hyperprivileged mode (HPSTATE.priv = 1), regardless of the values of TL and PSTATE.ie.

**Outstanding in Nonprivileged or Privileged mode, destined for delivery in Hyperprivileged mode.** An outstanding disrupting trap condition detected while in either nonprivileged mode or privileged mode and destined for delivery in hyperprivileged mode is never masked; it is delivered immediately.

**Outstanding in Hyperprivileged mode, destined for delivery in Hyperprivileged mode.** An outstanding disrupting trap condition detected in hyperprivileged mode and destined to be delivered in hyperprivileged mode is masked and held pending while PSTATE.ie = 0.

The above is summarized in TABLE 12-3.

**TABLE 12-3** Conditioning of Disrupting Traps

| Type of Disrupting Trap Condition | Current Virtual Processor Privilege Mode | Disposition of Disrupting Traps, based on privilege mode in which the trap is destined to be delivered | |
|---|---|---|---|
| | | Privileged | Hyperprivileged |
| *Interrupt_level_n* | Nonprivileged or Privileged | Held pending while PSTATE.ie = 0 or interrupt level ≤ PIL | — |
| | Hyperprivileged | Held pending while HPSTATE.hpriv = 1 | — |
| All other disrupting traps | Nonprivileged or Privileged | Held pending while PSTATE.ie = 0 | Delivered immediately |
| | Hyperprivileged | Held pending while HPSTATE.hpriv = 1 | Held pending while PSTATE.ie = 0 |

## 12.3.3.4 Trap Handler Actions for Disrupting Traps

Among the actions that trap-handler software might take to process a disrupting trap are:

- 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.

## 12.3.3.5 Clearing Requirement for Disrupting Traps

For each disrupting trap, a method must be provided for hyperprivileged software (or a service processor, if present) to detect and clear the pending disrupting trap without taking its corresponding hardware trap.

## 12.3.4 Reset Traps

A *reset trap* occurs when hyperprivileged 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:

- They are not maskable.
- Trap handler software for resets is generally not expected to resume execution of the program that was running when the reset trap occurred. After an SIR or XIR, execution of the interrupted program may resume.

All *reset traps* are delivered to the virtual processor in hyperprivileged mode.

**IMPL. DEP. #37-V8:** Some of a virtual processor's behavior during a reset trap is implementation dependent. See ***RED_state*** *Trap Processing* on page 501 for details.

The following reset traps are defined by the SPARC V9 architecture:

- **Power-on reset (POR)** — Used for initialization purposes (for example, when power is applied or reapplied to the virtual processor).
- **Watchdog reset (WDR)** — Initiated when the virtual processor enters `error_state` (impl. dep. #254-U3-Cs10). The WDR reset trap is taken instead of the trap request that caused entry to `error_state` at TL = *MAXTL*. TSTATE[*MAXTL*], TPC[*MAXTL*], TNPC[*MAXTL*] and TT[*MAXTL*] observed after a WDR reset trap are those associated with the trap request that caused entry to `error_state`. The value of TT[*MAXTL*] indicates the trap type of this trap. Machine state is consistent; however, software should not resume normal instruction processing at the address in TPC[TL] after the WDR reset trap. The values in TSTATE[*MAXTL*], TPC[*MAXTL*], TNPC[*MAXTL*] and TT[*MAXTL*] are accurate and are intended for debug purposes.
- **Externally initiated reset (XIR)** — Initiated in response to a signal or event that is external to the virtual processor. This reset trap is normally used for critical system events, such as power failure. The XIR reset trap is treated as an interrupt and processed similarly to a disrupting trap (but without masking). Software can resume the interrupted program at the conclusion of trap handler execution. If the XIR reset is detected when TL = *MAXTL*, the virtual processor enters `error_state` and triggers a WDR reset. Trap handler code for the resulting WDR reset can determine that the original cause of the entry to `error_state` was an XIR reset by observing that the trap type saved in TT[*MAXTL*] is 3 (indicating XIR).
- **Software-initiated reset (SIR)** — Initiated by software by executing the SIR instruction in hyperprivileged mode. In nonprivileged and privileged mode, the SIR instruction causes an *illegal_instruction* exception (which results in a trap to hyperprivileged mode). The SIR reset trap is processed similar to a precise trap. The PC saved in TPC[TL] points to the SIR instruction. If the SIR reset is detected when TL = *MAXTL*, the virtual processor enters `error_state` and triggers a WDR

reset. Trap handler code for the resulting WDR reset can determine that the original cause of the entry to `error_state` was an SIR reset by observing that the trap type saved in TT[*MAXTL*] is 4 (indicating SIR).

### 12.3.5 Uses of the Trap Categories

The SPARC V9 *trap model* stipulates the following:

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. See TABLE 12-4, TABLE 12-5, and *Exception and Interrupt Descriptions* on page 508 for identification of which traps are precise.
3. In an UltraSPARC Architecture implementation, all exceptions that occur as the result of program execution, except for errors on store instructions that occur after the store instruction that has passed the retirement point, are precise (impl. dep. #33-V8-Cs10).
4. An error detected after the initial access of a multiple-access load instruction (for example, LDTX or LDBLOCKF) should be precise. Thus, a trap due to the second memory access can occur. However, the processor state should not have been modified by the first access.
5. 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.

## 12.4 Trap Control

Several registers control how any given exception is processed, for example:

- The interrupt enable (ie) field in PSTATE and the Processor Interrupt Level (PIL) register control interrupt processing. See *Disrupting Traps* on page 473 for details.
- 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 hyperprivileged mode bit (hpriv) field in the HPSTATE register, which can affect how a trap is delivered. See *Conditioning of Disrupting Traps* on page 474 for details.
- 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`. It also affects whether the trap is processed in privileged mode or hyperprivileged mode.
- For a trap delivered to the virtual processor in privileged mode, PSTATE.tle determines whether implicit data accesses in the trap handler routine will be performed using big-endian or little-endian byte order. A trap delivered to the virtual processor in hyperprivileged mode always uses a default byte order of big-endian.

Between the execution of instructions, the virtual processor prioritizes the outstanding exceptions, errors, and interrupt requests. At any given time, only the highest-priority exception, error, or interrupt request is taken as a trap. When there are multiple interrupts outstanding, the interrupt with the highest interrupt level is selected. When there are multiple outstanding exceptions, errors, and/or interrupt requests, a trap occurs based on the exception, error, or interrupt with the highest priority (numerically lowest priority number in TABLE 12-5). See *Trap Priorities* on page 496.

## 12.4.1 PIL Control

When an interrupt request occurs, the virtual processor compares its interrupt request level against the value in the Processor Interrupt Level (PIL) register. If the interrupt request level is greater than PIL and no higher-priority exception is outstanding, then the virtual processor takes a trap using the appropriate *interrupt_level_n* trap vector.

## 12.4.2 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 FSR.tem is 1, the associated IEEE_754_exception can cause an *fp_exception_ieee_754* trap.

If a particular bit of FSR.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, FSR.fcc*n*, and FSR.aexc fields remain unchanged. However, if an IEEE_754_exception does not result in a trap, then the F register, FSR.fcc*n*, and FSR.aexc fields are updated to their new values.

# 12.5 Trap-Table Entry Addresses

Traps are delivered to the virtual processor in either privileged mode or hyperprivileged mode, depending on the trap type, the value of TL at the time the trap is taken, and the privilege mode at the time the exception was detected. See TABLE 12-4 on page 485 and TABLE 12-5 on page 491 for details.

Unique trap table base addresses are provided for traps being delivered in privileged mode and in hyperprivileged mode.

## 12.5.1 Trap-Table Entry Address to Privileged Mode

Privileged software initializes bits 63:15 of the Trap Base Address (TBA) register (its most significant 49 bits) with bits 63:15 of the desired 64-bit privileged trap-table base address.

At the time a trap to privileged mode is taken:

- Bits 63:15 of the trap vector address are taken from TBA{63:15}.
- Bit 14 of the trap vector address (the "TL>0" field) is set based on the value of TL just before the trap is taken; that is, if TL = 0 then bit 14 is set to 0 and if TL > 0 then bit 14 is set to 1.
- Bits 13:5 of the trap vector address contain a copy of the contents of the TT register (TT[TL]).
- Bits 4:0 of the trap vector address 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 12-3 illustrates the trap vector address for a trap delivered to privileged mode. In FIGURE 12-3, 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, as detailed in the following section, that there are two trap tables for traps to privileged mode: one for traps from TL = 0 and one for traps from TL > 0.

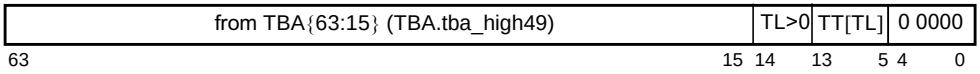



**FIGURE 12-3** Privileged Mode Trap Vector Address

## 12.5.2 Privileged Trap Table Organization

The layout of the privileged-mode trap table (which is accessed using virtual addresses) is illustrated in FIGURE 12-4.

| Value of TL (before trap) | Software Trap Type | Hardware Trap Type (TT[TL]) | Trap Table Offset (from TBA) | Contents of Trap Table |
|---|---|---|---|---|
| TL = 0 | — | $000_{16}$–$07F_{16}$ | $0_{16}$– $FE0_{16}$ | Hardware traps |
| | — | $080_{16}$–$0FF_{16}$ | $1000_{16}$–$1FE0_{16}$ | *Spill* / *fill* traps |
| | $0_{16}$– $7F_{16}$ | $100_{16}$–$17F_{16}$ | $2000_{16}$–$2FE0_{16}$ | Software traps to Privileged level |
| | — | $180_{16}$–$1FF_{16}$ | $3000_{16}$–$3FE0_{16}$ | *unassigned* |
| TL = 1 (TL = *MAXPTL*–1) | — | $000_{16}$–$07F_{16}$ | $4000_{16}$–$4FE0_{16}$ | Hardware traps |
| | — | $080_{16}$–$0FF_{16}$ | $5000_{16}$–$5FE0_{16}$ | *Spill* / *fill* traps |
| | $0_{16}$– $7F_{16}$ | $100_{16}$–$17F_{16}$ | $6000_{16}$–$6FE0_{16}$ | Software traps to Privileged level |
| | — | $180_{16}$–$1FF_{16}$ | $7000_{16}$–$7FE0_{16}$ | *unassigned* |

**FIGURE 12-4** Privileged-mode Trap Table Layout

The trap table for TL = 0 comprises 512 thirty-two-byte entries; the trap table for TL > 0 comprises 512 more thirty-two-byte entries. Therefore, the total size of a full privileged trap table is $2 \times 512 \times 32$ bytes (32 Kbytes). However, if privileged software does not use software traps (Tcc instructions) at TL > 0, the table can be made 24 Kbytes long.

## 12.5.3 Trap-Table Entry Address to Hyperprivileged Mode

Hyperprivileged software initializes bits 63:14 of the Hyperprivileged Trap Base Address (HTBA) register (its most significant 50 bits) with bits 63:14 of the desired 64-bit hyperprivileged trap table base address.

At the time a trap to hyperprivileged mode is taken:

- Bits 63:14 of the trap vector address are taken from HTBA{63:14}.
- Bits 13:5 of the trap vector address contain a copy of the contents of the TT register (TT[TL]).
- Bits 4:0 of the trap vector address 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 12-5 illustrates the trap vector address used for a trap delivered to hyperprivileged mode.

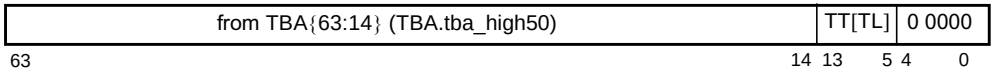



**FIGURE 12-5** Hyperprivileged Mode Trap Vector Address

## 12.5.4 Hyperprivileged Trap Table Organization

The layout of the hyperprivileged-mode trap table (which is accessed using physical addresses) is illustrated in FIGURE 12-6.

| Software Trap Type | Hardware Trap Type (TT[TL]) | Trap Table Offset (from HTBA) | Contents of Trap Table |
|---|---|---|---|
| — | $000_{16}$–$07F_{16}$ | $0_{16}$– $FE0_{16}$ | Hardware traps |
| — | $080_{16}$–$0FF_{16}$ | $1000_{16}$–$1FE0_{16}$ | *Spill* / *fill* traps |
| $0_{16}$– $7F_{16}$ | $100_{16}$–$17F_{16}$ | $2000_{16}$–$2FE0_{16}$ | Software traps from hyperprivileged level to hyperprivileged level |
| $80_{16}$– $FF_{16}$ | $180_{16}$–$1FF_{16}$ | $3000_{16}$–$3FE0_{16}$ | Software traps to hyperprivileged level |

**FIGURE 12-6** Hyperprivileged-mode Trap Table Layout

The hyperprivileged trap table comprises 512 thirty-two-byte entries. Therefore, the total size of a full hyperprivileged trap table is $512 \times 32$ bytes (16 Kbytes).

## 12.5.5 Trap Table Entry Address to `RED_state`

Traps occurring in `RED_state` or traps that cause the virtual processor to enter `RED_state` use an abbreviated trap vector, called the `RED_state` trap vector.

The `RED_state` trap vector is located at the following address, referred to as *RSTVADDR* (impl. dep. #114-V9-Cs10):

Physical Address — *RSTVADDR* = FFFF FFFF F000 $0000_{16}$
(highest 256 MB of physical address space)

In an implementation that implements fewer than 64 bits of physical addressing, unimplemented high-order bits of the above *RSTVADDR* are ignored.

FIGURE 12-7 illustrates the trap vector address used for a trap delivered to RED_state (in hyperprivileged mode).

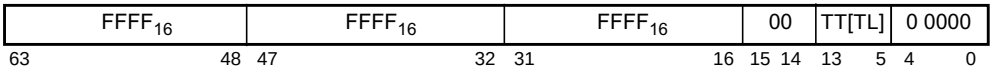



**FIGURE 12-7** RED_state Trap Vector Address

## 12.5.6 RED_state Trap Table Organization

The RED_state trap table is constructed so that it can overlay the hyperprivileged trap table (see FIGURE 12-6) if necessary. For a trap to RED_state, the trap table offset is added to the base address contained in *RSTVADDR* to yield the RED_state trap vector. FIGURE 12-8 illustrates the layout of the RED_state trap table.

| Hardware Trap Type (TT[TL]) | Trap Table Offset (from *RSTVADDR*) | Contents of Trap Table |
|---|---|---|
| 0 | $00_{16}$ | *Reserved* |
| 1 | $20_{16}$ | Power-on reset (POR) |
| TT† | $40_{16}$ | Watchdog reset (WDR) |
| 3 or TT‡ | $60_{16}$ | Externally initiated reset (XIR) |
| 4 | $80_{16}$ | Software-initiated reset (SIR) |
| TT* | $A0_{16}$ | All other exceptions in RED_state |

† TT = trap type of the exception that caused entry into error_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 = trap type of the exception. See TABLE 12-4 on page 485.

**FIGURE 12-8** RED_state Trap Table Layout

## 12.5.7 Trap Type (TT)

When a normal trap occurs, a value that uniquely identifies the type of 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 one of the following, depending on the trap's destination privilege mode:

- The TBA register, (TL > 0), and TT[TL] (see *Trap-Table Entry Address to Privileged Mode* on page 479)
- The HTBA register and TT[TL] (see *Trap-Table Entry Address to Hyperprivileged Mode* on page 480)

**Programming Note** The *spill_n_*, *fill_n_*, *clean_window*, and MMU-related traps (*fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection*) are spaced such that their trap-table entries are 128 bytes (32 instructions) long in the UltraSPARC Architecture. 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 467. Control is then transferred into the RED_state trap table at an address formed by *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 (caused by execution of a Tcc instruction) to privileged-mode trap handlers. TT values $180_{16}$–$1FF_{16}$ are used for software traps to trap handlers operating in hyperprivileged mode.

**IMPL. DEP. #35-V8-Cs20**: TT values $060_{16}$ to $07F_{16}$ were reserved for *implementation_dependent_exception_n* exceptions in the SPARC V9 specification, but are now all defined as standard UltraSPARC Architecture exceptions. See TABLE 12-4 for details.

The assignment of TT values to traps is shown in TABLE 12-4; TABLE 12-5 provides the same list, but sorted in order of trap priority. The key to both tables follows:

| Symbol | Meaning |
|---|---|
| ● | This trap type is associated with a feature that is architecturally required in an implementation of UltraSPARC Architecture 2007. Hardware must detect this exception or interrupt, trap on it (if not masked), and set the specified trap type value in the TT register. |
| ❍ | This trap type is associated with a feature that is architecturally defined in UltraSPARC Architecture 2007, but its implementation is optional. |
| P | Trap is taken via the Privileged trap table, in Privileged mode (PSTATE.priv = 1) |
| H | Trap is taken via the Hyperprivileged trap table, in Hyperprivileged mode (HSTATE.hpriv = 1) |
| $H^U$ | Trap is taken via the Hyperprivileged trap table, in Hyperprivileged mode (HSTATE.hpriv = 1). However, the trap is *unexpected*. While hardware can legitimately generate this trap, it should not do so unless there is a programming error or some other error. Therefore, occurrence of this trap indicates an actual error to hyperprivileged software. |
| -x- | Not possible. Hardware cannot generate this trap in the indicated running mode. For example, all privileged instructions can be executed in both privileged and hyperprivileged modes, therefore a *privileged_opcode* trap cannot occur in privileged or hyperprivileged mode. |
| — | This trap is reserved for future use. |
| (am) | Always Masked — when the condition occurs in this privilege mode, it is always masked out (but remains pending). |
| (ie) | When the outstanding disrupting trap condition occurs in this privilege mode, it may be conditioned (masked out) by PSTATE.ie = 0 (but remains pending). |
| (nm) | Never Masked — when the condition occurs in this running mode, it is never masked out and the trap is always taken. |
| (pend) | Held Pending — the condition can occur in this running mode, but can't be serviced in this mode. Therefore, it is held pending until the mode changes to one in which the exception *can* be serviced. |

**TABLE 12-4** Exception and Interrupt Requests, by TT Value *(1 of 6)*

| UA-2007 ●=Req'd. ❍=Opt'l | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| — | *Reserved* | $000_{16}$ | — | — | — | — | — |
| ● | *power_on_reset* | $001_{16}$ | reset | 0 | H (nm) | H (nm) | H (nm) |
| ● | *watchdog_reset* | TT♠ | reset | 1.2 | H (nm) | H (nm) | H (nm) |
| ● | *externally_initiated_reset* | $003_{16}$ | reset | 1.1 | H (nm) | H (nm) | H (nm) |
| ● | *software_initiated_reset* | $004_{16}$ | reset | 1.3 | -x- | -x- | H (nm) |
| — | *Reserved* | $005_{16}$ | — | — | — | — | — |
| ● | *RED_state_exception* | TT♣ | precise | ♣ | H (nm) | H (nm) | H (nm) |
| — | *implementation-dependent* | $006_{16}$ | — | — | — | — | — |
| ❍ | *store_error* | $007_{16}$ | deferred | 2.01 | H (nm) | H (nm) | H (nm) |
| ● | *IAE_privilege_violation* | $008_{16}$ | precise | 3.1 | H (nm) | -x- | -x- |
| ● | *instruction_access_MMU_miss*† | $009_{16}$ | precise | 2.08 | H (nm) | H (nm) | -x- |
| ● | *instruction_access_error* | $00A_{16}$ | precise | 4 | H (nm) | H (nm) | H (nm) |
| ● | *IAE_unauth_access* | $00B_{16}$ | precise | 3.2 | H (nm) | H (nm) | -x- |
| ● | *IAE_nfo_page* | $00C_{16}$ | precise | 3.3 | H (nm) | H (nm) | -x- |
| ❑ | *instruction_address_range* | $00D_{16}$ | precise | 2.06 | H (nm) | H (nm) | $H^U$ (nm) |
| ❑ | *instruction_real_range* | $00E_{16}$ | precise | 2.06 | H (nm) | H (nm) | $H^U$ (nm) |
| — | *Reserved* | $00F_{16}$ | — | — | — | — | — |
| ● | *illegal_instruction* | $010_{16}$ | precise | 6.2 | H (nm) | H (nm) | H (nm) |

**TABLE 12-4** Exception and Interrupt Requests, by TT Value *(2 of 6)*

| UA-2007 ●=Req'd. ❍=Opt'l | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ● | *privileged_opcode* | $011_{16}$ | precise | 7 | P (nm) | -x- | -x- |
| ❍ | *unimplemented_LDTW* | $012_{16}$ | precise | 6.3 | H (nm) | H (nm) | $H^U$ (nm) |
| ❍ | *unimplemented_STTW* | $013_{16}$ | precise | 6.3 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *DAE_invalid_asi* | $014_{16}$ | precise | 12.01 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *DAE_privilege_violation* | $015_{16}$ | precise | 12.04 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *DAE_nc_page* | $016_{16}$ | precise | 12.05 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *DAE_nfo_page* | $017_{16}$ | precise | 12.06 | H (nm) | H (nm) | $H^U$ (nm) |
| — | *Reserved* | $018_{16}$–$01F_{16}$ | — | — | — | — | — |
| ● | *fp_disabled* | $020_{16}$ | precise | 8 | P (nm) | P (nm) | $H^U$ (nm) |
| ❍ | *fp_exception_ieee_754* | $021_{16}$ | precise | 11.1 | P (nm) | P (nm) | $H^U$ (nm) |
| ❍ | *fp_exception_other* | $022_{16}$ | precise | 11.1 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *tag_overflow*$^D$ | $023_{16}$ | precise | 14 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *clean_window* | $024_{16}^{\ddagger}$ | precise | 10.1 | P (nm) | P (nm) | $H^U$ (nm) |
| — | *Reserved* | $025_{16}$–$027_{16}$ | — | — | — | — | — |
| ● | *division_by_zero* | $028_{16}$ | precise | 15 | P (nm) | P (nm) | $H^U$ (nm) |
| ❍ | *internal_processor_error* | $029_{16}$ | precise | ♦ | H (nm) | H (nm) | H (nm) |
| ❍ | *instruction_invalid_TSB_entry* | $02A_{16}$ | precise | 2.10 | H (nm) | H (nm) | -x- |

**TABLE 12-4** Exception and Interrupt Requests, by TT Value *(3 of 6)*

| UA-2007 ●=Req'd. ❍=Opt'l | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ❍ | *data_invalid_TSB_entry* | $02B_{16}$ | precise | 12.03 | H (nm) | H (nm) | H (nm) |
| — | *Reserved* | $02C_{16}$ | — | — | — | — | — |
| — | *implementation-dependent* | $02D_{16}$–$02F_{16}$ | — | — | — | — | — |
| ❑ | *mem_real_range* | $02D_{16}$ | precise | 11.3 | H (nm) | H (nm) | $H^U$ (nm) |
| ❑ | *mem_address_range* | $02E_{16}$ | precise | 11.3 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *DAE_side_effect_page* | $030_{16}$ | precise | 12.06 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *data_access_MMU_miss*† | $031_{16}$ | precise | 12.03 | H (nm) | H (nm) | H (nm) |
| ❍ | *data_access_error* | $032_{16}$ | precise | 12.10 | H (nm) | H (nm) | H (nm) |
| — | *data_access_protection* (no longer in use) | $033_{16}$ | precise | 12.07 | H (nm) | H (nm) | H (nm) |
| ● | *mem_address_not_aligned* | $034_{16}$ | precise | 10.2 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *LDDF_mem_address_not_aligned* | $035_{16}$ | precise | 10.1 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *STDF_mem_address_not_aligned* | $036_{16}$ | precise | 10.1 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *privileged_action* | $037_{16}$ | precise | 11.1 | H (nm) | H (nm) | -x- |
| ❍ | *LDQF_mem_address_not_aligned* | $038_{16}$ | precise | 10.1 | H (nm) | H (nm) | $H^U$ (nm) |
| ❍ | *STQF_mem_address_not_aligned* | $039_{16}$ | precise | 10.1 | H (nm) | H (nm) | $H^U$ (nm) |
| — | *Reserved* | $03A_{16}$ | — | — | — | — | — |
| — | *Reserved* | $03B_{16}$ | — | — | — | — | — |
| — | *implementation-dependent* | $03B_{16}$–$03D_{16}$ | — | — | — | — | — |

**TABLE 12-4** Exception and Interrupt Requests, by TT Value *(4 of 6)*

| UA-2007 ●=Req'd. ❍=Opt'l | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ● | *instruction_real_translation_miss* | $03E_{16}$ | precise | 2.08 | H (nm) | H (nm) | -x- |
| ● | *data_real_translation_miss* | $03F_{16}$ | precise | 12.03 | H (nm) | H (nm) | H (nm) |
| ❍ | *sw_recoverable_error* | $040_{16}$ | disrupting | 33.1 | H (nm) | H (nm) | H (ie) |
| ● | *interrupt_level_n* ($n$ = 1–15) | $041_{16}$ – $04F_{16}$ | disrupting | 32-$n$ (31 to 17) | P (ie) | P (ie) | (pend) |
| ❍ | *pic_overflow* (shares trap type $04F_{16}$ with *interrupt_level_15*) | $04F_{16}$ | disrupting | 16.00 | P (ie) | P (ie) | (pend) |
| — | *Reserved* | $050_{16}$– $05D_{16}$ | — | — | — | — | — |
| ● | *hstick_match* | $05E_{16}$ | disrupting | 16.01 | H (nm) | H (nm) | H (ie) |
| ● | *trap_level_zero* | $05F_{16}$ | precise | 2.02 | H | H | -x- |
| ❍ | *PA_watchpoint (RA_watchpoint)* | $061_{16}$ | precise | 12.09 | H (nm) | H (nm) | H (nm) |
| ❍ | *VA_watchpoint* | $062_{16}$ | precise | 11.2 | P (nm) | P (nm) | -x- |
| ❍ | *hw_corrected_error* | $063_{16}$ | disrupting | 33.2 | H (nm) | H (nm) | H (ie) |
| ● | *fast_instruction_access_MMU_miss* | $064_{16}$‡ | precise | 2.08 | H (nm) | H (nm) | -x- |
| — | *Reserved* | $065_{16}$– $067_{16}$ | — | — | — | — | — |
| ● | *fast_data_access_MMU_miss* | $068_{16}$‡ | precise | 12.03 | H (nm) | H (nm) | H (nm) |
| — | *Reserved* | $069_{16}$– $06B_{16}$ | — | — | — | — | — |
| ● | *fast_data_access_protection* | $06C_{16}$‡ | precise | 12.07 | H (nm) | H (nm) | H (nm) |

**TABLE 12-4** Exception and Interrupt Requests, by TT Value *(5 of 6)*

| UA-2007 ●=Req'd. ❍=Opt'l | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| — | *Reserved* | $06D_{16}$–$06F_{16}$ | — | — | — | — | — |
| ❍ | *implementation_dependent_exception_n* (impl. dep. #35-V8-Cs20) | $070_{16}$ | — | ∇ | — | — | — |
| ● | *instruction_access_MMU_error* | $071_{16}$ | precise | 2.07 | H (nm) | H (nm) | -x- |
| ● | *data_access_MMU_error* | $072_{16}$ | precise | 12.02 | H (nm) | H (nm) | $H^U$ (nm) |
| ❍ | *implementation_dependent_exception_n* (impl. dep. #35-V8-Cs20) | $073_{16}$ | — | ∇ | — | — | — |
| ● | *control_transfer_instruction* | $074_{16}$ | precise | 11.1 | P | P | $H^U$ |
| ❍ | *instruction_VA_watchpoint* | $075_{16}$ | precise | 2.05 | P (nm) | P (nm) | -x- |
| ● | *instruction_breakpoint* | $076_{16}$ | precise | 6.1 | H | H | H |
| ❑ | *implementation_dependent_exception_n* (impl. dep. #35-V8-Cs20) | $077_{16}$ – $078_{16}$ | — | ∇ | — | — | — |
| — | *Reserved* | $079_{16}$ | — | — | — | — | — |
| ● | *cpu_mondo* | $07C_{16}$ | disrupting | 16.08 | P (ie) | P (ie) | (pend) |
| ● | *dev_mondo* | $07D_{16}$ | disrupting | 16.11 | P (ie) | P (ie) | (pend) |
| ● | *resumable_error* | $07E_{16}$ | disrupting | 33.3 | P (ie) | P (ie) | (pend) |
| — | *nonresumable_error* (generated by hyperprivileged software, not by hardware) | $07F_{16}$ | — | — | — | — | — |
| ● | *spill_n_normal* ($n$ = 0–7) | $080_{16}$‡–$09C_{16}$‡ | precise | 9 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | (reserved for use by *spill_7_normal*; see footnote for trap type $09C_{16}$) | $09D_{16}$–$09F_{16}$ | — | — | — | — | — |
| ● | *spill_n_other* ($n$ = 0–7) | $0A0_{16}$‡–$0BC_{16}$‡ | precise | 9 | P (nm) | P (nm) | $H^U$ (nm) |

**TABLE 12-4** Exception and Interrupt Requests, by TT Value *(6 of 6)*

| UA-2007 ●=Req'd. ❍=Opt'l | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ● | (reserved for use by *spill_7_other* see footnote for trap type $0BC_{16}$) | $0BD_{16}$–$0BF_{16}$ | — | — | — | — | — |
| ● | *fill_n_normal* ($n$ = 0–7) | $0C0_{16}$‡–$0DC_{16}$‡ | precise | 9 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | (reserved for use by *fill_7_normal*; see footnote for trap type $0DC_{16}$) | $0DD_{16}$–$0DF_{16}$ | — | — | — | — | — |
| ● | *fill_n_other* ($n$ = 0–7) | $0E0_{16}$‡–$0FC_{16}$‡ | precise | 9 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | (reserved for use by *fill_7_other* see footnote for trap type $0FC_{16}$) | $0FD_{16}$–$0FF_{16}$ | — | — | — | — | — |
| ● | *trap_instruction* | $100_{16}$–$17F_{16}$ | precise | 16.02 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *htrap_instruction* | $180_{16}$–$1FF_{16}$ | precise | 16.02 | -x- | H (nm) | $H^U$ (nm) |
| ● | *guest_watchdog* ◊ | TT◊ | precise or disrupting◊ | ◊ | H (nm) | H (nm) | -x- |

* Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36-V8 on page 496), including relative priorities within a given priority level.

† This exception type is only used in UltraSPARC Architecture 2007 implementations that support hardware MMU table walking. See description of this exception in *Exception and Interrupt Descriptions* on page 508.

‡ 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.

◊ The *guest_watchdog* trap is caused when TL ≥ *MAXPTL* and any precise or disrupting trap occurs that is destined for privileged mode. *guest_watchdog* shares a trap table offset with *watchdog_reset* ($40_{16}$), but retains the trap type (TT) value and priority of the exception that caused the trap.

♠ *watchdog_reset* uses the trap vector entry for trap type $002_{16}$ (trap table offset $40_{16}$), but retains the trap type (TT) value of the exception that caused entry into error_state .

♣ *RED_state_exception* uses the trap vector entry for trap type $005_{16}$ (trap table offset $A0_{16}$), but retains the trap type (TT) value and priority of the exception that caused the trap.

♦ The priority of *internal_processor_error* is implementation dependent (impl. dep. # 402-S10)

D This exception is deprecated, because the only instructions that can generate it have been deprecated.

**TABLE 12-5** Exception and Interrupt Requests, by Priority *(1 of 5)*

| UA-2007 ●=Req'd. ❍=Opt'l ❑.=Impl-Specific | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered and (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ● | *power_on_reset* | $001_{16}$ | reset | 0 | H (nm) | H (nm) | H (nm) |
| ● | *externally_initiated_reset* | $003_{16}$ | reset | 1.1 | H (nm) | H (nm) | H (nm) |
| ● | *watchdog_reset* | TT♠ | reset | 1.2 | H (nm) | H (nm) | H (nm) |
| ● | *software_initiated_reset* | $004_{16}$ | reset | 1.3 | -x- | -x- | H (nm) |
| ❍ | *store_error* | $007_{16}$ | deferred | 2.01 | H (nm) | H (nm) | H (nm) |
| ● | *trap_level_zero* | $05F_{16}$ | precise | 2.02 | H | H | -x- |
| ❍ | *instruction_VA_watchpoint* | $075_{16}$ | precise | 2.05 | P (nm) | P (nm) | -x- |
| ❑ | *instruction_address_range* | $00D_{16}$ | precise | 2.06 | H (nm) | H (nm) | $H^U$ (nm) |
| ❑ | *instruction_real_range* | $00E_{16}$ | precise | | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *instruction_access_MMU_error* | $071_{16}$ | precise | 2.07 | H (nm) | H (nm) | -x- |
| ● | *instruction_real_translation_miss* | $03E_{16}$ | precise | 2.08 | H (nm) | H (nm) | -x- |
| ● | *instruction_access_MMU_miss*† | $009_{16}$ | precise | | H (nm) | H (nm) | -x- |
| ● | *fast_instruction_access_MMU_miss* | $064_{16}$‡ | precise | | H (nm) | H (nm) | -x- |
| ❍ | *instruction_invalid_TSB_entry* | $02A_{16}$ | precise | 2.10 | H (nm) | H (nm) | -x- |
| ● | *IAE_privilege_violation* | $008_{16}$ | precise | 3.1 | H (nm) | -x- | -x- |
| ● | *IAE_unauth_access* | $00B_{16}$ | precise | 3.2 | H (nm) | H (nm) | -x- |
| ● | *IAE_nfo_page* | $00C_{16}$ | precise | 3.3 | H (nm) | H (nm) | -x- |

**TABLE 12-5** Exception and Interrupt Requests, by Priority *(2 of 5)*

| UA-2007 ●=Req'd. ❍=Opt'l ❑.=Impl-Specific | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered and (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ● | *instruction_access_error* | $00A_{16}$ | precise | 4 | H (nm) | H (nm) | H (nm) |
| ● | *instruction_breakpoint* | $076_{16}$ | precise | 6.1 | H | H | H |
| ● | *illegal_instruction* | $010_{16}$ | precise | 6.2 | H (nm) | H (nm) | H (nm) |
| ❍ | *unimplemented_LDTW* | $012_{16}$ | precise | 6.3 | H (nm) | H (nm) | $H^U$ (nm) |
| ❍ | *unimplemented_STTW* | $013_{16}$ | precise | | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *privileged_opcode* | $011_{16}$ | precise | 7 | P (nm) | -x- | -x- |
| ● | *fp_disabled* | $020_{16}$ | precise | 8 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *spill_n_normal* ($n$ = 0–7) | $080_{16}$‡–$09C_{16}$‡ | precise | 9 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *spill_n_other* ($n$ = 0–7) | $0A0_{16}$‡–$0BC_{16}$‡ | precise | | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *fill_n_normal* ($n$ = 0–7) | $0C0_{16}$‡–$0DC_{16}$‡ | precise | | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *fill_n_other* ($n$ = 0–7) | $0E0_{16}$‡–$0FC_{16}$‡ | precise | | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *clean_window* | $024_{16}$‡ | precise | 10.1 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *LDDF_mem_address_not_aligned* | $035_{16}$ | precise | | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *STDF_mem_address_not_aligned* | $036_{16}$ | precise | | H (nm) | H (nm) | $H^U$ (nm) |
| ❍ | *LDQF_mem_address_not_aligned* | $038_{16}$ | precise | | H (nm) | H (nm) | $H^U$ (nm) |
| ❍ | *STQF_mem_address_not_aligned* | $039_{16}$ | precise | | H (nm) | H (nm) | $H^U$ (nm) |

**TABLE 12-5** Exception and Interrupt Requests, by Priority *(3 of 5)*

| UA-2007 ●=Req'd. ❍=Opt'l ❑.=Impl-Specific | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered and (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ● | *mem_address_not_aligned* | $034_{16}$ | precise | 10.2 | H (nm) | H (nm) | $H^U$ (nm) |
| ❍ | *fp_exception_other* | $022_{16}$ | precise | 11.1 | P (nm) | P (nm) | $H^U$ (nm) |
| ❍ | *fp_exception_ieee_754* | $021_{16}$ | precise | | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *privileged_action* | $037_{16}$ | precise | | H (nm) | H (nm) | -x- |
| ● | *control_transfer_instruction* | $074_{16}$ | precise | | P | H | $H^U$ |
| ❍ | *VA_watchpoint* | $062_{16}$ | precise | 11.2 | P (nm) | P (nm) | -x- |
| ❑ | *mem_real_range* | $02D_{16}$ | precise | 11.3 | H (nm) | H (nm) | $H^U$ (nm) |
| ❑ | *mem_address_range* | $02E_{16}$ | precise | | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *DAE_invalid_asi* | $014_{16}$ | precise | 12.01 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *data_access_MMU_error* | $072_{16}$ | precise | 12.02 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *data_real_translation_miss* | $03F_{16}$ | precise | 12.03 | H (nm) | H (nm) | H (nm) |
| ● | *data_access_MMU_miss*† | $031_{16}$ | precise | | H (nm) | H (nm) | H (nm) |
| ● | *fast_data_access_MMU_miss* | $068_{16}$‡ | precise | | H (nm) | H (nm) | H (nm) |
| ❍ | *data_invalid_TSB_entry* | $02B_{16}$ | precise | | H (nm) | H (nm) | H (nm) |
| ● | *DAE_privilege_violation* | $015_{16}$ | precise | 12.04 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *DAE_nc_page* | $016_{16}$ | precise | 12.05 | H (nm) | H (nm) | $H^U$ (nm) |

**TABLE 12-5** Exception and Interrupt Requests, by Priority *(4 of 5)*

| UA-2007 ●=Req'd. ❍=Opt'l ❑.=Impl-Specific | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered and (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ● | *DAE_nfo_page* | $017_{16}$ | precise | 12.06 | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *DAE_side_effect_page* | $030_{16}$ | precise | | H (nm) | H (nm) | $H^U$ (nm) |
| ● | *fast_data_access_protection* | $06C_{16}$‡ | precise | 12.07 | H (nm) | H (nm) | H (nm) |
| — | *data_access_protection* | $033_{16}$ | precise | | H (nm) | H (nm) | H (nm) |
| ❍ | *PA_watchpoint (RA_watchpoint)* | $061_{16}$ | precise | 12.09 | H (nm) | H (nm) | H (nm) |
| ❍ | *data_access_error* | $032_{16}$ | precise | 12.10 | H (nm) | H (nm) | H (nm) |
| ● | *tag_overflow*$^D$ | $023_{16}$ | precise | 14 | P (nm) | P (nm) | $H^U$ (nm) |
| ● | *division_by_zero* | $028_{16}$ | precise | 15 | P (nm) | P (nm) | $H^U$ (nm) |
| ❍ | *pic_overflow* | $04F_{16}$ | disrupting | 16.00 | P (ie) | P (ie) | (pend) |
| ● | *hstick_match* | $05E_{16}$ | disrupting | 16.01 | H (nm) | H (nm) | H (ie) |
| ● | *trap_instruction* | $100_{16}$–$17F_{16}$ | precise | 16.02 | P (nm) | P (nm) | H (nm) |
| ● | *htrap_instruction* | $180_{16}$–$1FF_{16}$ | precise | | -x- | H (nm) | $H^U$ (nm) |
| ● | *cpu_mondo* | $07C_{16}$ | disrupting | 16.08 | P (ie) | P (ie) | (pend) |
| ● | *dev_mondo* | $07D_{16}$ | disrupting | 16.11 | P (ie) | P (ie) | (pend) |
| ● | *interrupt_level_n* ($n$ = 1–15) | $041_{16}$–$04F_{16}$ | disrupting | 32-$n$ (31 to 17) | P (ie) | P (ie) | (pend) |

**TABLE 12-5** Exception and Interrupt Requests, by Priority *(5 of 5)*

| UA-2007 ●=Req'd. ❍=Opt'l ❑.=Impl-Specific | Exception or Interrupt Request | TT (Trap Type) | Trap Category | Priority (0 = High-est) | Mode in which Trap is Delivered and (and Conditioning Applied), based on Current Privilege Mode | | |
|---|---|---|---|---|---|---|---|
| | | | | | NP | Priv | HP |
| ❍ | *sw_recoverable_error* | $040_{16}$ | disrupting | 33.1 | H (nm) | H (nm) | H (ie) |
| ❍ | *hw_corrected_error* | $063_{16}$ | disrupting | 33.2 | H (nm) | H (nm) | H (ie) |
| ● | *resumable_error* | $07E_{16}$ | disrupting | 33.3 | P (ie) | P (ie) | (pend) |
| ● | *guest_watchdog* ◊ | TT◊ | precise or disrupting◊ | ◊ | H (nm) | H (nm) | -x- |
| ● | *RED_state_exception* | TT♣ | precise | ♣ | H (nm) | H (nm) | H (nm) |
| ❍ | *internal_processor_error* | $029_{16}$ | precise | ♦ | H (nm) | H (nm) | H (nm) |
| — | *nonresumable_error* (generated by hyperprivileged software, not by hardware) | $07F_{16}$ | — | — | — | — | — |

* Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36-V8 on page 496), including relative priorities within a given priority level.

† This exception type is only used in UltraSPARC Architecture 2007 implementations that support hardware MMU table walking. See description of this exception in *Exception and Interrupt Descriptions* on page 508.

‡ 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.

◊ The *guest_watchdog* trap is caused when TL ≥ *MAXPTL* and any precise or disrupting trap occurs that is destined for privileged mode. *guest_watchdog* shares a trap table offset with *watchdog_reset* ($40_{16}$), but retains the trap type (TT) value and priority of the exception that caused the trap.

♠ *watchdog_reset* uses the trap vector entry for trap type $002_{16}$ (trap table offset $40_{16}$), but retains the trap type (TT) value of the exception that caused entry into error_state .

♣ *RED_state_exception* uses the trap vector entry for trap type $005_{16}$ (trap table offset $A0_{16}$), but retains the trap type (TT) value and priority of the exception that caused the trap.

♦ The priority of *internal_processor_error* is implementation dependent (impl. dep. # 402-S10)

D This exception is deprecated, because the only instructions that can generate it have been deprecated.

### 12.5.7.1 Trap Type for Spi ll/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 12-9.

| 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 |

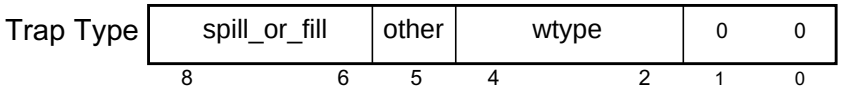



**FIGURE 12-9** Trap Type Encoding for Spill/Fill Traps

## 12.5.8 Trap Priorities

TABLE 12-4 on page 485 and TABLE 12-5 on page 491 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 $y$. Traps within the same priority class (0 to 33) are listed in priority order in TABLE 12-5 (impl. dep. #36-V8).

**IMPL. DEP. #36-V8:** The relative priorities of traps defined in the UltraSPARC Architecture 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 new traps are implementation dependent.

However, the TT values for the exceptions and interrupt requests shown in TABLE 12-4 and TABLE 12-5 must remain the same for every implementation.

The trap priorities given above always need to be considered within the context 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 detects *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.

# 12.6 Trap Processing

The virtual processor's action during trap processing depends on various virtual processor states, including the trap type, the current level of trap nesting (given in the TL register), HPSTATE, and PSTATE. When a trap occurs, the GL register is normally incremented by one (described later in this section), which replaces the set of eight global registers with the next consecutive set.

The following traps are processed in RED_state:

- POR and WDR reset requests
- SIR and XIR reset requests when TL < *MAXTL*
- Non-reset traps taken when TL = *MAXTL* – 1
- Traps taken when the virtual processor is in RED_state

All other traps are handled in execute_state using normal trap processing.

During normal operation, the virtual processor is in execute_state. It processes traps in execute_state and continues.

When a nonreset trap, externally initiated reset (XIR), or software-initiated reset (SIR) occurs with TL = *MAXTL*, there are no more levels on the trap stack, so the virtual processor enters the transitory state error_state. The virtual processor remains in error_state for an implementation-dependent duration, then generates a WDR reset (impl. dep. #254-U3-Cs10) to effect a change from error_state to RED_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 Organization* on page 482.

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 501. Finally, software can force the processor into RED_state by setting the HPSTATE.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, or XIR trap is taken with TL = *MAXTL*, which puts the virtual processor in error_state.

TABLE 12-6, TABLE 12-7, and TABLE 12-8 describe the virtual processor mode and trap-level transitions involved in handling traps.

**TABLE 12-6** Trap Received While in `execute_state`

| | New State, After Receiving Trap Type | | | | |
|---|---|---|---|---|---|
| **Original State** | **Nonreset Trap or Interrupt** | **POR** | **XIR** | **WDR ‡** | **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* | `error_state`<br>TL = *MAXTL* | ‡ | `error_state`<br>TL = *MAXTL* |

† This state occurs when software changes TL to *MAXTL* and leaves HPSTATE.red = 0, or if software sets HPSTATE.red ← 0 while TL = *MAXTL*.

‡ WDR can only be generated from `error_state`.

**TABLE 12-7** Trap Received While in `RED_state`

| | New State, After Receiving Trap Type | | | | |
|---|---|---|---|---|---|
| **Original State** | **Nonreset Trap or Interrupt** | **POR** | **XIR** | **WDR ‡** | **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* | `error_state`<br>TL = *MAXTL* | ‡ | `error_state`<br>TL = *MAXTL* |

‡ WDR can only be generated from `error_state`.

**TABLE 12-8** Reset Received While in `error_state`

| | New State, After Receiving Trap Type | | | | |
|---|---|---|---|---|---|
| **Original State** | **Nonreset Trap or Interrupt** | **POR** | **XIR** | **WDR** | **SIR** |
| `error_state`<br>TL = *MAXTL* | — | `RED_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 non-reset 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 *Nonreset Traps When the Virtual Processor Is in RED_state* on page 507.
- 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 467 and *Nonreset Traps with TL = MAXTL – 1* on page 501.
- 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 470.

Otherwise, the trap uses normal trap processing, described in the following section on *Normal Trap Processing*.

## 12.6.1 Normal Trap Processing

Normal traps comprise all traps processed in execute_state; that is, all non-RED_state and non-error_state traps.

A trap is delivered in either privileged mode or hyperprivileged mode, depending on the type of trap, the trap level (TL), and the privilege mode in effect when the exception was detected.

During normal trap processing, the following state changes occur (conceptually, in this order):

- The trap level is updated. 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    // note that if TL = MAXTL – 1 before this trap,
                            // trap would have been processed in
                            // RED_state, not here using normal trap
                            // processing.
```

- Existing state is preserved.

```
TSTATE[TL].gl  ← GL
TSTATE[TL].ccr  ← CCR
TSTATE[TL].asi ← ASI
TSTATE[TL].pstate ← PSTATE
TSTATE[TL].cwp  ← CWP
TPC[TL]         ← PC  // (upper 32 bits zeroed if PSTATE.am = 1)
TNPC[TL]        ← NPC // (upper 32 bits zeroed if PSTATE.am = 1)
HTSTATE[TL].hpstate ← HPSTATE //even for traps to privileged mode
```

- The trap type is preserved.

```
TT[TL]          ← the trap type
```

- The Global Level register (GL) is updated. This normally provides access to a fresh set of global registers:

```
if (the trap is being delivered in privileged mode)
then GL          ← min (GL + 1, MAXPGL)
else (trap is being delivered in hyperprivileged mode)
     GL          ← min (GL + 1, MAXGL)
endif
```

- The PSTATE register is updated to a predefined state (even for traps to hyperprivileged mode):

```
PSTATE.mm    is unchanged
PSTATE.pef   ← 1 // if an FPU is present, it is enabled
PSTATE.am    ← 0 // address masking is turned off
if (the trap is being delivered in privileged mode)
then PSTATE.priv ← 1 // the virtual processor enters privileged mode
     PSTATE.cle  ← PSTATE.tle //set endian mode for traps
else // trap is being delivered in hyperprivileged mode
     PSTATE.priv ← 0
     PSTATE.cle  ← 0
endif
PSTATE.ie    ← 0 // interrupts are disabled
PSTATE.tle   is unchanged
PSTATE.tct   ← 0 // trap on CTI disabled
```

- The HPSTATE register is updated:

```
if (the trap is to hyperprivileged mode)
then  HPSTATE.red ← 0
      HPSTATE.hpriv ← 1 // enter hyperprivileged mode
      HPSTATE.ibe   ← 0 disable instruction breakpoints
      HPSTATE.tlz is unchanged
endif
```

- 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] = 024₁₆ // a clean_window trap
then CWP  ← CWP + 1
endif

if (080₁₆ ≤ TT[TL] ≤ 0BF₁₆) // window spill trap
then CWP  ← CWP + CANSAVE + 2
endif

if (0C0₁₆ ≤ TT[TL] ≤ 0FF₁₆) // window fill trap
then CWP  ← CWP – 1
endif
```

For non-register-window traps, CWP is not changed.

- Control is transferred into the trap table:

```
// Note that at this point, TL has already been incremented (above)
if ( (trap is to privileged mode) and (TL ≤ MAXPTL) )
then
      //the trap is handled in privileged mode
      //Note: The expression "(TL > 1)" below evaluates to the
      //value 0₂ if TL was 0 just before the trap (in which
      //case, TL = 1 now, since it was incremented above,
      //during trap entry).  "(TL > 1)" evaluates to 1₂ if
      //TL was > 0 before the trap.
        PC ← TBA{63:15} :: (TL > 1) :: TT[TL] :: 0 0000₂
        NPC ← TBA{63:15} :: (TL > 1) :: TT[TL] :: 0 0100₂
else if ( (trap is to privileged mode) and (TL > MAXPTL) )
then    // this is the guest_watchdog case; the trap is handled in
        // hyperprivileged mode using trap table offset 40₁₆.
        PC ←  HTBA{63:14} :: 00₂ :: 040₁₆
        NPC ← HTBA{63:14} :: 00₂ :: 044₁₆
else    { trap is handled in hyperprivileged mode }
        PC ← HTBA{63:14} :: TT[TL] :: 0 0000₂
        NPC ← HTBA{63:14} :: TT[TL] :: 0 0100₂
endif
```

Interrupts are ignored as long as PSTATE.ie = 0.

**Programming Note** State in TPC[*n*], TNPC[*n*], TSTATE[*n*], and TT[*n*] is only changed autonomously by the 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*.

## 12.6.2 RED_state Trap Processing

The following conditions invoke RED_state trap processing, and cause the trap to be delivered in hyperprivileged mode:

- 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-V8:** Implementation-dependent registers may or may not be affected by the various reset traps.

### 12.6.2.1 Nonreset Traps with TL = *MAXTL* – 1

Nonreset traps that occur when TL = *MAXTL* – 1 are processed in RED_state.

The following state changes occur (conceptually, in this order) during a nonreset trap that occurs when TL = *MAXTL* – 1:

- The trap level is advanced.

  TL ← *MAXTL*

- Existing state is preserved.

  TSTATE[TL].gl ← GL
  TSTATE[TL].ccr ← CCR
  TSTATE[TL].asi ← ASI
  TSTATE[TL].pstate ← PSTATE
  TSTATE[TL].cwp ← CWP
  TPC[TL] ← PC // (upper 32 bits zeroed if PSTATE.am = 1)
  TNPC[TL] ← NPC // (upper 32 bits zeroed if PSTATE.am = 1)

  HTSTATE[TL].hpstate ← HPSTATE

- The trap type is preserved.

  TT[TL] ← the trap type

- The Global Level register is updated.

  GL ← **min** (GL + 1, *MAXGL*)

- The PSTATE register is set as follows:

  PSTATE.mm ← $00_2$ // TSO
  PSTATE.pef ← 1 // if an FPU is present, it is enabled
  PSTATE.am ← 0 // address masking is turned off
  PSTATE.priv ← 0 // entering hyperprivileged mode
  PSTATE.ie ← 0 // interrupts are disabled
  PSTATE.cle ← 0 // big-endian is default for hyperprivileged mode
  PSTATE.tle is unchanged // (*was unspecified in SPARC V9 specification)*
  PSTATE.tct ← 0 // trap on CTI disabled

- The HPSTATE register is updated:

  HPSTATE.red ← 1 // enter RED_state
  HPSTATE.hpriv ← 1 // enter hyperprivileged mode
  HPSTATE.ibe ← 0 // disable instruction breakpoints
  HPSTATE.tlz ← 0 // disable *trap_level_zero* exceptions

- 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
  endif

  If ($080_{16}$ ≤ TT[TL] ≤ $0BF_{16}$) // window *spill* trap
  then CWP ← CWP + CANSAVE + 2
  endif

```
If (0C0_16 ≤ TT[TL] ≤ 0FF_16)    // window fill trap
then CWP  ←  CWP – 1
endif
```

For non-register-window traps, CWP is not changed.

- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the RED_state trap table. See *Trap Table Entry Address to RED_state* on page 481 for further details of *RSTVADDR*.

```
PC     ← RSTVADDR{63:8} :: 1010 0000_2
NPC    ← RSTVADDR{63:8} :: 1010 0100_2
```

### 12.6.2.2 Power-On Reset (POR) Traps

A POR trap occurs 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 Chapter 16, *Resets* 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
```

- The Global Level register is updated.

```
GL     ← MAXGL
```

- The PSTATE register is set as follows:

```
PSTATE.mm     ←  00_2 // TSO
PSTATE.pef    ← 1 // if an FPU is present, it is enabled
PSTATE.am     ← 0 // address masking is turned off
PSTATE.priv   ← 0 // entering hyperprivileged mode
PSTATE.ie     ← 0 // interrupts are disabled
PSTATE.cle    ← 0 // big-endian is default for hyperprivileged mode
PSTATE.tle    ← 0 // big-endian mode for traps
PSTATE.tct    ← 0 // trap on CTI disabled
```

- The HPSTATE register is updated:

```
HPSTATE.red   ← 1 // enter RED_state
HPSTATE.hpriv ← 1 // enter hyperprivileged mode
HPSTATE.ibe   ← 0 // disable instruction breakpoints
HPSTATE.tlz   ← 0 // disable trap_level_zero exceptions
```

- The TICK register is protected.

```
TICK.npt     ← 1 // TICK is unreadable by nonprivileged software
```

- Implementation-specific state changes; for example, disabling an MMU.

- Control is transferred into the RED_state trap table.

  PC ← *RSTVADDR*{63:8} :: $0010\ 0000_2$
  NPC ← *RSTVADDR*{63:8} :: $0010\ 0100_2$

### 12.6.2.3 Watchdog Reset (WDR) Traps

Entry to error_state is caused by occurrence of a trap when TL = *MAXTL* (impl. dep. #39-V8-Cs10). See *error_state* on page 470.

To recover from error_state, the UltraSPARC Architecture provides *watchdog_reset* (WDR), which causes a transition from error_state to RED_state (impl. dep. #254-U3-Cs10).

The following virtual processor state changes occur during WDR (conceptually, in this order):

- The trap level is updated.

  TL ← min (TL + 1, *MAXTL*)

- Existing state is preserved.

  TSTATE[TL].gl ← GL
  TSTATE[TL].ccr ← CCR
  TSTATE[TL].asi ← ASI
  TSTATE[TL].pstate ← PSTATE
  TSTATE[TL].cwp ← CWP
  TPC[TL] ← PC // (upper 32 bits zeroed if PSTATE.am = 1)
  TNPC[TL] ← NPC // (upper 32 bits zeroed if PSTATE.am = 1)
  HTSTATE[TL].hpstate ← HPSTATE

- The trap type is set.

  TT[TL] ← the trap type that caused the WDR

- The Global Level register is updated.

  GL ← **min** (GL + 1, *MAXGL*)

- The PSTATE register is set as follows:

  PSTATE.mm ← $00_2$ // TSO
  PSTATE.pef ← 1 // if an FPU is present, it is enabled
  PSTATE.am ← 0 // address masking is turned off
  PSTATE.priv ← 0 // entering hyperprivileged mode
  PSTATE.ie ← 0 // interrupts are disabled
  PSTATE.cle ← 0 // big-endian is default for hyperprivileged mode
  PSTATE.tle is unchanged // (*was unspecified in SPARC V9 specification)*
  PSTATE.tct ← 0 // trap on CTI disabled

- The HPSTATE register is updated:

```
HPSTATE.red  ← 1 // enter RED_state
HPSTATE.hpriv ← 1 // enter hyperprivileged mode
HPSTATE.ibe  ← 0 // disable instruction breakpoints
HPSTATE.tlz  ← 0 // disable trap_level_zero exceptions
```

- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the RED_state trap table.

$$\text{PC} \leftarrow \mathit{RSTV_{ADDR}}\{63{:}8\} :: 0100\ 0000_2$$
$$\text{NPC} \leftarrow \mathit{RSTV_{ADDR}}\{63{:}8\} :: 0100\ 0100_2$$

### 12.6.2.4 Externally Initiated Reset (XIR) Traps

XIR traps are initiated by an external signal. They behave like an interrupt that cannot be masked by PSTATE.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 Chapter 16, *Resets* for further details.

If TL = *MAXTL*, then the virtual processor enters error_state.

The following virtual processor state changes occur during XIR (conceptually, in this order):

- The trap level is updated:

$$\text{TL} \leftarrow \min(\text{TL} + 1, \mathit{MAXTL})$$

- Existing state is preserved.

```
TSTATE[TL].gl  ← GL
TSTATE[TL].ccr  ← CCR
TSTATE[TL].asi  ← ASI
TSTATE[TL].pstate ← PSTATE
TSTATE[TL].cwp  ← CWP
TPC[TL]  ← PC  // (upper 32 bits zeroed if PSTATE.am = 1)
TNPC[TL]  ← NPC // (upper 32 bits zeroed if PSTATE.am = 1)
HTSTATE[TL].hpstate  ← HPSTATE
```

- The trap type is set.

$$\text{TT[TL]} \leftarrow 003_{16}$$

- The Global Level register is updated.

$$\text{GL} \leftarrow \mathbf{min}(\text{GL} + 1, \mathit{MAXGL})$$

- The PSTATE register is set as follows:

```
PSTATE.mm  ← 00₂ // TSO
PSTATE.pef  ← 1 // if an FPU is present, it is enabled
PSTATE.am  ← 0 // address masking is turned off
PSTATE.priv  ← 0 // entering hyperprivileged mode
PSTATE.ie  ← 0 // interrupts are disabled
```

PSTATE.cle ← 0 // big-endian is default for hyperprivileged mode
PSTATE.tle is unchanged // (*was unspecified in SPARC V9 specification)*
PSTATE.tct ← 0 // trap on CTI disabled

- The HPSTATE register is updated:

  HPSTATE.red ← 1 // enter RED_state
  HPSTATE.hpriv ← 1 // enter hyperprivileged mode
  HPSTATE.ibe ← 0 // disable instruction breakpoints
  HPSTATE.tlz ← 0 // disable *trap_level_zero* exceptions

- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the RED_state trap table.

  PC ← *RSTVADDR*{63:8} :: $0110\ 0000_2$
  NPC ← *RSTVADDR*{63:8} :: $0110\ 0100_2$

See *Externally Initiated Reset (XIR)* on page 629 and the documentation for specific processor implementations for more information.

### 12.6.2.5 Software-Initiated Reset (SIR) Traps

A software-initiated reset trap is initiated by execution of an SIR instruction in hyperprivileged mode. Hyperprivileged software uses the SIR trap as a panic operation or a metahypervisor trap. See Chapter 16, *Resets* for further details.

If TL = *MAXTL*, then the virtual processor enters error_state.

Otherwise, TL < *MAXTL* as trap processing begins and the following virtual processor state changes occur (conceptually, in this order):

- The trap level is updated.

  TL ← TL + 1

- Existing state is preserved.

  TSTATE[TL].gl ← GL
  TSTATE[TL].ccr ← CCR
  TSTATE[TL].asi ← ASI
  TSTATE[TL].pstate ← PSTATE
  TSTATE[TL].cwp ← CWP
  TPC[TL] ← PC // (upper 32 bits zeroed if PSTATE.am = 1)
  TNPC[TL] ← NPC // (upper 32 bits zeroed if PSTATE.am = 1)
  HTSTATE[TL].hpstate ← HPSTATE

- The trap type is set.

  TT[TL] ← $04_{16}$

- The Global Level register is updated.

  GL ← **min** (GL + 1, *MAXGL*)

- The PSTATE register is set as follows:

  PSTATE.mm ← $00_2$ // TSO
  PSTATE.pef ← 1 // if an FPU is present, it is enabled
  PSTATE.am ← 0 // address masking is turned off
  PSTATE.priv ← 0 // entering hyperprivileged mode
  PSTATE.ie ← 0 // interrupts are disabled
  PSTATE.cle ← 0 // big-endian is default for hyperprivileged mode
  PSTATE.tle is unchanged // (*was unspecified in SPARC V9 specification)*
  PSTATE.tct ← 0 // trap on CTI disabled

- The HPSTATE register is updated:

  HPSTATE.red ← 1 // enter RED_state
  HPSTATE.hpriv ← 1 // enter hyperprivileged mode
  HPSTATE.ibe ← 0 // disable instruction breakpoints
  HPSTATE.tlz ← 0 // disable *trap_level_zero* exceptions

- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the RED_state trap table.

  PC ← *RSTVADDR*{63:8} :: $1000\ 0000_2$
  NPC ← *RSTVADDR*{63:8} :: $1000\ 0100_2$

See *Software-Initiated Reset (SIR)* on page 630 and the documentation for specific processor implementations for more information.

### 12.6.2.6 Nonreset Traps When the Virtual Processor Is in RED_state

When a nonreset trap occurs while the virtual processor is in RED_state, if TL = *MAXTL*, then the virtual processor enters error_state.

Otherwise, TL < *MAXTL* as trap processing begins, the virtual processor remains in RED_state, and the following virtual processor state changes occur (conceptually, in this order):

- The trap level is updated.

  TL ← TL + 1

- Existing state is preserved.

  TSTATE[TL].gl ← GL
  TSTATE[TL].ccr ← CCR
  TSTATE[TL].ASI ← ASI
  TSTATE[TL].pstate ← PSTATE
  TSTATE[TL].cwp ← CWP
  TPC[TL] ← PC // (upper 32 bits zeroed if PSTATE.am = 1)
  TNPC[TL] ← NPC // (upper 32 bits zeroed if PSTATE.am = 1)
  HTSTATE[TL].hpstate ← HPSTATE

- The trap type is preserved.

  TT[TL] ← trap type

- The Global Level register is updated.

  GL ← **min** (GL + 1, *MAXGL*)

- The PSTATE register is set as follows:

  PSTATE.mm ← $00_2$ // TSO
  PSTATE.pef ← `1` // if an FPU is present, it is enabled
  PSTATE.am ← `0` // address masking is turned off
  PSTATE.priv ← `0` // entering hyperprivileged mode
  PSTATE.ie ← `0` // interrupts are disabled
  PSTATE.cle ← `0` // big-endian is default for hyperprivileged mode
  PSTATE.tle ← unchanged // (*was unspecified in SPARC V9 specification)*
  PSTATE.tct ← `0` // trap on CTI disabled

- The HPSTATE register is updated:

  HPSTATE.red ← `1` // enter `RED_state`
  HPSTATE.hpriv ← `1` // enter hyperprivileged mode
  HPSTATE.ibe ← `0` // disable instruction breakpoints
  HPSTATE.tlz is unchanged

- 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
  endif
  If ($080_{16}$ ≤ TT[TL] ≤ $0BF_{16}$) // window spill trap
  then CWP ← CWP + CANSAVE + 2
  endif
  If ($0C0_{16}$ ≤ TT[TL] ≤ $0FF_{16}$) // window fill trap
  then CWP ← CWP – 1
  endif

- For non-register-window traps, CWP is not changed.
- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the `RED_state` trap table.

  PC ← *RSTV*$_{ADDR}${63:8} :: $1010\ 0000_2$
  NPC ← *RSTV*$_{ADDR}${63:8} :: $1010\ 0100_2$

# 12.7 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.

All other trap types are reserved.

**Note** The encoding of trap types in the UltraSPARC Architecture 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. Chapter 7, *Instructions*, enumerates which traps can be generated by each instruction.

The following traps are generally expected to be supported in all UltraSPARC Architecture 2007 implementations. A given trap is not required to be supported in an implementation in which the conditions that cause the trap can never occur.

- ***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-V9:** 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.

- ***control_transfer_instruction*** [TT = $074_{16}$] (Precise) — This exception is generated if PSTATE.tct = 1 and the processor determines that a successful control transfer will occur as a result of execution of that instruction. If such a transfer will occur, the processor generates a *control_transfer_instruction* precise trap (trap type = $74_{16}$) instead of completing the control transfer. The pc stored in TPC[TL] is the address of the CTI, and the TNPC[TL] is set to the value of NPC before the CTI is executed. (impl. dep. #450-S20). PSTATE.tct is set to 0 as part of normal entry into a trap handler. When this exception occurs in nonprivileged or privileged mode, the trap is delivered in privileged mode.

- ***cpu_mondo*** [TT = $07C_{16}$] (Disrupting) — This interrupt is generated when another virtual processor has enqueued a message for this virtual processor. It is used to deliver a trap in privileged mode, to inform privileged software that an interrupt report has been appended to the virtual processor's CPU mondo queue. A direct message between virtual processors is sent via a CPU mondo interrupt, which is generated through software calls to hyperprivileged software. The standard software interface (API) to hyperprivileged software allows 64 bytes of data to be sent to one or more target virtual processors. When the CPU mondo queue has a valid entry, a *cpu_mondo* exception is sent to the target virtual processor.

**SPARC V9 Compatibility Note** The *data_access_exception* exception from SPARC V9 and UltraSPARC Architecture 2005 has been replaced by more specific exceptions, such as *DAE_invalid_asi*, *DAE_nc_page*, *DAE_nfo_page*, *DAE_privilege_violation*, and *DAE_side_effect_page*.

- ***DAE_invalid_asi*** [TT = $014_{16}$] (Precise) — An attempt was made to execute an invalid combination of instruction and ASI. See the instruction descriptions in Chapter 7 for a detailed list of valid ASIs for each instruction that can access alternate address spaces. The following invalid combinations of instruction, ASI, and virtual address cause a *DAE_invalid_asi* exception:
  - A load, store, load-store, or PREFETCHA instruction with either an invalid ASI or an invalid virtual address for a valid ASI.
  - A disallowed combination of instruction and ASI (see *Block Load and Store ASIs* on page 451 and *Partial Store ASIs* on page 452). This includes the following:
    - an attempt to use a (deprecated) atomic quad load ASI ($24_{16}$, $2C_{16}$, $34_{16}$, or $3C_{16}$) with any load alternate opcode other than LDTXA's (which is shared by LDDA)
    - an attempt to use a nontranslating ASI value with any load or store alternate instruction other than LDXA, LDDFA, STXA, or STDFA
    - an attempt to read from a write-only ASI-accessible register, or load from a store-only ASI (for example, a block commit store ASI, $E0_{16}$ or $E1_{16}$)
    - an attempt to write to a read-only ASI-accessible register
- ***DAE_nc_page*** [TT = $016_{16}$] (Precise) —An access to a noncacheable page (TTE.cp = 0) (including cases with the TLB disabled) was attempted by an atomic load-store instruction (CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA), an LDTXA instruction, a LDBLOCKF instruction, or a STPARTIALF instruction.
- ***DAE_nfo_page*** [TT = $017_{16}$] (Precise) — An attempt was made to access a non-faulting-only page (TTE.nfo = 1) by any type of load, store, load-store, or FLUSH instruction with an ASI other than a nonfaulting ASI (PRIMARY_NO_FAULT[_LITTLE] or SECONDARY_NO_FAULT[_LITTLE]).
- ***DAE_privilege_violation*** [TT = $015_{16}$] (Precise) — A privilege violation occurred, due to an attempt to access a privileged page (TTE.p = 1) by any type of load, store, or load-store instruction when executing in nonprivileged mode (PSTATE.priv = 0). This includes the special case of an access by privileged software using one of the ASI_AS_IF_USER_PRIMARY[_LITTLE] or ASI_AS_IF_USER_SECONDARY[_LITTLE] ASIs.
- ***DAE_side_effect_page*** [TT = $030_{16}$] (Precise) — An attempt was made to access a page which may cause side effects (TTE.e = 1) (including cases with the TLB disabled) by any type of load instruction with nonfaulting ASI.
- ***data_access_error*** [TT = $032_{16}$] (Precise) — A hardware error occurred during a data access. See Chapter 17, *Error Handling* for more details.
- ***data_access_MMU_error*** [TT = $072_{16}$] (Precise) — This exception is generated when, during a data access, the MMU detects any of
  (1) a data or tag parity error on a TLB (and/or µTLB) access, or
  (2) a multiple-tag-hit error on a TLB (and/or µTLB) access, or
  (3) an error during hardware tablewalk.

- ***data_access_MMU_miss*** [TT = $031_{16}$] (Precise) — During an attempted data access to memory,
  (1) hardware tablewalk was enabled, and
  (2) the MMU detects that a translation lookaside buffer did not contain a translation for the data's virtual address, and
  (3) the required TTE was not found in the configured TSBs.
- ***data_invalid_TSB _entry*** [TT = $02B_{16}$] (Precise) — During an attempted data access,
  (1) hardware tablewalk was enabled,
  (2) the MMU detected that a translation lookaside buffer did not contain a translation for the virtual address, and
  (3) the required TTE was found in the configured TSBs to be a real address, requiring real-to-physical address translation, and
  (4) the real address cannot be translated to a physical address by hardware.
- ***data_real_translation_miss*** [TT = $03F_{16}$] (Precise) — During an attempted real address data access, the MMU detected that a translation lookaside buffer (TLB) did not contain a translation for the real address (that is, a TLB miss occurred).
- ***dev_mondo*** [TT = $07D_{16}$] (Disrupting) — This interrupt causes a trap to be delivered in privileged mode, to inform privileged software that an interrupt report has been appended to its device mondo queue. When a virtual processor has appended a valid entry to a target virtual processor's device mondo queue, it sends a *dev_mondo* exception to the target virtual processor. The interrupt report contents are device specific.
- ***division_by_zero*** [TT = $028_{16}$] (Precise) — An integer divide instruction attempted to divide by zero.
- ***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 system-wide reset in multiprocessor systems.
- ***fast_data_access_MMU_miss*** [TT = $068_{16}$] (Precise) — During an attempted data access to memory,
  (1) hardware tablewalk was disabled (or is not implemented) and
  (2) the MMU detected that a translation lookaside buffer did not contain a translation for the virtual address.
  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 (TTE.w = 0)). Four trap vectors are allocated for this trap, allowing a trap handler of up to 32 instructions to fit within the trap vector area.

  Note that on an UltraSPARC Architecture virtual processor, an attempt to read or write to a privileged location while in nonprivileged mode causes the higher-priority *DAE_privilege_violation* instead of this exception.

- ***fast_instruction_access_MMU_miss*** [TT = $064_{16}$] (Precise) — During an attempted instruction virtual address access,
  (1) hardware tablewalk was disabled (or is not implemented) and
  (2) 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.
- ***fill_n_normal*** [TT = $0C0_{16}$–$0DF_{16}$] (Precise)
- ***fill_n_other*** [TT = $0E0_{16}$–$0FF_{16}$] (Precise)

  A RESTORE or RETURN instruction has determined that the contents of a register window must be restored from memory.
- ***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 disabled (PSTATE.pef = 0 or FPRS.fef = 0).
- ***fp_exception_ieee_754*** [TT = $021_{16}$] (Precise) — An FPop instruction generated an IEEE_754_exception and its corresponding trap enable mask (FSR.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. Example: execution of an FPop requires software assistance to complete. The floating-point exception type is encoded in FSR.ftt.
- ***guest_watchdog*** [TT = (see text)] (Precise, Disrupting) — The virtual processor was in nonprivileged or privileged mode, TL was ≥ *MAXPTL*, and a precise or disrupting exception to privileged mode occurred. *guest_watchdog* uses the same trap table entry (table offset $040_{16}$) as *watchdog_reset*. When a *guest_watchdog* trap occurs, the trap type (TT) value and priority of the exception that caused the trap are retained.
- ***hstick_match*** [TT = $05E_{16}$] (Disrupting) —This interrupt indicates that a match between the System Tick (STICK) and the Hypervisor System Tick Compare (HSTICK_CMPR) register has occurred (or that software has set HINTP.hsp = 1). The event is recorded in the hstick_match_pending (hsp) bit of the Hypervisor Interrupt Pending (HINTP) register. The *hstick_match* disrupting trap is recognized when HINTP.hsp = 1 and (PSTATE.ie = 1 or HPSTATE.hpriv = 0); otherwise, it remains pending. HINTP.hsp provides a mechanism for hyperprivileged software to determine that an *hstick_match* trap is pending while PSTATE.ie = 0 and to clear the condition without actually having to take the *hstick_match* trap.
- ***htrap_instruction*** [TT = $180_{16}$–$1FF_{16}$] (Precise) — A Tcc instruction was executed in privileged or hyperprivileged mode, the trap condition evaluated to TRUE, and the software trap number was greater than 127. The trap is delivered in hyperprivileged mode, using the hyperprivileged mode trap base address (HTBA). See also *trap_instruction* on page 520.

- ***hw_corrected_error*** [TT = $063_{16}$] (Disrupting) — Hardware detected an error asynchronous to instruction execution, or requests that information be logged for the error that was detected and corrected by the virtual processor.

  **SPARC V9 Compatibility Note** | The *hw_corrected_error* exception was called *ECC_error* in SPARC V9.

- ***IAE_nfo_page*** [TT = $00C_{16}$] (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a memory page which was marked for access only by nonfaulting loads (TTE.nfo = 1).
- ***IAE_privilege_violation*** [TT = $008_{16}$] (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a privileged memory page (TTE.p = 1) while the virtual processor was executing in nonprivileged mode.
- ***IAE_unauth_access*** [TT = $00B_{16}$] (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a memory page which was missing "execute" permission (TTE.ep = 0).
- ***illegal_instruction*** [TT = $010_{16}$] (Precise) — An attempt was made to execute an ILLTRAP instruction, an instruction with an unimplemented opcode, an instruction with invalid field usage, or an instruction that would result in illegal processor state.

  Examples of cases in which *illegal_instruction* is generated include the following:

  - An instruction encoding does not match any of the opcode map definitions (see Appendix A, *Opcode Maps*).
  - An instruction is not implemented in hardware.
  - 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, but is not required to be, generated. (See *Reserved Opcodes and Instruction Fields* on page 130.)
  - 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).
  - Illegal rd value for LDXFSR, STXFSR, or the deprecated instructions LDFSR or STFSR.
  - ILLTRAP instruction.
  - DONE or RETRY when TL = 0.

  All causes of an *illegal_instruction* exception are described in individual instruction descriptions in Chapter 7, *Instructions*.

- ***instruction_access_error*** [TT = $00A_{16}$] (Precise) — A hardware error occurred during an instruction access. See Chapter 17, *Error Handling* for more details.
- ***instruction_access_MMU_miss*** [TT = $009_{16}$] (Precise) — During an attempted instruction access (instruction fetch) from memory,
  (1) hardware tablewalk was enabled,
  (2) the MMU detected that a translation lookaside buffer did not contain a translation for the virtual address (that is, a TLB miss occurred), and
  (3) the required TTE was not found in the configured TSBs.

**SPARC V9 Compatibility Note** The *instruction_access_exception* exception from SPARC V9 has been replaced by more specific exceptions, such as *IAE_privilege_violation* and *IAE_unauth_access*.

- ***instruction_access_MMU_error*** [TT = $071_{16}$] (Precise) — This exception is generated when, during an instruction access, the MMU detects any of
  (1) a data or tag parity error on a TLB (and/or µTLB) access, or
  (2) a multiple-tag-hit error on a TLB (and/or µTLB) access, or
  (3) an error during hardware tablewalk.
- ***instruction_address_range*** [TT = $00D_{16}$] (Precise) — The *instruction_address_range* exception can only occur in implementations that do not implement full 64-bit instruction virtual addresses (impl. dep. #451-S20).

  This exception can only occur when PSTATE.am = 0, HPSTATE.hpriv = 0, HPSTATE.red = 0, and I/UMMU enable = 1 (a state said to "enable VA hole detection").

**Programming Note** Privileged software should not execute a write of PSTATE that changes PSTATE.am in the delay slot of a DCTI.

**Programming Note** Hyperprivileged software should not execute instructions that transition between VA hole detection states in the delay slot of a DCTI.

The *instruction_address_range* exception occurs upon either:

- an instruction fetch of a virtual address within an implementation-dependent region (no larger than 8 KB) immediately below the lowest virtual address that is not supported by the virtual processor and/or associated I/UMMU, or
- the fetch of a delayed control transfer instruction(DCTI)'s virtual address target that is not supported by the virtual processor and/or associated I/UMMU,
  VA hole detection is **not** enabled when the branch executes, and
  VA hole detection **is** enabled for the target fetch.

The second case can occur by either:

- execution of a state-changing instruction (for example,writing HPSTATE) in the delay slot of the DCTI, or

- an exception occuring on the delay slot of the DCTI, causing a trap, followed by state manipulation in the trap handler, which ends in a DONE or RETRY instruction, after which the target of the DCTI is fetched.

In the event that a trap handler modifies TPC or TNPC (via WRPR), the fetch of the instruction at the modified TPC or TNPC (after execution of DONE or RETRY) will not result in an *instruction_address_range* exception, even if the 64-bit address written to TPC or TNPC is not supported by the virtual processor and/or associated I/UMMU and VA hole detection is enabled when the unsupported virtual address is fetched. Instead, the unsupported bits of the address are silently ignored.

Implementations do not store state to create an *instruction_address_range* exception when the strand is executing from a PC whose 64-bit value is not supported by the virtual processor and/or associated I/UMMU, the strand is in a state in which VA hole detection is disabled (for example, when PSTATE.am = 1), and then software transitions the strand state to enable VA hole detection (for example, by setting PSTATE.am = 0) while the 64-bit PC is not supported by the virtual processor and/or associated I/UMMU. Instead, the unsupported bits of the address are silently ignored.

- ***instruction_breakpoint*** [TT = $076_{16}$] (Precise) — This exception is generated if HPSTATE.ibe = 1 and the processor has detected a breakpoint condition based on the values in the Instruction Breakpoint Control register for the current instruction. As part of the trap, the HPSTATE.ibe bit is cleared (set to 0).
- ***instruction_invalid_TSB_entry*** [TT = $02A_{16}$] (Precise) — During an attempted instruction access (instruction fetch),
  (1) hardware tablewalk was enabled,
  (2) the MMU detected that a translation lookaside buffer did not contain a translation for the virtual address,
  (3) the required TTE was found in the configured TSBs to be a real address, requiring real-to-physical address translation, and
  (4) the real address cannot be translated to a physical address by hardware.
- ***instruction_real_range*** [TT = $00E_{16}$] (Precise) — The *instruction_real_range* exception can only occur in implementations that do not implement full 56-bit instruction real addresses (impl. dep. #452-S20).

  This exception can only occur when HPSTATE.hpriv = 0, HPSTATE.red = 0, and I/UMMU enable = 0 (a state said to "enable RA hole detection").

**Programming Note** | Hyperprivileged software should not execute instructions that transition between RA hole detection states in the delay slot of a DCTI.

The *instruction_real_range* exception occurs when either:

- an instruction fetch of a real address within an implementation-dependent region (no larger than 8 KB) immediately below the lowest real address that is not supported by the virtual processor and/or associated I/UMMU, or

- the fetch of a delayed control transfer instruction(DCTI)'s real address target that is not supported by the virtual processor and/or associated I/UMMU, RA hole detection is **not** enabled when the branch executes, and RA hole detection **is** enabled for the target fetch.

The second case can occur by either:

- execution of a state-changing instruction (for example,writing HPSTATE) in the delay slot of the DCTI, or
- an exception occuring on the delay slot of the DCTI, causing a trap, followed by state manipulation in the trap handler, which ends in a DONE or RETRY instruction, after which the target of the DCTI is fetched.

In the event that a trap handler modifies TPC or TNPC (via WRPR), the fetch of the instruction at the modified TPC or TNPC (after execution of DONE or RETRY) will not result in an *instruction_real_range* exception, even if the 64-bit address written to TPC or TNPC is not supported by the virtual processor and/or associated I/UMMU and RA hole detection is enabled when the unsupported real address is fetched. Instead, the unsupported bits of the address are silently ignored.

Implementations do not store state to create an *instruction_real_range* exception when the strand is executing from a PC whose 64-bit value is not supported by the virtual processor and/or associated I/UMMU, the strand is in a state in which RA hole detection is disabled (for example, when HPSTATE.hpriv = 1), and then software transitions the strand state to enable RA hole detection (for example, by setting HPSTATE.hpriv = 0) while the 64-bit PC is not supported by the virtual processor and/or associated I/UMMU. Instead, the unsupported bits of the address are silently ignored.

- ***instruction_real_translation_miss*** [TT = $03E_{16}$] (Precise) — During an attempted real address instruction access (instruction fetch), the MMU detected a TLB miss.
- ***instruction_VA_watchpoint*** [TT = $075_{16}$] (Precise) — The virtual processor has detected that the Program Counter (PC) matches the VA Watchpoint register, when instruction VA watchpoints are enabled and the PC is being translated from a virtual address to a physical address. If the PC is not being translated from a virtual address (for example, the PC is being treated as a physical address), then an *instruction_VA_watchpoint* exception will not be generated, even if a match is detected between the VA Watchpoint register and the PC.
- ***internal_processor_error*** [TT = $029_{16}$] (Precise) — A serious internal error occurred in the virtual processor.

  **IMPL. DEP. #402-S10:** The trap priority of the *internal_processor_error* exception is implementation dependent. Furthermore, its priority may vary within an implementation, based on the cause of the error being reported.

- ***interrupt_level_n*** [TT = $041_{16}$–$04F_{16}$] (Disrupting) — SOFTINT{*n*} was set to 1 or an external interrupt request of level *n* was presented to the virtual processor and *n* > PIL.

  **Implementation Note** | interrupt_level_14 can be caused by (1) setting SOFTINT{14} to 1, (2) occurrence of a "TICK match", or (3) occurrence of a "STICK match" (see ***SOFTINT$^P$ Register (ASRs 20, 21, 22)*** on page 77).

- ***LDDF_mem_address_not_aligned*** [TT = $035_{16}$] (Precise) — An attempt was made to execute an LDDF or LDDFA instruction and the effective address was not doubleword aligned. (impl. dep. #109)

- ***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 444.)

- ***mem_address_range*** [TT = $02E_{16}$] (Precise) — The *mem_address_range* exception can only occur in implementations that do not implement full 64-bit virtual addresses (impl. dep. #451-S20).

  The *mem_address_range* exception occurs when either:

  - a memory-access instruction (load, store, or load-store) generates a memory virtual address that is not supported by the virtual processor and/or associated D/UMMU, or
  - a branch, JMPL, RETURN, or CALL instruction that is taken and generates a target virtual address that is not supported by the virtual processor and/or associated I/UMMU

  This exception can only occur if PSTATE.am = 0 at the time the instruction executes.

  For a memory-access instruction, this exception can only occur if the DMMU performs VA-to-PA translation for the effective memory address referenced by this instruction.

  For a branch, JMPL, RETURN, or CALL instruction, this exception can only occur if HPSTATE.hpriv = 0, HPSTATE.red = 0, and I/UMMU enable = 1 at the time of instruction execution.

***Programming Note*** No *mem_address_range* exception is triggered when control transfer to a virtual address not supported by the virtual processor and/or I/UMMU occurs by means not explicitly described above, such as:

- DONE or RETRY that enables VA hole detection and that redirects fetch to a virtual address that is not supported by the virtual processor and/or I/UMMU
- a DCTI that executes with VA hole detection disabled with a virtual target address that is not supported by the virtual processor and/or I/UMMU, with a delay slot instruction that enables VA hole detection

- ***mem_real_range*** [TT = $02D_{16}$] (Precise) — The *mem_real_range* exception can only occur in implementations that do not implement full 56-bit real addresses (impl. dep. #452-S20).

  The *mem_real_range* exception occurs when either:

  - a memory-access instruction (load, store, or load-store) generates a memory real address that is not supported by the virtual processor and/or associated D/UMMU, or
  - a branch, JMPL, RETURN, or CALL instruction that is taken and generates a target real address that is not supported by the virtual processor and/or associated I/UMMU

  For a memory-access instruction, this exception can only occur if the DMMU performs RA-to-PA translation for the effective memory address referenced by this instruction.

  For a branch, JMPL, RETURN, or CALL instruction, this exception can only occur if HPSTATE.hpriv = 0, HPSTATE.red = 0, and I/UMMU enable = 0 at the time of instruction execution.

***Programming Note*** No *mem_real_range* exception is triggered when control transfer to a real address not supported by the virtual processor and/or I/UMMU occurs by means not explicitly described above, such as:

- DONE or RETRY that enables RA hole detection and that redirects fetch to a real address that is not supported by the virtual processor and/or I/UMMU
- a DCTI that executes with RA hole detection disabled with a real target address that is not supported by the virtual processor and/or I/UMMU, with a delay slot instruction that enables RA hole detection

- ***nonresumable_error*** [TT = $07F_{16}$] (Disrupting) — There is a valid entry in the nonresumable error queue. This interrupt is not generated by hardware, but is used by hyperprivileged software to inform privileged software that an error report has been appended to the nonresumable error queue.
- ***PA_watchpoint*** [TT = $061_{16}$] (Precise) — The virtual processor has detected a load or store to a physical address specified by the PA Watchpoint register while PA watchpoints are enabled. Hyperprivileged software may reflect this trap back to privileged software as a synthetic *RA_watchpoint* exception.
- ***pic_overflow*** [TT = $04F_{16}$] (Disrupting) — A performance counter has overflowed and PIL < 15. Note that this exception shares a trap type, $04F_{16}$, with *interrupt_level_15*. The disrupting trap caused by *pic_overflow* is conditioned by PSTATE.ie.
  If PSTATE.ie = 1 and PIL < 15 when the possible counter overflow is detected and depending on the event being monitored by the counter, the disrupting trap may be reported prior to retirement of the instruction that incremented the counter to cause the possible counter overflow. Upon entry to the trap handler, TPC points to an instruction that increments the performance counter and the counter is within some epsilon of overflow.
  If PSTATE.ie = 0 or PIL = 15 when the possible overflow is detected, the trap remains pending and will be taken on the first instruction for which PSTATE.ie = 1 and PIL < 15. In this case, TPC may not point to an instruction that increments the counter.
- ***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 in nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), or an action defined to be hyperprivileged has been attempted while in nonprivileged or privileged mode (HPSTATE.hpriv = 0). Examples:
  - A data access by nonprivileged software using a restricted (privileged or hyperprivileged) ASI, that is, an ASI in the range $00_{16}$ to $7F_{16}$ (inclusively)
  - A data access by nonprivileged or privileged software using a hyperprivileged ASI, that is, an ASI in the range $30_{16}$ to $7F_{16}$ (inclusively)
  - Execution by nonprivileged software of an instruction with a privileged operand value
  - An attempt to read the TICK register by nonprivileged software when nonprivileged access to TICK is disabled (TICK.npt = 1).
  - An attempt to execute a nonprivileged instruction with an operand value requiring more privilege than available in the current privilege mode.
- ***privileged_opcode*** [TT = $011_{16}$] (Precise) — An attempt was made to execute a privileged instruction while PSTATE.priv = 0.

- ***RED_state_exception*** [TT = (see text)] (Precise) — Caused when TL = *MAXTL* – 1 and a trap occurs, an event that brings 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*.
- ***resumable_error*** [TT = $07E_{16}$] (Disrupting) — There is a valid entry in the resumable error queue. This interrupt is used to inform privileged software that an error report has been appended to the resumable error queue, and the current instruction stream is in a consistent state so that execution can be resumed after the error is handled.
- ***software_initiated_reset*** (SIR) [TT = $004_{16}$] (Precise) — 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.
- ***STDF_mem_address_not_aligned*** [TT = $036_{16}$] (Precise) — An attempt was made to execute an STDF or STDFA instruction and the effective address was not doubleword aligned. (impl. dep. #110)
- ***store_error*** [TT = $007_{16}$] (Deferred) — An error has been detected on a store instruction that prevents it from completing, but the error was detected after the store had passed its instruction retirement point. Since the store cannot be made globally visible, the software thread that issued the store must be terminated. Therefore, this is a termination deferred trap.
- ***sw_recoverable_error*** [TT = $040_{16}$] (Disrupting) — Indicates that one or more potentially recoverable errors have been detected in the virtual processor. A single *sw_recoverable_error* exception may indicate multiple errors and may occur asynchronously to instruction execution. When *sw_recoverable_error* causes a trap, the TPC and TNPC stacked by the trap do not necessarily indicate the instruction or data access that caused the error. (impl. dep. #31-V8-Cs10, #218-U3-Cs20) See Chapter 17, *Error Handling* for more details.

  **SPARC V9 Compatibility Note** | The *sw_recoverable_error* exception was called *async_data_error* in the SPARC V9 specification, which in turn superseded the earlier and less general SPARC V8 *data_store_error* exception.
- ***tag_overflow*** [TT = $023_{16}$] (Precise) (deprecated (C2) ) — 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, and the software trap number operand of the instruction is 127 or less.
- ***trap_level_zero*** [TT = $05F_{16}$] (Precise) — This exception indicates a simultaneous existence of three conditions as an instruction is about to be executed:

- *trap_level_zero* exceptions are enabled (HPSTATE.tlz = 1),
- the virtual processor is in nonprivileged or privileged mode (HPSTATE.hpriv = 0), and
- the trap level (TL) register's value is zero (TL = 0)

Upon entry to the trap handler for *trap_level_zero*, TPC points to the instruction that was about to be executed after all three of these conditions were met.

**Programming Note** The purpose of this trap is to improve efficiency when descheduling a virtual processor. When a descheduling event occurs and the virtual processor is executing in privileged mode at TL > 0, hyperprivileged software can choose to enable the *trap_level_zero* exception (set HPSTATE.tlz ← 1) and return to privileged mode, enabling privileged software to complete its TL > 0 processing. When privileged code returns to TL = 0, this exception enables the hyperprivileged code to regain control and deschedule the virtual processor with low overhead.

- ***unimplemented_LDTW*** [TT = $012_{16}$] (Precise) — An attempt was made to execute an LDTW instruction that is not implemented in hardware on this implementation (impl. dep. #107-V9).
- ***unimplemented_STTW*** [TT = $013_{16}$] (Precise) — An attempt was made to execute an STTW instruction that is not implemented in hardware on this implementation (impl. dep. #108-V9).
- ***watchdog_reset*** (WDR) [TT = $002_{16}$] (Reset) — This trap occurs in `error_state` and causes a transition to `RED_state` (impl. dep. #254-U3-Cs10).
- ***VA_watchpoint*** [TT = $062_{16}$] (Precise) — The virtual processor has detected an attempt to access a virtual address specified by the VA Watchpoint register, while VA watchpoints are enabled and the address is being translated from a virtual address to a physical address. If the load or store address is not being translated from a virtual address (for example, the address is being treated as a real address), then a *VA_watchpoint* exception will not be generated even if a match is detected between the VA Watchpoint register and a load or store address. This exception is always masked in hyperprivileged mode; therefore, a *VA_watchpoint* trap cannot occur in hyperprivileged mode (even if memory is accessed using `ASI_AS_IF_USER_PRIMARY` or `ASI_AS_IF_USER_SECONDARY`).

## 12.7.1 SPARC V9 Traps Not Used in UltraSPARC Architecture 2007

The following traps were optional in the SPARC V9 specification and are not used in UltraSPARC Architecture 2007:

- ***async_data_error*** [TT = $040_{16}$] (Disrupting) — This exception was superseded by the *sw_recoverable_error* exception.

- ***data_access_protection*** [TT = $033_{16}$] (Precise or Deferred) — This exception is generally superseded by *fast_data_access_protection* (see page 511).
- ***fast_ECC_error*** [TT = $070_{16}$] (Precise) — A single-bit or multiple-bit ECC error was detected. This exception is superseded by *hw_corrected_error* in UltraSPARC Architecture 2007.

  **IMPL. DEP. #202-U3:** 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 $070_{16}$.
- ***implementation_dependent_exception_n*** [TT = $077_{16}$ - $07A_{16}$] This range of implementation-dependent exceptions has been replaced by a set of architecturally-defined exceptions. (impl.dep. #35-V8-Cs20)
- ***LDQF_mem_address_not_aligned*** [TT = $038_{16}$] (Precise) — 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-V9-Cs10). 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 *Load Floating-Point Register* on page 253. (impl. dep. #111)
- ***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-V9-Cs10). 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 *Store Floating-Point* on page 342. (impl. dep. #112)

# 12.8 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.

## 12.8.1 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.

## 12.8.2 *clean_window* Trap

The virtual processor provides the *clean_window* trap so that system software can create a secure environment in which it is guaranteed that data cannot inadvertently leak through register windows from one software program to another.

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, from 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 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.

## 12.8.3 Vectoring of Fill/Spill Traps

To make handling of fill and spill traps efficient, the SPARC V9 architecture 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 *Trap-Table Entry Addresses* on page 479, for more details on how the trap address is determined.

## 12.8.4 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** *N_REG_WINDOWS*.

- 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** *N_REG_WINDOWS*

  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** *N_REG_WINDOWS*

**Implementation Note** | All spill traps can set CWP by using the calculation:
CWP ← (CWP + CANSAVE + 2) **mod** *N_REG_WINDOWS*
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** *N_REG_WINDOWS*

- On a *clean_window* trap, the window following CWP must be cleaned. Then

  CWP ← (CWP + 1) **mod** *N_REG_WINDOWS*

## 12.8.5 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)

CHAPTER 13

# Interrupt Handling

Virtual processors and I/O devices can interrupt a selected virtual processor by assembling and sending an interrupt packet. The contents of the interrupt packet are defined by software convention. Thus, hardware interrupts and cross-calls can have the same hardware mechanism for interrupt delivery and share a common software interface for processing.

The interrupt mechanism is a two-step process:

- sending of an interrupt request (through an implemenation-specific hardware mechanism) to an interrupt queue of the target virtual processor
- receipt of the interrupt request on the target virtual processor and scheduling software handling of the interrupt request

Privileged software running on a virtual processor can schedule interrupts to *itself* (typically, to process queued interrupts at a later time) by setting bits in the privileged SOFTINT register (see *Software Interrupt Register (SOFTINT)* on page 526).

**Programming Note** An interrupt request packet is sent by an interrupt source (through an implementation-specific mechanism) and is received by the specified target in an interrupt queue. 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 itself to later handle the interrupt request by posting an interrupt in the SOFTINT register at the desired interrupt level.

In the following sections, the following aspects of interrupt handling are described:


- **Interrupt Packets** on page 526.
- **Software Interrupt Register (SOFTINT)** on page 526.
- **Interrupt Queues** on page 527.
- **Interrupt Traps** on page 529.
- **Strand Interrupt ID Register (STRAND_INTR_ID)** on page 530.

# 13.1 Interrupt Packets

Each interrupt is accompanied by data, referred to as an "interrupt packet". An interrupt packet is 64 bytes long, consisting of eight 64-bit doublewords. The contents of these data are defined by software convention.

# 13.2 Software Interrupt Register (SOFTINT)

To schedule interrupt vectors for processing at a later time, privileged software running on a virtual processor can send itself signals (interrupts) by setting bits in the privileged SOFTINT register. Similarly, hyperprivileged software can schedule interrupt vectors for privileged software running on the same virtual processorby setting bits in SOFTINT.

See *SOFTINT$^P$ Register (ASRs 20, 21, 22)* on page 77 for a detailed description of the SOFTINT register.

**Programming Note** The SOFTINT register (ASR $16_{16}$) is used for communication from nucleus (privileged, TL > 0) software to privileged software running with TL = 0. 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*.

**Programming Note** The SOFTINT mechanism is independent of the "mondo" interrupt mechanism mentioned in *Interrupt Queues* on page 527. The two mechanisms do not interact.

## 13.2.1 Setting the Software Interrupt Register

SOFTINT{*n*} is set to 1 by executing a WRSOFTINT_SET$^P$ instruction (WRasr using ASR 20) with a '1' in bit *n* of the value written (bit *n* corresponds to interrupt level *n*). The value written to the SOFTINT_SET register is effectively **or**ed into the SOFTINT register. This approach allows the interrupt handler to set one or more bits in the SOFTINT register with a single instruction.

See *SOFTINT_SET$^P$ Pseudo-Register (ASR 20)* on page 78 for a detailed description of the SOFTINT_SET pseudo-register.

### 13.2.2 Clearing the Software Interrupt Register

When all interrupts scheduled for service at level *n* have been serviced, kernel software executes a WRSOFTINT_CLR$^{P}$ instruction (WRasr using ASR 21) with a '1' in bit *n* of the value written, to clear interrupt level *n* (impl. dep. 34-V8a). The complement of the value written to the SOFTINT_CLR register is effectively **and**ed with the SOFTINT register. This approach allows the interrupt handler to clear one or more bits in the SOFTINT register with a single instruction.

**Programming Note** To avoid a race condition between operating system kernel software clearing an interrupt bit and nucleus software setting it, software should (again) examine the queue for any valid entries after clearing the interrupt bit.

See *SOFTINT_CLR$^{P}$ Pseudo-Register (ASR 21)* on page 79 for a detailed description of the SOFTINT_CLR pseudo-register.

## 13.3 Interrupt Queues

Interrupts are indicated to privileged mode via circular interrupt queues, each with an associated trap vector. There are 4 interrupt queues, one for each of the following types of interrupts:

- Device mondos[1]
- CPU mondos
- Resumable errors
- Nonresumable errors

New interrupt entries are appended to the tail of a queue (by hardware or by hyperprivileged software) and privileged software reads them from the head of the queue.

**Programming Note** Software conventions for cooperative management of interrupt queues and the format of queue entries are specified in the separate *Hypervisor API Specification* document.

### 13.3.1 Interrupt Queue Registers

The active contents of each queue are delineated by a 64-bit head register and a 64-bit tail register.

1. "mondo" is a historical term, referring to the name of the original UltraSPARC 1 bus transaction in which these interrupts were introduced

**IMPL. DEP. #421-S10:** It is implementation dependent whether interrupt queue head and tail registers (a) are datatype-agnostic "scratch registers" used for communication between privileged and hyperprivileged software, in which case their contents are defined purely by software convention, or (b) are maintained to some degree by virtual processor hardware, imposing a fixed meaning on their contents.

**Programming Note** If the contents of Queue Head and Tail registers are set only by software convention in a given implementation, software could place any type of data in them (such as addresses, address offsets, or index values).

It is expected that Queue Head and Tail registers will typically contain a byte offset from the base of an appropriately-aligned queue region in memory.

The interrupt queue registers are accessed through ASI `ASI_QUEUE` ($25_{16}$). The ASI and address assignments for the interrupt queue registers are provided in TABLE 13-1.

**TABLE 13-1** Interrupt Queue Register ASI Assignments

| Register | ASI | Virtual Address | Privileged mode Access | Hyper-privileged mode Access |
|---|---|---|---|---|
| CPU Mondo Queue Head | $25_{16}$ (`ASI_QUEUE`) | $3C0_{16}$ | RW | R/W |
| CPU Mondo Queue Tail | $25_{16}$ (`ASI_QUEUE`) | $3C8_{16}$ | R or RW† | R/W |
| Device Mondo Queue Head | $25_{16}$ (`ASI_QUEUE`) | $3D0_{16}$ | RW | R/W |
| Device Mondo Queue Tail | $25_{16}$ (`ASI_QUEUE`) | $3D8_{16}$ | R or RW† | R/W |
| Resumable Error Queue Head | $25_{16}$ (`ASI_QUEUE`) | $3E0_{16}$ | RW | R/W |
| Resumable Error Queue Tail | $25_{16}$ (`ASI_QUEUE`) | $3E8_{16}$ | R or RW† | R/W |
| Nonresumable Error Queue Head | $25_{16}$ (`ASI_QUEUE`) | $3F0_{16}$ | RW | R/W |
| Nonresumable Error Queue Tail | $25_{16}$ (`ASI_QUEUE`) | $3F8_{16}$ | R or RW† | R/W |

† see **IMPL. DEP.**#422-S10

**IMPL. DEP. #422-S10:** It is implementation dependent whether tail registers are writable in privileged mode. If a tail register is read-only in privileged mode, an attempt to write to it causes a *DAE_invalid_asi* exception. If a tail register is writable in privileged mode, an attempt to write to it results in undefined behavior.

**Implementation Note** Although Queue Head and Tail registers behave as registers, they may or may not be implemented using actual *hardware* registers. For example, they may reside in memory, mapped by a mechanism visible only to hyperprivileged software. In any case, the means by which Queue Head and Tail registers are implemented is not visible to privileged software.

The status of each queue is reflected by its head and tail registers:

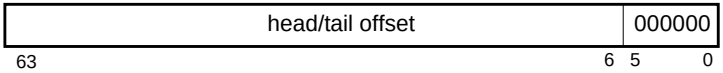
- A Queue Head Register indicates the location of the oldest interrupt packet in the queue
- A Queue Tail Register indicates the location where the next interrupt packet will be stored

An event that results in the insertion of a queue entry causes the tail register for that queue to refer to the following entry in the circular queue. Privileged code is responsible for updating the head register appropriately when it removes an entry from the queue.

A queue is *empty* when the contents of its head and tail registers are equal. A queue is *full* when the insertion of one more entry would cause the contents of its head and tail registers to become equal.

**Programming Note**

By current convention, the format of a Queue Head or Tail register is as follows:




Under this convention:

- updating a Queue Head register involves incrementing it by 64 (size of a queue entry, in bytes)
- Queue Head and Tail registers are updated using modular arithmetic (modulo the size of the circular queue, in bytes)
- bits 5:0 always read as zeros, and attempts to write to them are ignored
- the maximum queue offset for an interrupt queue is implementation dependent
- behavior when a queue register is written with a value larger than the maximum queue offset (queue length minus the length of the last entry) is undefined

This is merely a convention and is subject to change.

## 13.4 Interrupt Traps

The following interrupt traps are defined in the UltraSPARC Architecture 2007: *cpu_mondo*, *dev_mondo*, *resumable_error*, and *nonresumable_error*. The first three (*cpu_mondo*, *dev_mondo*, and *resumable_error*) are all generated by hardware, while *nonresumable_error* is generated by hyperprivileged software. See Chapter 12, *Traps*, for details.

UltraSPARC Architecture 2007 also supports the *interrupt_level_n* traps defined in the SPARC V9 specification.

How interrupts are delivered is implementation-specific; see the relevant implementation-specific Supplement to this specification for details.

## 13.5 Strand Interrupt ID Register (STRAND_INTR_ID)

The STRAND_INTR_ID per-virtual-processor register allows 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 *Strand Interrupt ID Register (STRAND_INTR_ID)* on page 602 for details.

CHAPTER 14

# Memory Management

An UltraSPARC Architecture 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, simplified protection encoding, and multiple page sizes. In an UltraSPARC Architecture implementation, TLB miss processing can be achieved either by hardware page tablewalk or by privileged software.

**IMPL. DEP. # 451-S20**: The width of the virtual address supported is implementation dependent. If fewer than 64 bits are supported, the unsupported bits must have the same value as the most significant supported bit. For example, if the model supports 48 virtual address bits, then bits 63:48 must have the same value as bit 47.

This appendix describes the Memory Management Unit, as observed by hyperprivileged software, in these sections:


- **Virtual Address Translation** on page 532.
- **Hyperprivileged Memory Management Architecture** on page 536.
- **Context ID** on page 537.
- **TSB Translation Table Entry (TTE)** on page 539.
- **Translation Storage Buffer (TSB)** on page 543.
- **Hardware Support for TSB Access** on page 546.
- **Faults and Traps** on page 549.
- **MMU Operation Summary** on page 552.
- **ASI Value, Context ID, and Endianness Selection for Translation** on page 555.
- **Translation** on page 558.
- **SPARC V9 "MMU Attributes"** on page 565.
- **MMU Internal Registers and ASI Operations** on page 566.
- **Translation Lookaside Buffer Hardware** on page 590.

# 14.1 Virtual Address Translation

The MMUs may support up to eight page sizes: 8 KBytes, 64 KBytes, 512 KBytes, 4 MBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes. 8-KByte, 64-KByte and 4- MByte page sizes must be supported; the other page sizes are optional.

**IMPL. DEP. #310-U4:** Which, if any, of the following optional page sizes are supported by the MMU in an UltraSPARC Architecture 2007 implementation is implementation dependent: 512 KBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes.

An UltraSPARC Architecture MMU supports a 64-bit virtual address (VA) space.

**IMPL. DEP. #452-S20:** The number of real address (RA) and physical address (PA) bits supported is implementation dependent. A minimum of 40 bits and maximum of 56 bits can be provided for both real addresses (RA) and physical addresses (PA). See implementation-specific documentation for details.

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 14-1 and FIGURE 14-2.

**IMPL. DEP. #453-S20:** It is implementation dependent whether there is a unified MMU (UMMU) or a separate IMMU (for instruction accesses) and DMMU (for data accesses). The UltraSPARC Architecture supports both configurations.

Each MMU consists of one or more Translation Lookaside Buffers (TLBs), and may include micro-TLB structures. Separate Instruction and Data MMUs (IMMU and DMMU, respectively) may be provided to enable concurrent virtual-to-physical address translations for instruction and data.

**IMPL. DEP. #222-U3:** TLB organization is implementation dependent.

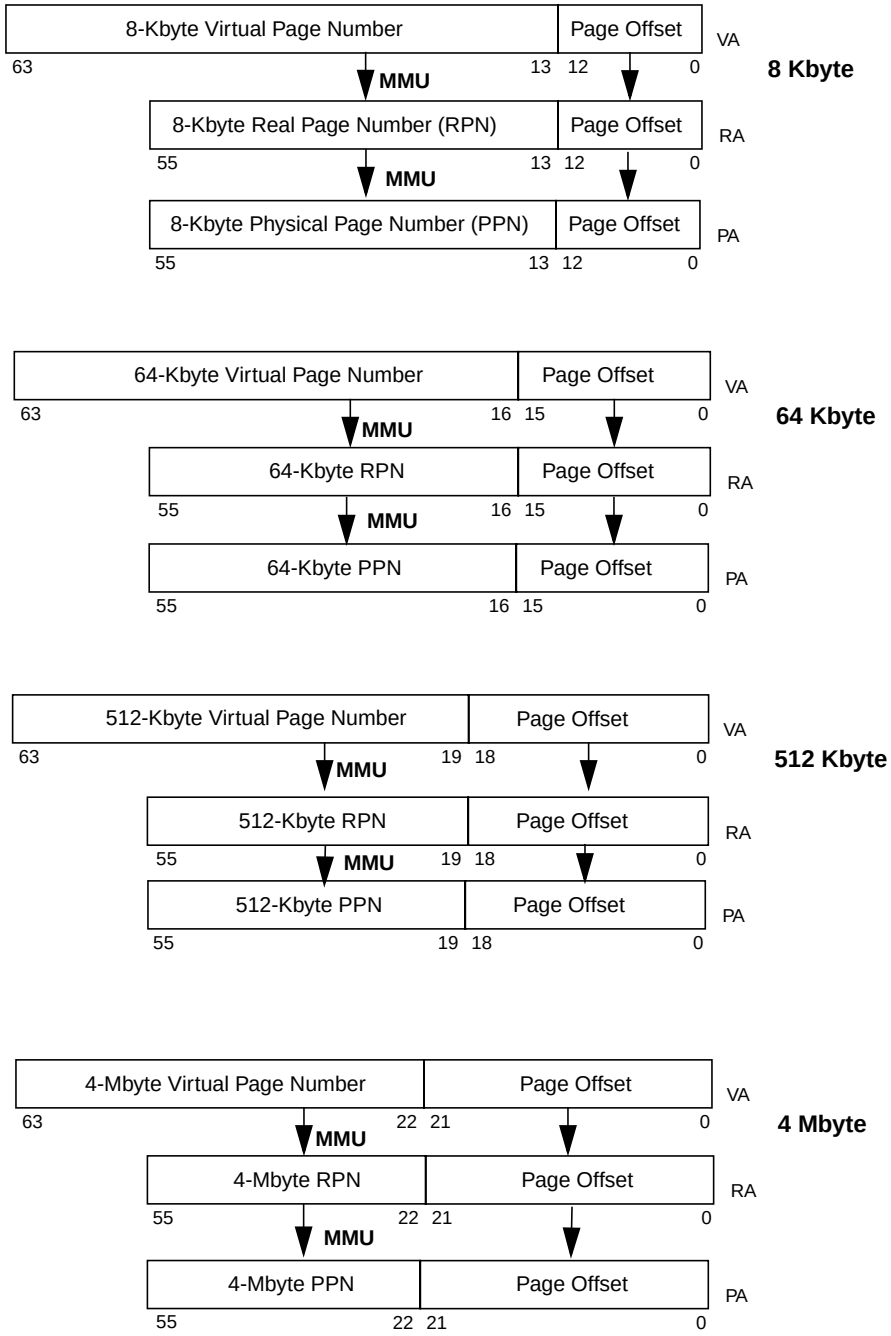



**FIGURE 14-1** Virtual-to--Physical Address Translation for 8-Kbyte, 64-Kbyte, 512-Kbyte, and 4-Mbyte Page Sizes

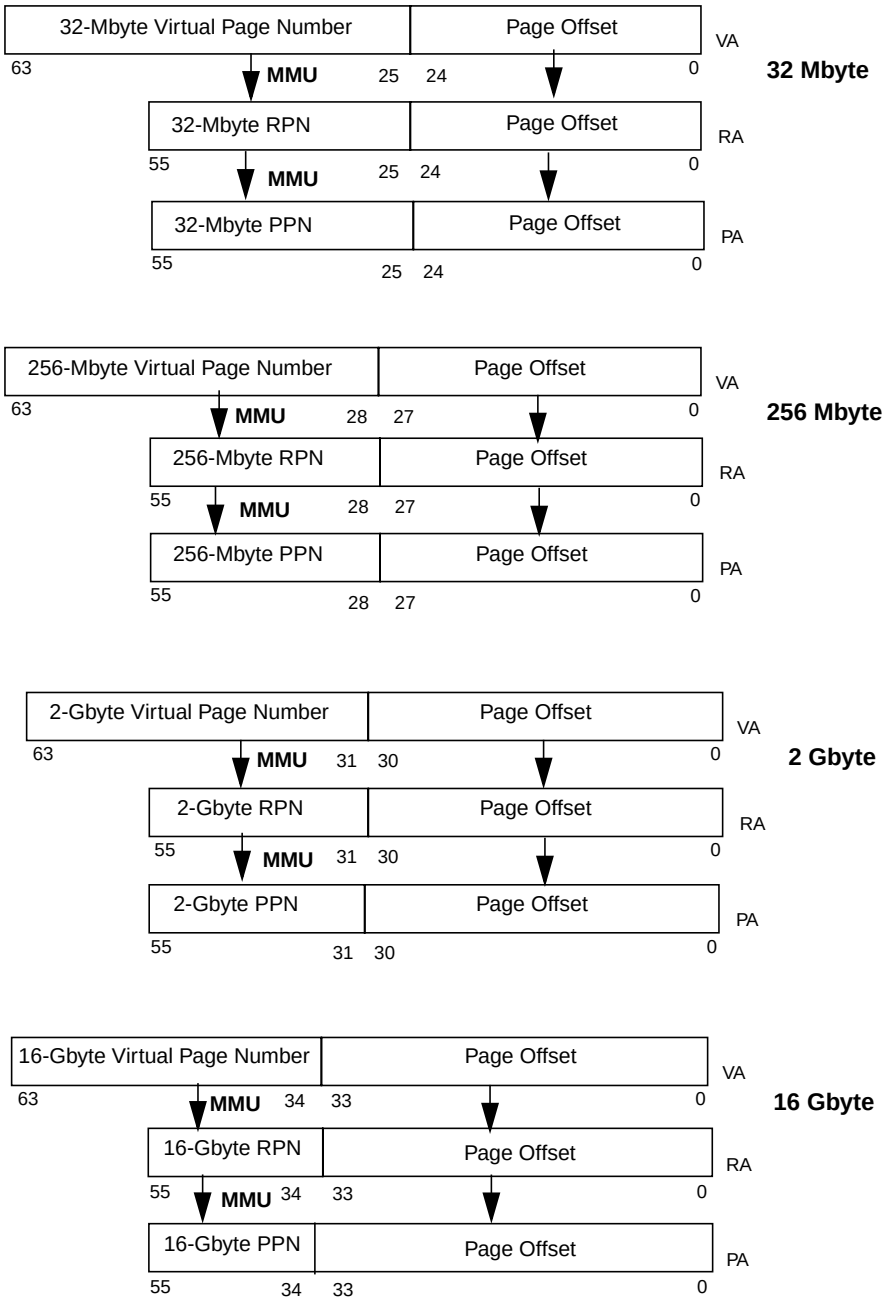



**FIGURE 14-2** Virtual-to-Physical Address Translation for 32-Mbyte, 256-Mbyte, 2-Gbyte, and 16-Gbyte Page Sizes

Privileged software manages virtual-to-real address translations. Hyperprivileged software manages real-to-physical address translations.

Privileged software maintains translation information in an arbitrary data structure, called the *software translation table*.

The Translation Storage Buffer (TSB) is an array of Translation Table Entries which serves as a cache of the software translation table, used to quickly reload the TLB in the event of a TLB miss.

The MMU TLBs act as independent caches of the software translation table, providing appropriate concurrency for virtual-to-physical address translation.

Hyperprivileged software maintains translation information for real-to-physical translations.

During a memory access, one or more TLBs are searched for a VA (or RA) translation. A TLB hit is indicated when the virtual address, context ID, and partition ID (or real address and partition ID) match an entry in the TLB.

A TLB miss is indicated when no such match occurs, and is handled as follows:

- With the hardware tablewalk unimplemented or disabled, the MMU immediately traps to hyperprivileged 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 546.
- With hardware tablewalk implemented and enabled, hardware processes the TLB miss directly.

A conceptual view of privileged-mode memory management the MMU is shown in FIGURE 14-3. 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 software translation table and the underlying memory management hardware. The TSB can be shared by all processes running on a virtual processor or can be process specific; the hardware does not require any particular scheme. There can be several TSBs.

The UltraSPARC Architecture provides a memory partitioning mechanism that allows for multiple partitions, each containing its own real address space. Hyperprivileged software provides real address to physical address translations. See *Real Address Translation* on page 537.

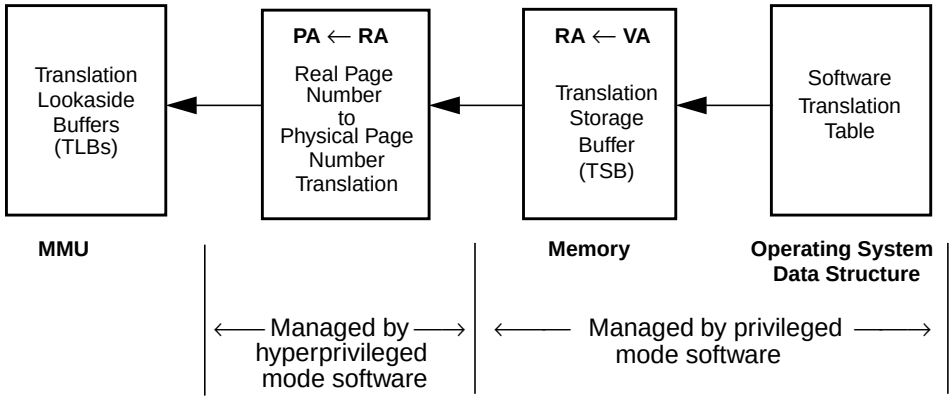



**FIGURE 14-3** Conceptual View of the MMU

Aliasing of multiple virtual addresses to the same physical address is supported. However, the reverse case of multiple mappings from one virtual address to multiple physical addresses producing a multiple TLB match is detected in hardware as a multiple tag hit TLB error. See Chapter 17, *Error Handling*, for details.

## 14.2 Hyperprivileged Memory Management Architecture

The intent of the hyperprivileged memory management architecture is to provide a memory addressing capability for a virtualized architecture, but at the same time removing the explicit dependence on hardware mechanisms for virtual memory management. Mechanisms are provided to allow privileged mode to manipulate the memory made available to it, and in turn to virtualize and make that memory available to its nonprivileged mode process.

### 14.2.1 Partition ID

The hyperprivileged memory architecture has a partition ID, which separates the real addresses of each partition in the same way that context IDs separate virtual address spaces within a single real address space. Hyperprivileged mode provides the partition ID to create multiple real address spaces. It uses the partition ID register to associate addresses with their partition ID.

The full representation of a memory address is:

virtual address: <partition ID > :: <context ID > :: <address>

real address: <partition ID > :: <address>

physical address: <address>

Nonprivileged mode only uses virtual addresses.

Privileged mode uses virtual addresses and real addresses, and manages the allocation of context IDs.

Hyperprivileged mode uses physical addresses (and explicit ASI virtual and real addresses) and manages the allocation of partition IDs.

The partition ID field is included in each TLB entry to allow multiple guest operating systems to share the MMU. The field is loaded with the contents of the partition ID register when the TLB entry is loaded. In addition, the partition ID stored in each entry of a TLB is compared against the partition ID to determine if a TLB hit occurs.

See *Partition ID Register* on page 569 for more details.

### 14.2.2 Real Address Translation

The memory system supports real addresses. TABLE 14-8 provides examples of the real addresses for data accesses. In addition, real addresses are provided when the MMU is disabled in privileged mode.

The MMU supports both virtual-to-physical (VA → PA) and real-to-physical (RA → PA) translations.

Hyperprivileged software controls the translation mechanisms from Real Page Numbers (RPNs) to Physical Page Numbers (PPNs).

## 14.3 Context ID

The MMU supports three contexts:

- Primary Context
- Secondary Context
- Nucleus Context (which has a fixed Context ID value of zero)

The context used for each access depends on the type of access, the ASI used, the current privilege mode, and the current trap level (TL). Details are provided in the following paragraphs and in TABLE 14-1.

For instruction fetch accesses, in nonprivileged and privileged mode when TL = 0 the Primary Context is used; when TL > 0, the Nucleus Context is used. Instruction accesses in hyperprivileged mode are physical addresses, so no context is provided.

For data accesses using *implicit* ASIs, in nonprivileged and privileged mode when TL = 0 the Primary Context is used; when TL > 0, the Nucleus Context is used. Data accesses using implicit ASIs in hyperprivileged mode are physical addresses, so no context is provided.

For data accesses using *explicit* ASIs:

- In nonprivileged mode the Primary Context is used for the ASI_PRIMARY* ASIs, and the Secondary Context is used for the ASI_SECONDARY* ASIs.
- In privileged mode, the Primary Context is used for the ASI_PRIMARY* and the ASI_AS_IF_USER_PRIMARY* ASIs, the Secondary Context is used for the ASI_SECONDARY* and the ASI_AS_IF_USER_SECONDARY* ASIs, and the Nucleus Context is used for ASI_NUCLEUS* ASIs.
- In hyperprivileged mode, the Primary Context is used for ASI_AS_IF_[USER|PRIV]_PRIMARY* ASIs, and the Secondary Context for the ASI_AS_IF_[USER|PRIV]_SECONDARY* ASIs, and the Nucleus Context for ASI_AS_IF_PRIV_NUCLEUS*.

The above paragraphs are summarized in TABLE 14-1.

**TABLE 14-1** Context Usage

| Access Type | Privilege Mode | Under What Conditions each Context is Used | | |
|---|---|---|---|---|
| | | Primary Context | Secondary Context | Nucleus Context |
| Instruction Access | Nonprivileged or Privileged | (when TL = 0) | † | (when TL > 0) |
| | Hyperprivileged | ‡ | ‡ | ‡ |
| Data access using *implicit* ASI | Nonprivileged or Privileged | (when TL = 0) | † | (when TL > 0) |
| | Hyperprivileged | ‡ | ‡ | ‡ |

**TABLE 14-1** Context Usage

| Access Type | Privilege Mode | Under What Conditions each Context is Used | | |
|---|---|---|---|---|
| | | Primary Context | Secondary Context | Nucleus Context |
| Data access using *explicit* ASI | Nonprivileged | ASI_PRIMARY* | ASI_SECONDARY* | † |
| | Privileged | ASI_PRIMARY* ASI_AS_IF_USER_PRIMARY* | ASI_SECONDARY* ASI_AS_IF_USER_SECONDARY* | ASI_NUCLEUS* |
| | Hyperprivileged | ASI_AS_IF_USER_PRIMARY* ASI_AS_IF_PRIV_PRIMARY* | ASI_AS_IF_USER_SECONDARY* ASI_AS_IF_PRIV_SECONDARY* | ASI_AS_IF_PRIV_NUCLEUS* |

† no context is listed becuase this case cannot occur

‡ no context is provided, because physcial addresses are used in this case in hyperprivileged mode

**Note** The UltraSPARC Architecture provides the capability of private and shared contexts. Multiple primary and secondary context IDs, which allow different processes to share TTEs within the TLB, are defined. See *Context ID Registers* on page 567 for details.

**Programming Note** Privileged software (operating sytems) intended to be portable across all UltraSPARC Architecture implementations should always ensure that, for memory accesses made in privileged mode, private and shared context IDs are set to the same value. The exception to this is privileged-mode accesses using the ASI_AS_IF_USER* ASIs, which remain portable even if the private and shared context IDs differ.

**IMPL. DEP. #___**: The UltraSPARC Architecture defines a 16-bit context ID. The size of the context ID field is implementation dependent. At least 13 bits must be implemented. If fewer than 16 bits are supported, the unused high order bits are ignored on writes to the context ID, and read as zeros.

## 14.4 TSB Translation Table Entry (TTE)

The Translation Storage Buffer (TSB) Translation Table Entry (TTE) is the equivalent of a page table entry as defined in the *Sun4v Architecture Specification*; 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 used by either the hardware tablewalker or privileged software.

The TTE configuration is illustrated in FIGURE 14-4 and described in TABLE 14-2.

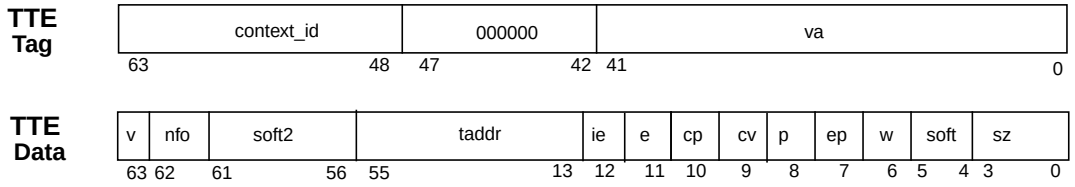



**FIGURE 14-4** Translation Storage Buffer (TSB) Translation Table Entry (TTE)

**TABLE 14-2** TSB TTE Bit Description *(1 of 4)*

| Bit | Field | Description |
|---|---|---|
| Tag– **63:48** | context_id | The 16-bit context ID associated with the TTE. |
| Tag– **47:42** | — | These bits must be zero for a tag match. |
| Tag– **41:0** | va | Bits 63:22 of the Virtual Address (the virtual page number). Bits 21:13 of the VA are not maintained because these bits index the minimally sized, direct-mapped TSBs. |
| Data – **63** | v | Valid. If v = 1, then the remaining fields of the TTE are meaningful, and the TTE can be used; otherwise, the TTE cannot be used to translate a virtual address.<br>**Programming Note** The explicit Valid bit is (intentionally) redundant with the software convention of encoding an invalid TTE with an unused context ID. The encoding of the context_id field is necessary to cause a failure in the TTE tag comparison, while the explicit Valid bit in the TTE data simplifies the TTE miss handler. |
| Data – **62** | nfo | No Fault Only. If nfo = 1, loads with ASI_PRIMARY_NO_FAULT{_LITTLE} or ASI_SECONDARY_NO_FAULT{_LITTLE} are translated. Any other data access with the D/UMMU TTE.nfo = 1 will trap with a *DAE_nfo_page* exception. An instruction fetch access to a page with the IMMU TTE.nfo = 1 results in an *IAE_nfo_page* exception. |
| Data – **61:56** | 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 any TLB (or uTLB), so when it is read from the TLB (uTLB), it may read as zero. |

**TABLE 14-2** TSB TTE Bit Description *(2 of 4)*

| Bit | Field | Description |
|---|---|---|
| Data – **55:13** | taddr | Target address; the underlying address (Real Address {55:13} or Physical Address {55:13}) to which the MMU will map the page.<br>UltraSPARC Architecture TLBs store physical addresses, not real addresses. Hyperprivileged software is responsible for translation between real and physical addresses. Whether this field contains a Real or Physical address is determined by the ra_not_pa bit in the corresponding MMU TSB Configuration register.<br><br>**IMPL. DEP. #441-S10:** Whether an implementation uses the most significant physical address bit to differentiate between memory and I/O addresses is implementation dependent. If that method is used, then the most significant bit of the physical address (PA) = 1 designates I/O space and the most significant bit of PA = 0 designates memory space .<br>**IMPL. DEP. #224-U3:** Physical address width support by the MMU is implementation dependent in the UltraSPARC Architecture; minimum PA width is 40 bits.<br>**IMPL. DEP. #238-U3:** When page offset bits for larger page sizes are stored in the TLB, it is implementation dependent whether the data returned from those fields by a Data Access read is zero or the data previously written to them. |
| Data – **12** | ie | Invert Endianness. If ie = 1 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 555 for details.<br><br>**Programming Notes** (1) The primary purpose of this bit is to aid in the mapping of I/O devices (through *noncacheable* memory addresses) whose registers contain and expect data in little-endian format. Setting TTE.ie = 1 allows those registers to be accessed correctly by big-endian programs using ordinary loads and stores, such as those typically issued by compilers; otherwise little-endian loads and stores would have be issued by hand-written assembler code.<br><br>(2) This bit can also be used when mapping *cacheable* memory. However, cacheable accesses to pages marked with TTE.ie = 1 may be slower than accesses to the page with TTE.ie = 0. For example, an access to a cacheable page with TTE.ie = 1 may perform as if there was a miss in the first-level data cache.<br><br>**Implementation Note** Some implementations may require cacheable accesses to pages tagged with TTE.ie = 1 to bypass the data cache, adding latency to those accesses.<br><br>**IMPL. DEP. #__:** The ie bit in the IMMU is ignored during ITLB operation. It is implementation dependent if it is implemented and how it is read and written. |

**TABLE 14-2** TSB TTE Bit Description *(3 of 4)*

| Bit | Field | Description |
|---|---|---|
| Data – **11** | e | Side effect. If the side-effect bit is set to 1, loads with ASI_PRIMARY_NO_FAULT, ASI_SECONDARY_NO_FAULT, and their *_LITTLE variations 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 to 1 for pages that map I/O devices having side effects. Note, also, that the e bit causes the prefetch instruction to be treated as a nop, but does not prevent normal (hardware) instruction prefetching.<br>**Note 1:** 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 to 1. If both the cp and cv bits are set to 1 along with the e bit, the result is undefined.<br>**Note 2:** The e bit and the nfo bit are mutually exclusive; both bits should never be set to 1 in any TTE. |
| Data – **10**<br>Data – **9** | cp,<br>cv | The cacheable-in-physically-indexed-cache bit and cacheable-in-virtually-indexed-cache bit determine the cacheability of the page. 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) | Meaning of TTE when placed in: D-TLB (Data Cache VA-indexed) |
|---|---|---|
| 00, 01 | Noncacheable | Noncacheable |
| 10 | Cacheable L2-cache, I-cache | Cacheable L2-cache |
| 11 | Cacheable L2-cache, I-cache | Cacheable L2-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-U3:** Whether the cv bit is supported in hardware is implementation dependent in the UltraSPARC Architecture. The cv bit in hardware should be provided if the implementation has virtually indexed caches, and the implementation should support hardware unaliasing for the caches.

| | | |
|---|---|---|
| Data – **8** | p | Privileged. If p = 1, only privileged and hyperprivileged software can access the page mapped by the TTE. If p = 1 and an access to the page is attempted by nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), then the MMU signals an*IAE_privilege_violation* exception or*DAE_privilege_violation* exception. |

**TABLE 14-2** TSB TTE Bit Description *(4 of 4)*

| Bit | Field | Description |
|---|---|---|
| Data – **7** | ep | Executable. If ep = 1, the page mapped by this TTE has execute permission granted. Instructions may be fetched and executed from this page. If ep = 0, an attempt to execute an instruction from this page results in an *IAE_unauth_access* exception.<br>**IMPL. DEP. #___:** An UltraSPARC Architecture ITLB implementation may elect to not implement the ep bit, and instead present the *IAE_unauth_access* exception if there is an attempt to load an ITLB entry with ep = 0 during a hardware tablewalk. In this case, the MMU miss trap handler software must also detect the ep = 0 case when the IMMU miss is handled by software. |
| Data – **6** | w | Writable. If w = 1, 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.<br>**IMPL. DEP. #___:** The w bit in the IMMU is ignored during ITLB operation. It is implementation dependent if the bit is implemented and how it is written and read. |
| Data – **5:4** | 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 any TLB (or uTLB), so when it is read from the TLB (or uTLB), it may read as zero. |
| Data – 3:**0** | sz | The page size of this entry, encoded as shown below.<br>**sz** / **Page Size**<br>0000 — 8 Kbyte<br>0001 — 64 Kbyte<br>0010 — 512 Kbyte<br>0011 — 4 Mbyte<br>0100 — 32 Mbyte<br>0101 — 256 Mbyte<br>0110 — 2 Gbyte<br>0111 — 16 Gbyte<br>1000-1111 — *Reserved* |

# 14.5 Translation Storage Buffer (TSB)

The Translation Storage Buffer (TSB) is an array of Translation Table Entries managed entirely by privileged 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 546, 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.

## 14.5.1 TSB Indexing Support

Hardware TSB indexing support via TSB pointers should be provided for the TTEs. Hardware tablewalk uses the TSB pointers. If the hardware tablewalk is disabled, the TLB miss handler software can use the TSB pointers.

## 14.5.2 TSB Cacheability and Consistency

The TSB exists as a 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 the dynamic sharing of the level-2 cache resource will provide a better overall solution than that provided by a fixed partitioning.

**Programming Note** When software updates the TSB, it is responsible for ensuring that the store(s) used to perform the update are made visible in the memory system (for access by subsequent loads, stores, and load-stores) by use of an appropriate MEMBAR instruction. Otherwise, since hardware tablewalk is not required to examine store buffers, a subsequent hardware tablewalk access to the TSB could retrieve stale data from the L2 cache.

Making a TSB update visible to fetches of instructions subsequent to the store(s) that updated the TSB may require execution of instructions such as FLUSH, DONE, or RETRY, in addition to the MEMBAR.

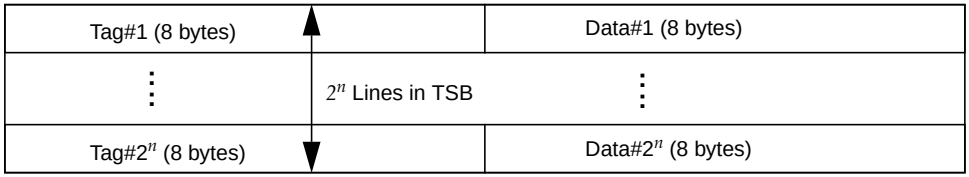
## 14.5.3 TSB Organization

The TSB is arranged as a direct-mapped cache of 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 14-5. The constant *n* is determined by the size field in the TSB register; it can range from 512 to an implementation-dependent number.

**FIGURE 14-5** TSB Organization




**IMPL. DEP. #227-U3:** The maximum number of entries in a TSB is implementation-dependent in the UltraSPARC Architecture (to a maximum of 16 million, limited by the size of the TSB Configuration register's tsb_size field).

## 14.5.4 TSB Configuration

The MMU provides hardware tablewalk support. Precomputed pointers into the TSB are provided for both zero and nonzero context IDs and are contained in the following ASIs:

```
ASI_MMU_ZERO_CONTEXT_TSB_CONFIG_0
ASI_MMU_ZERO_CONTEXT_TSB_CONFIG_1
...
ASI_MMU_ZERO_CONTEXT_TSB_CONFIG_n
ASI_MMU_NONZERO_CONTEXT_TSB_CONFIG_0
ASI_MMU_NONZERO_CONTEXT_TSB_CONFIG_1
...
ASI_MMU_NONZERO_CONTEXT_TSB_CONFIG_n
```

**IMPL. DEP. #___:** The number of TSB configuration registers is implementation dependent.

**IMPL. DEP. #__:** Hardware tablewalk supports an implementation dependent number of TSBs per virtual processor for zero context IDs and for nonzero context IDs. In some configurations, hardware tablewalk ignores the context ID match; see *Multiple context IDs* on page 547.

All TTEs in the TSB must have the size indicated in the TSB Configuration Register or larger. If the TSB page size is smaller than the size in the TSB Configuration Register, such an entry will never be matched.

Each TSB can be configured by hyperprivileged software in one of two different modes: context ID-match or context ID-ignore. The mode determines how a TSB entry is matched when the TSB is searched. Bits in the MMU_NONZERO_CONTEXTID_TSB_CONFIG_*n* registers control the mode for each context ID register. See TABLE 14-20 on page 572 for details.

In context ID-match mode, the context_id field of the TTE tag is matched against a context ID register, as specified by the actual or implied access ASI. This mode enables a TSB to be used for caching translation entries belonging to different context IDs. Matching with the context_id field allows only those translations belonging to the current context ID to be loaded into the TLB.

In context ID-ignore mode, the context_id field of a TSB is ignored when the TSB is searched. A TSB configured in this mode should have a context_id field of each translation entry set to 0. When a valid TSB entry is matched, it is loaded into the TLB with a context_id value provided from one of the primary or secondary context ID registers. The choice of primary or secondary is determined by the actual or implied access ASI. The index of the context ID is specified as part of the TSB configuration. Context ID-ignore mode enables TSB entries to be used with more than one context. See *Multiple context IDs* on page 547.

## 14.6 Hardware Support for TSB Access

The MMU hardware provides services to allow the TLB-miss handler to efficiently reload the TLB on a TLB miss. These services include:

- Hardware tablewalk — hardware loading of missing TTE entries
- 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.

### 14.6.1 Hardware Tablewalk

Hardware tablewalk is a hardware state machine that services reload requests from the TLB due to TLB misses. Hardware tablewalk accesses the TSBs to find TTEs that match the virtual address and one of the context IDs of the request. Hardware tablewalk may access multiple separate TSBs for each request.

Privileged code cannot access or control physical memory, so TTEs in the TSBs controlled by privileged code contain real page numbers, not physical page numbers. TTEs in the TSBs controlled by hyperprivileged code can contain real page numbers or physical page numbers. Hyperprivileged code controls the RA-to-PA translation within hardware tablewalk to permit hardware tablewalk to load privileged code TTEs into the TLB for VA-to-PA translation.

Real address requests are not translated by hardware tablewalk. In the event a real address misses in the TLB, an *instruction_real_translation_miss* (for instruction accesses) or a *data_real_translation_miss* (for data accesses) exception is generated for the request.

**IMPL. DEP. #__:** The hardware tablewalk is normally pipelined; it is provided on a virtual processor basis. The number of possible hardware tablewalks to be processed at a given time is model dependent. The number of simultaneous TSB accesses is also model dependent.

### 14.6.1.1 Typical Hardware Tablewalk Sequence

A typical TLB miss and reload sequence when hardware tablewalk is enabled is the following:

- Hardware tablewalk uses the TSB Configuration registers and the VA of the access to calculate the physical address of the TSB TTE to examine. The TSB Configuration register provides the base address of the TSB as well as the number of TTEs in the TSB and the size of the pages translated by the TTEs.[1] Hardware tablewalk uses a Nonzero Context ID TSB Configuration register if the context ID of the request is nonzero; otherwise, it uses a Zero Context ID TSB Configuration register. The context ID used to determine zero/nonzero context ID is always the content of the Context ID 0 register (in the event of a TLB miss on a Primary or Secondary Context access). Hardware tablewalk uses the page size from the TSB Configuration register to calculate the presumed VPN for the given VA. Hardware tablewalk then uses the number of TTE entries and the presumed VPN to generate an index into the TSB. This index is concatenated with the upper bits of the base address to generate the TTE address.
- Hardware tablewalk forwards a quadword load request for the TTE address to the L2 cache. At some later time, the L2 returns the TTE to hardware tablewalk.
- Hardware tablewalk compares the VPN and context ID of the request to that from the TTE, masking the VPN based on the page size in the TTE. If the VPN and context ID match, hardware tablewalk returns the TTE with the RPN translated into a PPN (see *Real Page Number To Physical Page Number Translation* on page 548). Hardware tablewalk checks the TTE from each enabled TSB until it either finds a match or has searched all enabled TSBs.
- If none of the TTE entries from the enabled TSBs match on page size, VPN, and context ID, hardware generates an *instruction_access_MMU_miss* or *data_access_MMU_miss* trap.

**Multiple context IDs.** Multiple Primary and Secondary context IDs permit different processes to share TTEs within the TLBs. The use_cid0 and use_cid1 bits in the TSB Configuration register disable the context ID match for hardware tablewalk. Hardware tablewalk ignores the context IDs in the TSB TTEs if either of these bits is set to 1 for requests with nonzero context IDs. If either bit is 1 and the page size and

1. This implies that all TTEs within a given TSB share a common page size.

VPN match, hardware tablewalk signals the TLB to write the appropriate context ID (depending on the bit setting) as the context ID of the TTE when it is loaded (instead of the context ID in the TTE itself). See TABLE 14-3 for details. Hardware tablewalk ignores these bits for requests with a 0 (nucleus) context ID value, and behaves as if the bits are zero (i.e. there is a context comparison).

**TABLE 14-3** Selection Control for Multiple Nonzero Context IDs

| use_cid0 | use_cid1 | Meaning |
|---|---|---|
| 0 | 0 | Hardware tablewalk compares context_id of TTE from the TSB with context ID of request and, if they match, stores the context ID of TTE into context_id field of TLB TTE. |
| 0 | 1 | Hardware tablewalk ignores context_id of TTE from the TSB and stores value of context ID register 1 in context_id field of TLB TTE. |
| 1 | X | Hardware tablewalk ignores context_id of TTE from the TSB and stores value of context ID register 0 in context_id field of TLB TTE. |

**Real Page Number To Physical Page Number Translation.** When hardware tablewalk fetches a TTE from a TSB, it can treat the taddr field as either an RA or a PA under control of the ra_not_pa field of the TSB Configuration register. If the ra_not_pa bit = 1, hardware tablewalk will translate the most significant bits of the real address in the TTE into the corresponding bits of the physical address. The TLBs store this PPN. The TLBs use this PPN to translate VAs into PAs. The hypervisor controls the RA-to-PA translation mechanism.

The RA-to-PA translation mechanism provides both range checking as well as mapping of address ranges from one location to another. The translation mechanism uses the RPN and page size in the TTE and calculates the starting and ending addresses for the specified real page. It then checks that these addresses lie in one of four ranges specified by the Real Range registers. If the real page lies completely inside one of the ranges (and the range is enabled), then the translation mechanism adds the RPN in the TTE to the corresponding field in the Physical Offset register to create the Physical Page Number. If the real page does not lie completely within any range, then an *instruction_invalid_TSB_entry* or *data_invalid_TSB_entry* trap is delivered to the virtual processor that initiated the hardware tablewalk. Each virtual processor has a model dependent number of dedicated ranges with corresponding physical offsets. The RA to PA translation does not depend on the context ID value being zero or nonzero.

**Note** | When the TSB Configuration register has ra_not_pa= 0, no range checking is provided for PPNs for that TSB.

## 14.6.2 Typical TLB Software Miss Sequence

A typical TLB miss sequence is the following:

1. A TLB miss causes either a *fast_instruction_access_MMU_miss* or a *fast_data_access_MMU_miss* exception when hardware tablewalk is disabled or unimplemented.
2. The appropriate TLB miss handler loads 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 is 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 ID of the load or store responsible for the MMU exception. See *TSB Translation Table Entry (TTE)* on page 539.

***Implementation 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.

***Note*** For proper operation, translations for guest OS TSB miss handlers must always be made available to hyperprivileged code or to the hardware tablewalk mechanism.

## 14.7 Faults and Traps

The traps recorded by the MMU are listed in TABLE 14-4. For a detailed description of each trap, see Chapter 12, *Traps*. All listed traps are precise traps.

**TABLE 14-4** MMU Trap Types, Causes, and Stored State Register Update Policy *(1 of 2)*

| Trap Name | Trap Cause | Registers Updated (Stored State in MMU) | | | |
|---|---|---|---|---|---|
| | | IMMU Tag Access | SFAR | DMMU Tag Access | UMMU Tag Access |
| *fast_instruction_access_MMU_miss* | I-TLB miss with hardware tablewalk disabled or unimplemented. | X | | | X |
| *IAE_nfo_page* | instruction access to nonfaulting load page (TTE.nfo = 1). | X | | | X |
| *IAE_privilege_violation* | Nonprivileged instruction access to privileged page (TTE.p = 1). | X | | | X |
| *IAE_unauth_access* | Instruction access to page without "execute" permission (TTE.ep = 0). | X | | | X |
| *instruction_access_error* | An error was detected on the access of instruction data. | X | | | X |
| *instruction_access_MMU_error* | An error was detected on the TLB entry or during hardware tablewalk for an instruction access. | X | | | X |
| *instruction_access_MMU_miss* | The hardware tablewalk for an instruction access could not find the required TTE in the enabled TSBs. | X | | | X |
| *instruction_invalid_TSB_entry* | A hardware tablewalk for an instruction access found the TTE in the enabled TSBs to be a real address, which cannot be translated to a physical address by hardware. | X | | | X |
| *instruction_real_translation_miss* | I-TLB miss on an instruction access using a real-address. | X | | | X |
| *instruction_address_range* [1] | Instruction virtual access out of range. | X | | | X |
| *instruction_real_range*[2] | Instruction real access out of range. | X | | | X |
| *instruction_VA_watchpoint* | Virtual instruction address matches the VA watchpoint register with VA watchpoints enabled. | | | | |
| *DAE_invalid_asi* | Invalid ASI for instruction. | | X | | |
| *DAE_nc_page* | Atomic access to noncacheable page (TTE.cp = 0). | | X | X | X |
| *DAE_nfo_page* | Data access to nonfaulting page (TTE.nfo = 1) with ASI other than a non-faulting ASI. | | X | X | X |

**TABLE 14-4** MMU Trap Types, Causes, and Stored State Register Update Policy *(2 of 2)*

| Trap Name | Trap Cause | Registers Updated (Stored State in MMU) | | | |
|---|---|---|---|---|---|
| | | IMMU Tag Access | SFAR | DMMU Tag Access | UMMU Tag Access |
| *DAE_privilege_violation* | Nonprivileged data access to privileged page (TTE.p = 1). | | X | X | X |
| *DAE_side_effect_page* | Non-faulting ASI data access to side-effect page (TTE.e = 1). | | X | X | X |
| *data_access_error* | An error is detected on a data access. | | X | | |
| *data_access_MMU_error* | An error was detected on the TLB entry or during hardware tablewalk for a data access. | | impl. dep. | | |
| *data_access_MMU_miss* | The hardware tablewalk for a data access could not find the required TTE in the enabled TSBs. | | | X | X |
| *data_invalid_TSB_entry* | A hardware tablewalk for a data access found the TTE in the enabled TSBs to be a real address, which cannot be translated to a physical address by hardware. | | | X | X |
| *data_real_translation_miss* | D-TLB miss on a data access using a real address. | | | X | X |
| *mem_address_range*[1] | Data or branch virtual access out of range. | | X | | |
| *mem_real_range*[2] | Data or branch real access out of range. | | X | | |
| *fast_data_access_MMU_miss* | D-TLB miss with hardware tablewalk disabled. | | | X | X |
| *fast_data_access_protection* | Store data access to page with write protection (TTE.w = 1). | | X | X | X |
| *privileged_action* | Data access by nonprivileged software, using a privileged or hyperprivileged ASI. | | | | |
| *PA_watchpoint* | Data access physical address matches the PA watchpoint register with PA watchpoints enabled. | | X | | |
| *mem_address_not_aligned*, **_mem_address_not_aligned* | Data access address is not properly aligned. | | (impl. dep. #237-U3) | | |
| *VA_watchpoint* | Data access virtual address matches the VA watchpoint register with VA watchpoints enabled. | | X | | |

1. Implementations that do not support 64-bit VAs in hardware require these exceptions. See section 12.7 of the Traps chapter for details.

2. Implementations that do not support 56 bits of real address require these exceptions. See section 12.7 of the Traps chapter for details.

**IMPL. DEP. #__:** It is implementation dependent whether D-SFAR is updated for MMU errors. See the Error Handling chapter of the model-specific PRM for details.

# 14.8 MMU Operation Summary

The behavior of the D/UMMU for data accesses is summarized in TABLE 14-5; the behavior of the I/UMMU for instruction accesses is summarized in TABLE 14-6. 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* or *data_access_MMU_miss* exception |
| Dasi | *DAE_invalid_asi* exception |
| Dpriv | *DAE_privilege_violation* exception |
| Dse | *DAE_side_effect_page* exception |
| Dreal | *data_real_translation_miss* exception |
| Dprot | *fast_data_access_protection* exception |
| Imiss | *fast_instruction_access_MMU_miss* or *instruction_access_MMU_miss* exception |
| Ipriv | *IAE_privilege_violation* exception |

The ASI is indicated by one of the following abbreviations:

| Abbreviation | Meaning |
|---|---|
| NUC | `ASI_NUCLEUS*` |
| PRI | Any ASI with PRIMARY translation, except `*NO_FAULT` |
| SEC | Any ASI with SECONDARY translation, except `*NO_FAULT` |
| PRI_NF | `ASI_PRIMARY_NO_FAULT*` |
| SEC_NF | `ASI_SECONDARY_NO_FAULT*` |
| AIU_PRI | `ASI_AS_IF_USER_PRIMARY*` |
| AIU_SEC | `ASI_AS_IF_USER_SECONDARY*` |
| AIP_PRI | `ASI_AS_IF_PRIV_PRIMARY*` |

| Abbreviation | Meaning |
|---|---|
| AIP_SEC | ASI_AS_IF_PRIV_SECONDARY* |
| AIP_NUC | ASI_AS_IF_PRIV_NUCLEUS* |
| REAL | ASI*REAL* |

**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 14-5.

- An attempt to execute an invalid combination of instruction and ASI; for example, ASI_PRIMARY_NOFAULT for a store or atomic load-store. The MMU signals a *DAE_invalid_asi* exception for these cases. For more details, see the description of this exception in *Exception and Interrupt Descriptions* on page 508.
- Attempted access using a restricted ASI in nonprivileged or privileged mode. The MMU signals a *privileged_action* exception for this case.
- An atomic instruction (including a 128-bit atomic integer load, LDTXA) 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 D/UMMU is disabled. The MMU signals a *DAE_nc_page* exception 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 *DAE_nfo_page* exception for this case.

**TABLE 14-5** D/UMMU Operation for Translations for Data Accesses

| Condition | | | | Behavior | | | | |
|---|---|---|---|---|---|---|---|---|
| **Opcode** | **Privilege Mode** | **ASI** | **W** | **TLB Miss** | **e = 0 p = 0** | **e = 0 p = 1** | **e = 1 p = 0** | **e = 1 p = 1** |
| Load | nonprivileged | PRI, SEC | — | Dmiss | OK | Dpriv | OK | Dpriv |
| | | PRI_NF, SEC_NF | — | Dmiss | OK | Dpriv | Dse | Dpriv |
| | privileged | PRI, SEC, NUC | — | Dmiss | OK | | | |
| | | PRI_NF, SEC_NF | — | Dmiss | OK | | Dse | |
| | | AIU_PRI, AIU_SEC | — | Dmiss | OK | Dpriv | OK | Dpriv |
| | | REAL | — | Dreal | OK | | | |
| | hyperprivileged | PRI, SEC, NUC[1] | — | — | OK | — | OK | — |
| | | PRI_NF, SEC_NF[1] | — | — | OK | — | Dse | — |
| | | AIU_PRI, AIU_SEC | — | Dmiss | OK | Dpriv | OK | Dpriv |
| | | AIP_PRI, AIP_SEC, AIP_NUC | — | Dmiss | OK | | | |
| | | REAL | — | Dreal | OK | | | |
| FLUSH | nonprivileged | | — | Dmiss[2] | OK | | | |
| | privileged | | — | Dmiss[2] | OK | | | |
| | hyperprivileged | | — | Dmiss[2] | OK | | | |
| Store or Atomic Load-store | nonprivileged | PRI, SEC | 0 | Dmiss | Dprot | Dpriv | Dprot | Dpriv |
| | | | 1 | Dmiss | OK | Dpriv | OK | Dpriv |
| | privileged | PRI, SEC, NUC | 0 | Dmiss | Dprot | | | |
| | | | 1 | Dmiss | OK | | | |
| | | AIU_PRI, AIU_SEC | 0 | Dmiss | Dprot | Dpriv | Dprot | Dpriv |
| | | | 1 | Dmiss | OK | Dpriv | OK | Dpriv |
| | | REAL | 0 | Dreal | Dprot | | | |
| | | | 1 | Dreal | OK | | | |
| | hyperprivileged | PRI, SEC, NUC[1] | 1 | — | OK | — | OK | — |
| | | AIU_PRI, AIU_SEC | 0 | Dmiss | Dprot | Dpriv | Dprot | Dpriv |
| | | | 1 | Dmiss | OK | Dpriv | OK | Dpriv |
| | | AIP_PRI, AIP_SEC, AIP_NUC | 0 | Dmiss | Dprot | | | |
| | | | 1 | Dmiss | OK | | | |
| | | REAL | 0 | Dreal | Dprot | | | |
| | | | 1 | Dreal | OK | | | |

1. In hyperprivileged mode, the address using these ASIs is treated as a physical address and the TLB is bypassed, with the default physical page attribute values applied. See *MMU Bypass* on page 564 for details.
2. The TLB miss exception is implementation dependent. .

The following cases are not covered in TABLE 14-6.

- An instruction access to a page marked as nonfaulting (TTE.nfo = 1). The MMU signals an *IAE_nfo_page* exception in this case.
- An instruction access to a page not marked with execute permission (TTE.ep = 0). The MMU signals an *IAE_unauth_access* exception in this case.

**TABLE 14-6** I/UMMU Operation for Translations for Instruction Accesses

| | Behavior | | |
|---|---|---|---|
| **Privilege mode** | **TLB Miss** | **p = 0** | **p = 1** |
| nonprivileged | Imiss | OK | Ipriv |
| privileged | Imiss | OK | OK |
| hyperprivileged | — | OK | OK |

# 14.9 ASI Value, Context ID, and Endianness Selection for Translation

The selection of the context ID 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, privilege level (PSTATE.priv and HPSTATE.hpriv) and the virtual processor endian mode (PSTATE.cle).
2. The context ID is determined directly from the ASI. The context ID value is read by the context ID selected by the ASI.

The ASI value and endianness (little or big) are determined for the IMMU and D/UMMU, respectively, according to TABLE 14-7 through TABLE 14-8, assuming that the MMUs are enabled.

When using the Primary Context ID, the values stored in the Primary Context IDs are used by the Instruction and Data (or Unified) MMUs. When using the Secondary Context ID, the values stored in the Secondary Context IDs are used by the Data (or Unified) MMU. The Secondary Context ID is never used by the Instruction MMU (or an instruction access to the Unified MMU).

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 D/UMMU “invert endianness” bit (ie). The D/UMMU ie bit inverts the endianness that is otherwise specified for the access.

**Note** The D/UMMU ie 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-endian if the ie bit = 1.

Accesses to ASIs which are not translated by the MMU (nontranslating ASIs) are not affected by the D/UMMU.ie bit.

**TABLE 14-7** ASI Mapping for Instruction Access (I/UMMU Enabled)

| Mode | TL | PSTATE.cle | Endianness | ASI Used | Resulting Address Type |
|---|---|---|---|---|---|
| Nonprivileged | 0 | — | Big | ASI_PRIMARY | VA |
| Privileged | 0 | — | Big | ASI_PRIMARY | VA |
| | 1–2 | — | Big | ASI_NUCLEUS | VA |
| Hyperprivileged | any | — | Big | — | PA |

**TABLE 14-8** ASI Mapping for Data Accesses (D/UMMU Enabled) ***(1 of 2)***

| Access Type | Privilege Mode | TL | PSTATE.cle | D/UMMU.ie | Endianness | ASI Used | Resulting Address Type |
|---|---|---|---|---|---|---|---|
| Load, Store, Atomic Load-Store, or Prefetch with implicit ASI | NP | 0[1] | 0 | 0 | Big | ASI_PRIMARY | VA |
| | | | | 1 | Little | | |
| | | 0[1] | 1 | 0 | Little | ASI_PRIMARY_LITTLE | VA |
| | | | | 1 | Big | | |
| | P | 0 | 0 | 0 | Big | ASI_PRIMARY | VA |
| | | | | 1 | Little | | |
| | | 0 | 1 | 0 | Little | ASI_PRIMARY_LITTLE | VA |
| | | | | 1 | Big | | |
| | | 1-2[1] | 0 | 0 | Big | ASI_NUCLEUS | VA |
| | | | | 1 | Little | | |
| | | 1-2[1] | 1 | 0 | Little | ASI_NUCLEUS_LITTLE | VA |
| | | | | 1 | Big | | |
| | HP | any | 0 | — | Big | — | PA |
| | | | 1 | — | Little | | |

**TABLE 14-8** ASI Mapping for Data Accesses (D/UMMU Enabled) ***(2 of 2)***

| Access Type | Privilege Mode | TL | PSTATE.cle | D/UMMU.ie | Endianness | ASI Used | Resulting Address Type |
|---|---|---|---|---|---|---|---|
| Load, Store, Atomic Load-Store, or Prefetch alternate with ASI name *not* ending in _LITTLE | NP | 0[1] | any | 0 | Big[2] | Explicitly specified in instruction (see TABLE 14-11 on page 559)) | VA |
| | | | | 1 | Little[1] | | |
| | P | 0-2[1] | any | 0 | Big[1] | Explicitly specified in instruction (seeTABLE 14-11) | VA |
| | | | | 1 | Little[1] | | |
| | | 0-2[1] | any | 0 | Big | ASI_*REAL* ASI | RA |
| | | | | 1 | Little | | |
| | | 0-2[1] | any | any | Big | Nontranslating ASIs (seeTABLE 14-11) | — |
| | HP | any | any | 0 | Big | ASI_AS_IF_USER* or ASI_AS_IF_PRIV* | VA |
| | | | | 1 | Little | | |
| | | any | any | 0 | Big | ASI_*REAL* ASI | RA |
| | | | | 1 | Little | | |
| | | any | any | any | Big | Other translating ASI (seeTABLE 14-11) | PA |
| | | any | any | any | Big | Nontranslating ASI (seeTABLE 14-11) | — |
| Load, Store, Atomic Load-Store, or Prefetch alternate with ASI name ending in _LITTLE | NP | 0[1] | any | 0 | Little | Explicitly specified in instruction (seeTABLE 14-11) | VA |
| | | | | 1 | Big | | |
| | P | 0-2[1] | any | 0 | Little | Explicitly specified in instruction (seeTABLE 14-11)) | VA |
| | | | | 1 | Big | | |
| | | 0-2[1] | any | 0 | Little | ASI_*REAL* ASI | RA |
| | | | | 1 | Big | | |
| | HP | any | any | 0 | Little | ASI_AS_IF_USER* or ASI_AS_IF_PRIV* ASI | VA |
| | | | | 1 | Big | | |
| | | any | any | 0 | Little | ASI_*REAL* ASIs | RA |
| | | | | 1 | Big | | |

1. *MAXPTL* = 2 for UltraSPARC Architecture 2007 processors. Privilege mode operation is valid only for TL = 0, 1 or 2. Nonprivileged mode operation is valid only for TL = 0. See section 5.6.7 for details.

2. Accesses to nontranslating ASIs are always made in big endian mode, regardless of the setting of D/UMMU.ie. See *ASI Values* on page 429 for information about nontranslating ASIs.

The Context ID used by the data and instruction MMUs is determined according to TABLE 14-9. The Context ID selection is not affected by the endianness of the access. For a comprehensive list of ASI values in the ASI map, see Chapter 10, *Address Space Identifiers (ASIs)*.

**TABLE 14-9** IMMU, DMMU and UMMU Context ID Usage

| ASI Value | Context ID Register |
|---|---|
| `ASI_*NUCLEUS*` (any ASI name containing the string `"NUCLEUS"`) | Nucleus ($0000_{16}$, hard-wired) |
| `ASI_*PRIMARY*` (any ASI name containing the string `"PRIMARY"`) | All Primary Context IDs |
| `ASI_*SECONDARY*` (any ASI name containing the string `"SECONDARY"`) | All Secondary Context IDs |
| All other ASI values | (Not applicable; no translation) |

# 14.10 Translation

The translation operation performed by the MMU for a given access is determined by:

- whether the access is for instruction(s) or data
- the current privilege mode
- whether the MMU is enabled (which in turn is determined by one or more implementation-dependent enable bits in the MMU control register).
- in the case of a data access, which ASI is associated with the access

TABLE 14-10 describes the operation of the IMMU.

**TABLE 14-10** I/U-MMU Translation for Instruction Accesses

| HPSTATE.hpriv | I/UMMU Enable bit (x = don't care) | HPSTATE.red | Resulting I/U-MMU Translation |
|---|---|---|---|
| Don't Care | x | 1 | PA[1] |
| 1 | x | 0 | PA |
| 0 | 0 | 0 | RA → PA[2] |
| 0 | 1 | 0 | VA → PA[3] |

1. VA{55:0} is passed directly through to PA{55:0}
2. VA{55:0} is passed directly through to RA{55:0} and RA{55:0} is translated by the IMMU.
3. VA{63:0} is translated via the IMMU.

TABLE 14-11 lists the UltraSPARC Architecture 2007-defined ASIs and the translation operation performed when each is used, depending on the current privilege mode and whether the D/UMMU is enabled or not. See implementation-specific documentation regarding operation with implementation-specific ASIs.

**TABLE 14-11** D/UMMU Translation for Explicit ASIs *(1 of 5)*

Key: *priv_act* = *privileged_action* exception; *inv_asi* = *DAE_invalid_asi* exception; non-T = nontranslating ASI

| ASI Value (hex) | ASI Name | DMMU disabled | | | DMMU enabled | | |
|---|---|---|---|---|---|---|---|
| | | Current Mode | | | Current Mode | | |
| | | Non-privileged | Privileged | Hyper-privileged | Non-privileged | Privileged | Hyper-privileged |
| 00-03 | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* |
| 04 | ASI_NUCLEUS | *priv_act* | RA → PA | PA | *priv_act* | VA → PA | PA |
| 05–0B | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* |
| 0C | ASI_NUCLEUS_LITTLE | *priv_act* | RA → PA | PA | *priv_act* | VA → PA | PA |
| 0D–0F | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* |
| 10 | ASI_AS_IF_USER_PRIMARY | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 11 | ASI_AS_IF_USER_SECONDARY | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 12–13 | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* |
| 14 | ASI_REAL | *priv_act* | RA → PA | RA → PA | *priv_act* | RA → PA | RA → PA |
| 15 | ASI_REAL_IO | *priv_act* | RA → PA | RA → PA | *priv_act* | RA → PA | RA → PA |
| 16 | ASI_BLOCK_AS_IF_USER_PRIMARY | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 17 | ASI_BLOCK_AS_IF_USER_SECONDARY | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 18 | ASI_AS_IF_USER_PRIMARY_LITTLE | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 19 | ASI_AS_IF_USER_SECONDARY_LITTLE | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 1A–1B | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* |
| 1C | ASI_REAL_LITTLE | *priv_act* | RA → PA | RA → PA | *priv_act* | RA → PA | RA → PA |
| 1D | ASI_REAL_IO_LITTLE | *priv_act* | RA → PA | RA → PA | *priv_act* | RA → PA | RA → PA |
| 1E | ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 1F | ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 20 | ASI_SCRATCHPAD | *priv_act* | non-T[2] | non-T[2] | *priv_act* | non-T[2] | non-T[2] |
| 21 | ASI_MMU_CONTEXTID | *priv_act* | non-T[2] | non-T[2] | *priv_act* | non-T[2] | non-T[2] |
| 22 | ASI_TWINX_AS_IF_USER_PRIMARY (ASI_TWINX_AIUP) | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 23 | ASI_TWINX_AS_IF_USER_SECONDARY (ASI_TWINX_AIUS) | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 24 | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* |
| 25 | ASI_QUEUE | *priv_act* | non-T[2] | non-T[2] | *priv_act* | non-T[2] | non-T[2] |
| 26 | ASI_TWINX_REAL | *priv_act* | RA → PA | RA → PA | *priv_act* | RA → PA | RA → PA |
| 27 | ASI_TWINX_NUCLEUS (ASI_TWINX_N) | *priv_act* | RA → PA | PA | *priv_act* | VA → PA | PA |
| 28–29 | *Reserved* | *priv_act* | *inv_asi* | *inv_asi* | *priv_act* | *inv_asi* | *inv_asi* |

**TABLE 14-11** D/UMMU Translation for Explicit ASIs *(2 of 5)*

Key: *priv_act* = *privileged_action* exception; *inv_asi* = *DAE_invalid_asi* exception; non-T = nontranslating ASI

| ASI Value (hex) | ASI Name | DMMU disabled | | | DMMU enabled | | |
|---|---|---|---|---|---|---|---|
| | | Current Mode | | | Current Mode | | |
| | | Non-privileged | Privileged | Hyper-privileged | Non-privileged | Privileged | Hyper-privileged |
| 2A | ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE (ASI_TWINX_AIUP_L) | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 2B | ASI_TWINX_AS_IF_USER_SECONDARY_LITTLE (ASI_TWINX_AIUS_L) | *priv_act* | RA → PA | VA[6] → PA | *priv_act* | VA → PA | VA → PA |
| 2C | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* |
| 2D | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* | *priv_act*[1] | *(impl. dep.)* | *(impl. dep.)* |
| 2E | ASI_TWINX_REAL_LITTLE (ASI_TWINXREAL_L) | *priv_act* | RA → PA | RA → PA | *priv_act* | RA → PA | RA → PA |
| 2F | ASI_TWINX_NUCLEUS_LITTLE (ASI_TWINXNL) | *priv_act* | RA → PA | PA | *priv_act* | VA → PA | PA |
| 30 | ASI_AS_IF_PRIV_PRIMARY | *priv_act* | *priv_act* | VA[6] → PA | *priv_act* | *priv_act* | VA → PA |
| 31 | ASI_AS_IF_PRIV_SECONDARY | *priv_act* | *priv_act* | VA[6] → PA | *priv_act* | *priv_act* | VA → PA |
| 32–35 | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 36 | ASI_AS_IF_PRIV_NUCLEUS | *priv_act* | *priv_act* | VA[6] → PA | *priv_act* | *priv_act* | VA → PA |
| 37 | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 38 | ASI_AS_IF_PRIV_PRIMARY_LITTLE | *priv_act* | *priv_act* | VA[6] → PA | *priv_act* | *priv_act* | VA → PA |
| 39 | ASI_AS_IF_PRIV_SECONDARY_LITTLE | *priv_act* | *priv_act* | VA[6] → PA | *priv_act* | *priv_act* | VA → PA |
| 3A–3D | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 3E | ASI_AS_IF_PRIV_NUCLEUS_LITTLE | *priv_act* | *priv_act* | VA[6] → PA | *priv_act* | *priv_act* | VA → PA |
| 3F | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 40 | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 41 | ASI_CMP_SHARED | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 42–4B | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 4C | Error Status and Enable Registers | *priv_act*[3] | *priv_act*[3] | non-T[2] *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | non-T[2] *(impl. dep.)* |
| 4D–4E | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 4F | ASI_HYP_SCRATCHPAD | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 50 | ASI_IMMU | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 51–53 | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 54 | ASI_MMU | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 55 | ASI_ITLB_DATA_ACCESS_REG | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 56 | ASI_ITLB_TAG_READ_REG | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 57 | ASI_IMMU_DEMAP | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |

**TABLE 14-11** D/UMMU Translation for Explicit ASIs *(3 of 5)*

Key: *priv_act* = *privileged_action* exception; *inv_asi* = *DAE_invalid_asi* exception; non-T = nontranslating ASI

| ASI Value (hex) | ASI Name | DMMU disabled | | | DMMU enabled | | |
|---|---|---|---|---|---|---|---|
| | | Current Mode | | | Current Mode | | |
| | | Non-privileged | Privileged | Hyper-privileged | Non-privileged | Privileged | Hyper-privileged |
| 58 | ASI_DMMU (ASI_UMMU) | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 59–5B | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 5C | ASI_DTLB_DATA_IN_REG (ASI_UTLB_DATA_IN_REG) | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 5D | ASI_DTLB_DATA_ACCESS_REG (ASI_UTLB_DATA_ACCESS_REG) | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 5E | ASI_DTLB_TAG_READ_REG (ASI_UTLB_TAG_READ_REG) | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 5F | ASI_DMMU_DEMAP (ASI_UMMU_DEMAP) | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 60–62 | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 63 | ASI_CMT_PER_STRAND | *priv_act* | *priv_act* | non-T[2] | *priv_act* | *priv_act* | non-T[2] |
| 64–7F | *Implementation dependent (impl. dep. #29-V8)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* | *priv_act*[3] | *priv_act*[3] | *(impl. dep.)* |
| 80 | ASI_PRIMARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| 81 | ASI_SECONDARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| 82 | ASI_PRIMARY_NO_FAULT | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| 83 | ASI_SECONDARY_NO_FAULT | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| 84–87 | *Reserved* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* |
| 88 | ASI_PRIMARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| 89 | ASI_SECONDARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| 8A | ASI_PRIMARY_NO_FAULT_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| 8B | ASI_SECONDARY_NO_FAULT_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| 8C–BF | *Reserved* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* |
| C0 | ASI_PST8_PRIMARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| C1 | ASI_PST8_SECONDARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| C2 | ASI_PST16_PRIMARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| C3 | ASI_PST16_SECONDARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| C4 | ASI_PST32_PRIMARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| C5 | ASI_PST32_SECONDARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| C6–C7 | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |
| C8 | ASI_PST8_PRIMARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| C9 | ASI_PST8_SECONDARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| CA | ASI_PST16_PRIMARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |

**TABLE 14-11** D/UMMU Translation for Explicit ASIs *(4 of 5)*

Key: *priv_act* = *privileged_action* exception; *inv_asi* = *DAE_invalid_asi* exception; non-T = nontranslating ASI

| ASI Value (hex) | ASI Name | DMMU disabled | | | DMMU enabled | | |
|---|---|---|---|---|---|---|---|
| | | Current Mode | | | Current Mode | | |
| | | Non-privileged | Privileged | Hyper-privileged | Non-privileged | Privileged | Hyper-privileged |
| CB | ASI_PST16_SECONDARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| CC | ASI_PST32_PRIMARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| CD | ASI_PST32_SECONDARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| CE–CF | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |
| D0 | ASI_FL8_PRIMARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| D1 | ASI_FL8_SECONDARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| D2 | ASI_FL16_PRIMARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| D3 | ASI_FL16_SECONDARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| D4–D7 | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |
| D8 | ASI_FL8_PRIMARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| D9 | ASI_FL8_SECONDARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| DA | ASI_FL16_PRIMARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| DB | ASI_FL16_SECONDARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| DC–DF | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |
| E0 | ASI_BLK_COMMIT_PRIMARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| E1 | ASI_BLK_COMMIT_SECONDARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| E2 | ASI_TWINX_PRIMARY (ASI_TWINX_P) | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| E3 | ASI_TWINX_SECONDARY (ASI_TWINX_S) | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| E4–E9 | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |
| EA | ASI_TWINX_PRIMARY_LITTLE (ASI_TWINX_PL) | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| EB | ASI_TWINX_SECONDARY_LITTLE (ASI_TWINX_SL) | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| EC–EF | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |
| F0 | ASI_BLK_PRIMARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| F1 | ASI_BLK_SECONDARY | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| F2–F7 | *Reserved* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* | *inv_asi* |
| F8 | ASI_BLK_PRIMARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |
| F9 | ASI_BLK_SECONDARY_LITTLE | RA → PA | RA → PA | PA | VA → PA | VA → PA | PA |

**TABLE 14-11** D/UMMU Translation for Explicit ASIs *(5 of 5)*

Key: *priv_act* = *privileged_action* exception; *inv_asi* = *DAE_invalid_asi* exception; non-T = nontranslating ASI

| ASI Value (hex) | ASI Name | DMMU disabled | | | DMMU enabled | | |
|---|---|---|---|---|---|---|---|
| | | Current Mode | | | Current Mode | | |
| | | Non-privileged | Privileged | Hyper-privileged | Non-privileged | Privileged | Hyper-privileged |
| FA–FB | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |
| FC–FD | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |
| FE–FF | *Implementation dependent (impl. dep. #29-V8)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* | *(impl. dep.)* |

1. Since this is an implementation-dependent ASI, its behavior is implementation-dependent; however, an attempt in nonprivileged mode to access an ASI in the range $00_{16}$-$2F_{16}$ is normally expected to generate a *privileged_action* exception.
2. Nontranslating ASIs access hardware registers, not memory. The virtual address for a nontranslating ASI is the physical address; the MMU does not perform any translation.
3. Since this is an implementation-dependent ASI, its behavior is implementation-dependent; however, an attempt in nonprivileged or privileged mode to access an ASI in the range $30_{16}$-$7F_{16}$ is normally expected to generate a *privileged_action* exception.
6. This translation is expected to change to "RA→ PA" in the next generation of the architecture

## 14.10.1 MMU Behavior During Reset and Upon Entering `RED_state`

A power-on-reset of the virtual processor does not reset or initialize anything in the MMU, including TLBs and μTLBs.

When the virtual processor enters `RED_state`, the IMMU, DMMU, and UMMU Enable bits are set to 0.

While in hyperprivileged mode or `RED_state`, all instruction accesses are passed through without translation, using MMU bypass. See TABLE 14-10 on page 558.

While in hyperprivileged mode, all data access addresses with `ASI_PRIMARY`, `ASI_SECONDARY` or `ASI_NUCLEUS` are passed through without translation (MMU bypass). See *MMU Bypass* on page 564 for details. For examples, see TABLE 14-11 on page 559.

**Note** A power on reset or entry into `RED_state` does not affect the TLB contents. Before the MMUs are enabled, the operating system software must either explicitly demap all TLB entries, or write each entry (either with a valid TLB entry or an entry with the valid bit set to 0). The operation of the IMMU, DMMU, or UMMU in enabled mode is undefined if the TLB valid bits have not been explicitly initialized by software.

### 14.10.1.1 MMU Bypass

In a bypass access, the MMU 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-U3). The physical page attribute bits are set as shown in TABLE 14-12.

**TABLE 14-12** Bypass Attribute Bits

| | Page Attribute Bits | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Address Space[1]** | **cp** | **ie** | **cv** | **e** | **p** | **w** | **nfo** | **ep** |
| Memory Space | 1[2] | 0 | 0[2] | 0[2] | 0 | 1 | 0 | 1 |
| I/O Space | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

1. Address Space assignment is implementation dependent.
2. If implementation-dependent data cache control register DCUCR.cp and DCUCR.cv bits are implemented, TTE.cp = DCUCR.cp, TTE.cv = DCUCR.cv and TTE.e = complement of the DCUCR.cp; otherwise, the TTE.cp, TTE.cv, and TTE.e values are as indicated in the table.

For instruction accesses, the I/UMMU truncates all instruction accesses to the physical address size of the implementation, and passes the default physically cacheable bit (cp = 1) or implementation-dependent Data Cache Unit Control register cp bit to the cache system. The access does not generate an *IAE_** exception.

**SPARC V8 Compatibility Note** Since a virtual address is wider than a physical address on UltraSPARC Architecture implementations, there is no need to use multiple ASIs to fill in the high-order physical address bits (as was done in SPARC V8 machines).

### 14.10.1.2 MMU Disabled Behavior

**IMPL. DEP. #__:** The enable bit(s) in the MMU control register is (are) implementation dependent.

An MMU is considered "disabled" when its enable bit in the MMU control register is set to 0.

When in nonprivileged or privileged mode:

- If the IMMU is disabled (see TABLE 14-10 on page 558), addresses for instruction fetches are treated as real addresses and translated to physical addresses.
- If the DMMU is disabled (see TABLE 14-11 on page 559), addresses for data accesses are treated as real addresses and translated to physical addresses.

When disabled, the D/UMMU correctly performs all LDXA and STXA operations, and traps are signalled just as if the MMU were enabled. For instance, if DCUCR.cp is implemented, executing a nonfaulting load when the D/UMMU is disabled and DCUCR.cp = 0 causes a *DAE_side_effect_page* exception (since e = 1).

**IMPL. DEP. #117-V9:** Whether PREFETCH and nonfaulting loads always succeed when the DMMU is disabled is implementation dependent. See the PREFETCH instruction description in the Instructions chapter for more information. Non-faulting loads are described in section 9.6.

## 14.11 SPARC V9 "MMU Attributes"

The UltraSPARC Architecture MMU complies completely with the SPARC V9 "MMU Attributes" as described in Appendix F.3.2.

With regard to Read, Write and Execute Permissions, SPARC V9 says "An MMU may allow zero or more of read, write and execute permissions, on a per-mapping basis. Read permission is necessary for data read accesses and atomic accesses. Write permission is necessary for data write accesses and atomic accesses. Execute permission is necessary for instruction accesses. At a minimum, an MMU must allow for 'all permissions', 'no permissions', and 'no write permission'; optionally, it can provide 'execute only' and 'write only', or any combination of 'read/write/execute' permissions."

TABLE 14-13 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 14-13** MMU SPARC V9 Appendix F.3.2 Protection Mode Compliance

| Condition | | | | | Resultant Protection Mode |
|---|---|---|---|---|---|
| TTE in DMMU | TTE in IMMU | TTE in UMMU | ep Bit | Writable Attribute Bit | |
| Yes | No | Yes | 0 | 0 | Read-only[1] |
| No | Yes | N/A | 1 | N/A | Execute-only[1] |
| Yes | No | Yes | 0 | 1 | Read/Write[1] |
| Yes | Yes | Yes | 1 | 0 | Read-only/Execute |
| Yes | Yes | Yes | 1 | 1 | Read/Write/Execute |
| No | No | No | N/A | N/A | No Access |

1. These protection modes are optional, according to SPARC V9.

# 14.12 MMU Internal Registers and ASI Operations

This section describes the MMU registers and how they are accessed:

- Context ID
- Partition ID register
- MMU Real Range registers
- MMU Physical Offset registers
- TSB Configuration registers
- I/D/U TSB Pointer registers
- DMMU Synchronous Fault Address register
- I/D/U TLB Tag Access, Data In, Data Access, and Tag Read registers.
- I/D/U MMU TLB Tag Target registers.
- I/D/U MMU Demap
- Tablewalk Pending Registers

## 14.12.1 Accessing MMU Registers

All internal MMU registers can be accessed directly by the virtual processor through defined ASIs, using LDXA and STXA instructions. UltraSPARC Architecture-compatible processors do not require a MEMBAR #Sync, FLUSH, DONE, or RETRY instruction after a store to an MMU register for proper operation.

TABLE 14-14 lists the MMU registers and provides references to sections with more details.

**TABLE 14-14** MMU Internal Registers and ASI Operations

| IMMU ASI | D/UMMU ASI | VA{63:0} | Access | Register or Operation Name |
|---|---|---|---|---|
| $21_{16}$ | | $8_{16}$ | RW | Primary Context ID 0 register |
| — | $21_{16}$ | $10_{16}$ | RW | Secondary Context ID 0 register |
| $21_{16}$ | | $108_{16}$ | RW | Primary Context ID 1 register |
| — | $21_{16}$ | $110_{16}$ | RW | Secondary Context ID 1 register |
| $50_{16}$ | | $0_{16}$ | R | I-/D-TSB Tag Target registers |
| — | $58_{16}$ | $20_{16}$ | R | DMMU Synchronous Fault Address register |
| $50_{16}$ | $58_{16}$ | $30_{16}$ | RW | I/D/U-TLB Tag Access registers |

**TABLE 14-14** MMU Internal Registers and ASI Operations *(Continued)*

| IMMU ASI | D/UMMU ASI | VA{63:0} | Access | Register or Operation Name |
|---|---|---|---|---|
| $52_{16}$ | | $108..120_{16}$ | RW | MMU Real Range registers |
| | | $208..220_{16}$ | RW | MMU Physical Offset registers |
| $54_{16}$ | | $10..48_{16}$ | RW | MMU TSB Configuration registers |
| | | $50..68_{16}$(I)<br>$70..88_{16}$(D) | RW | MMU I/D/U-TSB Pointer registers |
| | | $90_{16}$ | RW | Tablewalk Pending Control register |
| | | $98_{16}$ | RW | Tablewalk Pending Status register |
| $54_{16}$ | $5C_{16}$ | $0_{16}$ | W | I/D/U-TLB Data In registers |
| $55_{16}$ | $5D_{16}$ | See Section 14.12.9 | RW | I/D/U-TLB Data Access registers |
| $56_{16}$ | $5E_{16}$ | See Section 14.12.9 | R | I/D/U-TLB Tag Read register |
| $57_{16}$ | $5F_{16}$ | See Section 14.12.11 | W | I/D/UMMU Demap Operation |
| $58_{16}$ | | $80_{16}$ | RW | Partition ID |

## 14.12.2 Context ID Registers

The MMU architecture supports multiple primary and secondary context IDs. The address assignment of the context IDs is shown in TABLE 14-15.

**TABLE 14-15** Context ID ASI Assignments

| Register | ASI | Virtual Address |
|---|---|---|
| Primary Context ID 0 | $21_{16}$ | $008_{16}$ |
| Primary Context ID 1 | $21_{16}$ | $108_{16}$ |
| Secondary Context ID 0 | $21_{16}$ | $010_{16}$ |
| Secondary Context ID 1 | $21_{16}$ | $110_{16}$ |

**Programming Note** Since only the existence of Primary Context ID 0 and Secondary Context ID 0 is guaranteed, it is recommended that these be used as private context IDs, with any remaining context IDs used for possibly shared context address spaces. For platforms that implement more than one primary context ID and one secondary context ID, privileged code must ensure that no more than one page translation is allowed to match at any time. An illustration of erroneous behavior is as follows:

1. An operating system constructs a mapping for virtual address *A* valid for context ID *P*;
2. it then constructs a mapping for address *A* for context ID *Q*.

By setting Primary Context ID 0 to *P* and Primary Context ID 1 to *Q*, both mappings would be active simultaneously, with conflicting translations for address *A*. Care must be taken not to construct such scenarios.

UltraSPARC Architecture processors must prevent errors or data corruption due to multiple valid translations for a given virtual address using different contexts. TLBs may need to detect this scenario as a multiple tag hit error and cause an *instruction_access_MMU_error* exception for an instruction access or a *data_access_MMU_error* exception for a data access.

The UltraSPARC Architecture supports up to two primary context IDs and two secondary context IDs, which are shared by the IMMU and D/UMMU. Primary Context ID 0 and Primary Context ID 1 are the primary context IDs, and a TLB entry for a translating primary ASI can match the context_id field with either Primary Context ID 0 or Primary Context ID 1 to produce a TLB hit. Secondary Context ID 0 and Secondary Context ID 1 are the Secondary Context IDs, and a TLB entry for a translating secondary ASI can match the context_id field with either Secondary Context ID 0 or Secondary Context ID 1 to produce a TLB hit.

**Compatibility Note** To maintain backward compatibility with software designed for a single primary and single secondary context ID, writes to Primary (Secondary) Context ID 0 also update Primary (Secondary) Context ID 1.

The Primary Context ID 0 and Primary Context ID 1 registers are illustrated in FIGURE 14-6, where pcontext is the context ID for the primary address space.

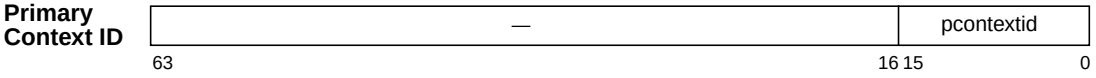



**FIGURE 14-6** IMMU, DMMU, and UMMU Primary Context ID 0/1

The Secondary Context ID 0 and Secondary Context ID 1 registers are illustrated in FIGURE 14-7, where scontextid is the context ID for the secondary address space.

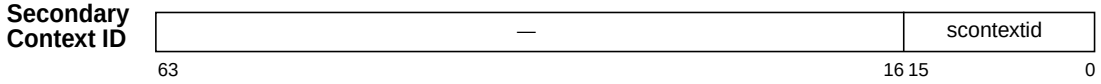



**FIGURE 14-7** D/UMMU Secondary Context ID 0/1

The Nucleus Context ID register is hardwired to zero, as illustrated in FIGURE 14-6.

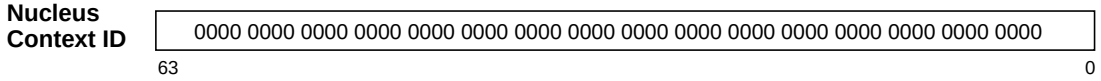



**FIGURE 14-8** IMMU, DMMU, and UMMU Nucleus Context ID

**IMPL. DEP. #415-S10:** The size of context ID fields in MMU context registers is implementation-dependent and may range from 13 to 16 bits.

## 14.12.3 Partition ID Register

ASI $58_{16}$ VA $80_{16}$

A partition ID is provided to allow multiple guest operating systems to share the same TLB. The partition ID register contents are compared in all TLB operations, such as demaps and translations, and are loaded into the PID field of the TLB tag during insertions. For more details on the partition ID, see *Real Address Translation* on page 537.

**IMPL. DEP. #416-S10:** The size of partition ID fields in MMU partition registers is implementation-dependent and must be large enough to uniquely encode the identities of all virtual processors that share the TLB.

The Partition ID register is defined in FIGURE 14-9, where partition_id is the 8-bit partition ID.

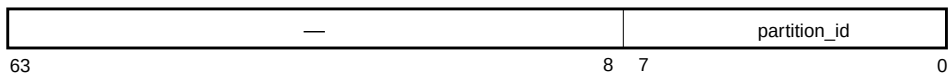



**FIGURE 14-9** Partition ID Register

## 14.12.4 MMU Real Range Registers

The Real Range registers specify the upper and lower bounds for real addresses.

- They consist of a single Real Range register that contains both the lower and upper bounds (which is possible if the real address field for each of the the upper and lower bounds is fewer than 32 bits)

**IMPL. DEP. #500-S30:** Whether each Real Range is specified in a single combined Real Range register or in a pair of Real Range registers (Real Range Lower and Real Range Upper) is implementation dependent.

An UltraSPARC Architecture 2007 processor must implement a minimum of four Real Range registers.

**IMPL. DEP. #501-S30:** The total number of (beyond the minimum of four) Real Range registers or Real Range Lower and Real Range Upper register pairs per virtual processor is implementation dependent.

The RPN-to-PPN translation associates each Real Range register Lower and Upper pair with its corresponding Physical Offset register. The RA-to-PA translation applies to TTEs from TSBs with the ra_not_pa = 1 in the TSB Configuration register, regardless of zero or nonzero context ID, as described in *Typical Hardware Tablewalk Sequence* on page 547.

If the enable field is 0 in the Real Range register, then the corresponding range base, bounds and offset are not used. If all are disabled, any hit in a TSB with the ra_not_pa bit = 1 in the TSB Configuration register results in a *instruction_invalid_TSB_entry* or *data_invalid_TSB_entry* exception.

**Note** | Ranges programmed into the Range Registers must not overlap. If overlapped, RPN to PPN translations have undefined behavior.

The ASI assignment for the Real Range registers is shown in TABLE 14-16.

**TABLE 14-16** MMU Real Range Register ASI and VA Assignments

| ASI | Register | Virtual Address |
|---|---|---|
| $52_{16}$ ASI_MMU_REAL | MMU Real Range 0 | $108_{16}$ |
| | MMU Real Range 1 | $110_{16}$ |
| | MMU Real Range 2 | $118_{16}$ |
| | MMU Real Range 3 | $120_{16}$ |

FIGURE 14-10 and TABLE 14-17 list the fields of the MMU Real Range register.

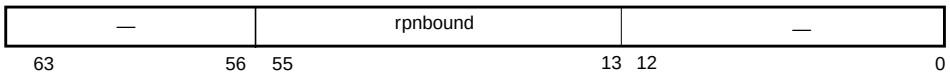



**FIGURE 14-10** MMU Real Range Register

**TABLE 14-17** MMU Real Range Register Format

| Bit | Field | Description |
|---|---|---|
| 63:56 | — | *Reserved* |
| 55:13 | rpnbound | RA{55:13}. Real address of the upper limit of the RPN range. |
| 12:0 | — | *Reserved* |

## 14.12.5 MMU Physical Offset Registers

**IMPL. DEP. #__:** The number of Physical Offset registers per virtual processor is implementation dependent.

The ASI assignment for the Physical Offset registers is shown in TABLE 14-18.

**TABLE 14-18** MMU_PHYSICAL_OFFSET_REGISTER_*n* ASI and VA Assignments

| ASI | Register | Virtual Address |
|---|---|---|
| $52_{16}$ (ASI_MMU_REAL) | MMU_PHYSICAL_OFFSET_REGISTER_0 | $208_{16}$ |
| | MMU_PHYSICAL_OFFSET_REGISTER_1 | $210_{16}$ |
| | MMU_PHYSICAL_OFFSET_REGISTER_2 | $218_{16}$ |
| | MMU_PHYSICAL_OFFSET_REGISTER_3 | $220_{16}$ |

The RPN-to-PPN translation associates each Real Range register pair (or Real Range register) with its corresponding Physical Offset register. The RA-to-PA translation applies to TTEs from TSBs with ra_not_pa = 1 in the TSB Config register, regardless of zero or nonzero context ID.

FIGURE 14-11 and TABLE 14-19 lists the fields of the MMU Physical Offset registers.

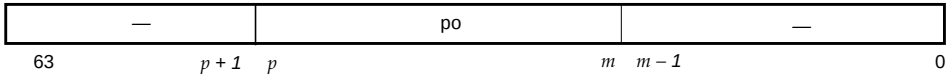



**FIGURE 14-11** MMU_PHYSICAL_OFFSET_REGISTER_*n*

**TABLE 14-19** MMU_PHYSICAL_OFFSET_REGISTER_*n* Format

| Bit | Field | Description |
|---|---|---|
| 63 : *p+1* | — | *Reserved* |
| *p* : *m* | po | Physical Offset.<br>Added to RA{*p*:*m*} of the request to generate PA{*p*:*m*}. |
| *m–1* : 0 | — | *Reserved* |

**IMPL. DEP. #__:** The bit positions (*p* and *m*, in FIGURE 14-11) of the ends of the po field are implementation dependent. The width of the physical address supported by the implementation determines *p*. The minimum physical offset supported determines *m*.

**Programming Note** | Hyperprivileged software must align the contents of the physical offset registers with a boundary of the largest page size that it is mapping.

## 14.12.6 TSB Configuration Registers

**IMPL. DEP. #__:** The number of TSB Configuration registers per virtual processor is implementation dependent.

The ASI and address assignments for the TSB Configuration registers are shown in TABLE 14-20.

**TABLE 14-20** MMU_[NON]ZERO_CONTEXT_ID_TSB_CONFIG_*n* Register ASI Assignments

| ASI | Register | Virtual Address |
|---|---|---|
| $54_{16}$ (ASI_MMU) | MMU_ZERO_CONTEXT_ID_TSB_CONFIG_0 | $10_{16}$ |
| | MMU_ZERO_CONTEXT_ID_TSB_CONFIG_1 | $18_{16}$ |
| | MMU_ZERO_CONTEXT_ID_TSB_CONFIG_2 | $20_{16}$ |
| | MMU_ZERO_CONTEXT_ID_TSB_CONFIG_3 | $28_{16}$ |
| | MMU_NONZERO_CONTEXT_ID_TSB_CONFIG_0 | $30_{16}$ |
| | MMU_NONZERO_CONTEXT_ID_TSB_CONFIG_1 | $38_{16}$ |
| | MMU_NONZERO_CONTEXT_ID_TSB_CONFIG_2 | $40_{16}$ |
| | MMU_NONZERO_CONTEXT_ID_TSB_CONFIG_3 | $48_{16}$ |

The Translation Storage Buffer (TSB) Configuration registers (MMU_[NON]ZERO_CONTEXT_ID_TSB_CONFIG_*n*) provide information for the hardware tablewalk and hardware formation of TSB pointers, to assist software in quickly handling TLB misses.

The register bits are illustrated in FIGURE 14-12 and described in TABLE 14-21.

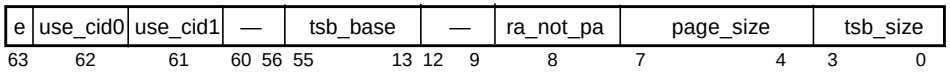



**FIGURE 14-12** MMU_[NON]ZERO_TSB_CONFIG_*n* Register Format

**TABLE 14-21** MMU_[NON]ZERO_CONTEXT_ID_TSB_CONFIG_*n* Register Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63 | e | RW | Enables the hardware tablewalk. If set to 1, hardware tablewalk will search this TSB on a TLB miss. |
| 62<br>61 | use_cid0<br>use_cid1 | RW | Controls whether hardware tablewalk checks the context ID value in the TTE from the TSB and what context ID value is written into the context_id field of the TTE in the TLB. If both bits are 0, hardware tablewalk compares the context ID in the TTE from the TSB with the context ID of the request and stores that context ID into the context_id field in the TLB if the TTE matches. If either bit is 1, hardware tablewalk ignores the context ID of the TTE from the TSB. If use_cid_0 = 1, the hardware tablewalk writes the value of Context ID register 0 into the context_id field of the TLB entry; otherwise, if use_cid_1 = 1, the hardware tablewalk writes the value of Context ID 1 register into the context_id field of the TLB entry.<br>**Note:** When the requesting context ID is zero (nucleus), hardware tablewalk ignores these bits. |
| 60:56 | — | R | *Reserved* |
| 55:13 | tsb_base | RW | Provides the base physical address of the Translation Storage Buffer. |
| 12:9 | — | R | *Reserved* |
| 8 | ra_not_pa | RW | If ra_not_pa = 1, RPN-to-PPN translation is enabled in hardware tablewalk.<br>**Note:** When hardware tablewalk is used for a TSB, the TSB may contain either real addresses or physical addresses, but not both. This bit should only be set to 1 when the TSB contains real addresses. |
| 7:4 | page_size | RW | Size of the pages mapped by the TTEs in the TSB. This page size is used to generate the TSB pointer. An attempt to store a reserved page size value in this field results in an *unsupported_page_size* exception. All TTEs in the TSB must be the size indicated in the TSB Configuration Register or greater. |
| 3:0 | tsb_size | RW | The tsb_size field provides the size of the TSB as follows:<br>• The number of entries in the TSB = $512 \times 2^{tsb_size}$.<br>• The number of entries in the TSB ranges from 512 entries at tsb_size = 0 (8-Kbyte TSB), to 16 M entries at tsb_size = 15 (256-Mbyte TSB). |

**Note** | Any update to the TSB Configuration register immediately affects the value returned from later reads of the TSB Pointer register.

## 14.12.7 I/D/U TSB Pointer Registers

The TSB Pointer registers are provided to allow software to easily access a TSB TTE given the VA that must be translated.

**IMPL. DEP. #__:** The number of TSB pointers provided is directly related to the number of TSB configuration registers. If an implementation has a unified MMU, it must provide one TSB pointer register per TSB configuration register. If an implementation has separate instruction and data MMUs, it must provide one instruction TSB pointer and one data TSB pointer per TSB configuration register.

The ASI assignment for the MMU I/D/U TSB Pointer registers are shown in TABLE 14-22.

**TABLE 14-22** MMU I/D/U TSB Pointer Register ASI Assignments

| ASI | Register | Virtual Address |
|---|---|---|
| $54_{16}$ (ASI_MMU) | IMMU TSB Pointer 0 | $50_{16}$ |
| | IMMU TSB Pointer 1 | $58_{16}$ |
| | IMMU TSB Pointer 2 | $60_{16}$ |
| | IMMU TSB Pointer 3 | $68_{16}$ |
| | DMMU TSB Pointer 0 or UMMU TSB Pointer 0 | $70_{16}$ |
| | DMMU TSB Pointer 1 or UMMU TSB Pointer 1 | $78_{16}$ |
| | DMMU TSB Pointer 2 or UMMU TSB Pointer 2 | $80_{16}$ |
| | DMMU TSB Pointer 3 or UMMU TSB Pointer 3 | $88_{16}$ |

The TSB Pointer registers are implemented as a reorder of the current data stored in the Tag Access register and the appropriate TSB Configuration register. If the Tag Access register or the TSB Configuration register is updated through a direct software write (via an STXA instruction), then the Pointer registers values will be updated as well.

The I/D/U-TSB Pointer registers are illustrated in FIGURE 14-13 and described in TABLE 14-23

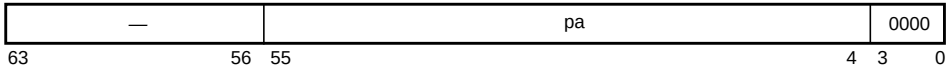



**FIGURE 14-13** I/D/U-TSB Pointer Registers

**TABLE 14-23** I/D/U-TSB Pointer Register Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:56 | — | R | *Reserved* |
| 55:4 | pa | R | The full physical address of the TTE in the TSB (PA{55:4}), as determined by the MMU hardware. |
| 3:0 | — | R | (always reads as 0) |

**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
- page_size

The TSB base address is provided in the TSB Configuration registers. Depending on the context ID that generated the TLB miss, an appropriate TSB Configuration register is selected. tsb_size and page_size is also supplied in the appropriate TSB Configuration register.

The virtual page number to be used for TSB pointer formation is in the I/D Tag Access register. If the Tag Access register holds a nonzero context, then the nonzero TSB Configuration register is used; if the Tag Access register holds a zero context, then the zero TSB configuration register is used.

The formula to generate the 56-bit pa field of the TSB Pointer register is as follows:

$$\begin{aligned} pa \leftarrow\ & tsb_base\{55 : (13+tsb_size)\} \\ & :: VA\{(21 + tsb_size + (3 \times page_size)) :: (13 + (3 \times page_size))\} \\ & :: 0000 \end{aligned}$$

# 14.12.8 Synchronous Fault Addresses

## 14.12.8.1 DMMU Synchronous Fault Address Register

**TABLE 14-24** DMMU_SYNCHRONOUS_FAULT_ADDRESS Register ASI Assignment

| Register | ASI | Virtual Address |
|---|---|---|
| DMMU_SYNCHRONOUS_FAULT_ADDRESS_REGISTER | $58_{16}$ | $20_{16}$ |

The Data Synchronous Fault Address register contains the virtual memory address of MMU exceptions detected on a data access. See TABLE 14-4 for details on which exceptions set the D-SFAR.

The D-SFAR register is illustrated in FIGURE 14-14

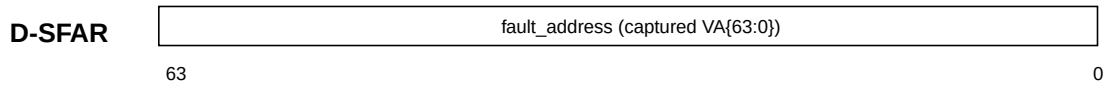



**FIGURE 14-14** MMU Data Synchronous Fault Address Register (D-SFAR)

The D-SFAR register is read-only; an attempt to write to the D-SFAR causes a *DAE_invalid_asi* exception.

**Note** | When PSTATE.am = 1, the upper 32 bits of the VA captured in this register will be zero.

### 14.12.8.2 Instruction Synchronous Fault Address

There is no IMMU Synchronous Fault Address register. Instead, software can derive the fault address for instruction access errors from the TPC register, as follows.

- For a *fast_instruction_access_MMU_miss* trap, TPC contains the virtual address that was not found in the IMMU TLB.
- For an *IAE_privilege_violation* trap, TPC contains the virtual address of the instruction in the privileged page that caused the exception.

## 14.12.9 I/D/U TLB Tag Access, Data In, Data Access, and Tag Read Registers

Access to the TLB is complicated because of the following needs:

- To provide an atomic write of a TLB entry (tag and data) that is larger than 64 bits
- To support an automatic, internal replacement index algorithm as well as to provide direct diagnostic access
- To allow multiple virtual processors that share the TLB to do a lock-free TLB update
- To have hardware assist in the TLB miss handler

TABLE 14-25 shows the effect of loads and stores on the Tag Access register and the TLB.

**TABLE 14-25** MMU TLB Access Summary

| Operation | to/from Register | Effect on MMU Physical Registers | | |
|---|---|---|---|---|
| | | **TLB tag array** | **TLB data array** | **Tag Access Register** |
| Load | Tag Read | No effect. Contents returned. | No effect | No effect |
| | Tag Access | No effect | No effect | No effect. Contents returned |
| | Data In | No effect (Trap with *DAE_invalid_asi)* ) | No effect (Trap with *DAE_invalid_asi)* ) | No effect (Trap with *DAE_invalid_asi)* ) |
| | Data Access | No effect | No effect. Contents returned. | No effect |
| Store | Tag Read | No effect (Trap with *DAE_invalid_asi*) | No effect (Trap with *DAE_invalid_asi)*) | No effect (Trap with *DAE_invalid_asi)* ) |
| | 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.

The hardware supports an autodemap function to handle the case in which two or more virtual processors sharing a TLB try to load the same translation into the TLB (for example, due to near-simultaneous TLB misses on the same page). A TLB replacement that attempts to add an already existing translation will cause the existing translation to be removed from the TLB.

**Notes** When a page is loaded into the TLB, autodemap will remove any TTE in the TLB that has the same virtual page number (VPN) and page size as the TTE being loaded. Autodemap will also remove any TTE in the TLB that maps a page that is larger than and overlaps the page of the new TTE. For example, an insertion of a 8-Kbyte page that lies within the virtual address range of a 64-Kbyte page will cause the 64-Kbyte page to be autodemapped.

Autodemap may or may not remove a TTE in the TLB that maps a page that is smaller than and is overlapped by the page of the new TTE. For example, an insertion of a 4-Mbyte page that overlaps the virtual address of one or more 64-Kbyte pages may or may not autodemap the overlapping 64-Kbyte pages. A subsequent multiple-hit error in the TLB could be generated as the result of a programming error that inserted a larger page in the TLB that overlapped smaller pages present in the TLB.

Autodemap only demaps entries that match on PID and real bit. If the real bit of the new TTE is 0, then only entries with matching context IDs will be demapped.

If a TLB replacement is attempted using a reserved page size value, an *unsupported_page_size* exception will be signalled instead.

If a TLB replacement is attempted with the valid bit equal to 0 (v = 0), the MMU will treat that the same as if the valid bit were 1 for purposes of allocating and overwriting a TLB entry and autodemapping matching pages, and the entry will be written into the TLB with the v bit set to 0.

The autodemap mechanism does not look at TTE data; it looks only at VA or RA, partition ID, real bit (r), and context ID if r = 0.

### 14.12.9.1 I/D/U MMU TLB Tag Access Registers

**TABLE 14-26** I/D/UMMU_TLB Tag Access Register ASI Assignments

| Register | ASI | Virtual Address |
|---|---|---|
| IMMU TLB Tag Access register | $50_{16}$ | $30_{16}$ |
| DMMU TLB Tag Access register or UMMU TLB Tag Access register | $58_{16}$ | $30_{16}$ |

In each MMU, the Tag Access registers are used as a temporary buffer for writing the TLB Entry tag information. The Tag Access registers hold the tag portion, and the Data In or Data Access register holds the data being accessed.

The IMMU ASIs are used for the ITLB, the DMMU ASIs for the DTLB, and the UMMU ASIs for the UTLB.

To support a 16-bit context ID value, two registers are assigned; a Lower Tag Access register and an Upper Tag Access register. Implementations with a context_id field less than 14 bits can use a single Tag Access register.

The Tag Access registers are updated during any of the following operations:

1. **When the MMU signals a trap due to a miss, exception, or protection violation.** The MMU hardware automatically writes the missing VA and the appropriate context (`ASI_PRIMARY_CONTEXTID_0` for primary context accesses, `ASI_SECONDARY_CONTEXTID_0` for secondary context accesses, `ASI_NUCLEUS_CONTEXTID` for other accesses) into the Tag Access registers to facilitate formation of the TSB Tag Target register. See TABLE 14-4 on page 550 for the Tag Access register update policy.

2. **An ASI write to the Tag Access registers.** 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 registers, but typically the value written into the Tag Access registers by the MMU hardware is appropriate.

3. **An I-TLB or D-TLB load by the hardware tablewalker.**

> **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 Upper Tag Access register fields are illustrated in FIGURE 14-15 and defined in TABLE 14-27.

**Upper Tag Access**

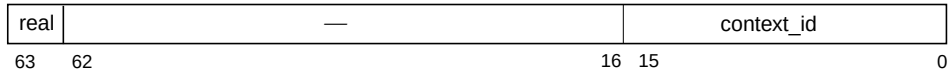



**FIGURE 14-15** I/D MMU TLB Upper Tag Access Register

**TABLE 14-27** TLB Upper Tag Access Register Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63 | real | RW | Real bit. If real = 1, this entry maps real-to-physical addresses, and the context_id field is ignored. If real = 0, this entry maps virtual-to-physical addresses, using the context_id field. |
| 62:16 | — | R | *Reserved* |
| 15:0 | context_id | RW | The 16-bit context ID. This field reads 0 when there is no associated context with the access. For real translations, the context_id field is undefined. (impl. dep. #415-S10) |

The TLB Lower Tag Access register fields are illustrated in FIGURE 14-16 and described in TABLE 14-28.

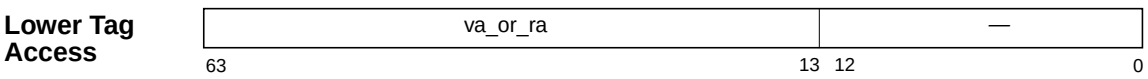



**FIGURE 14-16** I/D MMU TLB Lower Tag Access Registers

**TABLE 14-28** I/D MMU TLB Lower Tag Access Register Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:13 | va_or_ra | RW | If real = 0, this is a 51-bit virtual page number; if real = 1, this is a 43-bit real page number and software should insure that bits 63:56 are zeroes. |
| 12:0 | — | R | *Reserved* |

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 is obtained from the store data, and the TLB entry tag is obtained from the current contents of the TLB Tag Access registers as well as the Partition ID register.

## 14.12.9.2 I/D/UMMU TLB Data In Register

**TABLE 14-29** I/D/U MMU TLB Data In Register ASI Assignments

| Register | ASI | VA |
|---|---|---|
| IMMU_TLB_DATA_IN | $54_{16}$ | $0_{16}$ |
| DMMU_TLB_DATA_IN | $5C_{16}$ | $0_{16}$ |
| UMMU_TLB_DATA_IN | $5C_{16}$ | $0_{16}$ |

The TLB Data In register is used for TLB miss and TSB-miss handler automatic replacement writes.

The TLB Data In register uses the TTE format shown in TABLE 14-2 on page 540, except that bits 55:13 (taddr) contain a physical (not real) address. Refer to the description of the TTE data in *TSB TTE Bit Description* on page 540 for a complete description of the data fields.

ASI loads from the TLB Data In register are not supported and cause a*DAE_invalid_asi* exception. An attempt to write to the TLB Data In register with a nonzero VA address causes a *DAE_invalid_asi* exception.

An ASI store to the TLB Data In register initiates an automatic atomic replacement of the TLB entry pointed to by an implementation-dependent replacement algorithm. The TLB entry updates its tag with the contents of the Tag Access register, including the real bit, and the Partition ID register; the data for the TLB entry comes from the data stored to the Data In register.

**IMPL. DEP. #234-U3:** The replacement algorithm of a TLB entry is implementation dependent in UltraSPARC Architecture 2007.

## 14.12.9.3 I/D/U MMU TLB Data Access Register

**TABLE 14-30** I/D/U MMU TLB Data Access Register ASI Assignments

| Register | ASI |
|---|---|
| IMMU_TLB_DATA_ACCESS | $55_{16}$ |
| DMMU_TLB_DATA_ACCESS | $5D_{16}$ |
| UMMU_TLB_DATA_ACCESS | $5D_{16}$ |

The TLB Data Access register is used for operating system and diagnostic directed writes (writes to a specific TLB entry). It is also used to read entries from the TLB data array.

The format of the TLB Data Access register virtual address is shown in FIGURE 14-17; the fields are described in TABLE 14-31.

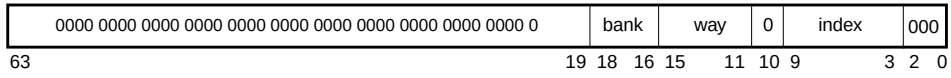



**FIGURE 14-17** I/D/U MMU TLB Data Access Virtual Address Format

**TABLE 14-31** I/D/U MMU TLB Data Access Register Field Description

| Bit | Field | Description |
|---|---|---|
| 18:16 | bank | The TLB Entry *bank* to be accessed.<br>**IMPL. DEP. #__:** Whether this field is supported is implementation dependent; otherwise, a nonzero value causes a *DAE_invalid_asi* exception. |
| 15:11 | way | The TLB Entry *way* to be accessed.<br>**IMPL. DEP. #__:** Whether this field is supported is implementation dependent; otherwise, a nonzero value causes a *DAE_invalid_asi* exception. |
| 9:3 | index | The TLB Entry *index* to be accessed.<br>**IMPL. DEP. #** Up to 128-TLB entries are supported. Implementations with less than 128-TLB entries should take a *DAE_invalid_asi* exception if unsupported bits are nonzero. |

The TLB Data Access register uses the TTE format shown in TABLE 14-2 on page 540. Refer to the description of the TTE data in *TSB TTE Bit Description* on page 540 for a complete description of the data fields.

When the data are stored in a TLB entry, bits 55:13 contain a physical (not real) address.

An ASI load from the TLB Data Access register initiates an internal read of the data portion of the specified TLB entry.

## 14.12.9.4 I/D/UMMU TLB Tag Read Register

**TABLE 14-32** I/D/U MMU TLB Tag Read Register ASI Assignments

| Register | ASI |
|---|---|
| IMMU_TLB_UPPER_TAG_READ | $56_{16}$ |
| DMMU_TLB_UPPER_TAG_READ | $5E_{16}$ |
| UMMU_TLB_UPPER_TAG_READ | $5E_{16}$ |
| IMMU_TLB_LOWER_TAG_READ | ? |
| DMMU_TLB_LOWER_TAG_READ | ? |
| UMMU_TLB_LOWER_TAG_READ | ? |
| IMMU_TLB_TAG_READ | $56_{16}$ |
| DMMU_TLB_TAG_READ | $5E_{16}$ |
| UMMU_TLB_TAG_READ | $5E_{16}$ |

The format for the Tag Read register virtual address is illustrated in FIGURE 14-18 and described in TABLE 14-33.

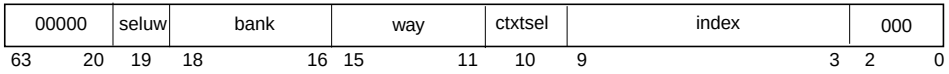



**FIGURE 14-18** I/D/U MMU TLB Tag Read Virtual Address Format

**TABLE 14-33** I/D/U MMU TLB Tag Read Register Field Description

| Bit | Field | Description |
|---|---|---|
| 19 | seluw | Select upper tag read register; when seluw = 1, select Upper Tag Read register; when seluw = 0, select Lower Tag Read register.<br>**IMPL. DEP. #__**: Whether this field is supported is implementation dependent; otherwise, a nonzero value causes a *DAE_invalid_asi* exception. |
| 18:16 | bank | The TLB Entry bank to be accessed.<br>**IMPL. DEP. #__:** Whether this field is supported is implementation dependent; otherwise, a nonzero value causes a *DAE_invalid_asi* exception. |
| 15:11 | way | The TLB Entry way to be accessed.<br>**IMPL. DEP. #__:** Whether this field is supported is implementation dependent; otherwise, a nonzero value causes a *DAE_invalid_asi* exception. |
| 10 | ctxtsel | If 0, context A is read out in the context field. If 1, context B is read out in the context field.<br>**Note:** A model's TLB may store duplicate copies of the context field, and ctxtsel allows software to examine both copies.<br>**IMPL. DEP. #__:** Whether this field is supported is implementation dependent; otherwise, a nonzero value causes a *DAE_invalid_asi* exception. |
| 9:3 | index | The TLB Entry index to be accessed.<br>**IMPL. DEP. #**Up to 128-TLB entries are supported. Implementations with less than 128-TLB entries should take a *DAE_invalid_asi* exception if unsupported bits are nonzero. |

To support the full range of address, context, and partition, two registers are defined, the Upper Tag Read register and the Lower Tag Read register. Selection is provided by bit 19 of the virtual address.

**IMPL. DEP. #___**: It is implementation dependent on whether the Tag Read registers are implemented as one register or two, depending on the number of context and partition bits supported.

The data format for the Upper Tag Read register is shown in FIGURE 14-19 and described in TABLE 14-34.

### Upper Tag Read

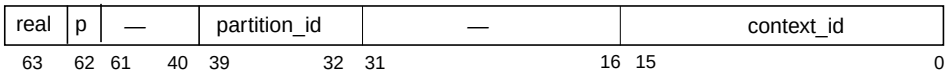



**FIGURE 14-19** I/D/U MMU TLB Upper Tag Read Register

**TABLE 14-34** I/D/U MMU TLB Upper Tag Read Register Field Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63 | real | R | Real bit. If real = 1, this entry maps real to physical, and the context_id field is ignored. If real = 0, this entry maps virtual to physical, using the context_id field. |
| 62 | p | R | Parity for the tag entry. Parity is generated across partition_id, real, VA, and the context_id field. |
| 61:40 | — | R | *Reserved* |
| 39:32 | partition_id | R | 8-bit partition ID used for all translation matches |
| 31:16 | — | R | *Reserved* |
| 15:0 | context_id | R | The 16-bit context ID. This field reads 0 when there is no associated context ID with the access. |

The MMU TLB Lower Tag Read register fields are illustrated in FIGURE 14-20 and described in TABLE 14-35.

**Lower Tag Read**

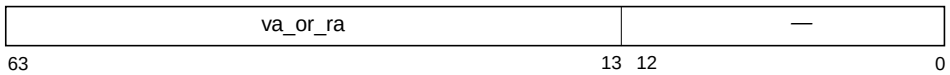



**FIGURE 14-20** I/D/U MMU TLB Lower Tag Read Registers

**TABLE 14-35** I/D/U MMU TLB Lower Tag Read Register Field Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:13 | va_or_ra | RW | If real = 0, this is a 51-bit virtual page number; if real = 1 , this is a 43-bit real page number. |
| 12:0 | — | R | *Reserved* |

Some implementations can fit the tag read information in a single Tag Read register. The data format is shown in FIGURE 14-21 and described in TABLE 14-36.

**Tag Read**

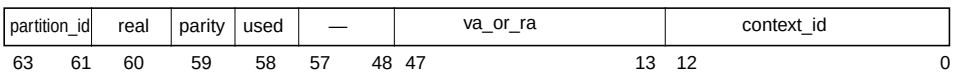



**FIGURE 14-21** Single I/D/U MMU TLB Tag Read Register

**TABLE 14-36** Single I/D/U MMU TLB Tag Read Bit Register Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:61 | partition_id | R | 3-bit partition ID. |
| 60 | real | R | Real. If 1, identifies an RA-to-PA translation instead of a VA-to-PA translation. |
| 59 | parity | R | Parity for the tag entry. Parity is generated across the PID, Real, va{47:13}, and the context field. |
| 58 | used | R | Used bit for replacement. |

**TABLE 14-36** Single I/D/U MMU TLB Tag Read Bit Register Description *(Continued)*

| Bit | Field | Type | Description |
|---|---|---|---|
| 57:48 | — | R | *Reserved* |
| 47:13 | va_or_ra | R | If real = 0, this is a 35-bit virtual page number; if real = 1, this is a 35-bit real page number. |
| 12:0 | context_id | R | The 13-bit context ID. |

## 14.12.10 I/D/UMMU TLB Tag Target Registers

**TABLE 14-37** I/D/U MMU_Tag Target Register ASI Assignments

| Register | ASI | VA |
|---|---|---|
| IMMU_TLB_TAG_TARGET | $50_{16}$ | $0_{16}$ |
| DMMU_TLB_TAG_TARGET | $58_{16}$ | $0_{16}$ |
| UMMU_TLB_TAG_TARGET | $58_{16}$ | $0_{16}$ |

The I/D/U MMU TLB Tag Target registers are bit-shifted versions of the data stored in the corresponding MMU TLB Tag Access registers. Since the appropriate MMU TLB Tag Access register is updated on I- or D-TLB misses, the appropriate MMU TLB Tag Target registers appear to software to also be updated on an I or D TLB miss. An attempt to write to this register results in a *DAE_invalid_asi* exception. The MMU TLB Tag Target registers are illustrated in FIGURE 14-22 and described in TABLE 14-38.

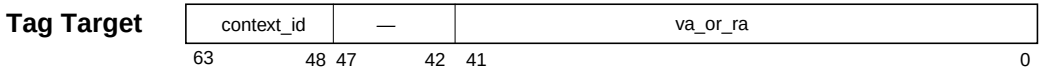



**FIGURE 14-22** I/D/U MMU TLB Tag Target Register format

**TABLE 14-38** I/D/U MMU TLB Tag Target Register Description

| Bit | Field | Type | Description |
|---|---|---|---|
| 63:48 | context_id | R | The context ID associated with the missing virtual address. For real translations, the context field is undefined. |
| 47:42 | — | R | *Reserved* |
| 41:0 | va_or_ra | R | VA{63:22} or RA{63:22}. |

## 14.12.11 I/D/UMMU Demap

Demap is an MMU *operation* (as opposed to an MMU register). The purpose of demap is to remove zero or more TTE entries from TLBs.

Demap is initiated by execution of a STXA instruction using one of the ASIs indicated in TABLE 14-39.

**TABLE 14-39** I/D/U MMU Demap ASI Assignments

| Operation | ASI | VA |
|---|---|---|
| IMMU Demap | $57_{16}$ | $0_{16}$ † |
| DMMU Demap | $5F_{16}$ | $0_{16}$ † |
| UMMU Demap | $5F_{16}$ | $0_{16}$ † |

† Virtual address is zero for implementations that encode Demap parameters in data, but nonzero for those that encode Demap parameters in the VA. See processor-specific documentation for details.

The format of data written by the STXA instruction in an UltraSPARC Architecture 2007 demap operation is illustrated in FIGURE 14-23 and described in TABLE 14-41.

An attempt to read (load) from a Demap ASI causes a *DAE_invalid_asi* exception.

Hardware should provide proper synchronization within each virtual processor. Previous memory accesses from the same virtual processor should have completed their TLB accesses, and the result of the demap should be visible to all instructions following (in program order) after the demap operation.

**Programming Note** A demap operation does not invalidate the TSB in memory. Software must modify the appropriate TTEs in the TSB *before* initiating a demap operation to remove those TTEs from TLB(s).

An MMU demap operation only affects TLB contents; it does not modify any other virtual processor registers.

Four types of demap operation are provided:

- Demap Page — There are two forms of the Demap Page operation:
  - If r = 1 in the Demap Page operation, it removes any TLB entry with TTE.r = 1 and that matches the specified virtual page number and Partition ID (Context ID is ignored in the match).
  - If r = 0 in the Demap Page operation, it removes any TLB entry with TTE.r = 0 and that matches the specified virtual page number, Partition ID, *and* Context ID.

  Which virtual page offset bits participate in the TLB entry match depends on the page size; see TABLE 14-40. (This is also true for a translation match.)

**TABLE 14-40** Virtual Page Offset Bits

| Page Size | Virtual Page Offset Bits that participate in match |
|---|---|
| 64 Kbytes | 15:13 |
| 512 Kbytes | 18:13 |
| 4 Mbytes | 21:13 |
| 32 Mbytes | 24:13 |
| 256 Mbyte | 27:13 |
| 2 Gbyte | 30:13 |
| 16 Gbytes | 33:13 |

**Note** | Each Demap Page operation removes zero, one, or more TLB entries. If no entries match, Demap Page removes no entries. If there is an error condition in the TLB that causes multiple TLB entries to match, Demap Page will remove all matching entries.

**Note** | If smaller pages are overlapped by a larger page, an attempt to demap a TTE for the larger page may or may not demap any of the TTEs for the smaller pages.

- Demap Context — Removes any TLB entry that matches the specified context ID and partition ID, and is a virtual-to-real translation entry (TTE.r = 0), regardless of virtual page number.

  Demap Context removes zero or more TLB entries. Demap Context will never demap a real-to-physical translation entry (r = 1).

  **Note** | The IMMU does not support a Demap Context operation with the context = $01_2$ (Secondary Context 0) encoding; an attempt to use it will cause the demap request to be ignored.

- Demap All — Removes any TLB entry that matches the partition ID. Context ID, real bit, and virtual page number are ignored when matching entries during a Demap All operation.
- Demap Real — Removes any TLB entry with its real bit (r) set to 1 that matches the partition ID, regardless of its context ID or virtual page number. Only real-to-physical translation entries in the TLB (TTE.r = 1) are demapped.

**Demap Data Format**

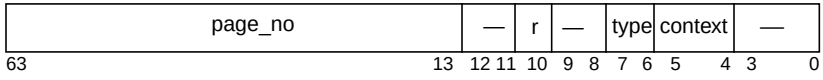



**FIGURE 14-23** MMU Demap Operation Data Format

**TABLE 14-41** Demap Data Format

| Bit(s) | Field | Type | Description |
|---|---|---|---|
| 63:13 | page_no | W | The virtual or real page number of the TTE to be removed from the TLB for Demap Page (VA{63:13} or RA {63:13}). This field is not used by the MMU for the Demap Context, Demap All, or Demap Real operations. |
| 12:11 | — | | This field is ignored by the MMU and should always be supplied as zeroes by software. |
| 10 | r | W | Selects between demapping real translations (r = 1) or virtual translations (r = 0). Valid for Demap Page only. |
| 9:8 | — | | This field is ignored by the MMU and should always be supplied as zeroes by software. |
| 7:6 | type | W | The type of demap operation:<br>**type** — **Demap Operation**<br>$00_2$ — Demap Page—see page 586<br>$01_2$ — Demap Context—see page 587<br>$10_2$ — Demap All—see page 587<br>$11_2$ — Demap Real—see page 587 |
| 5:4 | context | W | Context selection:<br>**context** — **Context Used in Demap**<br>$00_2$ — Primary 0<br>$01_2$ — Secondary 0 (DMMU only)<br>$10_2$ — Nucleus<br>$11_2$ — *Reserved*<br>For Demap Real and Demap All operations, the value of context is ignored.<br>For an IMMU Demap Context operation, context = $01_2$ (referencing the Secondary Context 0 register) is not supported; using context = $01_2$ in that situation causes the demap operation to be ignored.<br>For a Demap Page or Demap Context operation, use of the reserved value context = $11_2$ causes the demap operation to be ignored. |
| 3:0 | — | | This field is ignored by the MMU and should always be supplied as zeroes by software. |

## 14.12.12 Tablewalk Pending Registers

The Tablewalk Pending registers provide in-progress bits for hardware and software TLB loaders.

**TABLE 14-42** Tablewalk Pending Register ASI Assignments

| Register | ASI | VA |
|---|---|---|
| ASI_TABLEWALK_PENDING_CONTROL | $54_{16}$ | $90_{16}$ |
| ASI_TABLEWALK_PENDING_STATUS | $54_{16}$ | $98_{16}$ |

### 14.12.12.1 Tablewalk Pending Control Register

**FIGURE 14-24** Tablewalk Pending Control Register

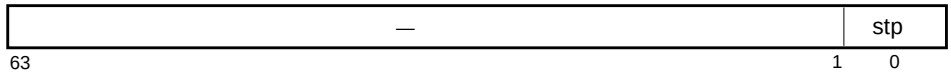



**TABLE 14-43** ASI Tablewalk Pending Control Register Description

| Bit(s) | Field | Type | Description |
|---|---|---|---|
| 63:1 | — | RW | *Reserved* |
| 0 | stp | RW | Indicates whether a software tablewalk is in progress. |

Note that this register is maintained completely by software.

**Programming Note** One Tablewalk Pending Control register is implemented per virtual processor. This register provides a mechanism whereby software tablewalk can record its status. Minimally, software tablewalk should write a 1 to stp before it fetches a TTE from a TSB, and should write a 0 to stp after it has written the TTE to the TLB or has determined that the TTE will not be written to the TLB.

### 14.12.12.2 Tablewalk Pending Status Register

**FIGURE 14-25** Tablewalk Pending Status Register

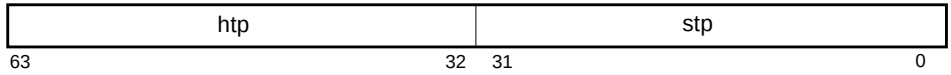



**TABLE 14-44** ASI Tablewalk Pending Status Register Description

| Bit(s) | Field | Type | Description |
|---|---|---|---|
| 63:32 | htp | RW | One bit of htp is required per supported concurrent hardware tablewalk. These concurrent hardware tablewalks may have implementation-dependent relationships to the strands. That is, a strand may be statically associated with one or more htp bits, or the strand to htp bit association may be dynamic and change over time. |
| 31:0 | stp | RW | Each bit corresponds to the stp bit in the Tablewalk Pending Control register for the corresponding virtual processor. |

**IMPL. DEP. #____**: The number of Tablewalk Pending Status registers present is implementation dependent, but at least one Tablewalk Pending Status register must be provided per IMMU/DMMU pair (or per UMMU).

This register allows software to identify when in-progress tablewalks have completed. Software can invalidate a TTE in a TSB and then poll this register to identify tablewalks that may be temporarily caching the TTE that has been invalidated. The bits that are 1 on this initial poll indicate pending tablewalks. A bit initially sampled as 1 and later sampled as 0 indicates an in progress tablewalk has completed. Once each of the bits that were initially 1 have been subsequently polled as 0, all tablewalks that were in-progress when the initial poll was taken have completed.

Because successive hardware tablewalks can set bits of htp again, it is possible for software to undersample. That is, polling software can miss a 1-to-0 transition if hardware clears and sets the bit between adjacent software polls.

**Implementation Note** Hardware must ensure that software can reliably detect the 1-to-0 transition of the htp bits. To reduce the likelihood of undersampling, an implementation may implement more than one htp bit per supported concurrent hardware tablewalk. Hardware may use these extra bits in round-robin fashion (or some other appropriate fashion) to indicate new tablewalks, thus extending the window of time provided for a software sample of the zeroed bits of the recently completed tablewalks. In such an implementation, any bit position in the HTP field can be used to indicate a hardware tablewalk. It is recommended that the same bits not be used successively, since the barrier determination operates only on the state change of individual bits.

# 14.13 Translation Lookaside Buffer Hardware

This section briefly describes the TLB hardware. For more detailed information, refer to the processor implementation documents or the corresponding microarchitecture specification.

## 14.13.1 TLB Operations

The TLB supports the following operations:

- **VA → PA translation.** The TLB receives a virtual address, a partition ID, and context ID as input and produces a physical address and page attributes as output.
- **RA → PA translation.** The TLB receives a real address and a partition ID 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 or real address, a partition ID, a "real" bit, a context ID (if the "real bit" is zero), and a demap type as input and sets the valid bit to zero for any matching 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 multiple reads. See *I/D/U TLB Tag Access, Data In, Data Access, and Tag Read Registers* on page 576.)
- **Write operation.** The TLB simultaneously writes the Tag and Data portion of the specified entry or the entry given by the implementation-dependent replacement policy.

CHAPTER 15

# Chip-Level Multithreading (CMT)

An UltraSPARC Architecture 2007 processor 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 chip-level multithreaded processors (CMTs). It addresses issues common to CMT processors, regardless of the microarchitecture of the individual physical processor cores, in the following sections:


- **Overview of CMT** on page 593.
- **Accessing CMT Registers** on page 597.
- **CMT Registers** on page 600.
- **Disabling and Parking Virtual Processors** on page 604.
- **Reset and Trap Handling** on page 614.
- **Error Handling in CMT Processors** on page 617.
- **Additional CMT Software Interfaces** on page 622.
- **Performance Issues for CMT Processors** on page 623.
- **Recommended Subset for Single-Strand Processors** on page 623.
- **Machine State Summary** on page 625.


## 15.1 Overview of CMT

A broad range of designs may fall under the definition of CMT. The interface specified here is intended to provide a set of common behaviors to enable operating system software and other privileged software to be common across UltraSPARC Architecture 2007 processors. This interface is not complete, as a range of implementation dependent features will exist to configure and control these processors.

The CMT programming model describes a set of privileged registers that are used for identification and configuration of CMT processors. Equally important, the CMT programming model describes certain behavior that is common across CMT implementations. The set of registers and the common behavior are covered in the following sections, grouped by topic.

UltraSPARC Architecture 2007 processors that are not CMT processors (are single-threaded) should implement a subset of the CMT interface. This enables those virtual processors to be more easily integrated into products that may also contain CMT processors and also enables more consistent software to be deployed across future products. See *Recommended Subset for Single-Strand Processors* on page 623 for additional information on non-CMT processor implementations.

## 15.1.1 CMT Definition

An UltraSPARC Architecture 2007 CMT processor is defined by its externally-visible nature and not by its internal organization. The following section gives some background terminology, followed by a description of the CMT definition.

### 15.1.1.1 Background Terminology

The following definitions expand on the abbreviated definitions provided in Chapter 2, *Definitions*.

**Thread.** Historically, the term *thread* is overused and ambiguous; software and hardware have used it differently. From a software (operating system) perspective, the term "thread" refers to an entity that:

- Can be executed on underlying hardware
- Is scheduled
- May or may not be actively running on hardware at any given time
- May migrate around the hardware of a system.

From the hardware perspective, the term "multithreaded processor" refers to a processor that can run multiple software threads simultaneously.

To avoid confusion, the term "thread" in UltraSPARC Architecture 2007 is used exclusively in the manner that it is used by software (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* refers to the state that hardware must maintain in order to execute a software thread. Specifically, a "strand" is the software-visible architected state (PC, NPC, general-purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.) of a thread plus 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 an operating system can schedule on that processor at any given time.

**Pipeline.** The term *pipeline* refers to an execution pipeline. It is a loose term for the basic collection of hardware needed to execute instructions. A pipeline may be used by one or more strands, in order to execute instruction from one or more threads.*Synonym:* **microcore**.

**Physical Core.** The term *physical processor core*, or just *physical core*, is similar to the term “pipeline” but represents a broader collection of hardware. A physical core includes one or more execution pipelines and associated structures, such as caches, that are required for executing 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 by time-multiplexing resources, partitioning resources, or any combination thereof.

The delineations among the terms strand, pipeline, and physical core are not precise. Among different microarchitecture organizations the scope of the terms may vary. In general, in a specific 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 more distant levels of the memory hierarchy and the system interface unit. A physical core will contain a defined number of strands, that is, a maximum number of software threads that may be 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. Physically, 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 (for example, an UltraSPARC I processor) would contain a single physical core which is single-stranded. A processor that follows the academic model of simultaneous multithreading (SMT) would contain a single physical core, where that physical core supports multiple strands in order to execute multiple simultaneous threads (multi-stranded physical core). A processor that follows the academic model of a chip multi-processor (CMP) would be a processor with multiple physical cores, each supporting only a single strand. A processor may also contain multiple physical cores, where each physical core is multi-stranded.

**Virtual Processor.** The term *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 multiple physical cores, each with multiple strands, may exist. In most respects a virtual processor appears to the system and to operating system

software as a processing unit equivalent to a traditional single-stranded processor (as in UltraSPARC I). Each virtual processor is capable of having interrupts directed specifically to it. At any given time, an operating system can have a different thread scheduled on each virtual processor.

The UltraSPARC Architecture 2007 CMT architecture (software interface) described in this chapter is independent of the specific method by which multiple virtual processors are implemented. The term "virtual processor" is generally used instead of "strand" because "strand" is commonly associated with multistranded physical cores.

**CPU.** The term *CPU* is ambiguous in reference to processors with multiple virtual processors. The term could potentially refer to a virtual processor or to an entire processor. Therefore, the term "CPU" is considered ambiguous and is not be used in this document.

**CMT.** *CMT* is an abbreviation for "Chip MultiThreading" or, as an adjective, "Chip MultiThreaded". A CMT processor is a processor containing more than one virtual processor.

### 15.1.1.2 CMT Definition

*CMT*, as defined in UltraSPARC Architecture 2007, applies to all SPARC virtual processors. A processor containing a single virtual processor (strand) is a special case, covered in *Recommended Subset for Single-Strand Processors* on page 623. The CMT interface is the same whether multiple strands are provided by multiple physical cores, a single physical core with multiple strands, or multiple physical cores each with multiple strands.

A *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, as defined elsewhere in this specification, to execute a thread (general purpose registers, floating-point registers, process state, status registers, condition codes, etc.). A virtual processor is the smallest unit to which an interrupt can be delivered. The addressability of interrupts to individual virtual processors is a very important aspect of the CMT programming interface. An UltraSPARC Architecture 2007 implementation must provide sufficient resources so that every virtual processor within the processor makes forward progress at a reasonable rate.

Each virtual processor contains a separate instance of all user-visible architected state; that is, nonprivileged architected state is per-virtual processor.

The privileged and hyperprivileged architected state of a processor falls into four classes (described in *Classes of CMT Registers* on page 598), based on the degree of sharing among virtual processors.

**Implementation Note** The UltraSPARC Architecture 2007 applies to a single physical processor chip. In a multiple-chip system, the UltraSPARC Architecture 2007 applies to each processor chip.

### 15.1.2 General CMT Behavior

In general, each virtual processor of a CMT processor behaves functionally as if it was an independent processor. This is an important aspect of CMT processors because user code running on a virtual processor does not need to know whether or not that virtual processor is part of a CMT processor. At a high level, most privileged code in an operating system can treat virtual processors of a CMT processor as if each was an independent processor. Some software (for example, boot, error, and diagnostic) must be aware that it is executing on a CMT processor. This chapter deals chiefly with the interface between this software and a CMT processor.

Each virtual processor of a CMT processor obeys the same memory model semantics as if it was an independent processor. All software designed to run in a multiprocessing environment, including thread libraries, must be able to operate on a CMT processor without modification.

There are significant performance implications of CMT processors, especially when shared resources (such as caches) exist within a CMT processor. The virtual processors' proximity will potentially mean drastically different costs for communicating between two virtual processors on the same CMT processor compared to communicating between two virtual processors on different CMT processors. This adds another degree of non-uniform 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 CMT processors. There may also be resource contention issues between virtual processors on the same CMT processor. *Performance Issues for CMT Processors* on page 623 discusses some key performance issues related to CMT processors.

## 15.2 Accessing CMT Registers

A key part of the CMT programming model is a set of privileged registers. This section covers how these registers are organized and accessed. The registers can be accessed by software running on a virtual processor of the CMT processor.

CMT-specific registers can be accessed by privileged software running on a virtual processor, using Load and Store Alternate (notably, LDXAs and STXAs) instructions that provide an address space identifier value and a (virtual) address. The CMT programming model defines address space identifiers and associated virtual addresses (VAs) for accessing the CMT-specific registers.

## 15.2.1 Classes of CMT Registers

Nonprivileged architected state, including registers visible to nonprivileged software, is (or at least appears to be) per-virtual-processor.

Privileged architected state, including registers visible to privileged software, is (or at least appears to be) per-virtual-processor.

The hyperprivileged architected state of a processor falls into four categories:

- Per-virtual-processor (per-strand) registers, of which each virtual processor has a private (not shared) copy
- Subset-shared registers, where a copy of each register is shared by a non-overlapping subset of virtual processors[1].
- Per-physical-core shared registers (a special case of subset-shared registers), where a copy of each register is shared by all virtual processors contained within a physical core.
- Processor-shared CMT registers, in which a single copy of each register is shared by all virtual processors in the processor

Registers that are read-only in privileged mode (for example, TICK) need not be strictly implemented as per-virtual-processor registers; they may be implemented in one of the "shared" categories above, such that their shared nature is not visible to privileged software.

CMT-specific registers of all classes can be accessed as ASI-mapped registers through hyperprivileged software running on a virtual processor. Software running on a given virtual processor can access:

- all the per-virtual processor registers belonging to the virtual processor on which it is running
- the per-physical-core shared registers belonging to the physical core on which it is running
- subset-shared registers for any group of virtual processors to which the virtual processor on which it is running belongs
- all processor-shared registers

[1] Currently, no architectural CMT registers fall into this category. It is defined here for completeness, because registers in this category may need to exist as implementation-specific registers

In nonprivileged or privileged mode, it is normally not possible for a virtual processor on one physical core to address (much less, read) the per-physical-core registers of another physical core. On some implementations it may be possible for a virtual processor on one physical core to address the per-physical-core registers of another physical core, but *only* in hyperprivileged mode or if hyperprivileged software grants such privileges to software running at a lower privilege level.

The semantics for accessing the CMT registers through the ASI interface are described in *Accessing CMT Registers Through ASIs* on page 599.

## 15.2.2 Accessing CMT Registers Through ASIs

Each CMT-specific register is accessible through a restricted ASI (accessible only in hyperprivileged software). The ASI number and virtual address corresponding to each CMT register are described later in this chapter.

Each virtual processor can access the per-physical-core CMT registers associated with that virtual processor. The implementation must guarantee that accesses to per-physical-core registers follow sequential semantics on the virtual processor with which they are associated.

Each virtual processor can access all the per-processor shared CMT registers on its processor. An update to a per-processor shared register from one virtual processor will be visible to all other virtual processors that share that register. The ordering of accesses to per-processor shared registers from different virtual processors is not defined, but an implementation must guarantee that:

- Accesses to a shared register from the same virtual processor follow sequential semantics.
- If multiple virtual processors attempt to store to a shared CMT register at the same time, the value observed in (readable from) the register will always be that written by one of those stores. That is, a store to a CMT register must be performed atomically on all bits of the register. In the case of the STRAND_RUNNING register, there is a third option — a write to the register may be dropped (ignored) entirely in certain situations (for details, see *Simultaneous Updates to the STRAND_RUNNING Register* on page 610).

There may be additional implementation-enforced restrictions on updates to some CMT registers.

All CMT registers are 64-bit registers, although some of the bits of individual registers can be reserved or defined to contain a fixed value in a given implementation. *Reserved* register fields should always be written by software with values of those fields previously read from that register or with zeroes and they should read as zero in hardware (see *Reserved Opcodes and Instruction Fields* on page

130). Software intended to run on future versions of CMTs should not assume that these fields will read as 0 or any other particular value. This convention simplifies future expansion of the CMT interface.

A CMT register is accessed through load and store instructions, using a defined ASI number and virtual address. CMT registers can only be accessed in hyperprivileged mode. An attempt to access a CMT register in nonprivileged or privileged mode results in a *privileged_action* exception.

Only the LDXA instruction can be used to read a CMT register. Only the STXA instruction can be used to store to a CMT register. An attempt to access a CMT register with any other instruction results in a *DAE_invalid_asi* exception. An attempt to write to a read-only CMT register with a STXA instruction results in a *DAE_invalid_asi* exception.

## 15.3 CMT Registers

In this section, the registers used to control operation of a processor in a CMT implementation are described. For each register defined in this document, a six-column quick-reference table is provided that specifies the key attributes of the register, as follows:

| Column Heading | Meaning of collumn contents |
|---|---|
| **Register Name** | The name of the CMT register |
| **ASI # (Name)** | The address space identifier number used for accessing the register from software running on the CMT processor (and the recommended ASI name for use in assembly-language hyperprivileged software) |
| **VA** | The virtual address used for accessing the register from software running on the CMT processor |
| **Scope** | The scope of sharing for the register — whether the register is a "per-virtual processor" (per-strand) register, or a single instance of a register that is "shared" among the virtual processors within a physical core (per-core), "shared" among a subset of virtual processors within a physical core (per-subset), or "shared" among all the virtual processors within a processor (per-proc). |
| **Access** | Whether software access to the register is read/write (RW), read-only (R only), write-only (W only), Write-1-to-Set (W1S), or Write-1-to-Clear (W1C) |
| **Note** | Any additional information |

## 15.3.1 Strand ID Register (STRAND_ID)

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| STRAND_ID | $63_{16}$ (ASI_CMT_PER_STRAND) | $10_{16}$ | per-strand | R only | |

**STRAND_ID**

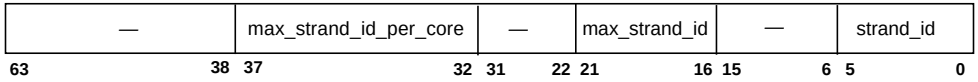



**FIGURE 15-1** STRAND_ID Register

STRAND_ID is a read-only, per-virtual processor register that holds the ID value assigned by hardware to each implemented virtual processor. The ID value is unique within the CMT processor.

As shown above, the STRAND_ID register has three fields:

1. strand_id, which represents this virtual processor's number, as assigned by hardware. The strand ID is encoded in 6 bits.
2. max_strand_id, which is the bit-position index (bit number) of the most significant '1' bit in the STRAND_AVAILABLE register. This is the Strand ID of the highest-numbered implemented virtual processor in this CMT processor.
3. max_strand_id_per_core, which specifies the number of strands minus one that are implemented on each physical core. For a single-stranded processor, max_strand_id_per_core will be 0.

Many other CMT-specific registers provide a bit mask in which each bit corresponds to an individual virtual processor. For these registers, the strand_id field indicates which bit of a bit mask corresponds to this specific virtual processor.

**Strand Numbering Convention.** The numbering of virtual processors (strands) may or may not be contiguous; system software may only assume that each strand ID is unique within a CMT processor. In general, virtual processors should be numbered in a sequential, contiguous series starting with strand number 0. When numbering the virtual processors within a CMT processor, this convention appears straightforward. There are cases, however, where this might not be so simple. This numbering convention is recommended but not required.

In a CMT processor designed with many virtual processors, some physical cores in a manufactured CMT processor may fail to function correctly. It is likely that there would be a desire to salvage a partially good CMT processor (one where a subset of the virtual processors and all the common area function correctly) and use it as a CMT processor with fewer than the maximum number of functional virtual processors. In such a case, it would be possible that the functional strands be

numbered contiguously, starting from 0, and that the STRAND_ID.max_strand_id field be set to the highest-numbered functional virtual processor. This requires some way to reassign the identity of individual virtual processors after manufacturing. If this is not practical, the functioning virtual processors may not be contiguously numbered.

#### 15.3.1.1 Exposing Stranding

If a processor implements multiple strands per physical core, the stranding is exposed in STRAND_ID.max_strand_id_per_core. This field encodes one less than the number of strands that are implemented on the physical processor core; for example, on a physical core with 4 strands, STRAND_ID.max_strand_id_per_core = 3. Every virtual processor within the physical core must observe the same value of max_strand_id_per_core. An implementation defines and count strands and physical processor cores as appropriate for that implementation.

When STRAND_ID.max_strand_id_per_core is nonzero, there are additional constraints on the numbering of virtual processors. virtual processors that correspond to strands on the same physical processor core must have contiguous STRAND_ID.strand_id values, with the lowest numbered virtual processor on a physical core having a strand_id value that is a multiple of the number of strands on each physical core.

It is important to expose stranding to software. From a performance standpoint, stranding must be exposed for the operating system to understand resource sharing and contention issues and to optimally schedule software threads on the processor. From a power management perspective, knowledge of stranding enables the facility to park or disable all strands on a physical core to obtain significant power savings.

### 15.3.2 Strand Interrupt ID Register (STRAND_INTR_ID)

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| STRAND_INTR_ID | $63_{16}$ (ASI_CMT_PER_STRAND) | $00_{16}$ | per-strand | RW | |

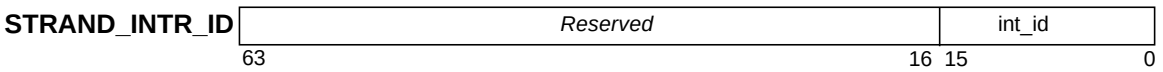



**FIGURE 15-2** STRAND_INTR_ID Register

The STRAND_INTR_ID register allows software to assign a 16-bit interrupt ID, unique within a system, to each virtual processor. This is necessary in order to enable virtual processors to receive interrupts. The identifier in this register is used

by other virtual processors (on the same and different CMT processors) and other bus agents to address interrupts to this specific virtual processor. It can also be used by this virtual processor to identify itself as the source of an interrupt it sends to other virtual processors and bus agents.

This register is Read/Write, accessible only in hyperprivileged mode (HPSTATE.hpriv = 1). It is expected that it will be modified only at boot or reconfiguration time. An attempt to access this register in privileged mode or nonprivileged mode results in a *privileged_action* exception.

The STRAND_INTR_ID register has only one field, a 16-bit interrupt ID field, named int_id.

If an implementation uses fewer than 16 bits for its interrupt ID, the unused bits read as zero and writes to them are ignored.

**IMPL. DEP. #:** It is implementation dependent whether any portion of the int_id field of the STRAND_INTR_ID register is read-only (see following subsection, *Assigning an Interrupt ID*).

### 15.3.2.1 Assigning an Interrupt ID

When assigning the interrupt ID to a virtual processor, software must be aware of interrupt routing conventions used in the system. Some portion of the interrupt 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 strand ID. This portion of the interrupt ID can be read-only in the STRAND_INTR_ID register. Such requirements are both processor- and system-platform-specific.

Each virtual processor in the CMT processor 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.

### 15.3.2.2 Dispatching and Receiving Interrupts

The mechanisms used to dispatch and receive interrupts must work with the interrupt ID register. A processor's interrupt dispatch mechanism must be able to specify the interrupt ID of the destination virtual processor to which the interrupt is to be delivered. When a destination interrupt ID is specified, the interrupt must be delivered to the virtual processor that has the matching ID in its STRAND_INTR_ID register.

### 15.3.2.3 Updating the Strand Interrupt ID Register

It is expected that the interrupt ID register of a virtual processor will be written once by software, when a virtual processor is initially booted. It is assumed that while a virtual processor is being booted, there will be no interrupt traffic in the system.

The latency from when software writes to STRAND_INTR_ID to when the write takes effect is implementation dependent. Use of a MEMBAR #Sync instruction after a write to STRAND_INTR_ID will cause the write to become visible before any instructions after the MEMBAR are executed on the virtual processor.

Updates to STRAND_INTR_ID are atomic: if STRAND_INTR_ID is 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 (addressed either to its old or new interrupt ID), the interrupt may or may not be received by the virtual processor. Once a virtual processor acknowledges an interrupt using its new interrupt ID, it 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 be dropped.

## 15.4 Disabling and Parking Virtual Processors

The CMT 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 running (not parked). This section also describes the interface for enabling/disabling virtual processors and parking/unparking virtual processors.

## 15.4.1 Strand Available Register (STRAND_AVAILABLE)

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| STRAND_AVAILABLE | $41_{16}$ (ASI_CMT_SHARED) | $00_{16}$ | per-proc | R only | |

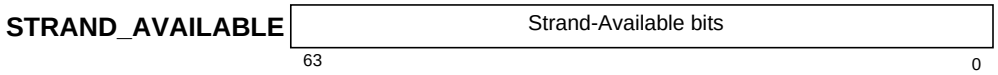



**FIGURE 15-3** STRAND_AVAILABLE Register

The STRAND_AVAILABLE register is a shared (one per processor) register that indicates which virtual processors are available for use (that is, are present and functional) in a CMT implementation.

The STRAND_AVAILABLE register is read-only, comprising a single 64-bit field. As illustrated in FIGURE 15-3, bit *n* corresponds to virtual processor *n*; therefore up to 64 virtual processors are supported per CMT. If a bit in the register is 1, the corresponding virtual processor is available for use in the CMT. If a bit in the register is 0, the corresponding virtual processor is not available for use. An "available" virtual processor is one that is present and functional, therefore can be enabled and used.

## 15.4.2 Enabling and Disabling Virtual Processors

The CMT programming model allows virtual processors to be enabled and disabled. Enabling or disabling a virtual processors is a heavyweight operation that in most cases requires either a *power_on_reset* (POR) or a *warm_reset* (WRM) for updates. A disabled virtual processor produces no architectural effects observable by other virtual processors, and does not participate in cache coherency. The behavior of any transaction (such as an interrupt) issued to a disabled virtual processor is undefined.

**IMPL. DEP. #322-U4:** Whether disabling a virtual processor reduces the power used by a CMT is implementation dependent. It is recommended that a disabled virtual processor consume a minimal amount of power.

**IMPL. DEP. #423-S10:** Whether disabling a virtual processor increases the performance of other virtual processors in the CMT is implementation dependent.

### 15.4.2.1 Strand Enable Status Register (STRAND_ENABLE_STATUS)

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| STRAND_ENABLE_STATUS | $41_{16}$ (ASI_CMT_SHARED) | $10_{16}$ | per-proc | R only | |

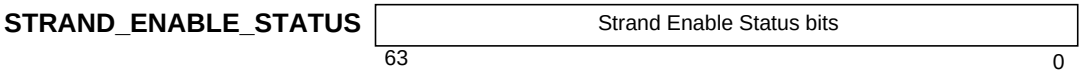



**FIGURE 15-4** STRAND_ENABLE_STATUS Register

The STRAND_ENABLE_STATUS register is a shared (one per processor) register that indicates which virtual processors are currently enabled. The register is a read-only register, in which each bit corresponds to a virtual processor.

As shown in FIGURE 15-4, bit *n* corresponds to virtual processor *n*. If a bit in the STRAND_ENABLE_STATUS register is 1, the corresponding virtual processor is available and enabled. A virtual processor indicated as "not available" in the STRAND_AVAILABLE register cannot be enabled, and its corresponding enabled bit in this register will be 0. An available, enabled virtual processor that is parked is still considered enabled.

**Programming Note** | Hyperprivileged software should never set bit STRAND_ENABLE{*n*} to 1 if STRAND_AVAILABLE{*n*} = 0.

**State After Reset.** The STRAND_ENABLE_STATUS register changes due to a *power_on_reset*. (POR) or a *warm_reset* (WRM). During a *power_on_reset*, the contents of its STRAND_AVAILABLE register are copied to the STRAND_ENABLE_STATUS register. During a *warm_reset* reset, the contents of the STRAND_ENABLE register are copied to the STRAND_ENABLE_STATUS register.

### 15.4.2.2 Strand Enable Register (STRAND_ENABLE)

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| STRAND_ENABLE | $41_{16}$ (ASI_CMT_SHARED) | $20_{16}$ | per-proc | RW | Changes take effect during reset |

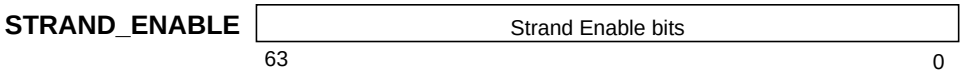



**FIGURE 15-5** STRAND_ENABLE Register

The STRAND_ENABLE register is a shared (one per processor) register, used by software to enable and disable a CMT's virtual processors. When disabled, a virtual processor and any structures private to that virtual processor behave as though they were not present.

**Programming Note** | When re-enabled, per-strand architectural state that existed when the virtual processor was previously enabled should be assumed to be lost. Therefore, hyperprivileged software must initialize any needed per-strand architectural state each time a virtual processor is enabled.

Changing a bit in the STRAND_ENABLE register does *not* take effect (cause a virtual processor to be enabled/disabled) immediately. Instead, it indicates a *pending* change to the STRAND_ENABLE_STATUS register, which will not take effect until the next *warm_reset* (WRM) reset — at which time, the contents of the STRAND_ENABLE register are copied to the STRAND_ENABLE_STATUS register. A change in the STRAND_ENABLE register may also take place at some other implementation-dependent time (see *Dynamically Enabling/Disabling Virtual Processors* on page 608 (impl. dep. #___).

As shown in FIGURE 15-5, the STRAND_ENABLE register contains one bit per possible virtual processor, with bit *n* corresponding to virtual processor *n*. If bit *n* is 1, then virtual processor *n* should be enabled after the next warm reset (if that virtual processor is available). If bit *n* is 0, then virtual processor *n* should be disabled after the next warm reset.

When bit *n* in the STRAND_AVAILABLE register is 0 (the virtual processor is unavailable), the corresponding bit (bit *n*) in the STRAND_ENABLE register is forced to 0 and attempts to write "1" to bit *n* in the STRAND_ENABLE register are ignored.

### Restrictions on Updating the STRAND_ENABLE Register.

**IMPL. DEP. #323-U4:** Whether an implementation provides a restriction that prevents software from writing a value of all zeroes (or zeroes corresponding to all available virtual processors) to the STRAND_ENABLE register is implementation dependent. This restriction avoids the dangerous case where all virtual processors become disabled and the only way to enable any virtual processor is a hard *power_on_reset* (a warm reset would not suffice). If such a restriction is implemented and software running on any virtual processor attempts to write a value of all zeroes (or zeroes corresponding to all available virtual processors) to the STRAND_ENABLE register, hardware forces the STRAND_ENABLE register to an implementation-dependent value which enables at least one of the available virtual processors.

**State After Reset.** Upon assertion of *power_on_reset*, the value of the STRAND_AVAILABLE register is copied to the STRAND_ENABLE register. The STRAND_ENABLE register does not change during any other reset, including system (or equivalent) resets.

#### 15.4.2.3 Dynamically Enabling/Disabling Virtual Processors

**IMPL. DEP. #424-S10:** Whether a CMT implementation provides the ability to dynamically enable and disable virtual processors is implementation dependent. It is tightly coupled to the underlying microarchitecture of a specific CMT implementation. This feature is implementation dependent because any implementation-independent interface would be too inefficient on some implementations.

### 15.4.3 Parking and Unparking Virtual Processors

Parking is a way to temporarily suspend the operation of a virtual processor, intended for use by critical diagnostic and recovery code. A parked virtual processor can be later unparked to allow it to resume running. A virtual processor can be parked or unparked at arbitrary times using the STRAND_RUNNING register and a WMR or POR reset is not required for parking/unparking to become effective. The STRAND_RUNNING_STATUS register can be used to determine whether a virtual processor that has been directed to park has completed the process of parking.

A parked virtual processor does not execute instructions and does not initiate any transactions on its own. If any portion of the memory system resides in a parked virtual processor, it will continue to be updated as necessary for it to remain coherent with the rest of the memory system while the virtual processor is parked.

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 software running on a virtual processor that it was ever parked (except for observable timing considerations).

While a virtual processor is parked, the STICK register continues to count.

**IMPL. DEP. #425-S10:** It is implementation dependent whether the TICK register continues to count while a virtual processor is parked.

Using the TICK or STICK counter to detect the parking of a virtual processor is not recommended.

An interrupt to a parked virtual processor behaves the same as if the virtual processor was too busy to accept the interrupt.

**IMPL. DEP. #324-U4:** It is implementation dependent whether parking a virtual processor reduces the power used by a CMT. It is recommended that a parked virtual processor use a reduced amount of power.

Parking a virtual processor should, when appropriate, reduce the contention for shared resources and enable other virtual processors to potentially run faster.

**IMPL. DEP. #426-S10**: The degree to which parking a virtual processor impacts the performance of other virtual processors is implementation dependent.

**Implementation Note** One possible way to implement virtual processor parking is to disable instruction fetching in a parked virtual processor. In such an implementation, after a virtual processor is parked, it will execute the instructions currently in its pipeline, complete pending transactions (such as draining the store queue), and then become idle (at which time, its status in the STRAND_RUNNING_STATUS register will change from "Running" to "Parked").

### 15.4.3.1 Strand Running Register (STRAND_RUNNING)

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| STRAND_RUNNING_RW | $41_{16}$ (ASI_CMT_SHARED) | $50_{16}$ | per-proc | RW | General RW access |
| STRAND_RUNNING_W1S | $41_{16}$ (ASI_CMT_SHARED) | $60_{16}$ | per-proc | W1S | Write 1s to set bits |
| STRAND_RUNNING_W1C | $41_{16}$ (ASI_CMT_SHARED) | $68_{16}$ | per-proc | W1C | Write 1s to clear bits |

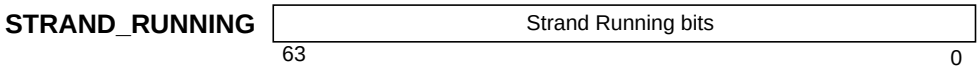



**FIGURE 15-6** STRAND_RUNNING Register

STRAND_RUNNING is a shared (one per processor) register, used by software to park and unpark selected virtual processors in a CMT implementation. When a virtual processor is parked, the virtual processor stops executing new instructions and will not initiate new transactions except in response to a coherency transaction initiated by another virtual processor.

**IMPL. DEP. #427-S10:** There may be an arbitrarily long, but bounded, delay ("skid") from the time when a virtual processor is directed to park or unpark (via an update to the STRAND_RUNNING register) until the corresponding virtual processor(s) actually park or unpark.

Multiple access methods are provided for writing bits in the STRAND_RUNNING register, distinguished by the virtual address used (listed above):

- STRAND_RUNNING_RW, for normal reading and writing of the entire register

- STRAND_RUNNING_W1S ("Write 1 to Set"), where writing '1' to a bit sets the destination bit to '1' and writing '0' to a bit leaves the destination bit unchanged
- STRAND_RUNNING_W1C ("Write 1 to Clear"), where writing '1' to a bit sets the destination bit to '0' (clears it) and writing '0' to a bit leaves the destination bit unchanged

A specific value can be atomically written to all bits of the STRAND_RUNNING register, using STRAND_RUNNING_RW, or bits can be individually modified, using STRAND_RUNNING_W1S or STRAND_RUNNING_W1C. When a virtual processor parks itself, software should write to STRAND_RUNNING_W1C. When a virtual processor wants to become the only active virtual processor (parking all other virtual processors in the CMT), it is more appropriate to write the desired value directly to STRAND_RUNNING_RW. A direct write eliminates the need to perform separate set and clear operations to write a specific value to the register.

As shown in FIGURE 15-6, the STRAND_RUNNING register contains one bit per possible virtual processor, with bit *n* corresponding to virtual processor *n*. Writing a value of 1 to bit position *n* activates (unparks) virtual processor *n* for normal execution, while writing a value of 0 to bit *n* parks virtual processor *n*. If bit *n* in the STRAND_ENABLE_STATUS register is 0 (not enabled), hardware forces the corresponding bit in the STRAND_RUNNING register to 0 and attempts to write to that bit are ignored.

**Updating the STRAND_RUNNING Register.** When a virtual processor parks itself by updating the STRAND_RUNNING register and follows the update with a FLUSH instruction, no instruction after the FLUSH instruction will be executed until the virtual processor is unparked. The virtual address specified in the FLUSH instruction is not important. The FLUSH instruction may be executed either 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 when a virtual processor parks itself.

**IMPL. DEP. #428-S10:** When a virtual processor writes to the STRAND_RUNNING register to park itself, the method by which completion of parking is assured (instructions stop being issued) is implementation dependent.

**Simultaneous Updates to the STRAND_RUNNING Register.** Hardware is not required to provide a mechanism for handling simultaneous updates from different strands to the STRAND_RUNNING register.

**Programming Note** It is the responsibility of hyperprivileged software to insure that a livelock condition, resulting from simultaneous updates from different strands to the STRAND_RUNNING register, does not occur.

After writing to STRAND_RUNNING with a STXA instruction, hyperprivileged software should check the STRAND_RUNNING_STATUS register to verify when the attempted parking/unparking of virtual processor(s) actually completed.

**At Least One Virtual Processor Must Remain Unparked.** Hardware enforces the restriction that an update to the STRAND_RUNNING register by software running on one of the virtual processors cannot cause all of the enabled virtual processors to become parked. This restriction is important to avoid the dangerous situation where all virtual processors become parked and there is no way to reactivate any of the virtual processors (without a warm reset or power-on reset).

**IMPL. DEP. #429-S10**: If an update to the STRAND_RUNNING register would cause all enabled virtual processors to become parked, it is implementation dependent which virtual processor is automatically unparked by hardware. The preferred implementation is that when an update to the STRAND_RUNNING register (STXA instruction) would cause all virtual processors to become parked, hardware silently ignores (discards) that STXA instruction.

***Implementation Note*** It is important that when a virtual processor attempts to issue an update to the STRAND_RUNNING register that would cause all virtual processors to become parked, that virtual processor is *not* parked. A virtual processor updating the STRAND_RUNNING register will be executing a section of software (error diagnostic or other special code) that is aware of the behavior and implications of parking. When an attempt is made to park all virtual processors, automatically unparking an *arbitrary* virtual processor would be problematic, because a virtual processor in the midst of running nonprivileged code could become the only unparked virtual processor. If this were to happen, the only active virtual processor in the CMT would be unaware of the state of the CMT and would not know to check the running status of other virtual processors.

**At Least One Virtual Processor Must Remain Unparked — Multiprocessor Configuration.** When there are multiple processors (chips) in the configuration, there is still a requirement to have at least one virtual processor unparked on each processor. However, from a testing point of view, it is desirable to be able to unpark all but one virtual processor in the entire multiprocessor configuration.

**IMPL. DEP. #430-S10**: In a multiprocessor configuration, whether all but one virtual processor can be parked is implementation dependent.

**State After Reset.** Upon power-on reset or warm reset, the STRAND_RUNNING register by default is initialized such that all the virtual processors are parked except for the lowest-numbered enabled virtual processor. This provides a default on-chip "boot master" virtual processor, reducing BootBus contention.

**Note** | For systems that use a system reset pin, the value of the STRAND_RUNNING register is updated upon assertion of the warm reset signal.

### 15.4.3.2 Strand Running Status Register (STRAND_RUNNING_STATUS)

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| STRAND_RUNNING_STATUS | $41_{16}$ (ASI_CMT_SHARED) | $58_{16}$ | per-proc | R only | |

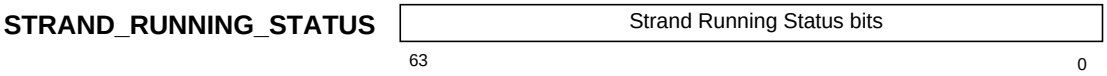



**FIGURE 15-7** STRAND_RUNNING_STATUS Register

STRAND_RUNNING_STATUS is a shared (one per processor) register. It indicates whether a virtual processor is still active (running) or has actually become parked. It is needed because there may be a delay between the time when a virtual processor is directed to park (via the STRAND_RUNNING register) and the time when it actually becomes parked. The STRAND_RUNNING_STATUS register is a shared, read-only register in which bit *n* indicates if strand *n* is active.

There is an implementation-dependent delay from the time virtual processor *n* is directed to park by writing 0 to bit *n* of the STRAND_RUNNING register until it actually becomes parked (impl. dep. #427-S10).

As shown in FIGURE 15-7, the STRAND_RUNNING_STATUS register has one 64-bit field (one bit per possible virtual processor), with bit *n* corresponding to virtual processor *n*.

- If virtual processor *n* is enabled (STRAND_ENABLE_STATUS{*n*} = 1):
  - a value of 0 in bit *n* of the STRAND_RUNNING_STATUS register indicates that virtual processor *n* is truly parked and will not execute any additional instructions or initiate new transactions until it is unparked.

- A value of 1 in bit *n* of the STRAND_RUNNING_STATUS register indicates that a virtual processor is active and can execute instructions and initiate transactions. All virtual processors that have a 1 in the STRAND_RUNNING register must have a 1 in the STRAND_RUNNING_STATUS register.

- If virtual processor *n* is disabled (STRAND_ENABLE_STATUS{*n*} = 0), bit *n* of the STRAND_RUNNING_STATUS register must be 0.

The STRAND_RUNNING_STATUS register indicates when a virtual processor that has been directed to park has actually parked, that is, is no longer executing instructions or initiating any transactions (except in response to coherency transactions generated by other virtual processors).

**IMPL. DEP. #431-S10:** The criteria used for determining whether a virtual processor is fully parked (corresponding bit set to '1' in the STRAND_RUNNING_STATUS register) are implementation dependent.

After bit *n* in the STRAND_RUNNING register has been changed from 1 to 0, hardware must guarantee that only a single transition from 1 to 0 in bit *n* of the STRAND_RUNNING_STATUS register will be observed.

**State After Reset.** The value of the STRAND_RUNNING_STATUS register is the same as the value of the STRAND_RUNNING register at the end of a system reset.

## 15.4.4 Virtual Processor Standby (or Wait) State

**IMPL. DEP. #432-S10:** Whether an implementation implements a `Standby` (or `Wait`) state for virtual processors, how that state is controlled, and how that state is observed are implementation-dependent.

In a `Standby` state, the virtual processor is suspended for a predetermined period of time and/or until an external interrupt is received. A `Standby` state may appear similar to a `Parked` state, but virtual processor `Standby` state (if implemented) must be completely orthogonal to parking. The details of the software interface to and implementation of `Standby`/`Wait` state is beyond the scope of this specification.

With respect to parking, the virtual processor is either `Running` or not running (`Parked`), as indicated in the STRAND_RUNNING_STATUS register. With respect to standby, the virtual processor is either in `Standby` or `Normal` state. Since these features are independent, the virtual processor can be in any of the four possible combinations of these states. A virtual processor is still considered running if it is in a `Standby` mode but is not `Parked`. If a virtual processor is in a `Standby` mode and becomes `Parked`, it will remain `Parked` even if an event causes it to change from `Standby` to `Normal` mode; it will not execute instructions until it is later unparked.

Implementing a `Standby` mode may provide performance and/or power-consumption benefits. A virtual processor in `Standby` mode may cause less resource contention with other running virtual processors and may consume less power.

## 15.5 Reset and Trap Handling

In a CMT, some resets apply globally to all virtual processors, some apply to an individual virtual processor, and some apply to an arbitrary subset of virtual processors. The following sections address how each type of reset affects the virtual processors in a CMT.

The reset nomenclature used in this section is generally consistent with that used for UltraSPARC Architecture 2007 processors. If future processors classify resets differently, this model should be extended appropriately to the new classifications.

Traps (as opposed to resets) apply to individual virtual processors and are discussed in *Traps* on page 463.

### 15.5.1 Per-Strand Resets (SIR and WDR Resets)

The only resets that affect only a single virtual processor are those that are internally generated by a virtual processor, such as software initiated reset (SIR) and watchdog reset (WDR). These resets are generated by an individual virtual processor and are not propagated to the other virtual processors in a CMT.

### 15.5.2 Full-Processor Resets (POR and WRM Resets)

There is a class of resets that are generated by an external agent and apply to all the virtual processors within a processor. This class includes all resets associated with fundamental CMT reconfigurations.

*power_on_reset* (POR) is one case of full-processor reset. Warm reset is another example of such a reset (warm reset may be either processor or physical strand-specific, depending on the implementation). Full-processor reset is required for certain reconfigurations of the processor.

Power-on reset and warm reset (or their equivalents in future processors) are global resets, sent to all strands in a CMT processor.

#### 15.5.2.1 Boot Sequence

As discussed in *Strand Running Register (STRAND_RUNNING)* on page 609, the default boot sequence is for all virtual processors except one (nominally, the lowest-numbered enabled virtual processor) to be set to `Parked` state at the beginning of full-processor reset. The single unparked virtual processor is the master virtual processor, which should arbitrate for the BootBus (if multiple CMT processors share the same BootBus). The master virtual processor (or service processor) should unpark the other virtual processors in the processor at the appropriate time in the booting process.

### 15.5.3 Partial Processor Resets (XIR Reset)

There is a class of resets, referred to here as "partial-processor resets," that are generated by an external agent and affect an arbitrary subset of virtual processors within a processor. The subset may be anything from all virtual processors to no virtual processors (impl. dep. #433-S10).

Externally-initiated reset (XIR) is a partial-processor reset. XIR is intended to reset a specific virtual processor in a system, primarily for diagnostic and recovery purposes.

**IMPL. DEP. #433-S10**: A mechanism must exist to specify which subset of virtual processors in a processor should be reset when a partial-processor reset (for example, XIR) occurs. The specific mechanism is implementation-dependent.

Possible methods of specifying the subset include the following:

1. Before the partial-processor reset occurs, set up a steering register that specifies the subset of virtual processors that should be affected. For systems using an XIR reset, the XIR Steering register described in *XIR Steering Register (XIR_STEERING)* on page 616 should be used.

2. Specify the subset of virtual processors concurrently with the reset request, across the same 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.

In an implementation that replaces the XIR reset with a different set of resets, the following rules apply for extending this CMT programming interface:

- Each partial-processor reset may use an interface where the set of virtual processors to reset is communicated along with the reset request.
- For partial-processor resets for which the set of virtual processors to be reset is *not* communicated along with the reset request:
  - The highest priority virtual processor will use the XIR_STEERING register to determine the subset of virtual processors to be reset.

- Each subsequent lower-priority virtual processor can either use the XIR_STEERING register or use an additional steering register (comparable to XIR_STEERING), specifically associated with that reset. Each additional steering register will be accessed using the same ASI number ($41_{16}$) as the XIR_STEERING register but with a distinct virtual address.

### 15.5.3.1 XIR Steering Register (XIR_STEERING)

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| XIR_STEERING | $41_{16}$ (ASI_CMT_SHARED) | $30_{16}$ | per-proc | RW | General access |

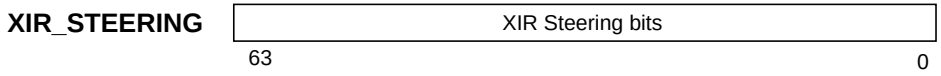



**FIGURE 15-8** XIR_STEERING Register

An externally initiated reset (XIR) can be steered to an arbitrary subset of virtual processors, using the XIR_STEERING register. The XIR_STEERING register is shared across virtual processors and is used by software to control which virtual processor(s) within a processor will receive the XIR reset signal when XIR is asserted for the processor module.

As shown in FIGURE 15-8, the XIR_STEERING register has one 64-bit field (one bit per possible virtual processor), in which bit *n* corresponds to virtual processor *n*.

When an external reset is asserted for the CMT, if bit *n* in the XIR_STEERING register is 1, virtual processor *n* receives an XIR reset; if bit *n* in the XIR_STEERING register is 0, virtual processor *n* continues execution, unaware of the external reset asserted for the CMT.

A virtual processor that is parked when it receives an XIR reset remains parked and will handle the XIR reset immediately after being unparked.

**IMPL. DEP. #325-U4a:** Whether XIR_STEERING{*n*} is a read-only bit or a read/write bit is implementation dependent. If XIR_STEERING{*n*} is read-only, then (1) writes to XIR_STEERING{*n*} are ignored and (2) XIR_STEERING{*n*} is set to 1 if virtual processor *n* is available and to 0 if it is not available (that is, XIR_STEERING{*n*} reads the same as STRAND_AVAILABLE{*n*}.

It may be desirable for an XIR to effectively unpark and reset all virtual processors in a CMT. If so, that effect can be generated by having the first action of software on virtual processor receiving an XIR to unpark all other virtual processors in the CMT.

**State After Reset.**

During *power_on_reset*, the contents of the STRAND_AVAILABLE register are copied to the XIR_STEERING register. During a warm reset, the contents of the STRAND_ENABLE register are copied to the XIR_STEERING register. This provides for a default condition in which all enabled virtual processors receive an XIR reset when an external reset is asserted for the processor. (impl. dep. #325-U4b)

## 15.6 Error Handling in CMT Processors

Errors in a structure *private* to a virtual processor are considered virtual-processor(strand)-specific and are reported to that virtual processor using its error-reporting mechanism.

When an error in a structure *shared* among virtual processors occurs:

- If the virtual processor initiating the request that caused or detected the error can be identified, the error is considered virtual-processor-specific and is reported back to the originating virtual processor.
- If the virtual processor initiating the request that caused or detected the error cannot be identified, the error is considered non-virtual-processor-specific.
- All virtual processors that share a structure are considered to be part of the *error-handling group* for that structure. This implies that any virtual processor in the group can be assigned to handle error traps associated with the structure and have diagnostic access to the structure for error recovery.

The following sections describe how a CMT processor handles both virtual-processor-specific and non-virtual-processor-specific errors.

### 15.6.1 Virtual-Processor-Specific Error Reporting

Errors specific to a particular virtual processor are reported to the virtual processor associated with the error, using the virtual processor's error reporting mechanism. A virtual-processor-specific error can be either synchronous or asynchronous. It may be an error that occurred in a shared structure but is traceable to the originating virtual processor. It is the responsibility of error handling software to recognize the implication of errors in shared structures and take appropriate action.

## 15.6.2 Reporting Errors on Shared Structures

Errors in shared structures are more complicated than virtual-processor-specific errors. When a non-virtual-processor-specific error occurs, it must be recorded and an exception must be generated on one of the virtual processors within the CMP to deal with the error. More precisely, the virtual processor that reports the exception must be part of the error-handling group for the shared structure in which the error was detected. The following subsections describe where the error should be recorded and in which virtual processor the exception should be generated.

### 15.6.2.1 Error Steering

When an error occurs in a shared resource, the error must be reported to a virtual processor that shares that resource and is part of its error-handling group. That virtual processor has the capability of issuing diagnostic reads and writes to the structure for diagnosis, correction, and error-clearing purposes. Error steering registers are used to determine which virtual processor will handle the error. Software configures an error steering register to specify which virtual processor should handle the error(s) associated with that error steering register. That is, an error steering register defines in which virtual processor an exception will be generated, to report and handle the error.

A given CMT implementation may contain resources shared by all the virtual processors of the CMT processor or shared by a subset of two or more virtual processors.

**IMPL. DEP. #434-S10:** Because of the range of implementation, the number of, organization of, and ASI assignments for error steering registers in a CMT processor are implementation dependent.

Error steering registers may be provided per shared resource or per level of sharing. In the case that all shared resources are shared by all virtual processors, it is recommended that a single error steering register be used and that error steering register should follow the behavior of the ERROR_STEERING register defined in *Error Steering Register (ERROR_STEERING)* on page 620. If a mechanism is used where error steering registers are used per level of sharing, it is recommended that the ERROR_STEERING register be used for the level at which all virtual processors share and provide error-handling groups.

**General Guidelines for Error Steering Registers.** An error steering register controls which virtual processor handles non-virtual-processor-specific errors. Such an error is recorded using the virtual processor's asynchronous error reporting mechanism (as relevant to the error) and generates an appropriate exception.

An error steering register is accessed through an ASI or a memory-mapped address. It must be accessible for both reading and writing by software (using load and store alternate instructions).

A processor contains one or more error steering registers. The number of error steering registers needed depends on how resources are shared and the ability of a virtual processor to diagnose errors in a resource it does not share.

An error steering register specifies a virtual processor by an encoded field, target_id, that corresponds to the strand_id of the targeted virtual processor. Use of an encoded representation guarantees that only one virtual processor can be specified. An error steering register should contain only one field, the target_id field, that encodes the strand_id of the virtual processor that should be informed of non-virtual-processor-specific errors in its sharing group.

**IMPL. DEP. #326-U4-Cs10a:** The number of implemented bits of ERROR_STEERING.target_id is nominally six, but is implementation dependent and must be sufficient to encode the highest implemented virtual processor ID.

It is the responsibility of software to ensure that an error steering register identifies an appropriate virtual processor for handling the error(s) assigned to it. If an error steering register identifies a virtual processor that is not available (per STRAND_AVAILABLE) or is disabled (per STRAND_ENABLE_STATUS), none of the enabled virtual processors in the error-handling group will be affected by the reporting of a non-virtual-processor-specific error to the disabled virtual processor. However, the behavior of the specified disabled virtual processor is undefined; for example, the error status register in 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 not part of the error-handling group, operation is also undefined. An example would be if the error steering register identifies a virtual processor in another error-handling group for a virtual-processor-specific error. To avoid this case, an error steering register should be assigned on a core basis for core errors that are non-virtual-processor-specific.

If an error steering register identifies a virtual processor that is parked, the non-virtual-processor-specific error is reported to that virtual processor and the virtual processor will observe the appropriate exception, but not until after it is unparked.

When an error steering register is written by software, the update becomes visible after an unspecified delay. If a store to the register is followed by a MEMBAR synchronization barrier instruction, it is guaranteed that the write to the error steering register will complete by the time the execution of the MEMBAR instruction completes.

When a non-virtual-processor-specific error occurs, the corresponding 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-virtual-processor-specific error occurs and at the same time target_id is being changed in the corresponding error steering register, the subsequent error report and the generated exception will occur together on the *same* virtual processor, either the virtual processor indicated by the old value in the error steering register or

the one indicated by the new value. That is, for non-virtual-processor-specific errors, the generation of an error report plus an exception is atomic with respect to changes to the contents of the error steering register.

**State of Error Steering Register After Reset.**

The target_id field of an error steering register is initialized during a power-on-reset and warm reset. After a power-on-reset, the value in the target_id field of an error steering register should refer to the lowest-numbered available virtual processor (as indicated by the STRAND_AVAILABLE register) that corresponds to the resource(s) covered by the steering register. After a warm reset, the value in the target_id field of an error steering register should refer to the lowest-numbered enabled virtual processor (as indicated by the STRAND_ENABLE register) that corresponds to the resource(s) covered by the steering register.

**Error Steering Register (ERROR_STEERING).** The ERROR_STEERING register is the recommended mechanism for specifying which virtual processor in an error-handling group should handle non-virtual-processor-specific errors in resources shared by all virtual processors of the error-handling group. ERROR_STEERING is a shared register, accessible from all virtual processors in the error-handling group.

When a non-virtual-processor-specific error occurs, the error is recorded using the asynchronous error reporting mechanism in the virtual processor indicated by ERROR_STEERING. The appropriate exception is generated in that same virtual processor.

| Register Name | ASI # (Name) | VA | Scope | Access | Note |
|---|---|---|---|---|---|
| ERROR_STEERING | | | per-proc | RW | |

The Error Steering register has only one field that encodes the strand ID of the strand that should be informed of non-virtual-processor-specific errors. When an error is detected that cannot be traced back to a specific virtual processor, the error is recorded in, and a trap is sent to, the virtual processor identified by the Error Steering register.

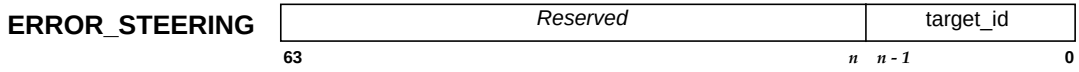



**FIGURE 15-9** ERROR_STEERING Register

**IMPL. DEP. #435-S10**: Although the ERROR_STEERING register is the recommended mechanism for steering non-virtual-processor-specific errors to a virtual processor for handling, the actual mechanism used in a given implementation is implementation dependent.

The ERROR_STEERING register contains one field, target_id, that encodes the virtual processor ID of the virtual processor that should be informed of non-virtual-processor-specific errors (see FIGURE 15-9).

**IMPL. DEP. #436-S10**: The width of the target_id field of the ERROR_STEERING register is implementation dependent.

The target_id field (refer to FIGURE 15-9) must be wide enough to encode the strand ID of the highest-numbered implemented virtual processor. If *n* bits of this field are implemented, the unused most-significant bits numbered 5 to 6-*n* read as zero and writes to those bits are ignored.

**IMPL. DEP. #437-S10:** An implementation may provide multiple target_id fields in an ERROR_STEERING register for different types of non-virtual-processor-specific errors.

### 15.6.2.2 Reporting Non-Virtual-Processor-Specific Errors

Before an exception can be generated for a non-virtual-processor-specific error, the error must be recorded. Non-virtual-processor-specific errors are recorded using the asynchronous error reporting mechanism of the virtual processor specified by the ERROR_STEERING register. The mechanism used is the same as that for reporting vitual processor-specific errors.

Each asynchronous error is defined as either virtual-processor-specific or non-virtual-processor-specific. If the same error can occur as either a virtual-processor-specific error or a non-virtual-processor-specific error, the two cases must be reported as two identifiably distinct errors.

**IMPL. DEP. #438-S10**: It is implementation dependent whether the error-reporting structures for errors in shared resources appear within a virtual processor in per-virtual-processor registers or are contained within shared registers associated with the shared structures in which the errors may occur.

**IMPL. DEP. #439-S10**: The type of exception generated in a virtual processor to handle each type of non-virtual-processor-specific error is implementation dependent. A virtual processor can choose to use the same exceptions used for corresponding virtual-processor-specific asynchronous errors or it can choose to generate different exceptions.

# 15.7 Additional CMT Software Interfaces

## 15.7.1 Diagnostic/RAS Registers

The CMT software interface defines how virtual processors are disabled or parked (for diagnostic and error recovery) and how errors are reported in a CMT processor. It is up to the implementation to provide appropriate diagnostic and recovery mechanisms, which are not specified here.

A future extension of the CMT programming model may include more common features for diagnostics and RAS. Increasing commonality without significantly limiting the implementation options is best.

## 15.7.2 Configuration Registers

Given the broad range of possible implementations, no common configuration interface is defined here.

At this time the CMT programming model does not specify any common configuration registers. A future extension of the CMT programming model may include some. Increasing commonality without significantly limiting the implementation options is best.

## 15.7.3 Performance Registers

At this time, no common performance registers are specified. A future extension of the CMT programming model may include some.

This is a specifically important area to have common features. A range of software tools rely on the performance registers and common features will enable software tools to be more quickly deployed on new architectures with less work.

## 15.7.4 Booting Support

Some of the registers previously described can be used by firmware for booting support. See *Strand Running Register (STRAND_RUNNING)* on page 609 for an example of such a register.

During a power-on-reset, only one enabled virtual processor per processor will be unparked. Only this virtual processor will begin fetching instructions after the reset.

**IMPL. DEP. #440-S10:** Which virtual processor is unparked during POR and whether it is unparked by processor hardware or by a service processor is implementation dependent. Conventionally, the virtual processor with the lowest-numbered strand_id is unparked.

In a recommended booting sequence, software determines when virtual processors become unparked after reset. The default behavior is for only one virtual processor to be unparked when the system reset signal is removed. That virtual processor, in turn, configures common registers and then unparks other virtual processors one at a time. This is only one possible boot sequence; software is free to implement other boot sequences.

## 15.8 Performance Issues for CMT Processors

Which resources are shared among which virtual processors in a CMT processor is implementation-dependent. Resources such as caches, TLBs, and even execution pipelines may be shared by virtual processors. From a performance perspective, there are significant issues that result from this sharing. In this section, hyperprivileged software issues of thread scheduling and configuration of inactive virtual processors is discussed. Issues of how to develop algorithms and approaches to take advantage of the low communication latencies between virtual processors are not covered here.

To understand and take advantage of performance issues in a CMT processor requires some knowledge of the underlying implementation. The existence of implementation dependencies is unavoidable, but hopefully abstract representations and general approaches can reduce the degree of implementation dependence in hyperprivileged software.

## 15.9 Recommended Subset for Single-Strand Processors

It is recommended that single-strand UltraSPARC Architecture 2007 processors implement a subset of the CMT interface. This enables them to more easily integrate into systems that may also contain CMT processors and enables more consistent software to be deployed across those and other future systems.

Single-strand UltraSPARC Architecture 2007 processors should implement all of the CMT registers described in this chapter, as follows:

- The Strand Interrupt ID register (STRAND_INTR_ID) should be fully implemented.
- All other registers can be implemented as read-only registers containing fixed values, writes to which are ignored.

TABLE 15-1 summarizes the recommended implementation of CMT registers for a single-strand processor implementation:

**TABLE 15-1** Recommended CMT Register Set for Single-Strand Processors

| ASI | VA | Register Name | Type | Note |
|---|---|---|---|---|
| $41_{16}$ | $00_{16}$ | STRAND_AVAILABLE | R only | Read value of $01_{16}$ |
| | $10_{16}$ | STRAND_ENABLE_STATUS | R only | Read value of $01_{16}$ |
| | $20_{16}$ | STRAND_ENABLE | R only | Read value of $01_{16}$ |
| | $30_{16}$ | XIR_STEERING | R only | Read value of $01_{16}$ |
| | $50_{16}$ | STRAND_RUNNING_RW | R only | Read value of $01_{16}$ |
| | $58_{16}$ | STRAND_RUNNING_STATUS | R only | Read value of $01_{16}$ |
| | $60_{16}$ | STRAND_RUNNING_W1S | W only (ignored) | Access (write) ignored |
| | $68_{16}$ | STRAND_RUNNING_W1C | W only (ignored) | Access (write) ignored |
| $63_{16}$ | $00_{16}$ | STRAND_INTR_ID | RW | Software assigned unique interrupt ID for virtual processor (read/write) |
| | $10_{16}$ | STRAND_ID | R only | Read value of $00_{16}$ |

# 15.10 Machine State Summary

TABLE 15-2 describes the ASI extensions that support CMT registers. The states of CMT registers after resets are enumerated in TABLE 16-2 on page 634.

**TABLE 15-2** ASI Extensions

| ASI | VA | Register Name | Scope | Type | Description |
|---|---|---|---|---|---|
| $41_{16}$ | $00_{16}$ | STRAND_AVAILABLE | per-proc | R | Bit mask of implemented virtual processors |
| | $10_{16}$ | STRAND_ENABLE_STATUS | per-proc | R | Bit mask of enabled virtual processors |
| | $20_{16}$ | STRAND_ENABLE | per-proc | RW | Bit mask of virtual processors to enable after next reset (read/write) |
| | $30_{16}$ | XIR_STEERING | per-proc | RW | Bit mask of virtual processors to propagate XIR to (read/write) |
| | $50_{16}$ | STRAND_RUNNING_RW | per-proc | RW | Bit mask to control which virtual processors are active and which are parked (read/write): 1= active, 0 = parked |
| | $58_{16}$ | STRAND_RUNNING_STATUS | per-proc | R | Bit mask of virtual processors that are currently active: 1 = active, 0 = parked |
| | $60_{16}$ | STRAND_RUNNING_W1S | per-proc | W1S | Pseudo-register for write-one-to-set access to STRAND_RUNNING |
| | $68_{16}$ | STRAND_RUNNING_W1C | per-proc | W1C | Pseudo-register for write-one-to-clear access to STRAND_RUNNING |
| $63_{16}$ | $00_{16}$ | STRAND_INTR_ID | per-strand | RW | Software assigned unique interrupt ID for virtual processor (read/write) |
| | $10_{16}$ | STRAND_ID | per-strand | R | Hardware assigned ID for virtual processor (read-only) |
| | $40_{16}$ and greater | *Reserved* | per-strand | Impl. Dep. | Reserved for implementation-specific per-strand registers |

CHAPTER 16

# Resets

## 16.1 Resets

The UltraSPARC Architecture 2007 defines 5 types of resets. Reset priorities, listed in order from highest to lowest, are as follows:

- power-on reset (POR)
- warm reset (WMR)
- externally initiated reset (XIR)
- watchdog reset (WDR), and
- software-initiated reset (SIR)

POR, WMR, and XIR resets are initiated external to the processor (chip). WDR and SIR resets are initiated by the virtual processor itself, in response to specific conditions.

POR resets are processor-wide (affect all virtual processors on the chip). WDR and SIR resets are directed to a specific virtual processor. XIR resets are directed to the virtual processor(s) indicated by the XIR_STEERING register. WMR resets are implementation dependent and may be either processor-wide or directed to specific virtual processor(s).

Resets are used to initialize a virtual processor and place it in an operating state, to attempt recovery of a failing or stuck virtual processor, to attempt recovery of failing operating system privileged software, and for debug purposes. The defined states for each reset show an increasing amount of resource reset, such that, for example, a XIR, WDR or SIR reset will leave most architectural and memory resources unchanged, while a WMR reset will leave most memory resources unchanged but reset certain architectural resources, and a POR reset will initialize all processor resources.

All resets are processed as traps and place the virtual processor in RED_state. `RED_state` (Reset, Error, and Debug state) is a restricted execution state reserved for processing hardware- and software-initiated resets. Please refer to *Reset Traps* on page 476 and the subsections regarding reset traps in *`RED_state` Trap Processing*, which begins on page 501.

## 16.1.1 Power-on Reset (POR)

A POR reset occurs when the assigned POR pin is asserted and deasserted. During this time, all other resets and traps are ignored. POR reset has the highest trap priority. POR causes any pending external transactions to be cancelled.

The POR reset is a processor-wide reset. It affects all virtual processors on the chip, as well as all IO, cache, and DRAM subsystems.

During a POR reset, hardware sets registers to a known state (see *Machine States* on page 630). All hardware-based initialization functions are performed, all logic (including the pipeline) is initialized, all architectural registers are placed in their reset state (as defined in TABLE 16-1 on page 631), and all entries in caches and TLBs are invalidated.

A service processor may also participate in the POR reset process. The POR reset functions provided by a service processor are documented in the relevant Service Processor specification.

After a POR reset is complete:

- The first available[1] virtual processor begins executing at physical address $RSTV_{ADDR} + 20_{16}$ (`RED_state` trap vector base address plus the POR offset of $20_{16}$), with a trap type of $01_{16}$.
- All other virtual processors are in "parked" state (see *Parking and Unparking Virtual Processors* on page 608)

***Implementation Note*** From the perspective of this specification, which describes a *processor* architecture, after a Power-On Reset (POR) execution begins on one strand of the processor. However, in a multiprocessor system, after POR a service processor might arrange for execution to initially occur on only one strand per *system*. If and how that that occurs is beyond the scope of this specification and would be described in system-level documentation.

[1] per the Strand Available register (see *Strand Available Register (STRAND_AVAILABLE)* on page 605)



**Sun Proprietary/Confidential: Need-To-Know**

***Programming Note*** After a POR reset, software must initialize values that are specified as "undefined" in TABLE 16-1. In particular, I-cache tags, D-cache tags and L2 cache tags must be initialized before enabling the caches. The ITLB, DTLB and UTLB also must be initialized before enabling memory management. If a service processor participates in the reset, software should also reference the Service Processor Specification to determine which machine state has been reset by the service processor.

## 16.1.2 Warm Reset (WMR)

A Warm Reset (WMR) occurs when software writes into a particular implementation-dependent reset register or when an implementation-dependent reset input pin is asserted and then deasserted. When a WMR reset is received, all other resets and traps except POR are ignored.

The extent to which the processor is reset by a WMR reset is implementation dependent. A WMR reset may be chip-wide or it may be core-wide (reseting all virtual processors on the core, but allowing virtual processors on other cores to continue processing and maintaining cache coherency).

A WMR, even if it is chip-wide, will not alter the contents of external memory. It may, however, alter on-chip portions of the memory system (for example, store queues or cache(s)).

Warm reset has the same trap type ($1_{16}$) and trap vector offset ($20_{16}$) as a POR reset. By what means hyperprivileged software can distinguish between WMR and POR resets is implementation dependent.

**IMPL. DEP. #420-S10**: The following aspects of Warm Reset (WMR) are implemenation dependent:
(a) by what means WMR can be applied (for example, write to reset register or assertion/deassertion of an input pin)
(b) the extent to which a processor is reset by WMR (for example, single physical core, entire processor (chip), and how the on-chip memory system is affected),
(c) by what means hyperprivileged software can distinguish between WMR and POR resets

## 16.1.3 Externally Initiated Reset (XIR)

An externally initiated reset (XIR) is sent by asserting and deasserting an input pin, setting and clearing a bit in a reset register, or both.

An XIR reset is sent to all virtual processors specified in the XIR_STEERING register (impl. dep. #304-U4-Cs10). It causes an XIR reset trap in each affected virtual processor. An XIR reset trap has trap type $3_{16}$ and uses a trap vector with a physical address offset of $60_{16}$.

Memory state, cache state, and most architectural state (see TABLE 16-1) are unchanged by an XIR reset. System coherency is guaranteed to be maintained during an XIR reset. The PC (NPC) saved in TPC (TNPC) observed after an XIR will be mutually consistent, such that execution could resume using the saved PC and NPC. In effect, XIR behaves like a non-maskable interrupt.

### 16.1.4 Watchdog Reset (WDR)

An UltraSPARC Architecture virtual processor enters `error_state` when a non-reset trap, SIR reset, or XIR reset occurs at TL = *MAXTL*.

The virtual processor signals itself internally to take a watchdog reset (WDR) and sets TT to the trap type of the trap that caused entry to `error_state`. The WDR causes a trap using a trap vector with a physical address offset of $40_{16}$. WDR only affects the virtual processor on which it occurs; no other virtual processors are affected.

On a watchdog reset trap caused by a register window-related trap, CWP register is updated the same as if a WDR had not occurred.

### 16.1.5 Software-Initiated Reset (SIR)

A software-initiated reset is initiated by an SIR instruction executing on a virtual processor. This virtual processor reset has a trap type 4 and uses a trap vector with a physical address offset of $80_{16}$. SIR affects only the virtual processor on which it executes; all other virtual processors are unaffected.

## 16.2 Machine States

Machine state changes when a trap is taken at TL = *MAXTL* – 1 or when a reset occurs.

TABLE 16-1 specifies the machine states observed by software after a trap is taken at TL = *MAXTL* – 1 or after a reset occurs. For details of how those machine states are set, see processor-specific documentation and/or relevant Service Processor documentation.

The UltraSPARC Architecture specifies the machine state that must be obeserved by software after reset.

**Implemenatation Note** | For the POR reset (and possibly the WMR reset), the change in machine state may be accomplished directly by processor hardware or with support from a service processor.



**Sun Proprietary/Confidential: Need-To-Know**

**Programming Note** Virtual processor states are *only* updated according to TABLE 16-1 if RED_state is entered because of a trap at TL = *MAXTL* − 1 or a reset. If RED_state is entered because the HPSTATE.red bit was explicitly set to 1 by software, then software is responsible for setting the appropriate machine state.

In the following tables, a value marked as "Undefined" may or may not be set to a known value by hardware (and/or a service processor) after reset.

**Programming Note** Values marked as "Undefined" after POR in the following tables should be initialized by software after the power-on reset.

**TABLE 16-1** Machine State After Reset or a Trap @ TL = *MAXTL* − 1 *(1 of 3)*

| Name | Fields | POR | WMR | WDR | XIR | SIR | Traps taken @TL=*MAXTL*-1 |
|---|---|---|---|---|---|---|---|
| Integer registers | | Undefined[1] | Unchanged | | | | |
| Floating-point registers | | Undefined | | | | | |
| | | | | | | | |
| *RSTVADDR* | | VA = FFFF FFFF F000 0000$_{16}$<br>PA = 0000 7FFF F000 0000$_{16}$<br>(impl. dep. #114) | | | | | |
| PC | | *RSTVADDR* \| 20$_{16}$ | | *RSTVADDR* \| 40$_{16}$ | *RSTVADDR* \| 60$_{16}$ | *RSTVADDR* \| 80$_{16}$ | *RSTVADDR* \| A0$_{16}$ |
| NPC | | *RSTVADDR* \| 24$_{16}$ | | *RSTVADDR* \| 44$_{16}$ | *RSTVADDR* \| 64$_{16}$ | *RSTVADDR* \| 84$_{16}$ | *RSTVADDR* \| A4$_{16}$ |
| PSTATE | tct | 0 (Trap on control transfer disabled) | | | | | |
| | mm | 00$_2$ (TSO) | | | | | |
| | pef | 1 (FPU on) | | | | | |
| | am | 0 (Full 64-bit address) | | | | | |
| | priv | 0 | | | | | |
| | ie | 0 (Disable interrupts) | | | | | |
| | cle | 0 (Current not little-endian) | | | | | |
| | tle | 0 (Trap little-endian) | | Unchanged | | | |
| HPSTATE | ibe | 0 (Instruction breakpoint disabled) | | | | | |
| | red | 1 (RED_state) | | | | | |
| | hpriv | 1 (Hyperprivileged mode) | | | | | |
| | tlz | 0 (*trap_level_zero* traps disabled) | | | | | |
| TBA<63:15> | tba_high49 | Undefined | Unchanged | | | | |
| HTBA<63:14> | htba_high50 | Undefined | Unchanged | | | | |
| Y | | Undefined | Unchanged | | | | |
| PIL | | Undefined | Unchanged | | | | |
| CWP | | Undefined | Unchanged except for register window traps | | | | |
| TT[TL] | | 1 | 1 | trap type | 3 | 4 | trap type |
| CCR | | Undefined | Unchanged | | | | |
| ASI | | Undefined | Unchanged | | | | |

**TABLE 16-1** Machine State After Reset or a Trap @ TL = *MAXTL* − 1 *(2 of 3)*

| Name | Fields | POR | WMR | WDR | XIR | SIR | Traps taken @TL=*MAXTL*-1 |
|---|---|---|---|---|---|---|---|
| TL | | *MAXTL* | | **min**(TL+1, *MAXTL*) | | | |
| GL | | *MAXGL* | | **min**(GL+1, *MAXGL*) | | | |
| TPC[TL] | | Undefined | (impl.dep.# 419-S10)‡ | PC | | | |
| TNPC[TL] | | Undefined | (impl.dep.# 419-S10)‡ | NPC | | | |
| TSTATE[TL] | gl | Undefined | (impl.dep.# 419-S10)‡ | GL | | | |
| | ccr | | | CCR | | | |
| | asi | | | ASI | | | |
| | pstate | | | PSTATE | | | |
| | cwp | | | CWP | | | |
| HTSTATE[TL] | ibe | Undefined | (impl.dep.# 419-S10)‡ | HPSTATE.ibe | | | |
| | red | | | HPSTATE.red | | | |
| | hpriv | | | HPSTATE.hpriv | | | |
| | tlz | | | HPSTATE.tlz | | | |
| TICK | npt | 1 | 1 | Unchanged | | | |
| | counter | Undefined | Count | | | | |
| CANSAVE | | Undefined | Unchanged | | | | |
| CANRESTORE | | Undefined | Unchanged | | | | |
| OTHERWIN | | Undefined | Unchanged | | | | |
| CLEANWIN | | Undefined | Unchanged | | | | |
| WSTATE | other | Undefined | Unchanged | | | | |
| | normal | | | | | | |
| HVER | manuf | *Implementation dependent* (impl. dep. #104-V9) | | | | | |
| | impl | *Implementation dependent* (impl. dep. # 13-V8) | | | | | |
| | mask | Mask dependent | | | | | |
| | maxgl | *MAXGL* | | | | | |
| | maxtl | *MAXTL* | | | | | |
| | maxwin | *N_REG_WINDOWS* − 1 | | | | | |
| FSR | all | Undefined | Unchanged | | | | |
| generaGSR | all | Undefined | Unchanged | | | | |
| FPRS | fef | Undefined | Unchanged | | | | |
| | du | | | | | | |
| | dl | | | | | | |
| SOFTINT | | Undefined | Unchanged | | | | |
| HINTP | hsp | Undefined | Unchanged | | | | |
| TICK_CMPR | int_dis | Undefined | Unchanged | | | | |
| | tick_cmpr | | | | | | |
| STICK | npt | Undefined | Unchanged | | | | |
| | counter | | Count | | | | |



**Sun Proprietary/Confidential: Need-To-Know**

**TABLE 16-1** Machine State After Reset or a Trap @ TL = *MAXTL* − 1 *(3 of 3)*

| Name | Fields | POR | WMR | WDR | XIR | SIR | Traps taken @TL=*MAXTL*-1 |
|---|---|---|---|---|---|---|---|
| STICK_CMPR | int_dis | 1 | Unchanged | | | | |
| | stick_cmpr | Undefined | Unchanged | | | | |
| HSTICK_CMPR | int_dis | 1 | Unchanged | | | | |
| | hstick_cmpr | Undefined | Unchanged | | | | |
| SCRATCHPAD_*n* | | Undefined | Unchanged | | | | |
| HYP_SCRATCHPAD_*n* | | Undefined† | Unchanged† | Unchanged | Unchanged† | Unchanged | |
| D_SFAR | | Undefined | Unchanged | | | | |
| | | | | | | | |
| I-cache controls | enable(s) | 0 (disable I$) | Unchanged | | | | |
| I-cache entries | | Invalidated | Unchanged | | | | |
| I-cache data | | Undefined | Unchanged | | | | |
| D-cache controls | enable(s) | 0 (disable D$) | Unchanged | | | | |
| D-cache entries | | Invalidated | Unchanged | | | | |
| D-cache data | | Undefined | Unchanged | | | | |
| MMU controls /Demap | enable(s) | 0 (disable MMU) | Unchanged | | | | |
| MMU registers (TABLE 14-14) | all | Undefined | Unchanged | | | | |
| ITLB/DTLB/UTLB entries | | Invalidated | Unchanged | | | | |
| store queue entries | | Invalidated | | Unchanged | | | |
| L2 cache controls | eanble(s) | 1 (enable L2$) | Unchanged | | | | |
| L2 cache entries | | Invalidated | Unchanged | | | | |
| L2 cache directory | | Invalidated | Unchanged | | | | |
| L2 cache data | | Undefined | Unchanged | | | | |
| Error enable registers | | Undefined | Unchanged | | | | |
| Error Trap enable registers | | Undefined | Unchanged | | | | |
| Error Status registers | error events | Undefined[2] | Unchanged | | | | |
| Watchpoint Controls | enables | 0 (disabled) | Unchanged | | | | |
| Interrupt Queue pointers | all | Undefined | Unchanged | | | | |
| Error Queue pointers | all | Undefined | Unchanged | | | | |

† If a service processor is present, it may change the value of hyperprivileged scratchpad register(s) before execution of the reset trap handler begins

‡ **IMPL. DEP. #419-S10:** It is implementation dependent whether, after a Warm Reset (WMR), the contents of TPC[TL], TNPC[TL], TSTATE[TL], and HTSTATE[TL] are unchanged from their values before the WMR, or contain the same values saved as during a WDR, XIR, or SIR reset. (The latter implementation is the preferred one.)

1. After POR, integer register R[0] must read as zero (with good ECC/parity). The value to which all other integer and all floating-point registers are set during a Power-On Reset (POR) is Undefined. For an implementation that protect these registers with ECC/parity, the registers must be initialized with good ECC/parity as part of a POR reset, either by hardware or software.
2. For a POR reset, the Error Status register(s) can be set either by hardware or by a service processor.

## 16.2.1 Machines States for CMT

TABLE 16-2 shows the CMT machine state set by hardware as a result of a trap taken at TL = *MAXTL* – 1 or when a reset occurs.

**TABLE 16-2** Machine State After Reset and in `RED_state` for CMT Registers

| Name | Fields | POR | WMR | WDR | XIR | SIR | Traps taken @TL= *MAXTL*-1 |
|---|---|---|---|---|---|---|---|
| ***Registers shared among Virtual Processors (Strands)*** | | | | | | | |
| STRAND_AVAILABLE | | Unchanged (Predefined value, set at time of manufacture) | | | | | |
| STRAND_ENABLE_STATUS | | Copied from STRAND_AVAILABLE† (but may be changed by a service processor during reset) | Copied from STRAND_ENABLE† (but may be changed by a service processor during reset) | Unchanged | | | |
| STRAND_ENABLE | | | Unchanged | | | | |
| XIR_STEERING | | Copied from STRAND_AVAILABLE† (impl. dep. #325-U4(b)) (but may be changed by a service processor during reset) | Copied from STRAND_ENABLE† (but may be changed by a service processor during reset) | Unchanged | | | |
| STRAND_RUNNING and STRAND_RUNNING_STATUS | | Set to 0†, then during the reset either (1) virtual processor hardware sets to 1 the bit in the position corresponding to lowest-numbered *implemented and available* virtual processor (as specified by STRAND_AVAILABLE), or (2) this register is initialized by a service processor. | Set to 0†, then during the reset either (1) virtual processor hardware sets to 1 the bit in the position corresponding to lowest-numbered *enabled* virtual processor (as specified by the value of STRAND_ENABLE before the reset), or (2) this register is initialized by a service processor. | Unchanged | | | |



**Sun Proprietary/Confidential: Need-To-Know**

**TABLE 16-2** Machine State After Reset and in `RED_state` for CMT Registers *(Continued)*

| Name | Fields | POR | WMR | WDR | XIR | SIR | Traps taken @TL= *MAXTL*-1 |
|---|---|---|---|---|---|---|---|
| ***Per-Strand Registers (not shared)*** | | | | | | | |
| STRAND_ID | max_strand_id | max strand ID † | Unchanged | | | | |
| | max_core_id | max core ID † | | | | | |
| | core_id | core ID † of this core | | | | | |
| CORE_INTR_ID | core_intr_id | interrupt ID † of this core | | | | | |

† if the implementation is *always* paired with a service processor and the service processor *always* initializes this register during reset, processor hardware can leave this register unchanged (or set it to 0) and allow the service processor to perform the initialization

**Sun Proprietary/Confidential: Need-To-Know**

CHAPTER 17

# Error Handling

When a virtual processor detects an error, it must capture the error information in one or more registers, called Error Status registers (ESRs). The virtual processor then reports the event via a precise trap, a deferred trap, disrupting exception, or fatal error signal. A single error can cause multiple traps.

**Implementation Note** Every attempt must be made to make forward progress through a stuck-at fault condition (both bits and signals). No stuck-at fault that creates a correctable error should cause the processor to hang. For example, if a stuck bit in an L2 cache is causing a software-correctable error and software has to correct it by flushing the cache line and retrying the operation, then on refetch, the critical packet of data must bypass the L2 cache and fill the L1 cache directly.

Error reporting, error information and ESRs, and error handling are described in these sections:


- **Error Reporting** on page 637.
- **Error Status Registers** on page 642.
- **Protection, Detection, Reporting, and Handling of Errors** on page 646.
- **Error Handling for Common Processor Errors** on page 653.


## 17.1 Error Reporting

Virtual processors report detected errors by means of the following:

- **Precise traps.** See also *Precise Traps* on page 471.
- **Deferred traps.** See also *Deferred Traps* on page 471.
- **Deferred exceptions.**
- **Deferred traps.** See also *Disrupting Traps* on page 473.
- **Fatal error signal.**

## 17.1.1 Precise Traps

Precise traps are generated for errors that require software intervention before normal execution can be resumed. Precise traps associated with errors are always generated on the virtual processor that issued the instruction that induced the error and are reported to hyperprivileged mode.

An implementation-dependent, trap-enable bit may control the taking of a precise trap. If masked off, the trap is not taken and does not remain pending. Note that the error information in the appropriate ESR may be lost when the trap is masked off.

Note that operation is unpredictable if the trap is masked off, because the processor may be executing with uncorrected data or instructions.

An implementation-dependent, error-recording enable bit may control the recording of the error information in the appropriate ESR; if masked off, the error information is not recorded in the ESR, and the error is ignored.

**Programming Note** The precise trap handler should correct the error if it is correctable, recover from it if it is uncorrectable but recoverable, log the error if no separate disrupting exception does so, and return control to the running program. In the case where the error is neither correctable nor recoverable, the handler should deal with the running program as it sees fit, for example, signaling it, terminating it, or gracefully rebooting the system.

## 17.1.2 Deferred Traps

All deferred traps associated with errors are reported to hyperprivileged mode.

An implementation-dependent, trap-enable bit may control the taking of a deferred trap. If masked off, the trap is not taken and does not remain pending. Note that the error information in the appropriate ESR may be lost when the trap is masked off.

Note that operation is unpredictable if the trap is masked off, because the processor may be executing with uncorrected data or instructions.

An implementation-dependent, error-recording-enable bit may control the recording of the error information in the appropriate ESR. If the ESR is masked off, the error information is not recorded in the ESR, and the error is ignored.

There are two classes of deferred traps: termination deferred traps and restartable deferred traps.

*Termination deferred traps* are reported for errors when an instruction cannot be retried, completed, or recovered and the application on the virtual processor must be terminated. One example is a store buffer tag parity error or control parity error that is detected after the store instruction retires. The store cannot be completed because


**Sun Proprietary/Confidential: Need-To-Know**

of the error (for a tag parity error, the address is unknown). The store instruction cannot be retried since it is beyond the retirement point. In this case, the termination deferred trap is taken on the virtual processor that executed the trap-inducing instruction.

**Programming Note** | The termination deferred trap handlers should terminate the program.

*Restartable deferred traps* are reported for errors when the hardware provides enough information for software to recover from the error and resume execution. They are taken on the virtual processor that executed the trap-inducing instruction.

**Programming Note** | The *restartable* deferred trap handlers attempt to resume operation, based on the information provided by the hardware.

## 17.1.3 Disrupting Exceptions

A disrupting exception causes an interrupt directed toward a service processor, or a disrupting trap directed towards the appropriate virtual processor. In the context of error handling, a disrupting exception directs software to log the error and, if both necessary and possible, to clear any stored error. Service processor interrupts are outside the scope of this document. Disrupting traps are discussed in the next section.

An implementation-dependent, trap-enable bit may control the taking of a disrupting trap. If masked off, the trap is not taken, but remains pending. Note that the error information in the appropriate ESR is still valid while the trap is masked off.

An implementation-dependent, error-recording enable bit may control the recording of the error information in the appropriate ESR. If masked off, the error information is not recorded in the ESR, and the error is ignored.

### 17.1.3.1 Disrupting Traps

In nonprivileged mode and privileged mode, all disrupting traps associated with errors are presented unmasked to hyperprivileged mode.

In hyperprivileged mode, all disrupting traps associated with errors are enabled by the PSTATE.ie bit. If masked off, the disrupting trap is not taken but remains pending. The associated error information is captured in an ESR even when the trap is masked off. If the ESR is cleared while the trap is masked off, then the pending trap is also cleared and no longer remains pending.

Two classes of disrupting traps are associated with disrupting exceptions: *hw_corrected_error* disrupting traps and *sw_recoverable_error* disrupting traps.

The *hw_corrected_error* disrupting trap should be used for hardware-corrected and hardware-cleared errors. All other disrupting exceptions cause *sw_recoverable_error* disrupting traps. Implementations may create additional classes of disrupting traps.

**Programming Notes**

The *hw_corrected_error* disrupting trap handler routine should log the error.

The *sw_recoverable_error* disrupting trap handler routine should try to correct, clear, and log the error condition. If correction is not possible, the trap handler provides recovery and then logs the error condition.

## 17.1.4 Fatal Error Signaling

By its nature, a fatal error indicates a serious situation wherein there is a loss of system coherency and it is imperative to cease operation as soon as possible to prevent further breach of data integrity. The standard signaling mechanism of generating an ERROR signal to induce a system reset could be too coarse-grained and not precise. There may be significant delay in the system response to effect a reset or halt. A virtual processor encountering a fatal error may potentially continue to execute and propagate corrupted data for many cycles before being reset by the system.

All SPARC processors must provide some guarantee that upon detecting a fatal error, the processor can reach a known safe state in a bounded time. This is not to say that a processor cannot rely on the ERROR signal mechanism. In fact, for certain system designs, the ERROR signal mechanism may be sufficient. The requirement is for processor designers to validate it and if found insufficient, provide additional mechanisms to ensure that data integrity is maintained in the event of fatal errors.

Examples of additional mechanisms used in the past include generating Service Processor interrupts, “nonmaskable” traps, use of a special “cease operation” mode by which processor transactions are prevented from going out to the system bus, or combinations of these mechanisms.



**Sun Proprietary/Confidential: Need-To-Know**

# 17.2 NotData Overview

## 17.2.1 Notdata Requirement

Notdata is a signaling method to avoid superfluous reports from error detectors and thus simplify diagnosis. Notdata signals absence of data, not incorrect data. Error detectors (as opposed to notdata detectors) observing notdata should not report a hardware error. Error correctors observing notdata should pass on notdata.

When some error corrector observes a UE, it must report the UE. Propagating and possibly storing the UE elsewhere in the memory subsystem might cause confusion later, as the stored UE might be interpreted as implying the storage itself has a fault. Notdata avoids this issue. When an error corrector observes a UE, it reports the UE, but propagates notdata instead.

The bit pattern used to represent notdata is implementation-dependent. It is also implementation-dependent whether the notdata indication affects just the correction block that contained the UE, the entire containing cache line, or some number of bytes in between the two.

Processors and I/O devices reading notdata must react in whatever way is provided for dealing with the absence of data, such as a data access exception or an instruction access exception, when notdata is encountered in normal execution. However, the exception handler is not required to undertake diagnosis. Diagnosis would have been triggered by the original report of the UE.

**Programming Note** The exception handler is merely required to manage the absence of data, which may include marking the page containing the notdata as toxic, to avoid further use of the page until it is scrubbed or taken out of service. The address of notdata encountered while fetching instructions can be found from the TPC. The address of notdata encounted while fetching data can be found by disassembling the instruction pointed to by TPC, or from an implementation-dependent register provided for this purpose.

**Implementation Note** An implementation may provide a register to capture the virtual, real, or physical address of notdata encountered while fetching data.

A notdata implementations must provide the same error protection to notdata that is applied to regular data. If regular data is protected against single-bit errors, then so must notdata. Spurious transformations to or from notdata because of bit errors are highly undesirable.

Systems making use of notdata must provide a method to clear notdata from stores (that is, memory and caches). As notdata covers multiple bytes, attempting to write a smaller unit than this into the block will not clear the notdata. Notdata merged with a valid byte still produces notdata. The entire notdata block must be overwritten at once to clear notdata, in an implementation-dependent fashion.

## 17.3 Error Status Registers

Each functional block reports the error status in an Error Status register (ESR) for each class of errors it detects.

Each type of error is reported in a corresponding ESR. There are at least two ways to organize this; the exact organization is implementation dependent.

- One way is to have each type of error reported in its own ESR, called an error-type ESR. That is, errors that cause precise traps are reported in precise ESRs, errors that cause deferred traps are reported in deferred ESRs, disrupting exceptions are reported in disrupting ESRs, and fatal error conditions are reported in fatal ESRs.
- The other method of organization is by logical block, called a block-type ESR, where there is one ESR per block and a single ESR can contain error reports of different types. This method of organization usually requires the ESR to accumulate reports of errors and be paired with a First ESR (FESR) that latches a report of the first detected error irrespective of type.

In a multiple-core processor with several virtual processors per core, it may be more practical to treat each virtual processor as a functional block, such that there is one precise ESR and one disrupting ESR (for correctable and uncorrectable errors) per virtual processor. The disrupting ESR captures errors as they are detected, sets the f bit, and presents the disrupting trap immediately if enabled. Errors that result in precise traps are staged through the pipeline and are presented at retirement. The precise ESR is architecturally visible at the end of the pipeline. Since normal program exceptions must be maintained on a virtual processor basis for precise traps, it is desirable that errors causing precise traps are handled in the same way.

There are four kinds of error-type ESRs:

- **Precise ESRs** — Errors that require software correction and clearing and result in a precise trap are reported in a precise ESR. Precise ESRs are provided for each virtual processor.
- **Deferred ESRs** — Errors that result in deferred traps are reported in a deferred ESR. Deferred ESRs are provided for each virtual processor.

- **Disrupting ESRs** — Hardware-corrected and clear errors that result in an *hw_corrected_error* disrupting trap are reported in a disrupting ESR. Software-correctable errors that result in a *sw_recoverable_error* disrupting trap are also reported in disrupting ESRs. Disrupting ESRs are provided for each virtual processor.
- **Fatal ESRs** — Fatal error conditions associated with the storage system are reported in fatal ESRs. Blocks provide one fatal ESR for all virtual processors.

## 17.3.1 Elements of an Event Status Register (ESR)

The first two bits of an ESR are the same for all ESRs: the Full bit and the Multiple Event bit. The remaining bits are associated with the specific errors detected by the functional block, and contain such information as address and ECC syndrome. ESR bits are illustrated in FIGURE 17-1 and described in TABLE 17-1.

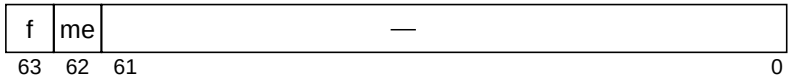



**FIGURE 17-1** Functional Block Error Status Register (ESR) Format

**TABLE 17-1** Error Status Register f and me Bit Description

| Bit | Field | Description |
|---|---|---|
| 63 | f | Full. Each ESR contains an f bit that is set to 1 by hardware the first time an error is detected for the specified ESR. Once the f bit is set to 1, it prevents further clocking by the hardware of all other bits in the register, except the me bit. The f bit must be written to 0 by software once the condition reported in the ESR has been handled. |
| 62 | me | Multiple Event. When hardware detects an error that would normally result in status being reported in an ESR but the f bit of that ESR is already 1, hardware makes no change to the ESR except writing 1 to the me bit. |

In general, hardware will write into the ESR only when it detects an error associated with that ESR. If the f bit is 0, hardware will set the f bit to 1, reset the me bit to 0, and update the rest of the status register with the error information. If the f bit is 1, hardware will just write the me bit to 1 and not modify any other bits in the register.

The setting of the f and me bits is shown in TABLE 17-2.

**TABLE 17-2** Block Error Status Register's f and me Bit State Transitions

| State Before Detection of Hardware Error | | | | State After Detection of Hardware Error | | |
|---|---|---|---|---|---|---|
| f | me | Error data | → | f | me | Error data |
| 0 | X[1] | Don't care | → | 1 | 0 | Capture error data |
| 1 | X[1] | First error data | → | 1 | 1 | First error data |

1. ‘X’ stands for either 0 or 1.

Note that in a processor that treats each virtual processor as a functional block, errors causing precise traps are staged to the end of instruction processing and presented at instruction retirement. Also, if multiple errors that cause precise traps are detected for the same instruction as it processes down the pipeline, the first error is reported, since it has the highest priority. This includes both errors and all other exception conditions associated with the instruction execution. The priority of all precise traps is listed in TABLE 12-5 on page 491.

Except for the specific case mentioned in the previous paragraph, if multiple errors are detected simultaneously, multiple bits may be set to 1 in the register. The additional error information reported in the ESR is for the highest-priority error and may or may not be correct for the other reported errors. The me bit is not set to 1 in this case. The precise behavior of a specific ESR in the event of multiple errors detected simultaneously must be documented. This does not apply to precise errors.

There is a small time window during which the me information may be lost. While servicing a reported first error, software needs to clear the f bit by writing a 0 to it. If a second error occurs and is reported by setting the me bit to 1 in the time window between software reading the ESR and writing the f bit to 0, the information that the me bit was set is lost. This can be fixed by providing "write one to clear" (W1C) access to the f and me bits. Software would have to read the ESR after clearing the f bit (but not the me bit, if it weren’t set to begin with); if the f bit were 0 but the me bit were 1, then another error occurred before the f bit was cleared.

Error-type tables are part of the detailed ESR descriptions in the model-specific *Programmer’s Reference Manual*. TABLE 17-3 depicts an example. The table column headings are explained below the table.

**TABLE 17-3** Example of an Error-Type Table

| Priority | Access | Error | FlagA | FlagB | FlagC | FieldX | FieldY | FieldZ | Corrected | Cleared |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Access A | Error condition 1 | 1 | X | 0 | ✔ | --- | --- | N | N |
| 1 | Access A | Error condition 2 | X | 1 | 0 | ✔ | ✔ | --- | Y | N |
| 2 | Access B | Error condition 3 | 0 | 0 | 1 | --- | --- | ✔ | Y | Y |

- **Priority:** The relative priority of error conditions, in case multiple errors are detected in the same clock cycle.
  - 0 indicates the highest, 2 (in the example above) indicates the lowest priority.
  - -- indicates that there are no relative priorities (not shown above).

  Several error conditions may be listed with the same relative priority. This implies either that these error conditions are mutually exclusive, or error information reported in overlapping fields is relevant to all indicated error conditions. For


**Sun Proprietary/Confidential: Need-To-Know**

some ESRs, additional priority information must be provided. In those cases, this information can be found in the *Priorities* section just following the *Error Types* section.

- **Access:** The access type (if any), such as load, store, atomic.
- **Error:** The error condition that is detected, for example, parity errors, ECC errors.
- **Flags:** The flags in the ESR that are updated for that particular error condition, such as data parity error, tag ECC error.
  - 1 indicates that this flag is set; 0 indicates that this flag is cleared.
  - X indicates that this flag may be set or cleared.

  In the example above, Error Conditions 1 and 2 may be detected and reported at the same time. However, Error Conditions 1/2 and Error Condition 3 are mutually exclusive.
- **Fields:** Supplemental error information fields in the ESR that report additional information about the error detected, such as data syndrome, way information, address.
  - ✔ indicates that this field reports relevant error information for this particular error condition.
  - --- indicates that this field does not contain relevant error information for this particular error condition and is unpredictable.

  In the example above, if Error Conditions 1 and 2 are detected at the same time, the supplemental error information reported in field X is the one for the higher-priority error condition. The relative priorities of error conditions, if any, are described as part of the detailed ESR definitions.
- **Corrected:** Indicates whether this error condition or notdata condition is corrected in hardware, such as by refetching a cache line in a clean cache which detected an error.
  - Y indicates that the error is corrected in hardware.
  - N indicates that the error is not corrected in hardware and requires software intervention.
- **Cleared:** Indicates whether this error condition or notdata condition is cleared in hardware, such as by overwriting the erroneous cache line in a cache, etc.
  - Y indicates that the error is cleared in hardware.
  - N indicates that the error is not cleared in hardware and may be reported multiple times if not cleared by software.

All unimplemented bits of the ESRs read as 0.

For *power_on_reset*, all bits of every ESR are reset to 0. For all other resets, the ESRs are not affected.

## 17.4 Protection, Detection, Reporting, and Handling of Errors

This section describes examples of hardware-software interfaces for common errors detected by the hardware for various areas of instruction processing. It is beyond the scope of this document to provide all of the cases of errors detected by each implementation of UltraSPARC Architecture processors. For a complete list of detected errors and the formats for associated Error Status registers, please refer to implementation-specific documentation.

### 17.4.1 L1 (Level-1) Caches

The L1 caches consist of the instruction cache and the data cache. There are generally three arrays of interest for the L1 caches: data, tag, and valid bits. Tag and data arrays are normally parity protected. Double-bit errors are normally not detected for the L1 caches. The Valid bit of each line is normally protected through duplication and matching of the two copies of the bit. A Valid bit error is indicated when the Valid bits have a different value.

For an L1 cache implemented with multiple associativities (ways), hardware also checks all the ways of the L1 cache index for multiple tag matches. If there are multiple clean (no tag parity/ECC error) tag matches and this is not an allowable condition for the cache (some cache designs allow multiple entries for the same tag), they should be reported as errors and all the ways of the index that match should be invalidated.

It is desirable that if there is no tag match but a tag parity error is found in one of the ways, a tag parity error is reported for the way (or ways) in error.

Error checking is provided for normal L1 cache accesses (instruction fetch to instruction cache, and data cache access for loads and stores), but not on diagnostic accesses. Diagnostic accesses must return associated parity or ECC bits.

If a tag parity/ECC error, data parity/ECC error, valid bit error, or multiple tag match error is detected, hardware should invalidate the L1 cache line (or lines), and force a cache miss. The missing cache line can be reaccessed and moved into the appropriate L1 cache from the L2 cache, L3 cache (if implemented) or memory.

The associated error information (data parity error, tag parity error, Valid bit error, multiple tag match, L1 cache index, and way) is captured in the virtual processor disrupting ESR, and a *hw_corrected_error* disrupting trap is reported, if enabled.


**Sun Proprietary/Confidential: Need-To-Know**

Alternatively, hardware can present a precise trap and allow software to invalidate the cache entry. In this case, hardware must provide a mechanism for software to invalidate the cache entry (entries) in error.

A preferred hardware implementation would be to include the valid bit and tag protection in the tag match. This would make all tag matches clean, that is, free of single-bit errors. A mechanism should then be provided to report errors in cache entries that do not match. Reporting errors on the entry to be replaced is a good choice.

## 17.4.2 TLB Errors

Processors generally provide a TLB for storage accesses when address translation is required. The TLB can either be unified, or there can be an instruction TLB (ITLB) for instruction accesses and a data TLB (DTLB) for data accesses. A micro-TLB containing local translations can also be maintained for instruction fetches (IμTLB) and data accesses (DμTLB) to improve performance. The TLB maintains address translations for the most recently used pages. There are normally one or more array structures for holding the translated addresses. Addresses can be translated either by software or by a hardware-assisted mechanism, called hardware tablewalk.

The TLB tag and data arrays are normally parity protected. The Valid bit is normally duplicated and checked.

Hardware normally checks for multiple clean (no tag parity error) tag matches. In some processor designs, this may be an allowable condition, and the processor can select the appropriate entry without reporting an error. Otherwise, the multiple hit error should be reported.

A preferred hardware implementation would be to include the valid bit and tag protection in the tag match. This would make all tag matches clean, that is, free of single-bit errors.

Error checking is provided for normal TLB accesses, but not on diagnostic TLB accesses. Diagnostic accesses must return associated parity.

Hardware has the option of correcting and clearing the errors or allowing software to do the correction and clearing.

### 17.4.2.1 Hardware-Corrected TLB Errors

If hardware does the correction and clearing, it forces a TLB miss, and clears the error condition by invalidating the TLB entry. If hardware tablewalk is enabled, hardware does the table lookup. If hardware tablewalk is disabled, hardware posts a *fast_instruction_access_MMU_miss* exception for instruction accesses, and a *fast_data_access_MMU_miss* exception for data accesses.

Hardware then reports a *hw_corrected_error* disrupting trap for logging the error, with the associated error information (data parity error, tag parity error, valid bit error, and memory address, if possible) captured in the virtual processor disrupting ESR.

#### 17.4.2.2 Software-Corrected TLB Errors

If software does the correction and clearing, a tag parity error, data parity error, or multiple tag match detected on a TLB access results in either an *instruction_access_MMU_error* (for instruction accesses) or *data_access_MMU_error* (for data accesses) exception being reported. There is no hardware correction or clearing. The associated error information (data parity error, tag parity error, multiple tag match, associated index or address, if possible) is captured in the virtual processor's precise ESR. The trap handler logs the error information and clears the error condition, using a mechanism provided by hardware (demap page or context register write, for example). The instruction at the failing address is then retried and the TLB entry is reloaded.

If other structures are associated with the TLB (such as the MMU register array or tablewalk registers), hardware detects correctable and uncorrectable errors and reports an *instruction_access_MMU_error* (for instruction accesses) or *data_access_MMU_error* (for data accesses) trap, with error information (such as the failing index) in the virtual processor's precise ESR.

### 17.4.3 Register File Errors

The integer register file (IRF) and the floating-point register file (FRF) are normally protected by a correctable error code. Hardware checks each operand's ECC. Hardware does not have to do the correction. See the model-specific Implementation Supplement to this specification for more details

If hardware does not do the correction and clearing, and an IRF correctable ECC error, IRF uncorrectable ECC error, FRF correctable ECC error, or FRF uncorrectable ECC error is detected, an *internal_processor_error* precise trap should be reported. The associated error information (IRF/FRF correctable/uncorrectable error, window (for IRF), index and ECC bits) should be captured in the precise ESR for the virtual processor. The trap handler should correct the IRF/FRF correctable ECC errors, based on the error information, using ASI accesses. For IRF/FRF uncorrectable ECC errors, the trap handler should take appropriate recovery actions. In both cases, the error should be logged.

If hardware does the correction and clearing, an *hw_corrected_error* disrupting trap is reported.



**Sun Proprietary/Confidential: Need-To-Know**

## 17.4.4 Execution Unit Errors

Execution units should be protected, either through replication or through internal arithmetic checking such as residue checking. Execution unit errors should cause an *internal_processor_error* precise trap. The associated error information should include the unique ID or IDs of the failing execution unit, or execution units if replication is used. Software should retry the instruction and ensure that forward progress is being made.

## 17.4.5 Other Core Errors Associated with Instruction Processing Before Instruction Retirement

Errors that are associated with instruction processing before the instruction retires and that require software correction should be reported as special precise error types. Error information should be provided in the precise ESR for the virtual processor.

Examples include the Store Buffer Data bypass error for loads, where the store data can have correctable or uncorrectable data ECC errors. Either an *internal_processor_error* or *data_access_error* precise trap is presented for the load. In the case of the correctable error, the trap handler can issue a MEMBAR #Sync to flush the store buffer and to allow hardware correction of the data prior to reissuing the load.

## 17.4.6 Store Errors

Store errors are errors associated with the processing of the store instruction after the store instruction has retired but before it has completed. The store data and associated address and control fields are held in buffers, waiting to be stored into the L2 cache. For large multithreaded processors, these store buffers are sufficiently large that parity or (preferably) ECC checking must be provided on the data, tag, and control fields. (If possible, SEC/DED ECC should be provided on tag, control, and data.)

Since the instruction has already retired, these errors cannot be reported as precise traps. Depending on the error condition and the hardware design, these errors can be fatal, disrupting, or deferred.

For correctable store buffer data or tag ECC errors, hardware does the correction and reports an *hw_corrected_error* disrupting trap. The error information should include the store buffer (SB) index containing the faulty data. For uncorrectable store buffer data ECC errors, hardware writes the store into the L2 cache with the notdata bit being set, and reports a *sw_recoverable_error* disrupting trap, with the associated error information (syndrome).

**Programming Note** Software should mark the page as toxic but not terminate the process at the time of the trap. If the toxic line is accessed and the appropriate trap occurs, software should take the appropriate action at that time.

If data parity is provided instead of SEC/DED for the store data buffer, all store data parity errors are treated the same as uncorrectable store buffer data ECC errors, as described in the previous paragraph.

For store buffer tag parity or control parity errors, there are two error reporting options: fatal error or terminating deferred trap. The fatal error is the least desirable since it brings down all virtual processors. If the terminating deferred trap option is selected, hardware must ensure that subsequent stores following the failing store are cancelled. If any subsequent store is stored into the L2 cache, cache coherency is compromised and hardware must signal a fatal error. Also, there should be no side effects to the store cancellation, such as lines remaining locked in the L2 cache for an atomic load/store.

The *store_error* terminating deferred trap is presented to the virtual processor with the failing store in this case. Hardware must provide a mechanism for purging the remaining store buffer entries without writing them to storage. Software terminates the virtual processor and other virtual processors in the partition in the trap handler. Software must ensure that the store buffers for the failing virtual processor are functional before reusing that virtual processor. This implies that the hyperprivileged code for the *store_error* trap handler is not run on the same virtual processor that detected the error.

## 17.4.7 Errors Not Associated with Instruction Processing

Errors not associated with instruction processing should be reported either as *hw_corrected_error* disrupting traps (if hardware does the correction and clearing), or *sw_recoverable_error* disrupting traps, with the associated error information (syndrome). Errors associated with coprocessors or with asynchronous functions are included in this category.

Implementations may choose to create other classes of disrupting traps to handle special cases.

Software can correct or reload these registers as part of the trap handler routine, and log the errors. Since these cases are unique to the processor, they must be clearly documented in the processor Implementation Supplement to this specification.



**Sun Proprietary/Confidential: Need-To-Know**

## 17.4.8 L2 Cache Errors

The L2 cache is normally a large, dirty (containing owned lines) cache and is normally the central point of coherency for the processor. The L2 cache services all memory requests from the L1 caches of each core and interfaces to an L3 cache, memory (controller), or both. The L2 cache contains the line state information (valid, modified, owned, shared, notdata, etc.).

The L2 cache tag array should be SEC/DED ECC protected for stores and snoops. For data to the L1 caches, DED ECC protection should be provided. If hardware also provides correction and clearing for L1 cache data, SEC is needed with a mechanism such as a cache scrubber for clearing the error from the L2 cache.

L2 cache data should be SEC/DED ECC protected for all writebacks and copybacks to memory. In-line correction is required on evictions. The line state should also be either SEC/DED ECC protected or replicated.

Uncorrectable tag and state errors normally result in fatal errors, which bring the processor down. Multiple "clean" tag matches also result in fatal errors. (A clean tag match is a tag match with no tag error in the matching way.) The error information (type, ECC syndrome, L2 cache index, and way) associated with the fatal error should be captured in the L2 cache fatal ESR. Some uncorrectable tag and state errors are recoverable. For example, uncorrectable tag errors on a clean line are recoverable. Uncorrectable state errors can be recoverable if a dual directory is provided. The degree of recoverability for uncorrectable tag and state errors is model dependent.

Hardware has the option of providing hardware correction and hardware clearing logic for correctable tag ECC errors for loads, instruction fetches, and TTE accesses. If hardware does the correction, it should also do the clearing; otherwise, software must be involved to clear the error.

A preferred hardware implementation would be to include the valid bit and tag protection in the tag match. This would make all tag matches clean. If there is no tag match on an L2-cache access but a correctable tag error is detected, the preferred hardware implementation would correct the tag or tags in error for the given index and then do the tag comparison again. Corrected tags should cause a *hw_corrected_error* disrupting trap being reported for logging with the associated error information (l2 $ index, way, syndrome).

Hardware can also allow software to correct and clear correctable tag and data ECC errors. If hardware does not provide the correction, correctable tag errors should be reported as a *data_access_error* precise trap for loads, an *instruction_access_error* precise trap for instruction fetches, a *data_access_MMU_error* precise trap for a TTE access for loads and an *instruction_access_MMU_error* for a TTE access for an instruction fetch. The trap is reported 1) if there was a correctable tag error on the matching way, or 2) if there is no match, but there is a tag error in a nonmatching

way for the same L2 cache index. In the latter case, the line will eventually be replaced through LRU replacement, but there is the exposure that a second cosmic event could make the tag error uncorrectable and result in a fatal error condition.

Hardware has the option of providing hardware correction and hardware clearing logic for correctable data ECC errors for loads, instruction fetches, and TTE accesses. If hardware does the correction, it should also do the clearing; otherwise, software must be involved to clear the error. If in-line ECC correction is provided from the L2-cache to the cores, the L2-cache should also provide an error-clearing mechanism, such as a scrubber.

Hardware can allow software to recover from uncorrectable or correctable data ECC errors. For these L2 cache errors, hardware reports either a *data_access_error* precise trap for loads, an *instruction_access_error* precise trap for instruction fetches, a *data_access_MMU_error* precise trap for a TTE access for loads with hardware tablewalk enabled, and an *instruction_access_MMU_error* for a TTE access for an instruction fetch with hardware tablewalk enabled. The appropriate error information (correctable data, uncorrectable data, notdata, L2 cache index and way, ECC syndrome) is captured in ESRS.

In the trap handler for the *data_access_error* and the *instruction_access_error*, software uses the prefetch invalidate variant (or equivalent) to invalidate the line from the L2 cache. If the line is clean, it is invalidated without a writeback. If the line is dirty, the writeback process causes the correctable tag and data errors to be corrected and cleared. When the instruction is reissued, the L2 cache is updated with a clean copy from memory. In the case of an uncorrectable data ECC error, if the line is dirty, the writeback causes the notdata bit to be set in memory. If software cannot recover from this condition, it terminates the user. It does not affect other virtual processors unless they are part of the same partition.

L2 cache errors are detected for operations asynchronous to instruction processing, such as snoops, copybacks, and writebacks. These errors include correctable store or snoop tag ECC error, correctable writeback and copyback data ECC error, and uncorrectable writeback and copyback data ECC error. If the hardware corrects and clears the error, it posts an *hw_corrected_error* disrupting trap. Software logs the error for this trap handler. If software participates in the correction or clearing, a *sw_recoverable_error* disrupting trap is reported. In this trap handler, software clears and also may correct the error by invalidating the line in the L2 cache. The error is then logged. The error information (error type, syndrome, virtual processor, L2 cache index, and way) is captured in the L2 cache disrupting ESR.

## 17.4.9 External Interface and Bus Errors

Errors detected on the external interface that result in system data coherency being compromised are captured in the External Interface Unit fatal ESR. The processor signals a fatal error. In a multiple processor configuration, the error may cause the processor to stop sending new packets on the interface and also to stop picking new

bus transactions. The processor will subsequently stall, waiting for data, and become isolated from the system, thereby preventing the propagation of the error to other processors in the configuration.

Uncorrectable data errors detected on data accesses in the External Interface Unit may require unique disrupting trap types to report the termination of the access. See the model-specific PRM for details.

## 17.5 Error Handling for Common Processor Errors

TABLE 17-4 provides the error handling process for the most common core and processor errors. A model's Implementation Supplement to this specification should contain a detailed list of all precise traps and the other information contained in this table.

**TABLE 17-4** Error Handling Table *(1 of 3)*

| ERROR | TRAP VECTOR | ESR INFO | HW Corr | HW Clr | Trap Handler Action |
|---|---|---|---|---|---|
| Icache Valid bit | *hw_corrected_error* disrupting | index, set | Y | Y | log error |
| Icache tag parity/ECC | *hw_corrected_error* disrupting | index, set | Y | Y | log error |
| Icache multiple hit | *hw_corrected_error* disrupting | index, set | Y | Y | log error |
| Icache data parity/ECC; Option#1 | *hw_corrected_error* disrupting | index, set | Y | Y | log error |
| Icache data parity/ECC; Option#2 | *inst_access__error* precise | index, set | N | N | SW invalidates line in Icache |
| ITLB Valid bit: Option #1 | *inst_access_MMU_error* precise | valid | N | N | SW demaps page |
| ITLB Valid bit: Option #2 | *hw_corrected_error* disrupting | valid, instr address | Y | Y | log error |
| ITLB tag parity: Option #1 | *inst_access_MMU_error* precise | tag parity | N | N | SW demaps page |
| ITLB tag parity: Option #2 | *hw_corrected_error* disrupting | tag parity, instr address | Y | Y | log error |
| ITLB multiple tag hit: Option #1 | *inst_access_MMU_error* precise | multiple hit, way | N | N | SW demaps all |
| ITLB multiple tag hit: Option #2 | *hw_corrected_error* disrupting | multiple hit, way, instr addr | Y | Y | log error |
| ITLB data parity: Option #1 | *inst_access_MMU_error* precise | data parity | N | N | SW demaps page |
| ITLB data parity: Option #2 | *hw_corrected_error* disrupting | data parity, instr addr | Y | Y | log error |
| Data cache Valid bit | *hw_corrected_error* disrupting | valid, index, set | Y | Y | SW logs error |
| Data cache tag parity | *hw_corrected_error* disrupting | tag parity, index, set | Y | Y | SW logs error |
| Data cache multiple hit | *hw_corrected_error* disrupting | multiple hit, way, index, | Y | Y | SW logs error |
| Data cache data parity/ ECC; Option#1 | *hw_corrected_error* disrupting | data parity, index, set | Y | Y | SW logs error |
| Data cache data parity/ ECC; Option#2 | *data_access__error* precise | index, set | N | N | SW invalidates line in Data cache |
| DTLB Valid bit: Option #1 | *data_access_MMU_error* precise | valid, data address | N | N | SW demaps page |
| DTLB Valid bit: Option #2 | *hw_corrected_error* disrupting | valid, data address | Y | Y | SW log error |
| DTLB tag parity: Option #1 | *data_access_MMU_error* precise | tag parity, data address | N | N | SW demaps page |
| DTLB tag parity: Option #2 | *hw_corrected_error* disrupting | tag parity, data address | Y | Y | SW log error |
| DTLB multiple tag hit: Option #1 | *data_access_MMU_error* precise | Multiple hit, way, data addr | N | N | SW demaps all |



**Sun Proprietary/Confidential: Need-To-Know**

**TABLE 17-4** Error Handling Table *(2 of 3)*

| ERROR | TRAP VECTOR | ESR INFO | HW Corr | HW Clr | Trap Handler Action |
|---|---|---|---|---|---|
| DTLB multiple tag hit: Option #2 | *hw_corrected_error* disrupting | Multiple hit, way, data addr | Y | Y | SW log error |
| DTLB data parity: Option #1 | *data_access_MMU_error* precise | data parity, data address | N | N | SW demaps page |
| DTLB data parity: Option #2 | *hw_corrected_error* disrupting | data parity, data address | Y | Y | SW log error |
| Integer RF correctable ECC | *internal_processor_error* precise | window, index, ECC bits | N | N | SW should correct |
| Integer RF uncorrectable ECC Option#1 | *internal_processor_error* precise | window, index, ECC bits | N | N | SW should retry and, if unsuccessful, terminate user |
| Integer RF uncorrectable ECC Option#2 | *hw_corrected_error* disrupting | window, index, ECC bits | Y | Y | SW log error |
| Floating Point RF correctable ECC | *internal_processor_error* precise | index, ECC bits | N | N | SW should correct |
| Floating Point RF uncorrectable ECC | *internal_processor_error* precise | index, ECC bits | N | N | SW should terminate user |
| Integer residue error/ Integer result mismatch | *internal_processor_error* precise | integer unit ID/coreID | N | N | SW should retry the failing instruction one or more times |
| Floating Point residue error | *internal_processor_error* precise | FP unit ID/ coreID | N | N | SW should retry the failing instruction |
| Correctable Store Buffer data ECC Error | *hw_corrected_error* disrupting | SB corr ECC, SB index | Y | Y | SW logs error |
| Uncorrectable Store Buffer data Parity/ECC Error | *sw_recoverable_error* disrupting | SB uncorr ECC, SB index | N | N | SW logs the error and marks the page as toxic |
| Store Buffer tag/control parity/ECC error: Option #1 | *store_error* deferred | SB tag/ctrl parity, SB index | N | N | SW terminates user |
| Store Buffer tag/control parity/ECC error: Option #2 | fatal error | SB tag/ctrl parity, SB index | N | N | fatal signal |
| L2 cache correctable ECC tag/data on ifetch:Option #1 | *inst_access_error* precise | index, way, syndrome | N | N | SW invalidates line in L2$, and retries instruction. |
| L2 cache correctable ECC tag/data on ifetch:Option #2 | *hw_corrected_error* disrupting | index, way, syndrome | Y | Y | SW log error |
| L2 cache uncorrectable ECC data on ifetch | *inst_access_error* precise | index, way, syndrome | N | N | SW invalidates line in L2$ and tries to recover |
| L2 cache notdata on ifetch | *inst_access_error* precise | index, way, syndrome | N | N | SW invalidates line in L2$ and tries to recover |
| L2 cache TTE correctable ECC tag/data on ifetch:Option #1 | *inst_access_MMU_error* precise | index, way, syndrome | N | N | SW attempts to refill the TLB and retry |
| L2 cache TTE correctable ECC tag/data on ifetch:Option #2 | *hw_corrected_error* disrupting | index, way, syndrome | Y | Y | SW log error |
| L2 cache TTE uncorrectable ECC data on ifetch | *inst_access_MMU_error* precise | index, way, syndrome | N | N | SW attempts to refill the TLB and retry |
| L2 cache TTE notdata on ifetch | *inst_access_MMU_error* precise | index, way, syndrome | N | N | SW attempts to refill the TLB, and retry |

**TABLE 17-4** Error Handling Table *(3 of 3)*

| ERROR | TRAP VECTOR | ESR INFO | HW Corr | HW Clr | Trap Handler Action |
|---|---|---|---|---|---|
| L2 cache correctable ECC tag/data on load: Option #1 | *data_access_error* precise | index, way, syndrome | N | N | SW invalidates line in L2$ and retries instruction |
| L2 cache correctable ECC tag/data on load: Option #2 | *hw_corrected_error* disrupting | index, way, syndrome | Y | Y | SW log error |
| L2 cache uncorrectable ECC data on load: | *data_access_error* precise | index, way, syndrome | N | N | SW invalidates line in L2$ and tries to recover |
| L2 cache notdata on load: | *data_access_error* precise | index, way, syndrome | N | N | SW invalidates line in L2$ and tries to recover |
| L2 cache TTE correctable ECC tag/data on load:Option #1 | *data_access_MMU_error* precise if hardware tablewalk enabled | index, way, syndrome | N | N | SW attempts to refill the TLB, retries instruction |
| L2 cache TTE correctable ECC tag on load: Option #2 | *hw_corrected_error* disrupting | index, way, syndrome | Y | Y | SW log error |
| L2 cache TTE uncorrectable ECC data on load | *data_access_MMU_error* precise if hardware tablewalk enabled | index, way, syndrome | N | N | SW attempts to refill the TLB, retries instruction |
| L2 cache TTE notdata on load | *data_access_MMU_error* precise | index, way, syndrome | N | N | SW terminates user |



**Sun Proprietary/Confidential: Need-To-Know**

APPENDIX **A**

# Opcode Maps

This appendix contains the UltraSPARC Architecture 2007 instruction opcode maps.

In this appendix and in Chapter 7, *Instructions*, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE 7-1 on page 134. For preferred substitute instructions for deprecated opcodes, see the individual opcodes in Chapter 7 that are labeled "Deprecated".

In the tables in this appendix, *reserved* (—) and shaded entries (as defined below) indicate opcodes that are not implemented in UltraSPARC Architecture 2007 strands.

| Shading | Meaning |
|---|---|
| | An attempt to execute opcode will cause an *illegal_instruction* exception. |

An attempt to execute a reserved opcode behaves as defined in *Reserved Opcodes and Instruction Fields* on page 130.

**TABLE A-1** op{1:0}

| op {1:0} | | | |
|---|---|---|---|
| **0** | **1** | **2** | **3** |
| Branches and SETHI (*See* TABLE A-2) | CALL | Arithmetic & Miscellaneous (*See* TABLE A-3) | Loads/Stores (*See* TABLE A-4) |

**TABLE A-2** op2{2:0} (op = 0)

| op2 {2:0} | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| ILLTRAP | BPcc (*See* TABLE A-7) | Bicc$^{D}$ (*See* TABLE A-7) | BPr (bit 28 = 0) (*See* TABLE A-8) | SETHI | FBPfcc (*See* TABLE A-7) | FBfcc$^{D}$ (*See* TABLE A-7) | — |
| | | | — (bit 28 = 1)[1] | NOP[2] | | | |

1. See the footnote regarding bit 28 on page 159.
2. rd = 0, imm22 = 0

**TABLE A-3** op3{5:0} (op = $10_2$) *(1 of 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>SIR[H] (rd = 15, rs1 = 0, i = 1)<br>— (rd = 15) **and** (rs1 ≠ 0 **or** i ≠ 1))<br>— (rd = 7 – 14)<br>WRFPRS (rd = 6)<br>WRasr[PASR] (7 ≤ rd ≤ 14)<br>WRPCR[P] (rd = 16)<br>WRPIC (rd = 17)<br>— (rd = 18)<br>WRGSR (rd = 19)<br>WRSOFTINT_SET[P] (rd = 20)<br>WRSOFTINT_CLR[P] (rd = 21)<br>WRSOFTINT[P] (rd = 22)<br>WRTICK_CMPR[P] (rd = 23)<br>WRSTICK[H] (rd = 24)<br>WRSTICK_CMPR[P] (rd = 25)<br>— (rd = 26 - 31) |
| | **1** | AND | ANDcc | TSUBcc | SAVED[P] (fcn = 0)<br>RESTORED[P] (fcn = 1)<br>ALLCLEAN[P] (fcn = 2)<br>OTHERW[P] (fcn = 3)<br>NORMALW[P] (fcn = 4)<br>INVALW[P] (fcn = 5)<br>— (fcn ≥ 6) |
| | **2** | OR | ORcc | TADDccTV[D] | — |
| | **2** | OR | ORcc | TADDccTV[D] | WRPR[P] (rd = 0-14 **or** 16)<br>— (rd = 15 **or** 17–31) |
| | **3** | XOR | XORcc | TSUBccTV[D] | WRHPR[H] |
| | **4** | SUB | SUBcc | MULScc[D] | FPop1 (*See* TABLE A-5) |
| | **5** | ANDN | ANDNcc | SLL (x = 0), SLLX (x = 1) | FPop2 (*See* TABLE A-6) |
| | **6** | ORN | ORNcc | SRL (x = 0), SRLX (x = 1) | IMPDEP1 (VIS) (*See* TABLE A-12) |
| | **7** | XNOR | XNORcc | SRA (x = 0), SRAX (x = 1) | |

**TABLE A-3** op3{5:0} (op = $10_2$) *(2 of 2)*

| | | op3{5:4} | | | |
|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** |
| **op3 {3:0}** | **8** | ADDC | ADDCcc | RDY[D] (rs1 = 0, i = 0) | JMPL |
| | | | | — (rs1 = 1, i = 0) | |
| | | | | RDCCR (rs1 = 2, i = 0) | |
| | | | | RDASI (rs1 = 3, i = 0) | |
| | | | | RDTICK[Pnpt] (rs1 = 4, i = 0) | |
| | | | | RDPC (rs1 = 5, i = 0) | |
| | | | | RDFPRS (rs1 = 6, i = 0) | |
| | | | | RDasr[PASR] (7 ≤ rd ≤ 14, i = 0) | |
| | | | | MEMBAR (rs1 = 15, rd = 0, i = 1, instruction bit 12 = 0) | |
| | | | | — (rs1 = 15, rd = 0, i = 1, instruction bit 12 = 1) | |
| | | | | — (i = 1, (rs1 ≠ 15 or rd ≠ 0)) | |
| | | | | — (rs1 = 15, rd = 0, i = 0) | |
| | | | | — (rs1 = 15 **and** rd > 0 **and** i = 0) | |
| | | | | RDPCR[P] (rs1 = 16 **and** i = 0) | |
| | | | | RDPIC (rs1 = 17 **and** i = 0) | |
| | | | | — (rs1 = 18 **and** i = 0) | |
| | | | | RDGSR (rs1 = 19 **and** i = 0) | |
| | | | | — (rs1 = 20 or 21) **and** (i = 0)) | |
| | | | | RDSOFTINT[P] (rs1 = 22 **and** i = 0) | |
| | | | | RDTICK_CMPR[P] (rs1 = 23 **and** i = 0) | |
| | | | | RDSTICK (rs1 = 24 **and** i = 0) | |
| | | | | RDSTICK_CMPR[P] (rs1 = 25 **and** i = 0) | |
| | | | | — ((rs1 = 26 – 31) **and** (i = 0)) | |
| | **9** | MULX | — | RDHPR[H] | RETURN |
| | **A** | UMUL[D] | UMULcc[D] | RDPR[P] (rs1 = 1–14 or 16) | Tcc ((i = 0 **and** inst{10:5} = 0) **or** ((i = 1) **and** (inst{10:8} = 0))) (*See* TABLE A-7) |
| | | | | — (rs1 = 15 or 17 – 31) | — (bit 29 = 1) — ((i = 0 **and** (inst{10:5} ≠ 0)) **or** (i = 1 **and** (inst{10:8} ≠ 0)) |
| | **B** | SMUL[D] | SMULcc[D] | FLUSHW | FLUSH |
| | **C** | SUBC | SUBCcc | MOVcc | SAVE |
| | **D** | UDIVX | — | SDIVX | RESTORE |
| | **E** | UDIV[D] | UDIVcc[D] | POPC (rs1 = 0) | DONE[P] (fcn = 0) |
| | | | | — (rs1 > 0) | RETRY[P] (fcn = 1) |
| | | | | | — (fcn = 2..15) |
| | | | | | — (fcn = 16..31) |
| | **F** | SDIV[D] | SDIVcc[D] | MOVr (*See* TABLE A-8) | — |

**op3 {3:0}**

**TABLE A-4** op3{5:0} (op = $11_2$)

| | | op3{5:4} | | | |
|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** |
| **op3 {3:0}** | **0** | LDUW | LDUWA[PASI] | LDF | LDFA[PASI] |
| | **1** | LDUB | LDUBA[PASI] | (rd = 0) LDFSR[D]<br>(rd = 1) LDXFSR<br>— (rd > 1) | — |
| | **2** | LDUH | LDUHA[PASI] | LDQF | LDQFA[PASI] |
| | **3** | LDTW[D]<br>— (rd odd) | LDTWA[D, PASI]<br>LDTXA<br>— (rd odd) | LDDF | LDDFA[PASI]<br>LDBLOCKF<br>LDSHORTF |
| | **4** | STW | STWA[PASI] | STF | STFA[PASI] |
| | **5** | STB | STBA[PASI] | STFSR[D], STXFSR<br>— (rd > 1) | — |
| | **6** | STH | STHA[PASI] | STQF | STQFA[PASI] |
| | **7** | STTW[D]<br>— (rd odd) | STTWA[PASI]<br>— (rd odd) | STDF | STDFA[PASI]<br>STLBLOCKF<br>STPARTIALF<br>STSHORTF |
| | **8** | LDSW | LDSWA[PASI] | *Reserved* | *Reserved* |
| | **9** | LDSB | LDSBA[PASI] | *Reserved* | *Reserved* |
| | **A** | LDSH | LDSHA[PASI] | *Reserved* | *Reserved* |
| | **B** | LDX | LDXA[PASI] | *Reserved* | *Reserved* |
| | **C** | *Reserved* | *Reserved* | — | CASA[PASI] |
| | **D** | LDSTUB | LDSTUBA[PASI] | PREFETCH<br>— (fcn = 5 – 15) | PREFETCHA[PASI]<br>— (fcn = 5 – 15) |
| | **E** | STX | STXA[PASI] | — | CASXA[PASI] |
| | **F** | SWAP[D] | SWAPA[D, PASI] | *Reserved* | *Reserved* |

**TABLE A-5** opf{8:0} (op = $10_2$, op3 = $34_{16}$ = FPop1)

| | opf{3:0} | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **opf{8:4}** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| $00_{16}$ | — | FMOVs | FMOVd | FMOVq | — | FNEGs | FNEGd | FNEGq |
| $01_{16}$ | — | — | — | — | — | — | — | — |
| $02_{16}$ | — | — | — | — | — | — | — | — |
| $03_{16}$ | — | — | — | — | — | — | — | — |
| $04_{16}$ | — | FADDs | FADDd | FADDq | — | FSUBs | FSUBd | FSUBq |
| $05_{16}$ | — | — | — | — | — | — | — | — |
| $06_{16}$ | — | — | — | — | — | — | — | — |
| $07_{16}$ | — | — | — | — | — | — | — | — |
| $08_{16}$ | — | FsTOx | FdTOx | FqTOx | FxTOs | — | — | — |
| $09_{16}$ | — | — | — | — | — | — | — | — |
| $0A_{16}$ | — | — | — | — | — | — | — | — |
| $0B_{16}$ | — | — | — | — | — | — | — | — |
| $0C_{16}$ | — | — | — | — | FiTOs | — | FdTOs | FqTOs |
| $0D_{16}$ | — | FsTOi | FdTOi | FqTOi | — | — | — | — |
| $0E_{16}$–$1F_{16}$ | — | — | — | — | — | — | — | — |

| | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
|---|---|---|---|---|---|---|---|---|
| $00_{16}$ | — | FABSs | FABSd | FABSq | — | — | — | — |
| $01_{16}$ | — | — | — | — | — | — | — | — |
| $02_{16}$ | — | FSQRTs | FSQRTd | FSQRTq | — | — | — | — |
| $03_{16}$ | — | — | — | — | — | — | — | — |
| $04_{16}$ | — | FMULs | FMULd | FMULq | — | FDIVs | FDIVd | FDIVq |
| $05_{16}$ | — | — | — | — | — | — | — | — |
| $06_{16}$ | — | FsMULd | — | — | — | — | FdMULq | — |
| $07_{16}$ | — | — | — | — | — | — | — | — |
| $08_{16}$ | FxTOd | — | — | — | FxTOq | — | — | — |
| $09_{16}$ | — | — | — | — | — | — | — | — |
| $0A_{16}$ | — | — | — | — | — | — | — | — |
| $0B_{16}$ | — | — | — | — | — | — | — | — |
| $0C_{16}$ | FiTOd | FsTOd | — | FqTOd | FiTOq | FsTOq | FdTOq | — |
| $0D_{16}$ | — | — | — | — | — | — | — | — |
| $0E_{16}$–$1F_{16}$ | — | — | — | — | — | — | — | — |

**TABLE A-6** opf{8:0} (op = $10_2$, 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) | — | † ‡ | † ‡ | † ‡ | — |
| $01_{16}$ | — | — | — | — | — | — | — | — | — |
| $02_{16}$ | — | — | — | — | — | FMOVRsZ ‡ | FMOVRdZ ‡ | FMOVRqZ ‡ | — |
| $03_{16}$ | — | — | — | — | — | — | — | — | — |
| $04_{16}$ | — | FMOVs (fcc1) | FMOVd (fcc1) | FMOVq (fcc1) | — | FMOVRsLEZ ‡ | FMOVRdLEZ ‡ | FMOVRqLEZ ‡ | — |
| $05_{16}$ | — | FCMPs | FCMPd | FCMPq | — | FCMPEs ‡ | FCMPEd ‡ | FCMPEq ‡ | — |
| $06_{16}$ | — | — | — | — | — | FMOVRsLZ ‡ | FMOVRdLZ ‡ | FMOVRqLZ ‡ | — |
| $07_{16}$ | — | — | — | — | — | — | — | — | — |
| $08_{16}$ | — | FMOVs (fcc2) | FMOVd (fcc2) | FMOVq (fcc2) | — | † | † | † | — |
| $09_{16}$ | — | — | — | — | — | — | — | — | — |
| $0A_{16}$ | — | — | — | — | — | FMOVRsNZ ‡ | FMOVRdNZ ‡ | FMOVRqNZ ‡ | — |
| $0B_{16}$ | — | — | — | — | — | — | — | — | — |
| $0C_{16}$ | — | FMOVs (fcc3) | FMOVd (fcc3) | FMOVq (fcc3) | — | FMOVRsGZ ‡ | FMOVRdGZ ‡ | FMOVRqGZ ‡ | — |
| $0D_{16}$ | — | — | — | — | — | — | — | — | — |
| $0E_{16}$ | — | — | — | — | — | FMOVRsGEZ ‡ | FMOVRdGEZ ‡ | FMOVRqGEZ ‡ | — |
| $0F_{16}$ | — | — | — | — | — | — | — | — | — |
| $10_{16}$ | — | 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

**TABLE A-7** cond{3:0}

| | | BPcc<br>op = 0<br>op2 = 1 | Bicc<br>op = 0<br>op2 = 2 | FBPfcc<br>op = 0<br>op2 = 5 | FBfcc$^{D}$<br>op = 0<br>op2 = 6 | Tcc<br>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 A-8** Encoding of rcond{2:0} Instruction Field

| | | BPr<br>op = 0<br>op2 = 3 | MOVr<br>op = 2<br>op3 = $2F_{16}$ | FMOVr<br>op = 2<br>op3 = $35_{16}$ |
|---|---|---|---|---|
| rcond {2:0} | 0 | — | — | — |
| | 1 | BRZ | MOVRZ | FMOVR<s \| d \| q>Z |
| | 2 | BRLEZ | MOVRLEZ | FMOVR<s \| d \| q>LEZ |
| | 3 | BRLZ | MOVRLZ | FMOVR<s \| d \| q>LZ |
| | 4 | — | — | — |
| | 5 | BRNZ | MOVRNZ | FMOVR<s \| d \| q>NZ |
| | 6 | BRGZ | MOVRGZ | FMOVR<s \| d \| q>GZ |
| | 7 | BRGEZ | MOVRGEZ | FMOVR<s \| d \| q>GEZ |

**TABLE A-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 A-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 A-11** cc Fields (BPcc and Tcc)

| cc1 | cc0 | Condition Code Selected |
|---|---|---|
| 0 | 0 | icc |
| 0 | 1 | — |
| 1 | 0 | xcc |
| 1 | 1 | — |

**TABLE A-12** IMPDEP1: opf{8:0} for VIS opcodes (op = $10_2$, 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$^{D,P}$ |
| | **0** | EDGE8 | ARRAY8 | FCMPLE16 | — | | FPADD16 | FZERO | FAND | SHUTDOWN$^{D,P}$ |
| | **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 | FALIGN DATA | — | FANDNOT1 | FSRC2 | — |
| | **9** | EDGE32N | BMASK | — | FMULD 8ULx16 | — | — | FANDNOT1S | FSRC2S | — |
| | **A** | EDGE32L | ALIGN ADDRESS _LITTLE | FCMPEQ16 | FPACK32 | — | — | FNOT1 | FORNOT1 | — |
| | **B** | EDGE32LN | — | — | FPACK16 | FPMERGE | — | FNOT1S | FORNOT1S | — |
| | **C** | — | — | FCMPGT32 | — | BSHUFFLE | — | FXOR | FOR | — |
| | **D** | — | — | — | FPACKFIX | FEXPAND | — | FXORS | FORS | — |
| | **E** | — | — | FCMPEQ32 | PDIST | — | — | FNAND | FONE | — |
| | **F** | — | — | — | — | — | — | FNANDS | FONES | — |

**TABLE A-14** IMPDEP1: opf{8:0} for VIS opcodes (op = $10_2$, op3 = $36_{16}$) *(3 of 3)*

| | | opf {8:4} | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16–1F |
| opf {3:0} | 0 | — | — | — | — | — | — | — | — |
| | 1 | — | — | — | — | — | — | — | — |
| | 2 | — | — | — | — | — | — | — | — |
| | 3 | — | — | — | — | — | — | — | — |
| | 4 | — | — | — | — | — | — | — | — |
| | 5 | — | — | — | — | — | — | — | — |
| | 6 | — | — | — | — | — | — | — | — |
| | 7 | — | — | — | — | — | — | — | — |
| | 8 | — | — | — | — | — | — | — | — |
| | 9 | — | — | — | — | — | — | — | — |
| | A | — | — | — | — | — | — | — | — |
| | B | — | — | — | — | — | — | — | — |
| | C | — | — | — | — | — | — | — | — |
| | D | — | — | — | — | — | — | — | — |
| | E | — | — | — | — | — | — | — | — |
| | F | — | — | — | — | — | — | — | — |

**TABLE A-13** op5{3:0} (op = $10_2$, op3 = $37_{16}$ = FMAf (IMPDEP2)

| | | op5{1:0} | | | |
|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** |
| **op5{3:2}** | **0** | — | FMADDs | FMADDd | — |
| | **1** | — | FMSUBs | FMSUBd | — |
| | **2** | — | FNMSUBs | FNMSUBd | — |
| | **3** | — | FNMADDs | FNMADDd | — |

APPENDIX **B**

**Note: This chapter is undergoing final review; please check back later for a copy of UltraSPARC Architecture 2007 containing the final version of this chapter.**

# 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 B-1 on page 671.

The appendix contains these sections:


- **Definition of an Implementation Dependency** on page 669.
- **Hardware Characteristics** on page 670.
- **Implementation Dependency Categories** on page 670.
- **List of Implementation Dependencies** on page 671.


## B.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 *UltraSPARC Architecture 2007 Compliance with SPARC V9 Architecture* on page 25.

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.

## B.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

## B.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-V8).

- **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 the Version register (VER) (impl. dep. #13-V8) and the FSR.ver field (impl. dep. #19-V8).
- **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-V8-Cs10).
- **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-V8) and some alternate address spaces (impl. dep. #29-V8).

# B.4 List of Implementation Dependencies

TABLE B-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; bold face indicates the main page on which the implementation dependency is described.

**TABLE B-1** SPARC V9 Implementation Dependencies *(1 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **1-V8** | f | **Software emulation of instructions**<br>Whether an instruction complies with UltraSPARC Architecture 2007 by being implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent. | **25** |
| **2-V8** | v | **Number of IU registers**<br>An UltraSPARC Architecture implementation may contain from 72 to 640 general-purpose 64-bit R registers. This corresponds to a grouping of the registers into *MAXGL* + 1 sets of global R registers plus a circular stack of *N_REG_WINDOWS* sets of 16 registers each, known as register windows. The number of register windows present (*N_REG_WINDOWS*) is implementation dependent, within the range of 3 to 32 (inclusive). | **26**, 50 |
| **3-V8** | 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. In this case, software running in a higher privilege mode shall emulate any functionality not present in the hardware. | **130** |
| **4**, **5** | | *Reserved*. | |

**TABLE B-1** SPARC V9 Implementation Dependencies *(2 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **6-V8** | f | **I/O registers privileged status**<br>Whether I/O registers can be accessed by nonprivileged code is implementation dependent. | **29** |
| **7-V8** | t | **I/O register definitions**<br>The contents and addresses of I/O registers are implementation dependent. | **29** |
| **8-V8-Cs20** | t | **RDasr/WRasr target registers**<br>Ancillary state registers (ASRs) in the range 0–27 that are not defined in UltraSPARC Architecture 2007 are reserved for future architectural use. ASRs in the range 28–31 are available to be used for implementation-dependent purposes. | **31**, 69, 305, 381 |
| **9-V8-Cs20** | f | **RDasr/WRasr privileged status**<br>The privilege level required to execute each of the implementation-dependent read/write ancillary state register instructions (for ASRs 28–31) is implementation dependent. | **31**, 69, 305, 381 |
| **10-V8–12-V8** | | *Reserved.* | |
| **13-V8** | a | **HVER**.impl<br>HVER.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. | **106** |
| **14-V8–15-V8** | | *Reserved.* | |
| **16-V8-Cu3** | | *Reserved.* | |
| **17-V8** | | *Reserved.* | |

**TABLE B-1** SPARC V9 Implementation Dependencies *(3 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **18-V8-Ms10** | 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 FSR.ns = 1 and a floating-point *source operand* is subnormal, an implementation may treat the subnormal operand as if it were a floating-point zero value of the same sign.<br>The cases in which this replacement is performed are implementation dependent. However, if it occurs,<br>(1) it should *not* apply to FABS, FMOV, or FNEG instructions and<br>(2) FADD, FSUB, and FCMP should give identical treatment to subnormal source operands.<br>Treating a subnormal source operand as zero may generate an IEEE 754 floating-point "inexact", "division by zero", or "invalid" condition (see *Current Exception (cexc)* on page 66). Whether the generated condition(s) trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem. | **394** |
| | | **b:** When a floating-point operation generates a subnormal *result* value, an UltraSPARC Architecture implementation may either write the result as a subnormal value or replace the subnormal result by a floating-point zero value of the same sign and generate IEEE 754 floating-point "inexact" and "underflow" conditions. Whether these generated conditions trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem. | **395** |
| | | **c:** If an FPop (notably, FMAu) generates an *intermediate* result value, the intermediate value is subnormal, and FSR.ns = 1, it is implementation dependent whether (1) the operation continues, using the subnormal value (possibly with some loss of accuracy), or (2) the virtual processor replaces the subnormal intermediate value with a floating-point zero value of the same sign, generates IEEE 754 floating-point "inexact" and "underflow" conditions, completes the instruction, and writes a final result (possibly with some loss of accuracy). Whether generated IEEE conditions trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem. | **395** |
| **19-V8** | a | **FPU version, FSR.ver**<br>Bits 19:17 of the FSR, FSR.ver, identify one or more implementations of the FPU architecture. | **62** |
| **20-V8–21-V8** | | *Reserved.* | |
| **22-V8** | f | **FPU tem, cexc, and aexc**<br>An UltraSPARC Architecture implementation implements the tem, cexc, and aexc fields in hardware, conformant to IEEE Std 754-1985. | 69 |
| **23-V8** | | *Reserved.* | |
| **24-V8** | | *Reserved.* | |
| **25-V8** | f | **RDPR of FQ with nonexistent FQ**<br>An UltraSPARC Architecture implementation does not contain a floating-point queue (FQ). Therefore, FSR.ftt = 4 (sequence_error) does not occur, and an attempt to read the FQ with the RDPR instruction causes an *illegal_instruction* exception. | 65, 310 |
| **26-V8–28-V8** | | *Reserved.* | |

**TABLE B-1** SPARC V9 Implementation Dependencies *(4 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **29-V8** | 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 the UltraSPARC Architecture. Others remain implementation dependent in the UltraSPARC Architecture. See *ASI Assignments* on page 430 and *Block Load and Store ASIs* on page 451 for details. | **119** |
| **30-V8-Cu3** | f | **ASI address decoding**<br>In SPARC V9, an implementation could choose to decode only a subset of the 8-bit ASI specifier. In UltraSPARC Architecture implementations, all 8 bits of each ASI specifier must be decoded. Refer to Chapter 10, *Address Space Identifiers (ASIs)*, of this specification for details. | 119 |
| **31-V8-Cs10** | f | This implementation dependency is no longer used in the UltraSPARC Architecture, since "catastrophic" errors are now handled using normal error-reporting mechanisms. | — |
| **32-V8-Ms10** | t | **Restartable deferred traps**<br>Whether any restartable deferred traps (and associated deferred-trap queues) are present is implementation dependent. | 473 |
| **33-V8-Cs10** | f | **Trap precision**<br>In an UltraSPARC Architecture implementation, all exceptions that occur as the result of program execution, except for *store_error*, are precise. | 477 |
| **34-V8** | f | **Interrupt clearing**<br>**a**: The method by which an interrupt is removed is now defined in the UltraSPARC Architecture (see *Clearing the Software Interrupt Register* on page 527).<br>**b**: How quickly a virtual processor responds to an interrupt request, like all timing-related issues, is implementation dependent. | **527** |
| **35-V8-Cs20** | t | **Implementation-dependent traps**<br>Trap type (TT) values $060_{16}$–$07F_{16}$ were reserved for *implementation_dependent_exception_n* exceptions in SPARC V9 but are now all defined as standard UltraSPARC Architecture exceptions. | **483** |
| **36-V8** | f | **Trap priorities**<br>The relative priorities of traps defined in the UltraSPARC Architecture 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 new traps are implementation dependent. | **496** |
| **37-V8** | f | **Reset trap**<br>Some of a virtual processor's behavior during a reset trap is implementation dependent. | 476 |
| **38-V8** | f | **Effect of reset trap on implementation-dependent registers**<br>Implementation-dependent registers may or may not be affected by the various reset traps. | **501** |

**TABLE B-1** SPARC V9 Implementation Dependencies *(5 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **39-V8-Cs10** | f | **Entering `error_state` on implementation-dependent errors**<br>The virtual processor enters `error_state` when a trap occurs while the virtual processor is already at its maximum supported trap level — that is, it enters `error_state` when a trap occurs while TL = *MAXTL*. No other conditions cause entry into `error_state` on an UltraSPARC Architecture virtual processor. | 470, 504 |
| **40-V8** | f | **`error_state` virtual processor 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, an UltraSPARC Architecture virtual processor may have other `error_state` entry traps that are implementation dependent. | **470** |
| **41-V8** | | *Reserved.* | |
| **42-V8-Cs10** | t, f, v | **FLUSH instruction**<br>FLUSH is implemented in hardware in all UltraSPARC Architecture 2007 implementations, so never causes a trap as an unimplemented instruction. | |
| **43-V8** | | *Reserved.* | |
| **44-V8-Cs10** | f | **Data access FPU trap**<br>**a**: If a load floating-point instruction generates an exception that causes a non-precise trap, it is implementation dependent whether the contents of the destination floating-point register(s) or floating-point state register are undefined or are guaranteed to remain unchanged.<br>**b**: If a load floating-point alternate instruction generates an exception that causes a non-precise trap, it is implementation dependent whether the contents of the destination floating-point register(s) are undefined or are guaranteed to remain unchanged. | **254**, **276**<br>**259** |
| **45-V8**–**46-V8** | | *Reserved.* | |
| **47-V8-Cs20** | t | **RDasr**<br>RDasr instructions with rd in the range 28–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For an RDasr instruction with rs1 in the range 28–31, the following are implementation dependent:<br>• the interpretation of bits 13:0 and 29:25 in the instruction<br>• whether the instruction is nonprivileged or privileged or hyperprivileged (impl. dep. #9-V8-Cs20)<br>• whether an attempt to execute the instruction causes an *illegal_instruction* exception | **306** |
| **48-V8-Cs20** | t | **WRasr**<br>WRasr instructions with rd in the range 16-18 and 26–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRasr instruction with rd in the range 26–31, 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 nonprivileged or privileged or hyperprivileged (impl. dep. #9-V8-Cs20)<br>• whether an attempt to execute the instruction causes an *illegal_instruction* exception | **382** |
| **49-V8**–**54-V8** | | *Reserved.* | |

**TABLE B-1** SPARC V9 Implementation Dependencies *(6 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **55-V8-Cs10** | 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 UltraSPARC Architecture implementations, tininess is detected before rounding. | 68 |
| **56–100** | | *Reserved.* | |
| **101-V9-CS10** | | **Maximum trap level (*MAXPTL*, *MAXTL*)**<br>The architectural parameter *MAXPTL* is a constant for each implementation; its legal values are from 2 to 6 (supporting from 2 to 6 levels of saved trap state visible to privileged software). In a typical implementation *MAXPTL* = *MAXPGL* (see impl. dep. #401-S10).<br>The architectural parameter *MAXTL* is a constant for each implementation; its legal values are from 3 to 7 (supporting from 3 to 7 levels of saved trap state).<br>Architecturally, *MAXPTL* must be ≥ 2, *MAXTL* must be ≥ 4, and *MAXTL* must be > *MAXPTL*. | **97**, 99 |
| **102-V9** | 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. | **509** |
| **103-V9-Ms10** | f | **Prefetch instructions**<br>The following aspects of the PREFETCH and PREFETCHA instructions are implementation dependent: | |
| | | **a**: the attributes of the block of memory prefetched: its size (minimum = 64 bytes) and its alignment (minimum = 64-byte alignment) | **298** |
| | | **b**: whether each defined prefetch variant is implemented (1) as a NOP, (2) with its full semantics, or (3) with common-case prefetching semantics | **299**, 301 |
| | | **c**: whether and how variants 16, 18, 19 and 24–31 are implemented; if not implemented, a variant must execute as a NOP | **304**C |
| | | The following aspects of the PREFETCH and PREFETCHA instructions used to be (but are no longer) implementation dependent: | |
| | | **d**: while in nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0), an attempt to reference an ASI in the range $0_{16}..7F_{16}$ by a PREFETCHA instruction executes as a NOP; specifically, it does not cause a *privileged_action* exception. | — |
| | | **e**: PREFETCH and PREFETCHA have no observable effect in privileged code | — |
| | | **f**: In UltraSPARC Architecture 2007, PREFETCH and PREFETCHA can cause a *data_access_MMU_miss* exception on a Strong prefetch operation | — |
| | | **g**: while in privileged mode (PSTATE.priv = 1 and HPSTATE.hpriv = 0), an attempt to reference an ASI in the range $30_{16}..7F_{16}$ by a PREFETCHA instruction executes as a NOP (specifically, it does not cause a *privileged_action* exception) | — |

**TABLE B-1** SPARC V9 Implementation Dependencies *(7 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **104-V9** | a | **HVER**.manuf<br>HVER.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 HVER.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 HVER.manuf value. | **106** |
| **105-V9** | f | **TICK register**<br>**a**: If an accurate count cannot always be returned when TICK is read, any inaccuracy should be small, bounded, and documented.<br>**b**: 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. | **74** |
| **106-V9** | 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. | **239** |
| **107-V9** | f | **Unimplemented LDTW(A) trap**<br>**a**: It is implementation dependent whether LDTW is implemented in hardware. If not, an attempt to execute an LDTW instruction will cause an *unimplemented_LDTW* exception.<br>**b**: It is implementation dependent whether LDTWA is implemented in hardware. If not, an attempt to execute an LDTWA instruction will cause an *unimplemented_LDTW* exception. | **267**<br>**270** |
| **108-V9** | f | **Unimplemented STTW(A) trap**<br>**a**: It is implementation dependent whether STTW is implemented in hardware. If not, an attempt to execute an STTW instruction will cause an *unimplemented_STTW* exception.<br>**b**: It is implementation dependent whether STDA is implemented in hardware. If not, an attempt to execute an STTWA instruction will cause an *unimplemented_STTW* exception. | **357**<br>**360** |

**TABLE B-1** SPARC V9 Implementation Dependencies *(8 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **109-V9-Cs10** | f | ***LDDF(A)_mem_address_not_aligned*** | |
| | | **a**: LDDF requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid (i = 1 or instruction bits 12:5 = 0) LDDF instruction may cause an *LDDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the LDDF instruction and return.<br>(In an UltraSPARC Architecture processor, the *LDDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDDF instruction) | 112, 112, 254, 517 |
| | | **b**: LDDFA requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid (i = 1 or instruction bits 12:5 = 0) LDDFA instruction may cause an *LDDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the LDDFA instruction and return.<br>(In an UltraSPARC Architecture processor, the *LDDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDDFA instruction) | 257 |
| **110-V9-Cs10** | f | ***STDF(A)_mem_address_not_aligned*** | |
| | | **a**: STDF requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid (i = 1 or instruction bits 12:5 = 0) STDF instruction may cause an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STDF instruction and return.<br>(In an UltraSPARC Architecture processor, the *STDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STDF instruction) | 112, **342**, 520 |
| | | **b**: STDFA requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid (i = 1 or instruction bits 12:5 = 0) STDFA instruction may cause an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STDFA instruction and return.<br>(In an UltraSPARC Architecture processor, the *STDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STDFA instruction) | 346 |

**TABLE B-1** SPARC V9 Implementation Dependencies *(9 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **111-V9-Cs10** | f | ***LDQF(A)_mem_address_not_aligned*** | |
| | | **a**: LDQF requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an LDQF instruction may cause an *LDQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the LDQF instruction and return.<br>(In an UltraSPARC Architecture processor, the *LDQF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDQF instruction)<br>(this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the LDQF instruction in hardware) | 113, 112, 254, 522 |
| | | **b**: LDQFA requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an LDQFA instruction may cause an *LDQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the LDQF instruction and return.<br>(In an UltraSPARC Architecture processor, the *LDQF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDQFA instruction)<br>(this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the LDQFA instruction in hardware) | 257 |
| **112-V9-Cs10** | f | ***STQF(A)_mem_address_not_aligned*** | |
| | | **a**: STQF requires only word alignment in memory. However, if the effective address is word aligned but not quadword aligned, an attempt to execute an STQF instruction may cause an *STQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STQF instruction and return.<br>(In an UltraSPARC Architecture processor, the *STQF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STQF instruction)<br>(this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the STQF instruction in hardware) | 113, 343, 522 |
| | | **b**: STQFA requires only word alignment in memory. However, if the effective address is word aligned but not quadword aligned, an attempt to execute an STQFA instruction may cause an *STQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STQFA instruction and return.<br>(In an UltraSPARC Architecture processor, the *STQF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STQFA instruction)<br>(this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the STQFA instruction in hardware) | 346 |

**TABLE B-1** SPARC V9 Implementation Dependencies *(10 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **113-V9-Ms10** | f | **Implemented memory models**<br>Whether memory models represented by PSTATE.mm = $10_2$ or $11_2$ are supported in an UltraSPARC Architecture processor is implementation dependent. If the $10_2$ model is supported, then when PSTATE.mm = $10_2$ the implementation must correctly execute software that adheres to the RMO model described in *The SPARC Architecture Manual-Version 9*. If the $11_2$ model is supported, its definition is implementation dependent. | **93**, 418 |
| **114-V9-Cs10** | f | **RED_state trap vector address (*RSTVADDR*)**<br>The RED_state trap vector is located at an address referred to as *RSTVADDR*. In thr UltraSPARC Architecture, *RSTVADDR* is bound to the following address:<br>Physical Address — FFFF FFFF F000 $0000_{16}$ (the highest 256MB of physical address space)<br>In an implementation that implements fewer than 64 bits of physical addressing, unimplemented high-order bits of the above *RSTVADDR* are ignored. | **481**, 631 |
| **115-V9** | f | **RED_state**<br>What occurs after the processor enters RED_state is implementation dependent. | **469** |
| **116-V9** | 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 UltraSPARC Architecture virtual processors, the SIR_enable control flag does not explicitly exist; the SIR instruction always generated an *illegal_instruction* exception in nonprivileged and privileged modes. SIR only causes a *software_initiated_reset* trap when executed in hyperprivleged mode. | 328 |
| **117-V9** | f | **MMU disabled prefetch/nonfaulting load behavior**<br>Whether PREFETCH[A] and nonfaulting loads always succeed when the DMMU is disabled is implementation dependent. | 298, **565** |
| **118-V9** | f | **Identifying I/O locations**<br>The manner in which I/O locations are identified is implementation dependent. | **410** |
| **119-Ms10** | f | **Unimplemented values for PSTATE.mm**<br>The effect of an attempt to write an unsupported memory model designation into PSTATE.mm is implementation dependent; however, it should never result in a value of PSTATE.mm value greater than the one that was written. In the case of an UltraSPARC Architecture implementation that only supports the TSO memory model, PSTATE.mm always reads as zero and attempts to write to it are ignored. | **93**, **419** |
| **120-V9** | 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. | **410** |
| **121-V9** | 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. | **410** |

**TABLE B-1** SPARC V9 Implementation Dependencies *(11 of 11)*

| Nbr | Category | Description | Page |
|---|---|---|---|
| **122-V9** | 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. | 187, **427** |
| **123-V9** | f | **Input/output (I/O) semantics**<br>The semantic effect of accessing I/O registers is implementation dependent. | **29** |
| **124-V9** | 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 UltraSPARC Architecture 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. | 413 |
| **125-V9-Cs10** | f | **Address masking**<br>(1) When PSTATE.am = 1, only the less-significant 32 bits of the PC register are stored in the specified destination register(s) in CALL, JMPL, and RDPC instructions, while the more-significant 32 bits of the destination registers(s) are set to 0.<br>((2) When PSTATE.am = 1, during a trap, only the less-significant 32 bits of the PC and NPC are stored (respectively) to TPC[TL] and TNPC[TL]; the more-significant 32 bits of TPC[TL] and TNPC[TL] are set to 0. | **95**, **95**, 161, 242, 306, 499 |
| **126-V9-Ms10** | | **Register Windows State registers width**<br>Privileged registers CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN contain values in the range 0 to *N_REG_WINDOWS* − 1. An attempt to write a value greater than *N_REG_WINDOWS* − 1 to any of these registers causes an implementation-dependent value between 0 and *N_REG_WINDOWS* − 1 (inclusive) to be written to the register. Furthermore, an attempt to write a value greater than *N_REG_WINDOWS* − 2 violates the register window state definition in *Register Window Management Instructions* on page 127.<br>Although the width of each of these five registers is architecturally 5 bits, the width is implementation dependent and shall be between $\lceil \log_2(N_REG_WINDOWS) \rceil$ 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.<br>For UltraSPARC Architecture 2007 processors, = 8. Therefore, each register window state register is implemented with 3 bits, the maximum value for CWP and CLEANWIN is 7, and the maximum value for CANSAVE, CANRESTORE, and OTHERWIN is 6. When these registers are written by the WRPR instruction, bits 63:3 of the data written are ignored. | **82** |
| **127–199** | | *Reserved.* | — |

TABLE B-2 provides a list of implementation dependencies that, in addition to those in TABLE B-1, apply to UltraSPARC Architecture processors. Bold face indicates the main page on which the implementation dependency is described. See Appendix C in the Extensions Documents for further information.

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(1 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **200–201** | *Reserved.* | — |
| **202-U3** | ***fast_ECC_error* trap**<br>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 $070_{16}$. | **522** |
| **203-U3-Cs10** | **Dispatch Control register (DCR) bits 13:6 and 1**<br>*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | |
| **204-U3-CS10** | **DCR bits 5:3 and 0**<br>*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | |
| **205-U3-Cs10** | **Instruction Trap Register**<br>*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | |
| **206-U3-Cs10** | **SHUTDOWN instruction**<br>On an UltraSPARC Architecture implementation executing in privileged or hyperprivileged mode, SHUTDOWN behaves like a NOP. | 326 |
| **208-U3** | **Ordering of errors captured in instruction execution**<br>The order in which errors are captured in instruction execution is implementation dependent. Ordering may be in program order or in order of detection. | — |
| **209-U3** | **Software intervention after instruction-induced error**<br>Precision of the trap to signal an instruction-induced error of which recovery requires software intervention is implementation dependent. | — |
| **210-U3** | **ERROR output signal**<br>The following aspects of the ERROR output signal are implementation dependent in the UltraSPARC Architecture:<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 | — |
| **211-U3** | **Error logging registers' information**<br>The information that the error logging registers preserves beyond the reset induced by an ERROR signal is implementation dependent. | — |
| **212-U3-Cs10** | ***Trap with fatal error***<br>*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **213-U3** | **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. | — |
| **214-U3** | **Enable/disable control for deferred traps**<br>Whether an implementation provides an enable/disable control feature for deferred traps is implementation dependent. | — |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(2 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **215-U3** | **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. | — |
| **216-U3** | ***data_access_error* trap precision**<br>The precision of a *data_access_error* trap is implementation dependent. | — |
| **217-U3** | ***instruction_access_error* trap precision**<br>The precision of an *instruction_access_error* trap is implementation dependent. | — |
| **218-U3-Cs20** | ***async_data_error***<br>The *async_data_error* exception has been superseded by *sw_recoverable_error*, so this implementation dependency no longer applies. | — |
| **219-U3** | **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 AFARs is implementation dependent. | — |
| **220-U3** | **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. | — |
| **221-U3** | **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. | — |
| **222-U3** | **TLB organization**<br>TLB organization is implementation dependent in UltraSPARC Architecture processors. | **532** |
| **223-U3** | **TLB multiple-hit detection**<br>Whether TLB multiple-hit detection is supported in an UltraSPARC Architecture implementation is implementation dependent. | — |
| **224-U3** | **MMU physical address width**<br>Physical address width support by the MMU is implementation dependent in the UltraSPARC Architecture; minimum PA width is 40 bits. | 446, 447, **541** |
| **225-U3** | **TLB locking of entries**<br>The mechanism by which entries in TLB are locked is implementation dependent in UltraSPARC Architecture implementations. | — |
| **226-U3** | **TTE support for cv bit**<br>Whether the cv bit is supported in TTE is implementation dependent in the UltraSPARC Architecture. 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. | **542** |
| **227-U3** | **TSB number of entries**<br>The maximum number of entries in a TSB is implementation dependent in the UltraSPARC Architecture (to a maximum of 16 million, limited by the size of the TSB Configuration register's tsb_size field). | **545** |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(3 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **228-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **229-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.***TSB Base address generation**<br>Whether the implementation generates the TSB Base address by **exclusive-OR**ing the TSB Base register and a TSB register or by taking the tsb_base field directly from a TSB register is implementation dependent in UltraSPARC Architecture. This implementation dependency existed for UltraSPARC III/IV, only to maintain compatibility with the TLB miss handling software of UltraSPARC I/II. | — |
| **230** | *Reserved.* | — |
| **230-U3-Cs20** | *This implementation dependency no longer applies, in UltraSPARC Architecture 2007* | — |
| **231-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **232-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **233-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **234-U3** | **TLB replacement algorithm**<br>The replacement algorithm for a TLB entry is implementation dependent in UltraSPARC Architecture 2007. | **581** |
| **235-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **236-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.***t** | — |
| **237-U3** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2007.* | —— |
| **238-U3** | **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. | **541** |
| **239-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | |
| **240-U3-Cs10** | *Reserved.* | — |
| **241-U3** | **Address Masking and D-SFAR**<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 (D-SFAR) is implementation dependent. | 95 |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(4 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **242-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **243-U3** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **244-U3-Cs10** | **Data Watchpoint** Reliability<br>Data Watchpoint traps are completely implementation-dependent in UltraSPARC Architecture processors. | — |
| **245-U3-Cs10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **246-U3** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **247-U3** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **248-U3** | **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. | **64** |
| **249-U3-Cs10** | **Data Watchpoint for Partial Store Instruction**<br>For an STPARTIAL 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. | **353** |
| **250-U3-Cs10** | *Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2007.* | — |
| **251** | *Reserved.* | |
| **252-U3-Cs10** | *Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **253-U3-Cs10** | *Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **254-U3-Cs10** | **Means of exiting `error_state`**<br>A virtual processor, upon entering `error_state`, automatically generates a *watchdog_reset* (WDR). | 470, 476, 504, 521, 630 |
| **255-U3-Cs10** | **LDDFA with ASI $E0_{16}$ or $E1_{16}$ and misaligned destination register number**<br>If an LDDFA opcode is used with an ASI of $E0_{16}$ or $E1_{16}$ (Block Store Commit ASI, an illegal combination with LDDFA) and a destination register number rd is specified which is not a multiple of 8 ("misaligned" rd), an UltraSPARC Architecture virtual processor generates an *illegal_instruction* exception. | 258 |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(5 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **256-U3** | **LDDFA with ASI $E0_{16}$ or $E1_{16}$ and misaligned memory address**<br>If an LDDFA opcode is used with an ASI of $E0_{16}$ or $E1_{16}$ (Block Store Commit ASI, an illegal combination with LDDFA) and a memory address is specified with less than 64-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the exception generated is *DAE_invalid_asi*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned*. | **258** |
| **257-U3** | **LDDFA with ASI $C0_{16}$–$C5_{16}$ or $C8_{16}$–$CD_{16}$ and misaligned memory address**<br>If an LDDFA opcode is used with an ASI of $C0_{16}$–$C5_{16}$ or $C8_{16}$–$CD_{16}$ (Partial Store ASIs, which are an illegal combination with LDDFA) and a memory address is specified with less than 8-byte alignment, the virtual processor generates n exception. It is implementation dependent whether the exception generated is *DAE_invalid_asi*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned*. | **259** |
| **258-U3-Cs10** | **ASI_SERIAL_ID**<br>(This register is not defined in the UltraSPARC Architecture, so this implementation dependency does not apply to UltraSPARC Architecture 2007.) | — |
| **259–299** | *Reserved.* | — |
| **300-U4-Cs10** | **Attempted access to ASI registers with LDTWA**<br>If an LDTWA instruction referencing a non-memory ASI is executed, it generates a *DAE_invalid_asi* exception. | 271 |
| **301-U4-Cs10** | **Attempted access to ASI registers with STTWA**<br>If an STTWA instruction referencing a non-memory ASI is executed, it generates a *DAE_invalid_asi* exception. | 360 |
| **302-U4-Cs10** | **Scratchpad registers**<br>An UltraSPARC Architecture processor includes eight privileged Scratchpad registers (64 bits each, read/write accessible). | 453 |
| **303-U4-CS10** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **304-U4-Cs10** | **XIR**<br>XIR affects only the virtual processors identified in the XIR_STEERING register (not a whole system). | **629** |
| **305-U4-Cs10** | *Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **306-U4-Cs10** | **Trap type generated upon attempted access to noncacheable page with LDTXA**<br>When an LDTXA instruction attempts access from an address that is not mapped to cacheable memory space, a *DAE_nc_page* exception is generated. | 273 |
| **307-U4-Cs10** | *Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **308-U3-Cs10** | *Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **309-U4-Cs10** | *Reserved.* | — |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(6 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **310-U4** | **Large page sizes**<br>Which, if any, of the following optional page sizes are supported by the MMU in an UltraSPARC Architecture implementation is implementation dependent: 512 KBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes. | **532** |
| **311–319** | *Reserved.* | |
| | **Strand Interrupt ID register**<br>Whether any portion of the int_id field of the Strand Interrupt ID register is read-only is implementation dependent. | **603** |
| **321-U4** | *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | |
| **322-U4** | **Power used by CMT**<br>Whether disabling a virtual processor reduces the power used by a CMT processor is implementation dependent. | **605** |
| **323-U4** | **Updating Strand Enable Register**<br>Whether an implementation provides a restriction that prevents software from writing a value of all zeroes (or zeroes corresponding to all available virtual processors) to the STRAND_ENABLE register is implementation dependent. This restriction avoids the dangerous case where all virtual processors become disabled and the only way to enable any virtual processor is a hard *power_on_reset* (a warm reset would not suffice). If such a restriction is implemented and software running on any virtual processor attempts to write a value of all zeroes (or zeroes corresponding to all available virtual processors) to the STRAND_ENABLE register, hardware forces the STRAND_ENABLE register to an implementation-dependent value which enables at least one of the available virtual processors. | **607** |
| **324-U4** | **Parking a virtual processor**<br>Whether parking a virtual processor reduces the power used by a CMT processor is implementation dependent. | **609** |
| **325-U4** | **XIR Steering register (XIR Reset)**<br>**a:** Whether XIR_STEERING{*n*} is a read-only bit or a read/write bit is implementation dependent. If XIR_STEERING{*n*} is read-only, then (1) writes to XIR_STEERING{*n*} are ignored and (2) XIR_STEERING{*n*} is set to 1 if virtual processor *n* is available and to 0 if it is not available (that is, XIR_STEERING{*n*} reads the same as STRAND_AVAILABLE{*n*}.<br>**b:** If XIR_STEERING{*n*} is read/write, upon de-assertion of reset the value of STRAND_AVAILABLE{*n*} is copied to XIR_STEERING{*n*} for all UltraSPARC Architecture implementations. | **616**<br><br>617, 634 |
| **326-U4-Cs10** | *Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.* | — |
| **327–399** | *Reserved* | |
| **400-S10** | **Global Level register (GL) implementation**<br>Although GL is defined as a 4-bit register, an implementation may implement any subset of those bits sufficient to encode the values from 0 to *MAXGL* for that implementation. If any bits of GL are not implemented, they read as zero and writes to them are ignored. | **100** |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(7 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **401-S10** | **Maximum Global Level (*MAXPGL*, *MAXGL*)**<br>The architectural parameter *MAXPGL* is a constant for each implementation; its legal values are from 2 to 15 (supporting from 3 to 16 sets of global registers visible to privileged software). In a typical implementation *MAXPGL* = *MAXPTL* (see impl. dep. #101-V9-CS10). The architectural parameter *MAXGL* is a constant for each implementation; its legal values are from 4 to 15 (supporting from 5 to 16 sets of global registers). Architecturally, *MAXPTL* must be ≥ 2 and *MAXGL* must be > *MAXPGL*. | 97, **99**, 100 |
| **402-S10** | **Priority of *internal_processor_error***<br>The trap priority of the *internal_processor_error* exception is implementation dependent. Furthermore, its priority may vary within an implementation, based on the cause of the error being reported. | 490, 495, **516** |
| **403-S10** | **Setting of "dirty" bits in FPRS**<br>A "dirty" bit (du or dl) in the FPRS register must be set to '1' if any of its corresponding F registers is actually modified. If an instruction that normally writes to an F register is executed and causes an *fp_disabled* exception, FPRS.du and FPRS.dl are unchanged. Beyond that, the specific conditions under which a dirty bit is set are implementation dependent. | 76, 76 |
| **404-S10** | **Privileged Scratchpad registers 4 through 7**<br>The degree to which Scratchpad registers 4–7 are accessible to privileged software is implementation dependent. Each may be (1) fully accessible, (2) accessible, with access much slower than to scratchpad register 0–3(emulated by *DAE_invalid_ASI* trap to hyperprivileged software), or (3) inaccessible (cause a *DAE_invalid_asi* exception). | **453** |
| **405-S10** | **Virtual address range**<br>An UltraSPARC Architecture implementation may support a full 64-bit virtual address space or a more limited range of virtual addresses. In an implementation that does not support a full 64-bit virtual address space, the supported range of virtual addresses is restricted to two equal-sized ranges at the extreme upper and lower ends of 64-bit addresses; that is, for $n$-bit virtual addresses, the valid address ranges are 0 to $2^{n-1} - 1$ and $2^{64} - 2^{n-1}$ to $2^{64} - 1$. (see also impl. dep. #451-S20) | **28** |
| **406-S10** | **HTBA high-order bits**<br>It is implementation dependent whether all 50 bits of HTBA{63:14} are implemented or if only bits *n-1*:0 are implemented. If the latter, writes to bits 63:$n$ are ignored and when HTBA is read, bits 63:$n$ read as sign-extended copies of the most significant implemented bit, HTBA{$n - 1$}. | **105** |
| **407-S10** | **Hyperprivileged Scratchpad register aliasing**<br>It is implementation dependent whether any of the hyperprivileged Scratchpad registers are aliased to the corresponding privileged Scratchpad register or is an independent register. | **454** |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(8 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **409-S10-Cs20** | **FLUSH instruction and memory consistency**<br>The implementation of the FLUSH instruction is implementation dependent.<br>If the implementation automatically maintains consistency between instruction and data memory,<br>(1) the FLUSH address is ignored and<br>(2) the FLUSH instruction cannot cause any data access exceptions, because its effective address operand is not translated or used by the MMU.<br>On the other hand, if the implementation does not maintain consistency between instruction and data memory, the FLUSH address is used to access the MMU and the FLUSH instruction can cause data access exceptions. | **188**, 190, 190 |
| **410-S10** | **Block Load behavior**<br>The following aspects of the behavior of block load (LDBLOCKF) instructions are implementation dependent:<br>• What memory ordering model is used by LDBLOCKF (LDBLOCKF is not required to follow TSO memory ordering)<br>• Whether LDBLOCKF follows memory ordering with respect to stores (including block stores), including whether the virtual processor detects read-after-write and write-after-read hazards to overlapping addresses<br>• Whether LDBLOCKF appears to execute out of order, or follow LoadLoad ordering (with respect to older loads, younger loads, and other LDBLOCKFs)<br>• Whether LDBLOCKF follows register-dependency interlocks, as do ordinary load instructions<br>•<br>• Whether the MMU ignores the side-effect bit (TTE.e) for LDBLOCKF accesses (in which case, LDBLOCKFs behave as if TTE.e = 0)<br>• Whether *VA_watchpoint* and *PA_watchpoint* exceptions are recognized on accesses to all 64 bytes of a LDBLOCKF (the recommended behavior), or only on accesses to the first eight bytes | **250**<br><br><br><br><br><br>410<br><br>251, 251, 252 |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(9 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **411-S10** | **Block Store behavior**<br>The following aspects of the behavior of block store (STBLOCKF) instructions are implementation dependent:<br>• The memory ordering model that STBLOCKF follows (other than as constrained by the rules outlined on page 340).<br>• Whether *VA_watchpoint* and *PA_watchpoint* exceptions are recognized on accesses to all 64 bytes of a STBLOCKF (the recommended behavior), or only on accesses to the first eight bytes.<br>• Whether STBLOCKFs to non-cacheable (TTE.cp = 0) pages execute in strict program order or not. If not, a STBLOCKF to a non-cacheable page causes a *DAE_nc_page* exception.<br>• Whether STBLOCKF follows register dependency interlocks (as ordinary stores do).<br>• Whether a non-Commit STBLOCKF forces the data to be written to memory and invalidates copies in all caches present (as the Commit variants of STBLOCKF do). | **340**, 341 |
| | • Whether the MMU ignores the side-effect bit (TTE.e) for STBLOCKF accesses (in which case, STBLOCKFs behave as if TTE.e = 0) | 410 |
| | • Any other restrictions on the behavior of STBLOCKF, as described in implementation-specific documentation. | |
| **412-S10** | **MEMBAR behavior**<br>An UltraSPARC Architecture implementation may define the operation of each MEMBAR variant in any manner that provides the required semantics. | **279** |
| **413-S10** | **Load Twin Extended Word behavior**<br>It is implementation dependent whether *VA_watchpoint* and *PA_watchpoint* exceptions are recognized on accesses to all 16 bytes of a LDTXA instruction (the recommended behavior) or only on accesses to the first 8 bytes. | **273** |
| **414** | *Reserved.* | — |
| **415-S10** | **Size of ContextID fields**<br>The size of context ID fields in MMU context registers is implementation-dependent and may range from 13 to 16 bits. | **569** |
| **416-S10** | **Size of PartitionID fields**<br>The size of partition ID fields in MMU partition registers is implementation-dependent and must be large enough to uniquely encode the identities of all virtual processors that share the TLB. | **569** |
| **417-S10** | **Behavior of DONE and RETRY when TSTATE[TL].pstate.am = 1**<br>If (1) TSTATE[TL].pstate.am = 1 and (2) a DONE or RETRY instruction is executed (which sets PSTATE.am to '1' by restoring the value from TSTATE[TL].pstate.am to PSTATE.am), it is implementation dependent whether the DONE or RETRY instruction masks (zeroes) the more-significant 32 bits of the values it places into PC and NPC. | 96, **166315** |
| **418** | —— *unused* —— | , |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(10 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **419-S10** | **Contents of TPC[TL], TNPC[TL], TSTATE[TL], and HTSTATE[TL] after a Warm Reset (WMR)**<br>It is implementation dependent whether, after a Warm Reset (WMR), the contents of TPC[TL], TNPC[TL], TSTATE[TL], and HTSTATE[TL] are unchanged from their values before the WMR, or are contain the same values saved as during a WDR, XIR, or SIR reset. (The latter implementation is the preferred one.) | **633** |
| **420-S10** | **Implementation Dependent Aspects of a Warm Reset (WMR)**<br>The following aspects of Warm Reset (WMR) are implemenation dependent:<br>(a) by what means WMR can be applied (for example, write to reset register or assertion/deassertion of an input pin)<br>(b) the extent to which a processor is reset by WMR (for example, single physical core, entire processor (chip), and how the on-chip memory system is affected),<br>(c) by what means hyperprivileged software can distinguish between WMR and POR resets | **629** |
| **421-S10** | **Interrupt Queue Head and Tail Register Contents**<br>It is implementation dependent whether interrupt queue head and tail registers (a) are datatype-agnostic "scratch registers" used for communication between privileged and hyperprivileged software, in which case their contents are defined purely by software convention, or (b) are maintained to some degree by virtual processor hardware, imposing a fixed meaning on their contents. | **528** |
| **422-S10** | **Interrupt Queue Tail Register Writability**<br>It is implementation dependent whether tail registers are writable in privileged mode. If a tail register is read-only in privileged mode, an attempt to write to it causes a *DAE_invalid_asi* exception. If a tail register is writable in privileged mode, an attempt to write to it results in undefined behavior. | 528, **528** |
| **423-S10** | **Performance Impact of Disabling a Virtual Processor**<br>Whether disabling a virtual processor increases the performance of other virtual processors in the CMT is implementation dependent. | **605** |
| **424-S10** | **Ability to Dynamically Enable/Disable a Virtual Processor**<br>Whether a CMT implementation provides the ability to dynamically enable and disable virtual processors is implementation dependent. It is tightly coupled to the underlying microarchitecture of a specific CMT implementation. This feature is implementation dependent because any implementation-independent interface would be too inefficient on some implementations. | **608** |
| **425-S10** | **TICK Register Counting While a Virtual Processor is Parked**<br>It is implementation dependent whether the TICK register continues to count while a virtual processor is parked. | **608** |
| **426-S10** | **Performance Impact of Parking a Virtual Processor**<br>The degree to which parking a virtual processor impacts the performance of other virtual processors is implementation dependent. | **609** |
| **427-S10** | **Latency to Park or Unpark a aVirtual Processor**<br>There may be an arbitrarily long, but bounded, delay ("skid") from the time when a virtual processor is directed to park or unpark (via an update to the STRAND_RUNNING register) until the corresponding virtual processor(s) actually park or unpark. | **609**, 612 |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(11 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **428-S10** | **Method by Which Self-Parking is Assured**<br>When a virtual processor writes to the STRAND_RUNNING register to park itself, the method by which completion of parking is assured (instructions stop being issued) is implementation dependent. | **610** |
| **429-S10** | **Which Virtual Processor is Automatically Unparked**<br>If an update to the STRAND_RUNNING register would cause all enabled virtual processors to become parked, it is implementation dependent which virtual processor is automatically unparked by hardware. The preferred implementation is that when an update to the STRAND_RUNNING register (STXA instruction) would cause all virtual processors to become parked, hardware silently ignores (discards) that STXA instruction. | **611** |
| **430-S10** | **Parking All But One Virtual Processor in a Multiprocessor Configuration**<br>In a multi'fio<br>configuration, whether all but one virtual processor can be parked is implementation dependent. | **612** |
| **431-S10** | **Criteria for Completion of Park/Unpark**<br>The criteria used for determining whether a virtual processor is fully parked (corresponding bit set to '1' in the STRAND_RUNNING_STATUS register) are implementation dependent. | **613** |
| **432-S10** | **Standby/Wait state**<br>Whether an implementation implements a `Standby` (or `Wait`) state for virtual processors, how that state is controlled, and how that state is observed are implementation-dependent. | **613** |
| **433-S10** | **Partial-Processor Reset Subsetting Mechanism**<br>A mechanism must exist to specify which subset of virtual processors in a processor should be reset when a partial-processor reset (for example, XIR) occurs. The specific mechanism is implementation-dependent. | **615** |
| **434-S10** | **Error Steering Register(s)**<br>Because of the range of implementation, the number of, organization of, and ASI assignments for error steering registers in a CMT processor are implementation dependent. | **618** |
| **435-S10** | **Error Steering Register Alternatives**<br>Although the ERROR_STEERING register is the recommended mechanism for steering non-virtual-processor-specific errors to a virtual processor for handling, the actual mechanism used in a given implementation is implementation dependent. | **620** |
| **436-S10** | **Error Steering Register**<br>The width of the target_id field of the ERROR_STEERING register is implementation dependent. | **621** |
| **437-S10** | **Error Steering Register targetid Field Plurality**<br>An implementation may provide multiple target_id fields in an ERROR_STEERING register for different types of non-virtual-processor-specific errors. | **621** |
| **438-S10** | **Non-Virtual Processor-Specific Errors in Shared Resources**<br>It is implementation dependent whether the error-reporting structures for errors in shared resources appear within a virtual processor in per-virtual-processor registers or are contained within shared registers associated with the shared structures in which the errors may occur. | **621** |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(12 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **439-S10** | **Exception Generated for Each Non-Virtual Processor-Specific Error**<br>The type of exception generated in a virtual processor to handle each type of non-virtual-processor-specific error is implementation dependent. A virtual processor can choose to use the same exceptions used for corresponding virtual-processor-specific asynchronous errors or it can choose to generate different exceptions. | **621** |
| **440-S10** | **Which Virtual Processor Unparked During Power-on-Reset (POR)**<br>Which virtual processor is unparked during POR and whether it is unparked by processor hardware or by a service processor is implementation dependent. Conventionally, the virtual processor with the lowest-numbered strand_id is unparked | **623** |
| **441-S10** | **Use of Physical Address Bit to Distinguish Memory from I/O Addresses**<br>Whether an implementation uses the most significant physical address bit to differentiate between memory and I/O addresses is implementation dependent. If that method is used, then the most significant bit of the physical address (PA) = 1 designates I/O space and the most significant bit of PA = 0 designates memory space . | **541** |
| **442-S10** | **STICK register**<br>**a**: If an accurate count cannot always be returned when STICK is read, any inaccuracy should be small, bounded, and documented.<br>**b**: An implementation may implement fewer than 63 bits in STICK.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. | **81** |
| **443-S10** | **PSTATE.am for Physical Addresses in Hyperprivileged Mode**<br>In hyperprivileged mode, when PSTATE.am = 1 and physical addressing is being used, it is implementation-dependent whether the more-significant 32 bits of addresses are masked (treated as zero). | **96** |
| **444–449** | *Reserved for UltraSPARC Architecture 2005* | |
| **450-S20** | **Availability of *control_transfer_instruction* exception feature**<br>Availability of the *control_transfer_instruction* exception feature is implementation dependent. If not implemented, trap type $074_{16}$ is unused, PSTATE.tct always reads as zero, and writes to PSTATE.tct are ignored. | **92**,509 |
| **451-S20** | **Width of Virtual Addresses supported**<br>The width of the virtual address supported is implementation dependent. If fewer than 64 bits are supported, the unsupported bits must have the same value as the most significant supported bit. For example, if the model supports 48 virtual address bits, then bits 63:48 must have the same value as bit 47. (see also impl. dep. #405-S10) | **531**, 514, 517 |
| **452-S20** | **Width of Real and Physical Addresses supported**<br>The number of real address (RA) and physical address (PA) bits supported is implementation dependent. A minimum of 40 bits and maximum of 56 bits can be provided for both real addresses (RA) and physical addresses (PA). See implementation-specific documentation for details. | **532**, 515, 518 |
| **453-S20** | **Unified vs. Split Instruction and Data MMUs**<br>It is implementation dependent whether there is a unified MMU (UMMU) or a separate IMMU (for instruction accesses) and DMMU (for data accesses). The UltraSPARC Architecture supports both configurations. | **532** |

**TABLE B-2** UltraSPARC Architecture Implementation Dependencies *(13 of 13)*

| Nbr | Description | Page |
|---|---|---|
| **453-S20** | **Unified vs. Split Instruction and Data MMUs**<br>It is implementation dependent whether there is a unified MMU (UMMU) or a separate IMMU (for instruction accesses) and DMMU (for data accesses). The UltraSPARC Architecture supports both configurations. | **532** |
| **454-499** | *Reserved for UltraSPARC Architecture 2007* | |
| **500 and up** | *Reserved for future use* | |

APPENDIX **C**

# Assembly Language Syntax

This appendix supports Chapter 7, *Instructions*. Each instruction description in Chapter 7 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 UltraSPARC Architecture assemblers for the convenience of assembly language programmers.

The appendix contains these sections:


- **Notation Used** on page 695.
- **Syntax Design** on page 702.
- **Synthetic Instructions** on page 702.


## C.1 Notation Used

The notations defined here are also used in the assembly language syntax descriptions in Chapter 7, *Instructions*.

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.

## C.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`, ... `%f31`

`%f32`, `%f34`, ... `%f60`, `%f62` (even-numbered only, from `%f32` to `%f62`)

`%d0`, `%d2`, `%d4`, ... `%d60`, `%d62` (`%d`*n*, where *n* **mod** 2 = 0, only)

`%q0`, `%q4`, `%q8`, ... `%q56`, `%q60` (`%q`*n*, where *n* **mod** 4 = 0, only)

See *Floating-Point Registers* on page 54 for a detailed description of how the single-precision, double-precision, and quad-precision floating-point registers overlap.

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 `fcc`*n* specifies a set of floating-point condition codes. It can have any of the following values:

```
%fcc0
%fcc1
%fcc2
%fcc3
```

## C.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 (ASI) register |
| `%canrestore` | Restorable Windows register |
| `%cansave` | Savable Windows register |
| `%ccr` | Condition Codes register |
| `%cleanwin` | Clean Windows register |
| `%cwp` | Current Window Pointer (CWP) register |
| `%fprs` | Floating-Point Registers State (FPRS) register |
| `%fsr` | Floating-Point State register |
| `%gsr` | General Status Register (GSR) |
| `%hintp` | Hyperprivileged Interrupt Pending (HINTP) register |
| `%hpstate` | Hyperprivileged State (HSTATE) register |
| `%hstick_cmpr` | Hyperprivileged System Tick Compare (HSTICK_CMPR) register |
| `%htba` | Hyperprivileged Trap Base Address (HTBA) register |
| `%htstate` | Hyperprivileged Trap State (HTSTATE) register |
| `%hver` | Hyperprivileged Version (HVER) register |
| `%otherwin` | Other Windows (OTHERWIN) register |
| `%pc` | Program Counter (PC) register |
| `%pil` | Processor Interrupt Level register |
| `%pstate` | Processor State register |
| `%softint` | Soft Interrupt register |
| `%softint_clr` | Soft Interrupt register (clear selected bits) |

| | |
|---|---|
| %softint_set | Soft Interrupt register (set selected bits) |
| %stick † | System Timer (STICK) register |
| %stick_cmpr † | System Timer Compare (STICK_CMPR) register |
| %tba | Trap Base Address (TBA) register |
| %tick | Cycle count (TICK) register |
| %tick_cmpr | Timer Compare (TICK_CMPR) register |
| %tl | Trap Level (TL) register |
| %tnpc | Trap Next Program Counter (TNPC) register |
| %tpc | Trap Program Counter (TPC) register |
| %tstate | Trap State (TSTATE) register |
| %tt | Trap Type (TT) register |
| %wstate | Window State register |
| %y | Y register |

† The original assembly language names for %stick and %stick_cmpr were, respectively, %sys_tick and %sys_tick_cmpr, which are now deprecated. Over time, assemblers will support the new %stick and %stick_cmpr names for these registers (which are consistent with %tick %tick_cmpr, and %hstick_cmpr). In the meantime, some existing assemblers may only recognize the original names.

The following special symbol names are prefix unary operators that perform the functions described, on an argument that is a constant, symbol, or expression that evaluates to a constant offset from a symbol:

| | |
|---|---|
| %hh | Extracts bits 63:42 (high 22 bits of upper word) of its operand |
| %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 |

For example, the value of "%lo(*symbol*)" is the least-significant 10 bits of *symbol*.

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 constant values to which they are bound are listed in TABLE C-1.

**TABLE C-1** Value Names and Values *(1 of 2)*

| Value Name in Assembly Language | Value | Comments |
|---|---|---|
| ***for PREFETCH instruction "fcn" field*** | | |
| #n_reads | 0 | |
| #one_read | 1 | |
| #n_writes | 2 | |
| #one_write | 3 | |
| #page | 4 | |

**TABLE C-1** Value Names and Values *(2 of 2)*

| Value Name in Assembly Language | Value | Comments |
|---|---|---|
| `#unified` | 17 ($11_{16}$) | |
| `#n_reads_strong` | 20 ($14_{16}$) | |
| `#one_read_strong` | 21 ($15_{16}$) | |
| `#n_writes_strong` | 22 ($16_{16}$) | |
| `#one_write_strong` | 23 ($17_{16}$) | |
| ***for MEMBAR instruction "mmask" field*** | | |
| `#LoadLoad` | $01_{16}$ | |
| `#StoreLoad` | $02_{16}$ | |
| `#LoadStore` | $04_{16}$ | |
| ***for MEMBAR instruction "cmask" field*** | | |
| `#StoreStore` | $08_{16}$ | |
| `#Lookaside` | $10_{16}$ | |
| `#MemIssue` | $20_{16}$ | |
| `#Sync` | $40_{16}$ | |

## C.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) |
| *siam_mode* | A 3-bit mode value for the SIAM instruction |
| *simm7* | A signed immediate constant that can be represented in 7 bits |
| *simm8* | A signed immediate constant that can be represented in 8 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 |

## C.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.

## C.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 (see TABLE 7-7 and TABLE 7-8 on page 278 for a complete list of mnemonics). |

*prefetch_fcn (prefetch function)* Can be any of the following:

0–31

Predefined constants (the values of which fall in the 0-31 range) useful as *prefetch_fcn* values can be found in TABLE C-1 on page 698.

*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`)
*$reg_{rs1}$* + *$reg_{rs2}$*
*$reg_{rs1}$* + *simm8*
*$reg_{rs1}$* – *simm8*
*simm8* (equivalent to `%g0` + *simm8*)
*simm8* + *$reg_{rs1}$* (equivalent to *$reg_{rs1}$* + *simm8*)

The resulting operand value (software trap number) must be in the range 0–255, inclusive.

## C.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.

# C.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 (for example, in a load, store, or load-store 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.

# C.3 Synthetic Instructions

TABLE C-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 C-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* | Originally envisioned as an encoding for an "instruction prefetch" operation, but functions as a NOP on all UltraSPARC Architecture implementations. ( See PREFETCH function 17 on page 297 for an alternative method of prefetching instructions.) |
| `tst` | *reg*$_{rs1}$ | `orcc` | `%g0,` *reg*$_{rs1}$`, %g0` | Test. |
| `ret` | | `jmpl` | `%i7+8, %g0` | Return from subroutine. |

**TABLE C-2** Mapping Synthetic to SPARC V9 Instructions *(2 of 3)*

| Synthetic Instruction | | SPARC V9 Instruction(s) | | Comment |
|---|---|---|---|---|
| `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 — | |
| | | `or` | `%g0,` *value*, $reg_{rd}$ | (When *0* ≤ *value* ≤ `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 ≤ *value* ≤ `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` | `%hh(`*value*`)`, *reg* | Create 64-bit constant. |
| | | `or` | *reg*, `%hm(`*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. |

**TABLE C-2** Mapping Synthetic to SPARC V9 Instructions *(3 of 3)*

| Synthetic Instruction | | SPARC V9 Instruction(s) | | Comment |
|---|---|---|---|---|
| `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. |
| `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*. |
| `decc` | *reg$_{rd}$* | `subcc` | *reg$_{rd}$*, `1`, *reg$_{rd}$* | Decrement 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*** | |

# Index


## A

a (annul) instruction field
  branch instructions, 153, 154, 156, 159, 174, 177
accesses
  cacheable, 409
  I/O, 409
  restricted ASI, 413
  with side effects, 409, 420
accrued exception (aexc) field of FSR register, **65**, 478, 673
ADD instruction, **145**
ADDC instruction, **145**
ADDcc instruction, **145**, 330
ADDCcc instruction, **145**
address
  aliasing, 536
  operand syntax, 700
  separation of virtual and real, 536
  space identifier (ASI), 429
address mask (am) field of PSTATE register
  description, **94**
address space, **7**, 22
address space identifier (ASI), **7**, 408
  accessing MMU registers, 566
  appended to memory address, 27, 110
  architecturally specified, **413**
  changed in, 455
  changed in UA
    ASI_REAL, 455
    ASI_REAL_IO, 455
    ASI_REAL_IO_LITTLE, 455
    ASI_REAL_LITTLE, 455
    ASI_TWINX_N, 455
    ASI_TWINX_NL, 455
    ASI_TWINX_NUCLEUS_LITTLE, 455
    ASI_TWINX_R, 455
    ASI_TWINX_REAL, 455
    ASI_TWINX_REAL_L, 455
    ASI_TWINX_REAL_LITTLE, 455
  definition, 7
  encoding address space information, 111
  explicit, **117**
  explicitly specified in instruction, 118
  implicit, *See* implicit ASIs
  load from TLB Data Access register, 582
  load from TLB Tag Read register, 577
  nontranslating, **13**, 271, 360
  nontranslating ASI, **430**
  operations, 566
  with prefetch instructions, 298
  real ASI, **430**
  real-translating ASIs, **430**
  restricted, 413, 429, 553
    hyperprivileged, 414
    privileged, 414
  restriction indicator, 73
  SPARC V9 address, 411
  translating ASI, **430**
  unrestricted, **414**, 429
  virtual-translating ASI, **430**
address space identifier (ASI) register
  for load/store alternate instructions, 73
  address for explicit ASI, 117
  and LDDA instruction, 256, 269
  and LDSTUBA instruction, 265
  load integer from alternate space instructions, 246

with prefetch instructions, 298
for register-immediate addressing, 414
restoring saved state, 165, 315
saving state, 463
and STDA instruction, 359
store floating-point into alternate space instructions, 345
store integer to alternate space instructions, 335
and SWAPA instruction, 366
after trap, 33
and TSTATE register, 89
and write state register instructions, 382
addressing modes, 22
ADDX instruction (SPARC V8), 145
ADDXcc instruction (SPARC V8), 145
AFAR, *See* Asynchronous Fault Address register (AFAR)
AFSR, *See* Asynchronous Fault Status register (AFSR)
alias
floating-point registers, 54
aliased, **7**
ALIGNADDRESS instruction, 146
ALIGNADDRESS_LITTLE instruction, 146
alignment
data (load/store), 28, **112**, 411
doubleword, 28, **112**, 411
extended-word, **112**
halfword, 28, **112**, 411
instructions, 28, **112**, 411
integer registers, 258, 268, 270
memory, 411, 517
quadword, 28, **112**, 411
word, 28, **112**, 411
ALLCLEAN instruction, **147**
alternate space instructions, 29, 73
ancillary state registers (ASRs)
access, 69
assembly language syntax, 696
I/O register access, 29
possible registers included, 306, 383
privileged, 31, 672
reading/writing implementation-dependent processor registers, 31, 672
writing to, 382
AND instruction, **148**
ANDcc instruction, **148**
ANDN instruction, **148**
ANDNcc instruction, **148**
annul bit
in branch instructions, 159
in conditional branches, 175
annulled branches, 159
application program, **7**, 69
architectural direction note, **5**
architecture, meaning for SPARC V9, 21
arithmetic overflow, 72
ARRAY16 instruction, 149
ARRAY32 instruction, 149
ARRAY8 instruction, 149
ASI, **7**
for IMMU, DMMU, UMMU, 579
invalid, and *DAE_invalid_asi*, 510
write to Tag Access register, 579
ASI register, 69
ASI, *See* address space identifier (ASI)
ASI_*REAL* ASIs, 412
ASI_AIPN, 435, 450
ASI_AIPN_L, 435, 450
ASI_AIPP, 435, 450
ASI_AIPP_L, 435, 450
ASI_AIPS, 435, 450
ASI_AIPS_L, 435, 450
ASI_AIUP, 432, 445
ASI_AIUPL, 432, 446
ASI_AIUS, 432, 445
ASI_AIUS_L, 272
ASI_AIUSL, 432, 446
ASI_AS_IF_PRIV_NUCLEUS, 435, 450, 538
ASI_AS_IF_PRIV_NUCLEUS_LITTLE, 435, 450
ASI_AS_IF_PRIV_PRIMARY, 435, 450, 538
ASI_AS_IF_PRIV_PRIMARY_LITTLE, 435, 450
ASI_AS_IF_PRIV_SECONDARY, 435, 450, 538
ASI_AS_IF_PRIV_SECONDARY_LITTLE, 435, 450
ASI_AS_IF_USER*, 94, 411, 412, 539
ASI_AS_IF_USER* ASIs, 411
ASI_AS_IF_USER_NONFAULT_LITTLE, 415
ASI_AS_IF_USER_PRIMARY, 432, 445, 510, 521, 538
ASI_AS_IF_USER_PRIMARY_LITTLE, 414, 432, 446, 510
ASI_AS_IF_USER_SECONDARY, 414, 432, 445, 510, 521, 538
ASI_AS_IF_USER_SECONDARY_LITTLE, 414, 432, 446, 510
ASI_AS_IF_USER_SECONDARY_NOFAULT_LITTLE, 415

ASI_BLK_AIUP, 432, 445
ASI_BLK_AIUPL, 432, 446
ASI_BLK_AIUS, 432, 445
ASI_BLK_AIUSL, 432, 446
ASI_BLK_COMMIT_P, 442
ASI_BLK_COMMIT_S, 442
ASI_BLK_P, 442
ASI_BLK_PL, 442
ASI_BLK_S, 442
ASI_BLK_SL, 443
ASI_BLOCK_AS_IF_USER_PRIMARY, 432, 445
ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE, 432, 446
ASI_BLOCK_AS_IF_USER_SECONDARY, 432, 445
ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE, 432, 446
ASI_BLOCK_COMMIT_PRIMARY, 442
ASI_BLOCK_COMMIT_SECONDARY, 442
ASI_BLOCK_PRIMARY, 442
ASI_BLOCK_PRIMARY_LITTLE, 442
ASI_BLOCK_SECONDARY, 442
ASI_BLOCK_SECONDARY_LITTLE, 443
ASI_CMT_PER_CORE, 440, 561
ASI_CMT_PER_STRAND, 440, 561, 601, 602
ASI_CMT_SHARED, 435, 605, 606, 609, 612, 616
ASI_DEVICE_ID+SERIAL_ID, 443
ASI_DMMU, 439
ASI_DMMU_DEMAP, 439
ASI_DTLB_DATA_ACCESS_REG, 439
ASI_DTLB_DATA_IN_REG, 439
ASI_DTLB_TAG_READ_REG, 439
ASI_FL16_P, 441
ASI_FL16_PL, 441
ASI_FL16_PRIMARY, 441
ASI_FL16_PRIMARY_LITTLE, 441
ASI_FL16_S, 441
ASI_FL16_SECONDARY, 441
ASI_FL16_SECONDARY_LITTLE, 442
ASI_FL16_SL, 442
ASI_FL8_P, 441
ASI_FL8_PL, 441
ASI_FL8_PRIMARY, 441
ASI_FL8_PRIMARY_LITTLE, 441
ASI_FL8_S, 441
ASI_FL8_SECONDARY, 441
ASI_FL8_SECONDARY_LITTLE, 441
ASI_FL8_SL, 441
ASI_IMMU, 437
ASI_IMMU_DEMAP, 439
ASI_ITLB_DATA_ACCESS_REG, 439
ASI_ITLB_TAG_READ_REG, 439
ASI_MMU, 438, **572**
ASI_MMU_CONTEXTID, 433, 559
ASI_MMU_NONZERO_CONTEXT_TSB_CONFIG_0/1, 545
ASI_MMU_REAL, 437, **570**
ASI_MMU_ZERO_CONTEXT_TSB_CONFIG_*n*, 545
ASI_MMU_ZERO_CONTEXTID_TSB_CONFIG_0/1, 545
ASI_N, 431
ASI_NL, 432
ASI_NUCLEUS, 118, 431, 538
ASI_NUCLEUS_LITTLE, 118, 432
ASI_NUCLEUS_QUAD_LDD (deprecated), 455
ASI_NUCLEUS_QUAD_LDD_L (deprecated), 455
ASI_NUCLEUS_QUAD_LDD_LITTLE (deprecated), 455
ASI_P, 440
ASI_PHY_BYPASS_EC_WITH_EBIT_L, 455
ASI_PHYS_BYPASS_EC_WITH_EBIT, 455
ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE, 455
ASI_PHYS_USE_EC, 455
ASI_PHYS_USE_EC_L, 455
ASI_PHYS_USE_EC_LITTLE, 455
ASI_PL, 440
ASI_PNF, 440
ASI_PNFL, 440
ASI_PRIMARY, 118, 413, 414, 440
ASI_PRIMARY_LITTLE, 118, 413, 440, 556
ASI_PRIMARY_NO_FAULT, 410, 427, 440
ASI_PRIMARY_NO_FAULT_LITTLE, 410, 427, 440, 510
ASI_PRIMARY_NOFAULT_LITTLE, 415
ASI_PST16_P, 351, 440
ASI_PST16_PL, 351, 441
ASI_PST16_PRIMARY, 440
ASI_PST16_PRIMARY_LITTLE, 441
ASI_PST16_S, 351, 440
ASI_PST16_SECONDARY, 440
ASI_PST16_SECONDARY_LITTLE, 441
ASI_PST16_SL, 351
ASI_PST32_P, 351, 440
ASI_PST32_PL, 351, 441
ASI_PST32_PRIMARY, 440
ASI_PST32_PRIMARY_LITTLE, 441
ASI_PST32_S, 351, 440
ASI_PST32_SECONDARY, 440

ASI_PST32_SECONDARY_LITTLE, 441
ASI_PST32_SL, 351, 441
ASI_PST8_P, 440
ASI_PST8_PL, 441
ASI_PST8_PRIMARY, 440
ASI_PST8_PRIMARY_LITTLE, 441
ASI_PST8_S, 440
ASI_PST8_SECONDARY, 440
ASI_PST8_SECONDARY_LITTLE, 441
ASI_PST8_SL, 351, 441
ASI_QUAD_LDD_L (deprecated), 455
ASI_QUAD_LDD_LITTLE (deprecated), 455
ASI_QUAD_LDD_PHYS (deprecated), 455
ASI_QUAD_LDD_REAL (deprecated), 434
ASI_QUAD_LDD_REAL_LITTLE (deprecated), 434
ASI_REAL, 432, 446, 455
ASI_REAL_IO, 432, 447, 455
ASI_REAL_IO_L, 432
ASI_REAL_IO_LITTLE, 432, 447, 455
ASI_REAL_L, 432
ASI_REAL_LITTLE, 432, 447, 455
ASI_S, 440
ASI_SECONDARY, 440, 538
ASI_SECONDARY_LITTLE, 440
ASI_SECONDARY_NO_FAULT, 427, 440, 510
ASI_SECONDARY_NO_FAULT_LITTLE, 427, 440, 510
ASI_SECONDARY_NOFAULT, 415
ASI_SL, 440
ASI_SNF, 440
ASI_SNFL, 440
ASI_TABLEWALK_PENDING_CONTROL, 588
ASI_TABLEWALK_PENDING_STATUS, 588
ASI_TWINX_AIUP, 272, 433, 448, 559
ASI_TWINX_AIUP_L, 272, 448
ASI_TWINX_AIUPL, 434
ASI_TWINX_AIUS, 272, 448, 559
ASI_TWINX_AIUS_L, 434, 448, 560
ASI_TWINX_AS_IF_USER_PRIMARY, 433, 448, 559
ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE, 434, 448
ASI_TWINX_AS_IF_USER_SECONDARY, 433, 448, 559
ASI_TWINX_AS_IF_USER_SECONDARY_LITTLE, 434, 448, 560
ASI_TWINX_N, 272, 434, 455
ASI_TWINX_NL, 272, 435, 448, 455
ASI_TWINX_NUCLEUS, 434, 448, 455, 559
ASI_TWINX_NUCLEUS[_L], 411
ASI_TWINX_NUCLEUS_LITTLE, 435, 448, 455
ASI_TWINX_P, 272, 442
ASI_TWINX_PL, 272, 442
ASI_TWINX_PRIMARY, 442, 451, 562
ASI_TWINX_PRIMARY_LITTLE, 442, 451, 562
ASI_TWINX_R, 434, 449, 455
ASI_TWINX_REAL, 272, 434, 449, 455
ASI_TWINX_REAL[_L], 411
ASI_TWINX_REAL_L, 434, 449, 455
ASI_TWINX_REAL_LITTLE, 434, 449, 455
ASI_TWINX_S, 272, 442
ASI_TWINX_SECONDARY, 442, 451, 562
ASI_TWINX_SECONDARY_LITTLE, 442, 451, 562
ASI_TWINX_SL, 272, 442
ASI_UMMU, 439
ASR, **7**
*asr_reg*, **696**
*async_data_error* exception, **521**
*async_data_error* exception (superseded), **520**, **683**
atomic
  memory operations, 273, **422**, 424
  store doubleword instruction, 357, 359
  store instructions, 333, 335
atomic load-store instructions
  compare and swap, **162**
  load-store unsigned byte, **264**, 366
  load-store unsigned byte to alternate space, **265**
  simultaneously addressing doublewords, 365
  swap R register with alternate space memory, **366**
  swap R register with memory, 162, **365**
atomicity, 410, 680
autodemap function, 577, 578
available (core), **7**

## B

BA instruction, 153, 154, 663
BCC instruction, 153, 663
bclrg synthetic instruction, 704
BCS instruction, 153, 663
BE instruction, 153, 663
Berkeley RISCs, 24
BG instruction, 153, 663
BGE instruction, 153, 663
BGU instruction, 153, 663
Bicc instructions, **153**, 657
big-endian, **7**

big-endian byte order, 28, 92, 113
in hyperprivileged mode, 478
binary compatibility, 24
BL instruction, 663
BLD, **7**
BLD, *See* LDBLOCKF instruction
BLE instruction, 153, 663
BLEU instruction, 153, 663
block load instructions, 55, **249**, 451
block store instructions, 55, **338**, 451
with commit, 258, 339, 451
blocked byte formatting, 150
block-type ESR, 642
BMASK instruction, 155
BN instruction, 153, 663
BNE instruction, 153, 663
BNEG instruction, 153, 663
BP instructions, 663
BPA instruction, 156, 663
BPCC instruction, 156, 663
BPcc instructions, 72, **156**, 664
BPCS instruction, 156, 663
BPE instruction, 156, 663
BPG instruction, 156, 663
BPGE instruction, 156, 663
BPGU instruction, 156, 663
BPL instruction, 156, 663
BPLE instruction, 156, 663
BPLEU instruction, 156, 663
BPN instruction, 156, 663
BPNE instruction, 156, 663
BPNEG instruction, 156, 663
BPOS instruction, 153, 663
BPPOS instruction, 156, 663
BPr instructions, **159**, 663
BPVC instruction, 156, 663
BPVS instruction, 156, 663
branch
annulled, 159
delayed, 109
elimination, 126
fcc-conditional, 175, 177
icc-conditional, 154
instructions
on floating-point condition codes, **174**
on floating-point condition codes with prediction, **176**
on integer condition codes with prediction (BPcc), 156
on integer condition codes, *See* Bicc instructions
when contents of integer register match condition, **159**
prediction bit, 159
unconditional, 153, 157, 174, 177
with prediction, 22
BRGEZ instruction, 159
BRGZ instruction, 159
BRLEZ instruction, 159
BRLZ instruction, 159
BRNZ instruction, 159
BRZ instruction, 159
`bset` synthetic instruction, 704
BSHUFFLE instruction, 155
BST, **7**
BST, *See* STBLOCKF instruction
`btog` synthetic instruction, 704
`btst` synthetic instruction, 704
BVC instruction, 153, 663
BVS instruction, 153, 663
byte, **7**
addressing, 117
data format, **35**
order, 28
order, big-endian, 28
order, little-endian, 28
byte order
big-endian, 92
in hyperprivileged mode, 478
implicit, 93
in trap handlers, 478
little-endian, 92

## C

cache
coherency protocol, 409
data, 417
instruction, 417
miss, 304
nonconsistent instruction cache, 417
cacheable accesses, 408
caching, TSB, 544
CALL instruction
description, 161
displacement, 30, 31
does not change CWP, 52
and JMPL instruction, 242

writing address into R[15], 54
call synthetic instruction, 702
CANRESTORE (restorable windows) register, **84**
and *clean_window* exception, 127
and CLEANWIN register, 84, 86, 523
counting windows, 86
decremented by RESTORE instruction, 311
decremented by SAVED instruction, 321
detecting window underflow, 52
if registered window was spilled, 312
incremented by SAVE instruction, 319
modified by NORMALW instruction, 291
modified by OTHERW instruction, 293
range of values, 82, 681
RESTORE instruction, 127
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 632
window underflow, 522
CANSAVE (savable windows) register, **83**
decremented by SAVE instruction, 319
detecting window overflow, 52
FLUSHW instruction, 191
if equals zero, 127
incremented by RESTORE, 311
incremented by SAVED instruction, 321
range of values, 82, 681
SAVE instruction, 524
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 632
window overflow, 522
CAS synthetic instruction, 424
CASA instruction, **162**
32-bit compare-and-swap, 423
alternate space addressing, 29
and *DAE_nc_page* exception, 510
atomic operation, 264
hardware primitives for mutual exclusion of CASXA, 422
in multiprocessor system, 265, 365, 366
R register use, 111
word access (memory), 112
cas*n* synthetic instructions, 704
CASX synthetic instruction, 423, 424
CASXA instruction, **162**
64-bit compare-and-swap, 423
alternate space addressing, 29
and *DAE_nc_page* exception, 510
atomic operation, 265
doubleword access (memory), 112
hardware primitives for mutual exclusion of CASA, 422
in multiprocessor system, 264, 265, 365, 366
R register use, 111
catastrophic error exception, **464**
cc0 instruction field
branch instructions, 156, 177
floating point compare instructions, 182
move instructions, 283, 664
cc1 instruction field
branch instructions, 156, 177
floating point compare instructions, 182
move instructions, 283, 664
cc2 instruction field
move instructions, 283, 664
CCR (condition codes register), **8**
CCR (condition codes) register, **71**
32-bit operation (icc) bit of condition field, **71**, **72**
64-bit operation (xcc) bit of condition field, **71**, **72**
ADD instructions, 145
ASR for, 69
carry (c) bit of condition fields, **72**
icc field, *See* CCR.icc field
MULScc instruction, 287
negative (n) bit of condition fields, **72**
overflow bit (v) in condition fields, **72**
restored by RETRY instruction, 165, 315
saved after trap, 463
saving after trap, 33
state after reset, 631
TSTATE register, 89
write instructions, 382
xcc field, *See* CCR.xcc field
zero (z) bit of condition fields, **72**
CCR.icc field
add instructions, 145, 368
bit setting for signed division, 324
bit setting for signed/unsigned multiply, 331, 379
bit setting for unsigned division, 378
branch instructions, 154, 157, 283
integer subtraction instructions, 364
logical operation instructions, 148, 292, 389
MULScc instruction, 287
Tcc instruction, 372
CCR.xcc field

add instructions, 145, 368
bit setting for signed/unsigned divide, 324, 378
bit setting for signed/unsigned multiply, 331, 379
branch instructions, 157, 283
logical operation instructions, 148, 292, 389
subtract instructions, 364
Tcc instruction, 372
clean register window, 319, 509
clean tag match, 651
clean window, **8**
and window traps, 86, 522
CLEANWIN register, 86
definition, **523**
number is zero, 127
trap handling, 524
*clean_window* exception, 84, 127, 320, 483, **509**, 523, 676
CLEANWIN (clean windows) register, **84**
CANSAVE instruction, 127
clean window counting, 84
incremented by trap handler, 524
range of values, 82, 681
specification for RDPR instruction, 309
specification for WRPR instruction, 386
specifying number of available clean windows, 523
state after reset, 632
value calculation, 86
cleared, **8**, 645
clock cycle, counts for virtual processor, 73
clock tick registers, *See* TICK *and* STICK registers
clock-tick register (TICK), 519
`clr`*n* synthetic instructions, 704
CMP
disabling a core, **605**
parking a core, **608**
`cmp` synthetic instruction, 364, 702
CMT, **8**, **593**, **596**
enabling a core, **605**
ERROR_STEERING register, 618, **620**
Programming Model, 594
registers, **600**
STRAND_AVAILABLE register, 601, **605**
STRAND_ENABLE register, **606**
STRAND_ENABLE_STATUS register, **606**
STRAND_ID register, **601**
STRAND_INTR_ID register, 530, **602**
STRAND_RUNNING register
simultaneous updates, 610
STRAND_RUNNING register, **609**
STRAND_RUNNING_STATUS, **612**
unparking a core, **608**
XIR_STEERING register, **616**
code
self-modifying, 424
coherence, **8**
between processors, 680
data cache, 417
domain, 409
memory, 410
unit, memory, 411
compare and swap instructions, **162**
comparison instruction, 120, 364
compatibility note, **5**
completed (memory operation), **8**
compliance
SPARC V9, 565
compliant SPARC V9 implementation, **25**
cond instruction field
branch instructions, 154, 156, 175, 177
floating point move instructions, 196
move instructions, 283
condition codes
adding, 368
effect of compare-and-swap instructions, 163
extended integer (xcc), 72
floating-point, 175
icc field, 71
integer, 71
results of integer operation (icc), 72
subtracting, 364, 374
trapping on, 372
xcc field, 71
condition codes register, *See* CCR register
conditional branches, 154, 175, 177
conditional move instructions, 32
conforming SPARC V9 implementation, **25**
consistency
between instruction and data spaces, 424
processor, 417, 421
processor self-consistency, 420
sequential, 410, 418, 419
strong, 419
const22 instruction field of ILLTRAP instruction, 238
constants, generating, 325
context, **8**

during TLB miss, 549, 575
nucleus, 189
selection for translation, 555
context identifier, **412**
Context register
determination of, 555
Nucleus, 569
Primary, 568
Secondary, 569
control transfer
pseudo-control-transfer via WRPR to PSTATE.am, 96
*control_transfer_instruction* exception, **509**
CALL/JMPL instructions, 161, 242
DONE/RETRY instructions, 166, 316, 373
RETURN, 318
and Tcc instruction, 373
with branch instructions, 154, 158, 160, 175, 177
control-transfer instructions, 30
control-transfer instructions (CTIs), 30, 165, 315
conventions
font, 2
notational, 3
conversion
between floating-point formats instructions, 234
floating-point to integer instructions, 232, 393
integer to floating-point instructions, 186, 237
planar to packed, 222
copyback, **8**
core, **8**
correctable, **8**
corrected, **8**, 645
CPI, **8**
CPU, pipeline draining, 83, 87
*cpu_mondo* exception, **509**
cross-call, **9**
CTI, **9**, 18
current exception (cexc) field of FSR register, **66**, 130, 673
current window, **9**
current window pointer register, *See* CWP register
current_little_endian (cle) field of PSTATE register, **92**, 413
CWP (current window pointer) register
and instructions
CALL and JMPL instructions, 52
FLUSHW instruction, 191
RDPR instruction, 309
RESTORE instruction, 127, 311
SAVE instruction, 127, 311, 319
WRPR instruction, 386
and traps
after spill trap, 524
after spill/fill trap, 33
on window trap, 524
saved by hardware, 463
CWP (current window pointer) register, **83**
clean windows, 84
definition, 9
incremented/decremented, 51, 311, 319
overlapping windows, 51
range of values, 82, 681
restored during RETRY, 165, 315
specifying windows for use without cleaning, 523
state after reset, 631
and TSTATE register, 89
updated during a WDR reset, 630

## D

D superscript on instruction name, **134**
d16hi instruction field
branch instructions, 159
d16lo instruction field
branch instructions, 159
*DAE_invalid_ASI* exception
with load instructions and ASIs, 259, 448, 451, 452, 453
with store instructions and ASIs, 259, 448, 449, 451, 452, 453
*DAE_invalid_asi* exception, **510**
accessing noncacheable page, 423
ASI loads from TLB Data In register, 581
MMU operation summary, 552
register update policy, 550
signaled by MMU, 553
SWAP/SWAPA, 367
TLB Data Access register fields, 582, 583
with compare-and-swap instructions, 164
with load alternate instructions, 163, 247, 248, 257, 258, 265, 270, 336, 346, 347, 360, 367
with load instructions, 259, 266, 271
with load instructions and ASIs, 258
with nonfaulting load, 427
with store instructions, 266, 337, 348, 361
write to D-SFAR register, 576
*DAE_invalid_ASI* exception (replacing SPARC V9

*data_access_exception*), **509**
*DAE_invalid_ASIn* exception
with load instructions and ASIs, 449
*DAE_nc_page* exception, **510**
accessing noncacheable page, 422
register update policy, 550
signaled by MMU, 553
with compare-and-swap instructions, 164
with load instructions, 251, 264, 266, 273
with store instructions, 266
with SWAP instructions, 367
*DAE_nc_page* exception (replacing SPARC V9 *data_access_exception*), **509**
*DAE_nfo_page* exception, **510**
and TTE.nfo, 540
register update policy, 550
signaled by MMU, 553
with compare-and-swap instructions, 164
with FLUSH instructions, 190
with LDTXA instructions, 274
with load alternate instructions, 248
with load instructions, 245, 252, 255, 259, 261, 263, 264, 266, 268, 271, 276, 358
with nonfaulting load, 427
with store instructions, 266, 333, 337, 341, 343, 348, 350, 353, 356, 361, 363
with SWAP instruction, 365
with SWAP instructions, 367
*DAE_nfo_page* exception (replacing SPARC V9 *data_access_exception*), **509**
*DAE_noncacheable_page* exception
with LDTXA instructions, 274
with LDXFSR instructions, 261, 276
*DAE_privilege_violation* exception, **510**
and TTE.p, 542
MMU operation summary, 552
read/write to privileged location, 511
register update policy, 551
with load alternate instructions, 248
with load instructions, 252, 259, 261, 266, 268, 271, 341, 358
with store instructions, 266, 333, 337, 343, 348, 350, 353, 356, 361, 363, 365
with SWAP instructions, 164, 245, 367
*DAE_privilege_violation* exception (replacing SPARC V9 *data_access_exception*), **509**
*DAE_side_effect_page*
with nonfaulting loads, 410
*DAE_side_effect_page* exception, **510**
load from MMU Data In register, 577
MMU operation summary, 552, 565
register update policy, 551
with load alternate instructions, 248
with load instructions, 252, 259, 271
with nonfaulting load, 427
*DAE_side_effect_page* exception (replacing SPARC V9 *data_access_exception*), **509**
data
access, **9**
cache coherence, 417
conversion between SIMD formats, 43
flow order constraints
memory reference instructions, **416**
register reference instructions, **416**
formats
byte, **35**
doubleword, **35**
halfword, **35**
Int16 SIMD, **44**
Int32 SIMD, **44**
quadword, **35**
tagged word, **35**
Uint8 SIMD, **44**
word, **35**
memory, 426
types
floating-point, 35
signed integer, 35
unsigned integer, 35
width, 35
watchpoint exception, 352
Data Cache Unit Control register, *See* DCUCR
data cache, error detecting/reporting, 646
Data Synchronous Fault Address register, *See* D-SFAR
Data Synchronous Fault Status register, *See* D-SFSR
*data_access_error* exception, **510**
correctable tag errors, 651
L2 cache, hardware reported, 652
register update policy, 551
with compare-and-swap instructions, 164
with load instructions, 252, 255, 263, 264, 266, 268, 274, 276
with store instructions, 266, 334, 341, 343, 348, 350, 354, 356, 358, 361, 363
*data_access_exception* exception (SPARC V9), **509**
*data_access_MMU_error* exception, **510**
detection by hardware, 648

multiple-tag hit error, 568
on PREFETCH, 299, 300, 304
register update policy, 551
software correction, 648
with CASA instruction, 164
with load instructions, 245, 248, 252, 255, 259, 261, 263, 264, 266, 268, 271, 274, 276
with store instructions, 333, 337, 341, 343, 348, 350, 354, 356, 358, 361, 363
with SWAP instruction, 365, 367
*data_access_MMU_miss* exception
register update policy, 551
TTE entries, 547
with integer load instructions, 245
with load alternate instructions, 248
with load instructions, 252
with PREFETCH instruction, 299
with store instructions, 333
*data_access_MMU_miss* exception (superseded), **511**
*data_access_protection* exception (superseded), **522**
*data_invalid_TSB_entry* exception, **511**, 548, 551, 570
*data_real_translation_miss* exception, **511**, 547, 551
*data_store_error* exception (SPARC V8), **520**
DCTI couple, **125**
DCTI instructions, **9**
behavior, 109
RETURN instruction effects, 317
`dec` synthetic instructions, 704
`decccg` synthetic instructions, 704
deferred ESR, **642**
deferred trap, **471**, **638**
distinguishing from disrupting trap, 473
floating-point, 310
handler, 639
restartable, 639
implementation dependency, 473
software actions, 473
termination, **638**
delay instruction
and annul field of branch instruction, 175
annulling, 30
conditional branches, 177
DONE instruction, 165
executed after branch taken, 159
following delayed control transfer, 30
RETRY instruction, 315
RETURN instruction, 317
unconditional branches, 177
with conditional branch, 157
delayed branch, 109
delayed control transfer, 159
delayed CTI, *See* DCTI
demap, **9**
DMMU, 586
IMMU, 586
UMMU, 586
demap operation, **585**
Demap All, **587**
Demap Context, **587**
Demap Page, **586**
Demap Real, **587**
denormalized number, 9
deprecated, **9**
deprecated exceptions
*tag_overflow*, **520**
deprecated instructions
FBA, 174
FBE, 174
FBG, 174
FBGE, 174
FBL, 174
FBLE, 174
FBLG, 174
FBN, 174
FBNE, 174
FBO, 174
FBU, 174
FBUE, 174
FBUGE, 174
FBUL, 174
FBULE, 174
LDFSR, 260
LDTW, 267
LDTWA, 269
MULScc, 70, 287
RDY, 69, 70, 305
SDIV, 70, 323
SDIVcc, 70, 323
SMUL, 70, 331
SMULcc, 70, 331
STFSR, 349
STTW, 357
STTWA, 359
SWAP, 365
SWAPA, 366

TADDccTV, 369
TSUBccTV, 375
UDIV, 70, 377
UDIVcc, 70, 377
UMUL, 70, 379
UMULcc, 70, 379
WRY, 69, 70, 381
*dev_mondo* exception, **511**
diagnostic access
L1 cache, 646
disable (core), **9**
disabled (core), **9**
disabling CMP core, **605**
disp19 instruction field
branch instructions, 156, 177
disp22 instruction field
branch instructions, 153, 175
disp30 instruction field
word displacement (CALL), 161
disrupting ESR, **643**
and hardware-corrected TLB error, 648
associated error infor for L1 cache, 646
and error info for L2 cache, 652
disrupting exception, **639**
disrupting trap, **473**
differences from reset trap, 476
enabling, 639
hardware correction, 648
hyperprivileged mode, 639
types, 639
divide instructions, 30, 289, 323, 377
*division_by_zero* exception, 121, 289, **511**
division-by-zero bits of FSR.aexc/FSR.cexc fields, 68
DMMU
ASIs, 579
context register usage, 558
determining ASI value and endianness, 555
Enable bit, 563
memory operation summary, 552
Secondary Context register, 568, 569
DMMU demap, 586
DMMU Tag Access register
when updatable, 579
DMMU TLB Tag Access register, 578
DMMU_SYNCHRONOUS_FAULT_ADDRESS_REGISTER, 575
DMMU_TLB_DATA_ACCESS, 581
DMMU_TLB_DATA_IN, 580
DMMU_TLB_LOWER_TAG_READ, 582
DMMU_TLB_TAG_READ, 582
DMMU_TLB_TAG_TARGET, 585
DMMU_TLB_UPPER_TAG_READ, 582
DONE instruction, **165**
effect on HTSTATE, 104
effect on TNPC register, 88
effect on TSTATE register, 90
executed in RED_state, 468
generating *illegal_instruction* exception, 513
modifying CCR.XCC condition codes, 72
return from trap, 463
return from trap handler with different GL value, 101
target address, 31
doubleword, **9**
addressing, 116
alignment, 28, 112, 411
data format, **35**
definition, 9
D-SFAR
Fault Address field, **576**
*illustrated*, 576
state after reset, 633
DuTLB, disabled, 510

## E

*ECC_error* exception, **513**
EDGE16 instruction, 168
EDGE16L instruction, 168
EDGE16LN instruction, 170
EDGE16N instruction, 170
EDGE32 instruction, 168
EDGE32L instruction, 168
EDGE32LN instruction, 170
EDGE32N instruction, 170
EDGE8 instruction, 168
EDGE8L instruction, 168
EDGE8LN instruction, 170
EDGE8N instruction, 170
emulating multiple unsigned condition codes, 126
enable (core), **9**
enable floating-point
*See* FPRS register, fef field
*See* PSTATE register, pef field
enabled (core), **9**
enabling CMP core, **605**
error, **9**

access type, 645
condition detected, 645
priority, 644
reporting multiple errors, 644
error handling
by software, 640
fatal error conditions, 640
instructions, 649
*power_on_reset*
effect on ESR registers, 645
precise traps, 642
register files, 648
stores, 650
TLB errors, 648
ERROR signal, 640
`error_state`, **470**
definition, 466
effects when entering, 675
entering, 497, 499, 505, 506, 507
exiting, 497
recognizing interrupts, 498
and `RED_state`, 467
ERROR_STEERING register, **620**
errors
detecting, 637
fatal, 640
reporting, 638, 642
status reporting, 642
store errors, 649
error-type ESR, 642
ESR (Error Status Register), **9**
ESR (error status) register, **637**
after *power_on_reset*, 645
bit state transitions, 643
block type, 642
error info for register file errors, 648
error information fields, 645
error types, 643
errors reported, organization, 642
f field, **643**, 644
flags updated for error condition, 645
information loss, 638
me field, **643**, 644
reading by software, 644
unimplemented bits, 645
writing to by hardware, 643
ESRs
deferred, **642**
disrupting, **643**
fatal, **643**
precise, **642**
even parity, **10**
exception, **10**
exceptions
*See also* individual exceptions
*async_data_error*, **521**
*async_data_error* (superseded), **520**
catastrophic error, 464
causing traps, 463
*clean_window*, 483, **509**, 676
*control_transfer_instruction*, **509**
*cpu_mondo*, **509**
*DAE_invalid_asi*, **510**
*DAE_invalid_ASI* (replacing SPARC V9 *data_access_exception*), 509
*DAE_nc_page*, **510**
*DAE_nc_page* (replacing SPARC V9 *data_access_exception*), 509
*DAE_nfo_page*, **510**
*DAE_nfo_page* (replacing SPARC V9 *data_access_exception*), 509
*DAE_privilege_violation*, **510**, 511
*DAE_privilege_violation* (replacing SPARC V9 *data_access_exception*), 509
*DAE_side_effect_page*, **510**
*DAE_side_effect_page* (replacing SPARC V9 *data_access_exception*), 509
*data_access_error*, **510**
*data_access_exception* (SPARC V9), **509**
*data_access_MMU_error*
on PREFETCH, 300
on *PREFETCH*, 299
*data_access_MMU_error*, 164, 245, 248, 252, 255, 259, 261, 263, 264, 266, 268, 271, 274, 276, 304, 333, 337, 341, 343, 348, 350, 354, 356, 358, 361, 363, 365, **510**
*data_access_MMU_miss*, 299
*data_access_MMU_miss* (superseded), **511**
*data_access_protection* (superseded), **522**
*data_invalid_TSB_entry*, **511**
*data_real_translation_miss*, **511**
*data_store_error* (SPARC V8), **520**
definition, 464
*dev_mondo*, **511**
*division_by_zero*, **511**
*ECC_error*, **513**
*fast_data_access_MMU_miss*, 299, 300, **511**
*fast_data_access_protection*, **511**

*fast_ECC_error*, **522**
*fill_n_normal*, **512**
*fill_n_other*, **512**
*fp_disabled*
and GSR, 76
*fp_disabled*, **512**
*fp_exception_ieee_754*, **512**
*fp_exception_other*, **512**
*guest_watchdog*, **512**
*hstick_match*, 104, 107, **512**
*htrap_instruction*, **512**
*hw_corrected_error*, **513**, 522
*IAE_privilege_violation*, **513**
*illegal_instruction*
and SIR instruction, 476
*illegal_instruction*, 102, **513**
*instruction_access_error*, **514**
*instruction_access_exception* (SPARC V9), **514**
*instruction_access_MMU_error*, **514**
*instruction_access_MMU_miss*, **514**
*instruction_address_range*, **514**
*instruction_breakpoint*, **515**
*instruction_invalid_TSB_entry*, **515**
*instruction_real_range*, **515**
*instruction_real_translation_miss*, **515**, **516**
*instruction_VA_watchpoint*, **516**
*internal_processor_error*, **516**
*interrupt_level_14*
and SOFTINT.int_level, 78
and STICK_CMPR.stick_cmpr, 82
and TICK_CMPR.tick_cmpr, 80
*interrupt_level_14*, 517
*interrupt_level_15*
and SOFTINT.int_level, 78
*interrupt_level_15*, 519
*interrupt_level_n*
and SOFTINT register, 77
and SOFTINT.int_level, 78
*interrupt_level_n*, 474, **517**
*LDDF_mem_address_not_aligned*, **517**
*LDQF_mem_address_not_aligned*, **522**
*mem_address_not_aligned*, **517**
*mem_address_range*, **517**
*mem_real_range*, **518**
*nonresumable_error*, **519**
*PA_watchpoint*, **519**
pending, 33
*pic_overflow*, **519**
*power_on_reset*, **519**
*privileged_action*, **519**
*privileged_opcode*
and access to register-window PR state registers, 82, 87, 98, 101
and access to SOFTINT, 77
and access to SOFTINT_CLR, 79
and access to SOFTINT_SET, 79
and access to STICK_CMPR, 81
and access to TICK_CMPR, 79
*privileged_opcode*, **519**
*RA_watchpoint*, 488, 494
*RED_state_exception*, **520**
*resumable_error*, **520**
*software_initiated_reset*, **520**
*spill_n_normal*, 320, **520**
*spill_n_other*, 320, **520**
*STDF_mem_address_not_aligned*, **520**
*store_error*, **520**
*STQF_mem_address_not_aligned*, **522**
*sw_recoverable_error*, **520**, 521
*tag_overflow* (deprecated), **520**
*trap_instruction*, **520**
*trap_level_zero*
state after reset, 631
*trap_level_zero*, **520**
*unimplemented_LDTW*, **521**
*unimplemented_STTW*, 521
*VA_watchpoint*, **521**
*watchdog_reset*
and *guest_watchdog*, 465
*watchdog_reset*, **521**
window fill, 483
window spill, 483
execute unit, 415
`execute_state`
and `error_state`, 499
and `RED_state`, 499
returning to, 497
trap processing, 466, 497
execution unit, error protection/checking, 649
explicit ASI, **10**, **117**, 431
extended word, **10**
addressing, 116
*externally_initiated_reset* (XIR), **505**, **511**, **629**
causing entry into `RED_state`, 467
entering `error_state`, 466
and `error_state`, 482
for critical system events, 476
for debugging, 469

partial-processor, **615**
RED_state trap processing, 501
to virtual processor, 629
virtual processor trap processing, 497
when TL = *MAXTL*, 497

## F

F registers, **10**, 26, 129, 391, 478
FABSd instruction, **171**, 661, 662
FABSq instruction, **171**, 171, 661, 662
FABSs instruction, **171**
FADD, 172
FADDd instruction, **172**
FADDq instruction, **172**, 172, 184
FADDs instruction, **172**
FALIGNDATA instruction, 173
FAND instruction, 230
FANDNOT1 instruction, 230
FANDNOT1S instruction, 230
FANDNOT2 instruction, 230
FANDNOT2S instruction, 230
FANDS instruction, 230
*fast_data_access_MMU_miss* exception, **511**
hardware-corrected, 647
MMU operation summary, 552
register update policy, 551
TLB miss-and-refill step, 549
with integer load instructions, 245
with load alternate instructions, 248
with PREFETCH instruction, 299, 300
with store instructions, 333
*fast_data_access_protection* exception, **511**
MMU operation summary, 552
register update policy, 551
with store instructions, 341
write permission not granted, 543
*fast_ECC_error* exception, **522**
*fast_instruction_access_MMU_miss* exception
hardware-corrected, 647
MMU operation summary, 552
register update policy, 550
TLB miss, 549
TTE access for instruction fetch, 651
fatal error
causes, 651
detecting, 640
in external interface or bus, 652
store buffer tag/control parity, 650
fatal ESR, **643**
associated error info for L2 cache, 651
external interface or bus error, 652
fault, **10**
fault, stuck-at, 637
FBA instruction, 174, 175, 663
FBE instruction, 174, 663
FBfcc instructions, 60, **174**, 512, 657, 663
FBG instruction, 174, 663
FBGE instruction, 174, 663
FBL instruction, 174, 663
FBLE instruction, 174, 663
FBLG instruction, 174, 663
FBN instruction, 174, 663
FBNE instruction, 174, 663
FBO instruction, 174, 663
FBPA instruction, 176, 177, 663
FBPE instruction, 176, 663
FBPfcc instructions, 60, **176**, 657, 663, 664
FBPG instruction, 176, 663
FBPGE instruction, 176, 663
FBPL instruction, 176, 663
FBPLE instruction, 176, 663
FBPLG instruction, 176, 663
FBPN instruction, 176, 177, 663
FBPNE instruction, 176, 663
FBPO instruction, 176, 663
FBPU instruction, 176, 663
FBPUE instruction, 176, 663
FBPUG instruction, 176, 663
FBPUGE instruction, 176, 663
FBPUL instruction, 176, 663
FBPULE instruction, 176, 663
FBU instruction, 174, 663
FBUE instruction, 174, 663
FBUG instruction, 174, 663
FBUGE instruction, 174, 663
FBUL instruction, 174, 663
FBULE instruction, 174, 663
fcc-conditional branches, 175, 177
fcc*n*, **10**
FCMP instructions, 664
FCMP* instructions, 60, 61, 182
FCMPd instruction, **182**, 662
FCMPE instructions, 664
FCMPE* instructions, 60, 61, 182
FCMPEd instruction, **182**, 662
FCMPEq instruction, **182**, 183, 662
FCMPEQ16 instruction, 179

FCMPEQ32 instruction, 179
FCMPEs instruction, **182**, 662
FCMPGT instruction, 179
FCMPGT16 instruction, 179
FCMPGT32 instruction, 179
FCMPLE16 instruction, 179
FCMPLE16 instruction, 179
FCMPLE32 instruction, 179
FCMPLE32 instruction, 179
FCMPNE16 instruction, 179, 180
FCMPNE32 instruction, 179, 180
FCMPq instruction, **182**, 183, 662
FCMPs instruction, **182**, 662
fcn instruction field
  DONE instruction, 165
  PREFETCH, 297
  RETRY instruction, 315
FDIVd instruction, **184**
FDIVq instruction, **184**
FDIVs instructions, 184
FdMULq instruction, **210**, 210
FdTOi instruction, **232**, 393
FdTOq instruction, **234**, 234
FdTOs instruction, **234**
FdTOx instruction, **232**, 662
fef field of FPRS register, **75**
  and access to GSR, 76
  and *fp_disabled* exception, 512
  branch operations, 175, 177
  byte permutation, 155
  comparison operations, 180, 183
  component distance, 294
  data formatting operations, 185, 213, 223
  data movement operations, 284
  enabling FPU, 94
  floating-point operations, 171, 172, 184, 186, 193, 195, 199, 202, 210, 212, 231, 232, 234, 236, 237, 253, 256, 260, 262, 275
  integer arithmetic operations, 204, 221, 226
  logical operations, 227, 228, 230
  memory operations, 251
  read operations, 307, 327, 340
  special addressing operations, 146, 173, 342, 349, 353, 355, 362, 383
fef, *See* FPRS register, fef field
FEXPAND instruction, 185
FEXPAND operation, 185
fill handler, 312
fill register window, 512
  overflow/underflow, 52
  RESTORE instruction, 86, 311, 522
  RESTORED instruction, 128, 313, 524
  RETRY instruction, 524
  selection of, 523
  trap handling, 523, 524
  trap vectors, 312
  window state, 86
*fill_n_normal* exception, 312, 318, 512, **512**
*fill_n_other* exception, 312, 318, **512**
FiTOd instruction, **186**
FiTOq instruction, **186**, 186, 237
FiTOs instruction, **186**
fixed values, 239
fixed-point scaling, 205
floating point
  absolute value instructions, **171**
  add instructions, **172**
  compare instructions, 60, 61, **182**, 182
  condition code bits, 175
  condition codes (fcc) fields of FSR register, 63, 175, 177, 182
  data type, 35
  deferred-trap queue (FQ), 310
  divide instructions, **184**
  exception, **10**
  exception, encoding type, 62
  FPRS register, 382
  FSR condition codes, 61
  move instructions, **195**
  multiply instructions, **210**
  multiply-add/subtract, 192
  negate instructions, **212**
  operate (FPop) instructions, **10**, **32**, 62, 66, 129, 260
  registers
    destination F, 391
    FPRS, *See* FPRS register
    FSR, *See* FSR register
    programming, 58
  rounding direction, 61
  square root instructions, **231**
  subtract instructions, **236**
  trap types, **10**
    IEEE_754_exception, 63, **64**, 65, 66, 69, 391, 392
    invalid_fp_register, 236
    unfinished_FPop, 63, **64**, 69, 172, 184, 211, 231, 235, 236, 392

results after recovery, 64
unimplemented_FPop, 69, 203, 392
traps
deferred, 310
precise, 310
floating-point condition codes (fcc) fields of FSR register, 478
floating-point operate (FPop) instructions, 512
floating-point register file (FRF), 648
floating-point trap types
IEEE_754_exception, 478, 512
floating-point unit (FPU), **10**, **26**
FLUSH instruction, 189
memory ordering control, 279
FLUSH instruction
memory/instruction synchronization, 187
FLUSH instruction, **187**, 426
data access, 9
immediacy of effect, 190
in multiprocessor system, 187
in self-modifying code, 189
latency, 681
flush instruction memory, *See* FLUSH instruction
flush register windows instruction, **191**
FLUSHW instruction, **191**, 520
effect, 32
management by window traps, 86, 522
spill exception, 129, 191, 524
FMA instructions
fused, 192
FMADDd instruction, 192
FMADDs instruction, 192
FMAu instruction, 673
FMOVcc instructions
conditionally moving floating-point register contents, 72
conditions for copying floating-point register contents, 125
copying a register, 60
encoding of opf<84> bits, 662
encoding of opf_cc instruction field, 664
encoding of rcond instruction field, 663
floating-point moves, **196**
FPop instruction, 130
used to avoid branches, 200, 283
FMOVccd instruction, 662
FMOVccq instruction, 662
FMOVd instruction, **195**, 661, 662
FMOVDfcc instructions, 196
FMOVdGEZ instruction, **201**
FMOVdGZ instruction, **201**
FMOVDicc instructions, 196
FMOVdLEZ instruction, **201**
FMOVdLZ instruction, **201**
FMOVdNZ instruction, **201**
FMOVdZ instruction, **201**
FMOVq instruction, **195**, 195, 661, 662
FMOVQfcc instructions, 196, 199
FMOVqGEZ instruction, **201**
FMOVqGZ instruction, **201**
FMOVQicc instructions, 196, 199
FMOVqLEZ instruction, **201**
FMOVqLZ instruction, **201**
FMOVqNZ instruction, **201**
FMOVqZ instruction, **201**
FMOVr instructions, 130, 663
FMOVRq instructions, 202
FMOVRsGZ instruction, **201**
FMOVRsLEZ instruction, **201**
FMOVRsLZ instruction, **201**
FMOVRsNZ instruction, **201**
FMOVRsZ instruction, **201**
FMOVs instruction, **195**
FMOVScc instructions, 198
FMOVSfcc instructions, 196
FMOVsGEZ instruction, **201**
FMOVSicc instructions, 196
FMOVSxcc instructions, 196
FMOVxcc instructions, 196, 199
FMSUBd instruction, 192
FMSUBs instruction, 192
FMUL8SUx16 instruction, 204, 207
FMUL8ULx16 instruction, 204, 207
FMUL8x16 instruction, 204, 205
FMUL8x16AL instruction, 204, 206
FMUL8x16AU instruction, 204, 206
FMULd instruction, **210**
FMULD8SUx16 instruction, 204, 208
FMULD8ULx16 instruction, 204, 209
FMULq instruction, **210**, 210, 231
FMULs instruction, **210**
FNAND instruction, 230
FNANDS instruction, 230
FNEG instructions, **212**
FNEGd instruction, **212**, 661, 662
FNEGq instruction, **212**, 212, 661, 662
FNEGs instruction, **212**
FNMADDd instruction, 192

FNMADDs instruction, 192
FNMSUBd instruction, 192
FNMSUBs instruction, 192
FNOR instruction, 230
FNORS instruction, 230
FNOT1 instruction, 228
FNOT1S instruction, 228
FNOT2 instruction, 228
FNOT2S instruction, 228
FONE instruction, 227
FONES instruction, 227
FOR instruction, 230
formats, instruction, 110
FORNOT1 instruction, 230
FORNOT1S instruction, 230
FORNOT2 instruction, 230
FORNOT2S instruction, 230
FORS instruction, 230
*fp_disabled* exception, **512**
absolute value instructions, 171, 172, 236
and GSR, 76
FPop instructions, 130
FPRS.fef disabled, 75
PSTATE.pef not set, 75, 76, 688
with branch instructions, 175, 178
with compare instructions, 181
with conversion instructions, 186, 233, 235, 237
with floating-point arithmetic instructions, 184, 194, 211, 221, 226
with FMOV instructions, 195
with load instructions, 259, 263
with move instructions, 200, 203, 284
with negate instructions, 212
with store instructions, 342, 343, 347, 349, 350, 353, 355, 356, 362, 363, 383
*fp_exception* exception, 65
*fp_exception_ieee_754* "invalid" exception, 232
*fp_exception_ieee_754* exception, **512**
and tem bit of FSR, 62
cause encoded in FSR.ftt, 63
FSR.aexc, 65
FSR.cexc, 67
FSR.ftt, 66
generated by FCMP or FCMPE, 61
and IEEE 754 overflow/underflow conditions, 66
trap handler, 392
when FSR.ns = 1, 394, 395, 673
when FSR.tem = 0, 478
when FSR.tem =1, 478
with floating-point arithmetic instructions, 172, 184, 194, 211, 236
*fp_exception_other* exception, 69, **512**
cause encoded in FSR.ftt, 63
FSUBq instruction, 236
incorrect IEEE Std 754-1985 result, 130, 671
supervisor handling, 392
trap type of unfinished_FPop, 64
when quad FPop unimplemented in hardware, 65
with floating-point arithmetic instructions, 184, 194, 211
FPACK instruction, 77
FPACK instructions, **213–217**
FPACK16 instruction, 213, 214
FPACK16 operation, 214
FPACK32 instruction, 213, 215
FPACK32 operation, 215
FPACKFIX instruction, 213, 217
FPACKFIX operation, 217
FPADD16 instruction, 219
FPADD16S instruction, 219
FPADD32 instruction, 219
FPADD32S instruction, 219
FPMERGE instruction, 222
FPop, **10**
FPop, *See* floating-point operate (FPop) instructions
FPRS register
*See also* floating-point registers state (FPRS) register
FPRS register, **75**
ASR summary, 69
definition, 10
fef field, 130, 477
RDFPRS instruction, 306
state after reset, 632
FPRS register fields
dl (dirty lower fp registers), **76**
du (dirty upper fp registers, **76**
fef, **75**
fef, *See also* fef field of FPRS register
FPSUB16 instruction, 224
FPSUB16S instruction, 224
FPSUB32 instruction, 224
FPSUB32S instruction, 224
FPU, **10**
FqTOd instruction, **234**, 234
FqTOi instruction, **232**, 232, 393

FqTOs instruction, **234**, 234
FqTOx instruction, **232**, 232, 661, 662
*freg*, **696**
FsMULd instruction, **210**
FSQRTd instruction, **231**
FSQRTq instruction, **231**
FSQRTs instruction, **231**
FSR (floating-point state) register
  fields
    aexc (accrued exception), 63, 64, **65**, 65, 391
    aexc (accrued exceptions), 193
      in user-mode trap handler, 392
    -- dza (division by zero) bit of aexc, 68
    -- nxa (rounding) bit of aexc, 68
    cexc (current exception), 61, 63, 64, **66**, 66, 67, 391, 512
    cexc (current exceptions), 193
      in user-mode trap handler, 392
    -- dzc (division by zero) bit of cexc, 68
    -- nxc (rounding) bit of cexc, 68
    fcc (condition codes), **60**, 63, 64, 392, 697
    fcc*n*, **61**
    ftt (floating-point trap type), 60, **62**, 65, 66, 130, 275, 349, 362, 512
      in user-mode trap handler, 392
    not modified by LDFSR/LDXFSR instructions, 60
    ns (nonstandard mode), 60, 260, 275
    qne (queue not empty), 60, **65**, 260, 275
      in user-mode trap handler, 392
    rd (rounding), **61**
    tem (trap enable mask), **61**, 65, 67, 193, 393, 394, 512
    ver, **62**
    ver (version), 60, 275
FSR (floating-point state) register, **60**
  after floating-point trap, 391
  compliance with IEEE Std 754-1985, 69
  LDFSR instruction, 260
  reading/writing, 60
  state after reset, 632
  values in ftt field, 63
  writing to memory, 349, 362
FSRC1 instruction, 228
FSRC1S instruction, 228
FSRC2 instruction, 228
FSRC2S instruction, 228
FsTOd instruction, **234**
FsTOi instruction, **232**, 393
FsTOq instruction, **234**, 234
FsTOx instruction, **232**, 661, 662
FSUBd instruction, **236**
FSUBq instruction, **236**, 236
FSUBs instruction, **236**
functional choice, implementation-dependent, 671
fused FMA instructions, 192
FXNOR instruction, 230
FXNORS instruction, 230
FXOR instruction, 230
FXORS instruction, 230
FxTOd instruction, **237**, 662
FxTOq instruction, **237**, 662
FxTOs instruction, **237**, 662
FZERO instruction, 227
FZEROS instruction, 227

## G

general status register, *See* GSR (general status) register
generating constants, 325
GL register, **99**
  access, 100
  during resets, 101
  during trap processing, 497
  function, 99
  reading with RDPR instruction, 309, 386
  relationship to TL, 100
  restored during RETRY, 165, 315
  SPARC V9 compatibility, 97
  and TSTATE register, 89
  value restored from TSTATE[TL], 101
  value restored from TSTATE[TL], 100, 165, 315
  and VER.maxgl, 106
  writing to, 100
global level register, *See* GL register
*global* registers, 22, 26, 48, **50**, 50, 671
graphics status register, *See* GSR (general status) register
GSR (general status) register
  fields
    align, **77**
    im (interval mode) field, **77**
    irnd (rounding), **77**
    mask, **77**
    scale, **77**
GSR (general status) register
  ASR summary, 70

*guest_watchdog* exception, **512**

## H

H superscript on instruction name, **134**
halfword, **11**
  alignment, 28, **112**, 411
  data format, **35**
hardware
  correcting ECC errors, 652
  correcting L2 cache errors, 651
  correcting TLB errors, 647
  dependency, **670**
  error correction, 645
  hardware-corrected error, 640
  L2 cache errors reported, 652
  restartable deferred traps, 639
  setting ESR bits, 643
  signalling fatal error in L2 cache, 650
  TLB, 590
  TLB error checking, 647
  traps, 483
hardware tablewalk, **546**, 647
  disabled, 544, 549
  enabled, 652
  handling multiple context IDs, 547–548
  loading of missing TTE entries, 546
  loading privileged code, 546
  MMU support for, 545
  and ra field of TTE, 548
  real address requests, 547
  and software tablewalk, 590
  TLB miss/reload sequence, 547
  and TSB pointers, 544
hardware trap stack, 33
HINTP register, **104**
HPR state registers (ASRs), 101–108
hpriv field of HPSTATE register, 468
HPSTATE register, **102**
  entering hyperprivileged execution mode, 463
  hpriv field, **103**
  hpriv field, *See also* hyperprivileged (hpriv) field of HPSTATE register
  and HTSTATE register, 103
  ibe field, 515
  ibe field, **102**
  red field, **102**, 468
  state after reset, 631
  tlz field, **103**
  tlz field, and *trap_level_zero* exception, 103, 521
HPSTATE register fields
  hpriv
    determining mode, 13
hsp (*hstick_match* pending) field of HINTP register, **105**, 107
HSTICK_CMPR register, **107**, 512
  and HINTP, 105
*hstick_match* exception, 104, 107, **512**
*hstick_match* pending (hsp) field of HINTP register, **105**, 107
HTBA (hyperprivileged trap base address) register, **105**, 465, 512
  establishing table address, 463
  initialization, 480
  state after reset, 631
*htrap_instruction* exception, 373, **512**
HTSTATE (hyperprivileged trap state) register, **103**
  number of copies for reading, 308
  number of copies for writing, 384
  reading, 308
  writing to, 384
HVER (version) register
  fields
    maxtl, **106**
    maxwin, **106**
HVER (version) register, **106**
  state after reset, 632
HVER (version) register fields
  impl, **62**, **106**
  manuf, 106
  mask, **106**
  maxgl, **106**
  maxwin, **106**
*hw_corrected_error* exception, **513**, 522, 651
*hw_corrected_error* trap, **639**
  disrupting traps, 650
  for correctable store buffer data, 649
  handler routine, 640
  L1 caches, 646
  L2 cache errors, 652
  logging, 648
  reporting, 643
hyperprivileged, **11**
  mode, 87
  registers, **101**
hyperprivileged (hpriv) field of HPSTATE register, 307, 360
  access to register-window PR state registers, 87

and trap control, 478
compare and swap instructions, 163, 367
disrupting trap condition detected, 474
load instructions, 247, 251, 257, 265, 270, 341
*privileged_action* exception, 413
store instructions, 335, 346, 360
*trap_level_zero* exception, 167, 316, 385, 387, 521
hyperprivileged mode
byte order, 478
hyperprivileged scratchpad registers
state after reset, 633
hypervisor (software), **11**

## I

i (integer) instruction field
arithmetic instructions, 287, 289, 292, 323, 331, 377, 379
floating point load instructions, 253, 256, 260, 275
flush memory instruction, 187
flush register instruction, 191
jump-and-link instruction, 242
load instructions, 244, 264, 265, 267, 269
logical operation instructions, 148, 292, 389
move instructions, 283, 285
POPC, 295
PREFETCH, 297
RETURN, 317
I/O
access, 409
memory, 408
memory-mapped, 409
*IAE_* exception* exception
when IMMU is disabled, 564
*IAE_nfo_page* exception, 550
*IAE_privilege_violation* exception, **513**
and TTE.p, 542
register update policy, 550
*IAE_unauth_access* exception, 543, 550
IEEE 754, **11**
IEEE Std 754-1985, 11, 21, 61, 64, 67, 69, 130, 391, 671
IEEE_754_exception floating-point trap type, **11**, 63, **64**, 65, 69, 391, 392, 478, 512
IEEE-754 exception, **11**
IER register (SPARC V8), 383
*illegal_instruction* exception, 191, **513**
and SIR instruction, 476
attempt to write in nonprivileged mode, 81
DONE/RETRY, 167, 316, 317
HTSTATE register, reading/writing, 102, 104
ILLTRAP, 238
not implemented in hardware, 144
POPC, 296
PREFETCH, 304
RETURN, 318
with BPr instruction, 160
with branch instructions, 158, 160
with CASA and CASXA instructions, 163, 292
with CASXA instruction, 164
with DONE instruction, 166
with FMOV instructions, 195
with FMOVcc instructions, 200
with FMOVR instructions, 203
with load instructions, 54, 251, 254, 268, 270, 276, 452
with move instructions, 284, 286
with OTHERW instruction, 326
with RDHPR instructions, 308
with read hyperprivileged register instructions, 308, 309
with read instructions, 306, 307, 308, 309, 387, 675
with store instructions, 258, 343, 350, 357, 358, 360, 363
with Tcc instructions, 373
with TPC register, 87
with TSTATE register, 89
with write instructions, 383, 384, 385, 388
write to ASR 5, 75
write to STICK register, 81
write to TICK register, 74
ILLTRAP instruction, **238**, 513
imm_asi instruction field
explicit ASI, providing, 117
floating point load instructions, 256
load instructions, 265, 267, 269
PREFETCH, 297
immediate CTI, 109
I-MMU
and instruction prefetching, 410
IMMU
ASIs, 579
context register usage, 558
determining ASI value and endianness, 555
Enable bit, 563
memory operation summary, 552

IMMU demap, 586
IMMU TLB Tag Access register, 578
IMMU_TLB_DATA_ACCESS, 581
IMMU_TLB_DATA_IN, 580
IMMU_TLB_LOWER_TAG_READ, 582
IMMU_TLB_TAG_READ, 582
IMMU_TLB_TAG_TARGET, 585
IMMU_TLB_UPPER_TAG_READ, 582
IMPDEP1 instruction, 239
IMPDEP1 instructions, 239, 665, 666
IMPDEP2 instruction, 240
IMPDEP2A instructions, 239, 513, 677
IMPDEP2B instructions, 130, 239, 513
implementation, **11**
implementation (impl) field of HVER register, **62**, **106**
implementation dependency, **669**
implementation dependent, **11**
implementation note, **5**
implementation-dependent functional choice, 671
implementation-dependent instructions, *See* IMPDEP2A instructions
implicit ASI, **11**, **117**, 430
implicit ASI memory access
  LDFSR, 260
  LDSTUB, 264
  load fp instructions, 253, 275
  load integer doubleword instructions, 267
  load integer instructions, 244
  STD, 357
  STFSR, 349
  store floating-point instructions, 342, 362
  store integer instructions, 333
  SWAP, 365
implicit byte order, 93
*in* registers, 49, 51, 319
inccc synthetic instructions, 704
inexact accrued (nxa) bit of aexc field of FSR register, 393
inexact current (nxc) bit of cexc field of FSR register, 393
inexact mask (nxm) field of FSR.tem, **67**
inexact quotient, 323, 377
infinity, 393, 394
initiated, **11**
input/output (I/O) locations
  access by nonprivileged code, 672
  behavior, 408
  contents and addresses, 672
  identifying, 680
  order, 408
  semantics, 681
  value semantics, 408
instruction cache
  error detection, 646
  error reporting (L1), 646
instruction fields, **11**
  *See also* individual instruction fields
  definition, 11
instruction group, **11**
instruction MMU, *See* I-MMU
instruction prefetch buffer, invalidation, 188
instruction set architecture (ISA), 11, **12**, 23
*instruction_access_error* exception, **514**, 550, 652
*instruction_access_exception* exception (SPARC V9), **514**
*instruction_access_MMU_error* exception, **514**, 550, 648
  multiple-tag hit error, 568
*instruction_access_MMU_miss* exception, **514**
  register update policy, 550
  TTE entries, 547
*instruction_address_range* exception, **514**
  register update policy, 550
*instruction_breakpoint* exception, **515**
*instruction_invalid_TSB_entry* exception, **515**, 548, 550, 570
*instruction_real_range* exception, **515**
*instruction_real_range* exception, register update policy, 550
*instruction_real_translation_miss* exception, **515**, **516**, 547, 550
*instruction_VA_watchpoint* exception, **516**, 550
instructions
  32-bit wide, 22
  alignment, **112**
  alignment, 28, 146, 411
  arithmetic, integer
    addition, 145, 368
    division, 30, 289, 323, 377
    multiplication, 30, 287, 289, 331, 379
    subtraction, 364, 374
    tagged, 30
  array addressing, 149
  atomic
    CASA/CASXA, 162
    load twin extended word from alternate space, 272
    load-store, 111, 162, 264, 265, 365, 366

load-store unsigned byte, 264, 265
successful loads, 244, 246, 268, 270
successful stores, 333, 335
branch
branch if contents of integer register match condition, **159**
branch on floating-point condition codes, **174**, **176**
branch on integer condition codes, **153**, **156**
cache, 417
causing illegal instruction, 238
compare and swap, **162**
comparison, 120, 364
conditional move, 32
control-transfer (CTIs), 30, 165, 315
conversion
convert between floating-point formats, **234**
convert floating-point to integer, **232**
convert integer to floating-point, **186**, **237**
floating-point to integer, 393
count of number of bits, 295
edge handling, 168
fetches, 112
floating point
compare, 60, 61, 182
floating-point add, **172**
floating-point divide, **184**
floating-point load, 111, 253
floating-point load from alternate space, 256
floating-point load state register, 253, 275
floating-point move, 195, **196**, 201
floating-point operate (FPop), 32, 260
floating-point square root, **231**
floating-point store, 111, 342
floating-point store to alternate space, 345
floating-point subtract, **236**
operate (FPop), 62, 66
short floating-point load, 262
short floating-point store, 355
status of floating-point load, 260
flush instruction memory, **187**
flush register windows, **191**
formats, **110**
generate software-initiated reset, 328
implementation-dependent, *See* IMPDEP2A instructions
jump and link, 30, **242**
loads
block load, 249
floating point, *See* instructions: floating point
integer, 111
integer from alternate space, 598
simultaneously addressing doublewords, 365
unsigned byte, 162, **264**
unsigned byte to alternate space, 265
logical operations
64-bit/32-bit, 228, 230
AND, **148**
logical 1-operand ops on F registers, 227
logical 2-operand ops on F registers, 228
logical 3-operand ops on F registers, 230
logical XOR, 389
OR, **292**
memory, 426
moves
floating point, *See* instructions: floating point
move integer register, **281**, **285**
on condition, 22
ordering MEMBAR, 120
permuting bytes specified by GSR.mask, 155
pixel component distance, **294**, 294
pixel formatting (PACK), 213
prefetch data, **297**
read hyperprivileged register, **308**
read privileged register, **309**
read state register, 31, **305**
register window management, 32
reordering, 416
reserved, 130
*reserved* fields, 144
RETRY
and restartable deferred traps, 473
RETURN vs. RESTORE, 317
sequencing MEMBAR, 120
set high bits of low word, 325
set interval arithmetic mode, 327
setting GSR.mask field, 155
shift, 30
shift, **329**
shift count, 329
shut down to enter power-down mode, 326
SIMD, **17**
simultaneous addressing of doublewords, 366
SIR, **328**
software-initiated reset, 328
stores
block store, 338
floating point, *See* instructions: floating point

integer, 111, 333
integer (except doubleword), 333
integer into alternate space, 335, 598
partial, 351
unsigned byte, 162
unsigned byte to alternate space, 265
unsigned bytes, 264
swap R register, **365**, **366**
synthetic (for assembly language programmers), 702–704
tagged addition, 368
test-and-set, 423
timing, 144
trap on integer condition codes, **371**
write hyperprivileged register, **384**
write privileged register, **386**
write state register, 382
integer register file (IRF), 648
integer unit (IU)
condition codes, 72
definition, **12**
description, 26
*internal_processor_error* exception, **516**, 648, 649
interrupt
enable (ie) field of PSTATE register, 474, 477
level, 99
request, **12**, 33, 463
*interrupt_level_14* exception, 78, 517
and SOFTINT.int_level, 78
and STICK_CMPR.stick_cmpr, 82
and TICK_CMPR.tick_cmpr, 80
*interrupt_level_15* exception, 519
and SOFTINT.int_level, 78
*interrupt_level_n* exception, 474, **517**
and SOFTINT register, 77
and SOFTINT.int_level, 78
inter-strand operation, **12**
intra-strand operation, **12**
invalid accrued (nva) bit of aexc field of FSR register, **68**
invalid ASI
and *DAE_invalid_asi*, 510
*DAE_invalid_asi* exception, 577
invalid current (nvc) bit of cexc field of FSR register, **68**, 393, 394
invalid mask (nvm) field of FSR.tem, **67**, 393, 394
*invalid_exception* exception, 232
invalid_fp_register floating-point trap type, 203, 212, 236
INVALW instruction, **241**
`iprefetch` synthetic instruction, 702
ISA, **12**
ISA, *See* instruction set architecture
issue unit, 415, **415**
issued, **12**
italic font, in assembly language syntax, 695
IU, **12**
`ixc` synthetic instructions, 704

## J

`jmp` synthetic instruction, 702
JMPL instruction, **242**
computing target address, 30
does not change CWP, 52
*mem_address_not_aligned* exception, 517
reexecuting trapped instruction, 317
jump and link, *See* JMPL instruction

## L

L1 cache
bit protection, 646
diagnostic access error checking, 646
error checking for accesses, 646
error handling by hardware, 646
L2 cache, **651**
and uncorrectable ECC errors, 649
error reporting, 651
errors detected, 652
invalidating line, 652
recoverable errors, 651
SEC/DED ECC protection, 651
LD instruction (SPARC V8), 244
LDBLOCKF instruction, **249**, 451
LDBLOCKF instruction, *DAE_nc_page* exception, 510
LDD instruction (SPARC V8 and V9), 268
LDDA instruction, 450
LDDA instruction (SPARC V8 and V9), 270
LDDF instruction, 112, **253**, 517
*LDDF_mem_address_not_aligned* exception, **517**
address not doubleword aligned, 678
address not quadword aligned, 679
LDDF/LDDFA instruction, 112
load instruction with partial store ASI and misaligned address, 259
with load instructions, 254, 257, 452

with store instructions, 346, 452
*LDDF_mem_not_aligned* exception, 59
LDDFA instruction, **256**, 353
alignment, 112
ASIs for fp load operations, 452
behavior with block store with Commit ASIs, 258
behavior with partial store ASIs, 254–??, 258, 259–??, 275–??, 452–??
causing *LDDF_mem_address_not_aligned* exception, 112, 517
for block load operations, 451
used with ASIs, 451
LDF instruction, 59, **253**
LDFA instruction, 59, **256**
LDFSR instruction, 60, 62, 63, **260**, 513
LDQF instruction, **253**, 522
*LDQF_mem_address_not_aligned* exception, **522**
address not quadword aligned, 679
LDQF/LDQFA instruction, 113
with load instructions, 257
LDQFA instruction, **256**
LDSB instruction, **244**
LDSBA instruction, **246**
LDSH instruction, **244**
LDSHA instruction, **246**
LDSHORTF instruction, 262
LDSTUB instruction, 111, **264**, 265, 423, 424
and *DAE_nc_page* exception, 510
hardware primitives for mutual exclusion of LDSTUB, 422
LDSTUBA instruction, 264, **265**
alternate space addressing, 29
and *DAE_nc_page* exception, 510
hardware primitives for mutual exclusion of LDSTUBA, 422
LDSW instruction, **244**
LDSWA instruction, **246**
LDTW instruction, 54, 112
LDTW instruction (deprecated), **267**
LDTWA instruction, 54, 112
LDTWA instruction (deprecated), **269**
LDTX instruction, 447
LDTX instruction, *DAE_nc_page* exception, 510
LDTXA instruction, 114, 116, **272**, 448
access alignment, 112
access size, 112
LDUB instruction, **244**
LDUBA instruction, **246**
LDUH instruction, **244**
LDUHA instruction, **246**
LDUW instruction, **244**
LDUWA instruction, **246**
LDX instruction, **244**
LDXA instruction, **246**, 271, 420, 598
reading from a CMP register, 600
LDXFSR instruction, 60, 62, 63, 260, **275**, **321**, 513
leaf procedure
modifying windowed registers, 128
little-endian byte order, **12**, 28, 92
load
block, *See* block load instructions
floating-point from alternate space instructions, **256**
floating-point instructions, **253**, 260
floating-point state register instructions, **253**, **275**
from alternate space, 29, 73, 117, 598
instructions, **12**
instructions accessing memory, 111
nonfaulting, 415, 565
short floating-point, *See* short floating-point load instructions
`LoadLoad` MEMBAR relationship, 278
`LoadLoad` MEMBAR relationship, 425
`LoadLoad` predefined constant, 700
loads
nonfaulting, 427
load-store alignment, 28, **112**, 411
load-store instructions
compare and swap, **162**
definition, **12**
and *fast_data_access_protection* exception, 511
load-store unsigned byte, 162, **264**, 365, 366
load-store unsigned byte to alternate space, **265**
memory access, 27
swap R register with alternate space memory, **366**
swap R register with memory, 162, **365**
`LoadStore` MEMBAR relationship, 278, 425
`LoadStore` predefined constant, 700
*local* registers, 49, 51, 311
logical XOR instructions, **389**
`Lookaside` predefined constant, 700
LSTPARTIALF instruction, 452

## M

machine state

after reset, 630, 634
in RED_state, 630, 634
manufacturer (manuf) field of HVER register, **106**
manufacturer (manuf) field of VER register, 677
mask number (mask) field of HVER register, **106**
masked-off trap
deferred trap, 638
disrupting trap, 639
precise trap, 638
*MAXGL*, 26, 48, 50, **97**, **99**, 100
maximum global levels maxgl field of HVER register, **106**
maximum trap levels maxtl field of HVER register, **106**
*MAXPGL*, **97**, **99**
*MAXTL*
and error_state, 499
and *MAXGL*, 100
and RED_state, 499
instances of HTSTATE register, 103
instances of TNPC register, 88
instances of TPC register, 87
instances of TSTATE register, 89
instances of TT register, 90
non-reset trap, 467
may (keyword), **12**
*mem_address_not_aligned* exception, **517**
generated by virtual processor, 258
JMPL instruction, 243
LDTXA, 448, 449, 451
load instruction with partial store ASI and misaligned address, 259
register update policy, 551
RETURN, 318
when recognized, 164
with CASA instruction, 163
with compare instructions, 164
with load instructions, 112–113, 244, 245, 247, 253, 258, 260, 263, 268, 270, 271, 275, 362, 451, 452
with store instructions, 112–113, 258, 333, 335, 337, 347, 350, 356, 358, 360, 451, 452
with swap instructions (deprecated), 365, 367
*mem_address_range* exception, **517**, 551
*mem_real_range* exception, **518**, 551
MEMBAR
#Sync
semantics, 280
instruction
#Sync to allow hardware correction of data, 649
#Sync to flush store buffer, 649
atomic operation ordering, 424
FLUSH instruction, 187, 426
functions, **277**, **423**–**425**
memory ordering, 279
memory synchronization, 120
side-effect accesses, 410
STBAR instruction, 279
write to error steering register, 619
mask encodings
#LoadLoad, 278, 425
#LoadStore, 278, 425
#Lookaside, 278, 426
#MemIssue, 278, 426
#StoreLoad, 278, 425
#StoreStore, 278, 425
#Sync, 278, 426
predefined constants
#LoadLoad, 700
#LoadStore, 700
#Lookaside, 700
#MemIssue, 700
#StoreLoad, 700
#StoreStore, 700
#Sync, 700
MEMBAR
#Lookaside, 420
#StoreLoad, 420
*membar_mask*, **700**
MemIssue predefined constant, 700
memory
access instructions, 27, 111
alignment, 411
atomic operations, 422
atomicity, 680
cached, 408
coherence, 410, 680
coherency unit, 411
data, 426
instruction, 426
location, 408
models, **407**
ordering unit, 411
real, 408
reference instructions, data flow order constraints, 416
synchronization, 279

virtual address, 408
virtual address 0, 428
memory management architecture
(hyperprivileged), **536**
address translation, 537
allocation of partition IDs, 537
separation of real and virtual addresses, 536
Memory Management Unit
definition, **13**
Memory Management Unit, *See* MMU
memory model
mode control, 419
partial store order (PSO), **418**
relaxed memory order (RMO), 279, **418**
sequential consistency, **419**
strong, 419
total store order (TSO), 279, **418**, **420**
weak, 419
memory model (mm) field of PSTATE register, **93**
memory order
pending transactions, 418
program order, 415
memory_model (mm) field of PSTATE register, 419
memory-mapped I/O, 409
metrics
for architectural performance, 459
for implementation performance, 459
*See also* performance monitoring hardware
MMU
accessing registers, 566
bypass, 429, **564**
contexts, **537**
definition, **13**
demap, **585**
context operation, 587
operation syntax, 587
page operation, 586
dTLB Tag Access Register *illustrated*, 569, 570, 571
I/D Tag Access registers, 579, 580, 583, 584
iTLB Tag Access Register *illustrated*, 569, 570, 571
page sizes, 532
SPARC V9 compliance, 565
Synchronous Fault Address registers, 576
MMU_NONZERO_CONTEXTID_TSB_CONFIG_*n*, 545
mode
hyperprivileged, 87, 414
MMU bypass, 429
nonprivileged, 24
privileged, 26, 87, 414
motion estimation, 294
MOVA instruction, 281
MOVCC instruction, 281
MOVcc instructions, **281**
conditionally moving integer register contents, 72
conditions for copying integer register contents, 125
copying a register, 60
encoding of cond field, 663
encoding of opf_cc instruction field, 664
used to avoid branches, 200, 283
MOVCS instruction, 281
move floating-point register if condition is true, **196**
move floating-point register if contents of integer register satisfy condition, **201**
MOVE instruction, 281
move integer register if condition is satisfied instructions, **281**
move integer register if contents of integer register satisfies condition instructions, **285**
move on condition instructions, 22
MOVFA instruction, 282
MOVFE instruction, 282
MOVFG instruction, 282
MOVFGE instruction, 282
MOVFL instruction, 282
MOVFLE instruction, 282
MOVFLG instruction, 282
MOVFN instruction, 282
MOVFNE instruction, 282
MOVFO instruction, 282
MOVFU instruction, 282
MOVFUE instruction, 282
MOVFUG instruction, 282
MOVFUGE instruction, 282
MOVFUL instruction, 282
MOVFULE instruction, 282
MOVG instruction, 281
MOVGE instruction, 281
MOVGU instruction, 281
MOVL instruction, 281
MOVLE instruction, 281
MOVLEU instruction, 281
MOVN instruction, 281
mov*n* synthetic instructions, 704
MOVNE instruction, 281

MOVNEG instruction, 281
MOVPOS instruction, 281
MOVr instructions, 126, **285**, 663
MOVRGEZ instruction, **285**
MOVRGZ instruction, **285**
MOVRLEZ instruction, **285**
MOVRLZ instruction, **285**
MOVRNZ instruction, **285**
MOVRZ instruction, **285**
MOVVC instruction, 281
MOVVS instruction, 281
multiple processors, fatal error, 652
multiple unsigned condition codes, emulating, 126
multiply instructions, 289, **331**, **379**
multiply-add instructions (fused), 192
multiply-subtract instructions (fused), 192
multiprocessor synchronization instructions, 162, 365, 366
multiprocessor system, **13**, 187, 303, 365, 366, 417, 681
MULX instruction, **289**
must (keyword), **13**

## N

N superscript on instruction name, **134**
*N_REG_WINDOWS*, **14**
  integer unit registers, 26, 671
  RESTORE instruction, 311
  SAVE instruction, 319
  value of, 48, 82
NaN (not-a-number)
  conversion to integer, 393
  converting floating-point to integer, 232
  signalling, 61, 182, 234
neg synthetic instructions, 704
negative infinity, 393, 394
negative multiply-add instructions (fused), 192
negative multiply-subtract instructions (fused), 192
nested traps, 22
next program counter register, *See* NPC register
NFO, **13**
noncacheable
  accesses, 408
nonfaulting load, **13**, 415, 565
nonfaulting loads
  behavior, 427
  use by optimizer, 428
non-faulting-only page
  illegal access to, 510
non-faulting-only page, illegal access to
  and TTE.nfo, 540
nonleaf routine, 242
nonprivileged, **13**
  mode, 7, **13**, 24, 26, 63
  software, 75
nonprivileged trap (npt) field of TICK register, 74, 307
*nonresumable_error* exception, **519**
nonstandard floating-point, *See* floating-point status register (FSR) NS field
nontranslating ASI, **13**, 271, 360, **430**
nonvirtual memory, 303
NOP instruction, 153, 174, 177, **290**, 298, 372
normal trap, **13**
normal traps, **467**, 482, 498, 501
NORMALW instruction, **291**
not synthetic instructions, 703
notdata bit, setting, 652
note
  architectural direction, **5**
  compatibility, **5**
  general, **5**
  implementation, **5**
  programming, **5**
NPC (next program counter) register, **75**
  and PSTATE.tct, 92
  control flow alteration, 18
  definition, **13**
  DONE instruction, 165
  instruction execution, 109
  relation to TNPC register, 88
  RETURN instruction, 315
  saving after trap, 33
  state after reset, 631
npt, **13**
nucleus context, 189
Nucleus Context register, 569
nucleus software, **14**
NUMA, **14**, 597
nvm (invalid mask) field of FSR.tem, **67**, 393, 394
*NWIN*, *See N_REG_WINDOWS*
nxm (inexact mask) field of FSR.tem, **67**

## O

octlet, **14**
odd parity, **14**

ofm (overflow mask) field of FSR.tem, **67**
op3 instruction field
  arithmetic instructions, 145, 157, 160, 162, 287, 289, 323, 331, 377, 379
  floating point load instructions, 253, 256, 260, 275
  flush instructions, 187, 191
  jump-and-link instruction, 242
  load instructions, 244, 264, 265, 267, 269
  logical operation instructions, 148, 292, 389
  PREFETCH, 297
  RETURN, 317
opcode
  definition, **14**
  format, 239
opf instruction field
  floating point arithmetic instructions, 172, 184, 210, 231
  floating point compare instructions, 182
  floating point conversion instructions, 232, 234, 237
  floating point instructions, 171
  floating point integer conversion, 186
  floating point move instructions, 195
  floating point negate instructions, 212
opf_cc instruction field
  floating point move instructions, 196
  move instructions, 664
opf_low instruction field, 196
optional, **14**
OR instruction, **292**
ORcc instruction, **292**
ordering MEMBAR instructions, 120
ordering unit, memory, 411
ORN instruction, **292**
ORNcc instruction, **292**
OTHERW instruction, **293**
OTHERWIN (other windows) register, **84**
  FLUSHW instruction, 191
  keeping consistent state, 86
  modified by OTHERW instruction, 293
  partitioned, 86
  range of values, 82, 681
  rd designation for WRPR instruction, 386
  rs1 designation for RDPR instruction, 309
  SAVE instruction, 320
  state after reset, 632
  zeroed by INVALW instruction, 241
  zeroed by NORMALW instruction, 291
OTHERWIN register trap vectors
  fill/spill traps, 523
  handling spill/fill traps, 523
  selecting spill/fill vectors, 523
*out* register #7, 54
*out* registers, 49, 51, 319
overflow
  bits
    (v) in condition fields of CCR, **121**
    accrued (ofa) in aexc field of FSR register, **68**
    current (ofc) in cexc field of FSR register, **68**
  causing spill trap, 522
  tagged add/subtract instructions, 121
overflow mask (ofm) field of FSR.tem, **67**

## P

p (predict) instruction field of branch instructions, 156, 159, 160, 177
P superscript on instruction name, **134**
PA_watchpoint, 356
*PA_watchpoint* exception, 430, **519**
packed-to-planar conversion, 222
packing instructions, *See* FPACK instructions
page fault, 303
page table entry (PTE), *See* translation table entry (TTE)
parity, even, **10**
parity, odd, **14**
park, **14**
parked, **14**
parking CMP core, **608**
partial store instructions, 351, 452
partial store order (PSO) memory model, **418**, 419
Partition ID register
  memory address representation, 536–537
  and TLB entries, 537
partition identifier, **412**, **536**
partitioned
  additions, 219
  subtracts, 224
$P_{ASI}$ superscript on instruction name, **134**
$P_{ASR}$ superscript on instruction name, **134**
PC (program counter) register, **15**, 69, **75**
  after instruction execution, 109
  and PSTATE.tct, 92
  CALL instruction, 161
  changed by NOP instruction, 290

copied by JMPL instruction, 242
saving after trap, 33
set by DONE instruction, 165
set by RETRY instruction, 315
state after reset, 631
Trap Program Counter register, 87
PDIST instruction, **294**
pef field of PSTATE register
and access to GSR, 76
and *fp_disabled* exception, 512
and FPop instructions, 130
branch operations, 175, 177
byte permutation, 155
comparison operations, 180, 183
component distance, 294
data formatting operations, 185, 213, 223
data movement operations, 284
enabling FPU, 75
floating-point operations, 171, 172, 184, 186, 193, 195, 199, 202, 210, 212, 231, 232, 234, 236, 237, 253, 256, 260, 262, 275
integer arithmetic operations, 204, 221, 226
logical operations, 227, 228, 230
memory operations, 251
read operations, 307, 327, 340
special addressing operations, 146, 173, 342, 349, 353, 355, 362, 383
trap control, 477
pef, *See* PSTATE, pef field
performance monitoring hardware
accuracy requirements, 459
classes of data reported, 459
counters and controls, 460
high-level requirements, 457
kinds of user needs, 457
*See also* instruction sampling
physical address, **14**
physical core, **14**
physical processor, **14**
*pic_overflow* exception, **519**
PIL (processor interrupt level) register, **99**
interrupt conditioning, 474
interrupt request level, 478
*interrupt_level_n*, 517
specification of register to read, 309
specification of register to write, 386
state after reset, 631
trap processing control, 477
pipeline, **14**
pipeline draining of CPU, 83, 87
PIPT, **14**
pixel instructions
compare, 179
component distance, **294**, 294
formatting, 213
pixel registers for storing values, 239
planar-to-packed conversion, 222
$P_{npt}$ superscript on instruction name, **134**
pointer registers, implementation, 549
POPC instruction, **295**
POR, **14**
POR (*power_on_reset*), **628**
machine state changes, 630
POR, *See* *power_on_reset* (POR)
positive infinity, 393, 394
power failure, 476, 505
*power_on_reset* (POR)
hard reset when POR pin activated, 628
*power_on_reset* (POR), **519**, **628**
effect on HTSTATE, 104
effect on STICK register fields, 81
effect on TNPC register, 88
effect on TPC, 88
effect on TT register, 90
enabling/disabling virtual processors, 605, 606
full-processor reset, 614
hard reset, 607, 687
machine state changes, 630
and `RED_state`, 467, 469, 501
STRAND_ENABLE_STATUS register, 617
system reset, 614
when initiated, 476
precise ESR, **642**
associated error info for register files, 648
associated error info for SW correction, 648
and SFSR, 642
precise floating-point traps, 310
precise trap, **471**
conditions for, 471
generation of, 638
handling, 638
software actions, 471
vs. disrupting trap, 473
predefined constants
`LoadLoad`, 700
`lookaside`, 700
`MemIssue`, 700
`StoreLoad`, 700

StoreStore, 700
Sync, 700
predict bit, 160
prefetch
for one read, 302
for one write, 303
for several reads, 302
for several writes, 302
page, 303
to nearest unified cache, 303
prefetch data instruction, **297**
PREFETCH instruction, 111, **297**, 298, 565, 676
*prefetch_fcn*, **700**
PREFETCHA instruction, **297**, 676
and invalid ASI or VA, 510
prefetchable, **15**
Primary Context ID 0, 567
Primary Context ID 1, 567
Primary Context register, 568
priority of traps, 478, 496
*priveleged_action* exception
read from TICK register when access disabled, 74
privilege violation, and *DAE_privilege_violation* exception, 510
privileged, **15**
mode, 26, 87, 414
registers, **87**
software, 25, 52, 63, 94, 119, 191, 480, 676
privileged (priv) field of PSTATE register, **96**, **103**, 163, 166, 247, 251, 256, 257, 265, 270, 307, 335, 341, 346, 360, 366, 367, 383, 414, 519
privileged mode, **15**
*privileged_action* exception, **519**
accessing restricted ASIs, 413
read from TICK register when access disabled, 74
register update policy, 551
restricted ASI access attempt, 119, 430, 553
TICK register access attempt, 73
with CASA instruction, 163
with compare instructions, 164
with load alternate instructions, 247, 251, 257, 265, 270, 335, 341, 346, 360, 367
with load instructions, 256
with RDasr instructions, 307
with read instructions, 307
with store instructions, 348
with swap instructions, 367
*privileged_opcode* exception, **519**
DONE instruction, 167
RETRY instruction, 316
SAVED instruction, 321
with DONE instruction, 166, 309, 316, 387
with write instructions, 388
WRPR in nonprivileged mode, 74
processor, **15**
execute unit, 415
issue unit, 415, **415**
privilege-mode transition diagram, 466
reorder unit, 415
self-consistency, **416**
state diagram, 467
processor cluster, *See* processor module
processor consistency, 417, 421
processor interrupt level register, *See* PIL register
processor self-consistency, 416, 420
processor state register, *See* PSTATE register
processor states
error_state, 467, **470**, 497, 498, 499
entering, 505, 506, 507
execute_state, 497, 499
RED_state, 467, 468, 469, 483, 497, 499, 501, 503, 507
processor states, *See* error_state, execute_state, *and* RED_state
program counter register, *See* PC register
program counters, saving, 463
program order, **415**, 416
programming note, **5**
PSO, *See* partial store order (PSO) memory model
PSR register (SPARC V8), 383
PSTATE register
entering privileged execution mode, 463
restored by RETRY instruction, 165, 315
saved after trap, 463
saving after trap, 33
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 631
and TSTATE register, 89
PSTATE register fields
ag
unimplemented, 97
am
CALL instruction, 161
description, **94**
masked/unmasked address, 166, 242, 315, 318
cle

and implicit ASIs, 118
and PSTATE.tle, 93
description, **92**
ie
description, 96, 97
enabling disrupting traps, 474, 639
interrupt conditioning, 474
masking disrupting trap, 484
mm
description, **93**
implementation dependencies, 93, 418, 680
reserved values, 93
pef
and FPRS.fef, 94
description, **94**
*See also* pef field of PSTATE register
priv
access to register-window PR state registers, 87
accessing restricted ASIs, 413
description, **96**
determining mode, 13, 15, 542
when processor in privileged mode, 103
tct
branch instructions, 154, 158, 160, 175, 177
CALL instruction, 161
description, **92**
DONE instruction, 166
JMPL instruction, 242
RETRY instruction, 316, 373
RETURN instruction, 318
tle
and PSTATE.cle, 93
description, **93**
PTE (page table entry), *See* translation table entry (TTE)

## Q

quadword, **15**
alignment, 28, 112, 411
data format, **35**
quiet NaN (not-a-number), 61, 182

## R

R register, **15**
#15, 54
special-purpose, 54
alignment, 268, 270
ra_not_pa field of TSB Config register, 541, 548, 570, 571, **573**, 573
*RA_watchpoint* exception, 488, 494
rational quotient, 377
RA-to-PA translation, **548**
R-A-W, *See* read-after-write memory hazard
rcond instruction field
branch instructions, 159
encoding of, 663
move instructions, 285
rd (rounding), **15**
rd instruction field
arithmetic instructions, 145, 157, 160, 162, 287, 289, 323, 331, 377, 379
floating point arithmetic, 172
floating point arithmetic instructions, 184, 210, 231
floating point conversion instructions, 232, 234, 237
floating point integer conversion, 186
floating point load instructions, 253, 256, 260, 275
floating point move instructions, 195, 196
floating point negate instructions, 212
floating-point instructions, 171
jump-and-link instruction, 242
load instructions, 244, 264, 265, 267, 269
logical operation instructions, 148, 292, 389
move instructions, 283, 285
POPC, 295
RDASI instruction, **69**, 73, **305**
RDasr instruction, **305**
accessing I/O registers, 29
implementation dependencies, 306, 675
reading ASRs, 69
RDCCR instruction, 69, 71, **305**, 305
RDFPRS instruction, 69, 75, 305
RDGSR instruction, 70, 76, 305
RDHPR instruction, 101, 102, 104, 106, **308**
hyperprivileged registers read, 308
RDPC instruction, 69, 305
reading PC register, 75
RDPR instruction, 69, **309**
accessing GL register, 100
accessing non-register-window PR state registers, 87
accessing register-window PR state registers, 82
and register-window PR state registers, 82

effect on TNPC register, 88
effect on TPC register, 88
effect on TSTATE register, 90
effect on TT register, 91
reading privileged registers, 87
reading PSTATE register, 91
reading the TICK register, 73, 74
registers read, 309
RDSOFTINT instruction, 70, 77, 305
RDSTICK instruction, 70, 80, 81, 305
RDSTICK_CMPR instruction, 70, 305
RDTICK instruction, 69, 73, 74, 305
RDTICK_CMPR instruction, 70, 305
RDY instruction, 71
read ancillary state register (RDasr) instructions, **305**
read state register instructions, 31
read-after-write memory hazard, 416
real address, **15**
real ASI, **430**
real memory, 408
Real Range registers, **569**
fields, 570
real-translating ASIs, **430**
recoverable, **16**
RED_state, **16**
catastrophic failure avoidance, 497
description, 466
entering, 469, 503, 680
entry conditions, 467
exiting, 102
MMU behavior, 563
red field of HPSTATE register, 467, 468, 469, 497, 498
reset of TLB, 563
restricted environment, 468
special trap processing, 501
trap processing, 469, 497, 499
trap table, 483
trap vector, 481, 680
RED_state trap, **16**
*RED_state_exception* exception, **520**
reference MMU, 695
*reg*, **696**
*reg_or_imm*, **701**
*reg_plus_imm*, **700**
*regaddr*, **700**
register reference instructions, data flow order constraints, 416
register window, **49**, **50**
register window management instructions, 32
register windows
clean, 84, 86, 127, 509, 522, 523, 524
fill, 52, 86, 127, 128, 312, 313, 321, 512, 522, 523, 524
management of, 24
overlapping, 51–53
spill, 52, 86, 127, 128, 129, 320, 321, 520, 522, 523, 524
registers
*See also* individual register (common) names
accessing MMU registers, 566
address space identifier (ASI), 414
ASI (address space identifier), **73**
chip-level multithreading, *See* CMT
clean windows (CLEANWIN), **84**
clock-tick (TICK), 519
current window pointer (CWP), **83**
error status (ESR), **642**
F (floating point), 391, 478
floating-point, 26
programming, 58
floating-point registers state (FPRS), **75**
floating-point state (FSR), **60**
general status (GSR), **76**
GL (global level), 106
*global*, 22, 26, 48, **50**, 50, 671
global level (GL), **99**
HSTICK_CMPR
and HINTP, 105
HSTICK_CMPR, 107
HTSTATE (hyperprivileged trap state), **103**
HVER (version register), **106**
hyperprivileged, **101**
IER (SPARC V8), 383
*in*, 49, 51, 319
*local*, 49, 51
next program counter (NPC)
and PSTATE.tct, 92
next program counter (NPC), **75**
other windows (OTHERWIN), **84**
*out*, 49, 51, 319
*out* #7, 54
pixel storage registers, 239
processor interrupt level (PIL)
and SOFTINT, 78
and STICK_CMPR, 82
and TICK_CMPR, 80

processor interrupt level (PIL), **99**
program counter (PC)
and PSTATE.tct, 92
program counter (PC), **75**
PSR (SPARC V8), 383
R register #15, 54
renaming mechanism, 416
restorable windows (CANRESTORE), **84**, 84
savable windows (CANSAVE), **83**
scratchpad
hyperprivileged, **454**
privileged, **453**
SOFTINT, 70
SOFTINT_CLR pseudo-register, 70, **79**
SOFTINT_SET pseudo-register, 70, **78**
STICK, **80**
STICK_CMPR
and HINTP, 105
ASR summary, 70
int_dis field, 78, **82**
stick_cmpr field, **82**
and system software trapping, 81
TBR (SPARC V8), 383
TICK, **73**
TICK_CMPR
int_dis field, 78, **80**
tick_cmpr field, **80**
TICK_CMPR, 70, **79**
TL (trap level), 106
trap base address (TBA), **91**
trap base address, *See* registers: TBA
trap level (TL), **97**
trap level, *See* registers: TL
trap next program counter (TNPC), **88**
trap next program counter, *See* registers: TNPC
trap program counter (TPC), **87**
trap program counter, *See* registers: TPC
trap state (TSTATE), **89**
trap state, *See* registers: TSTATE
trap type (TT), **90**, **482**
trap type, *See* registers: TT
VA_WATCHPOINT, 516, 521
visible to software in privileged mode, 87–101
WIM (SPARC V8), 383
window state (WSTATE), **85**
window state, *See* registers: WSTATE
Y (32-bit multiply/divide), **70**
relaxed memory order (RMO) memory model, 279, **418**
renaming mechanism, register, 416
reorder unit, 415
reordering instruction, 416
reserved, **16**
fields in instructions, 144
register field, 48
reset
after fatal error, 640
*externally_initiated_reset* (XIR), 466, 467, 469, 476, 482, 497, 501, **505**, 505, **511**, 615, **629**
*power_on_reset* (POR)
enabling/disabling virtual processors, 605, 606
machine state changes, 630
STRAND_ENABLE_STATUS register, 617
*power_on_reset* (POR), 467, 469, 476, 501, **519**, 614, **628**
power-on, 74
processing, 468
request, 519, 520
reset trap, 74, 90, 473, 476
*software_initiated_reset* (SIR), 466, 467, 469, 476, 482, 497, **506**, **520**, 614, **630**
trap, 674
trap vector address, *See* RSTVaddr
*warm_reset* (WMR)
and STRAND_ENABLE register, 607
enabling/disabling virtual processors, 605, 606
machine state changes, 630
*warm_reset* (WMR), **629**
*watchdog* (WDR), 614
*watchdog_reset* (POR), 469
*watchdog_reset* (WDR)
and *guest_watchdog*, 465
*watchdog_reset* (WDR), 501, 505, **521**, 614, **630**
XIR, 615
reset trap, **16**
Reset, Error, and Debug state, *See* `RED_state`
restartable deferred trap, **472**
restorable windows register, *See* CANRESTORE register
RESTORE instruction, 52, **311**–**312**
actions, 127
and current window, 54
decrementing CWP register, 51
fill trap, 512, 522
followed by SAVE instruction, 52
managing register windows, 32

operation, 311
performance trade-off, 311, 319
and restorable windows (CANRESTORE) register, 84
restoring register window, 311
role in register state partitioning, 86
`restore` synthetic instruction, 703
RESTORED instruction, 128, **313**
creating inconsistent window state, 313
fill handler, 312
fill trap handler, 128, 524
register window management, 32
restricted, **16**
restricted address space identifier, 119
restricted ASI, 413, 429, 553
*resumable_error* exception, **520**
`ret/retl` synthetic instructions, 702
RETRY instruction, **315**
and restartable deferred traps, 473
effect on HTSTATE, 104
effect on TNPC register, 88
effect on TPC register, 88
effect on TSTATE register, 90
executed in `RED_state`, 468
generating *illegal_instruction* exception, 513
modifying CCR.XCC, 72
reexecuting trapped instruction, 524
restoring gl value in GL, 101
return from trap, 463
returning to instruction after trap, 475
target address, return from privileged traps, 31
RETURN instruction, **317–318**
computing target address, 30
fill trap, 512
*mem_address_not_aligned* exception, 517
operation, 317
reexecuting trapped instruction, 317
RETURN vs. RESTORE instructions, 317
RMO, **17**
RMO, *See* relaxed memory order (RMO) memory model
rounding
for floating-point results, 61
in signed division, 323
rounding direction (rd) field of FSR register, 172, 184, 210, 231, 232, 234, 236, 237
routine, nonleaf, 242
rs1 instruction field
arithmetic instructions, 145, 157, 160, 162, 287, 289, 323, 331, 377, 379
branch instructions, 159
floating point arithmetic instructions, 172, 184, 210
floating point compare instructions, 182
floating point load instructions, 253, 256, 260, 275
flush memory instruction, 187
jump-and-link instruction, 242
load instructions, 244, 264, 265, 267, 269
logical operation instructions, 148, 292, 389
move instructions, 285
PREFETCH, 297
RETURN, 317
rs2 instruction field
arithmetic instructions, 145, 157, 160, 162, 287, 289, 292, 323, 331, 377, 379
floating point arithmetic instructions, 172, 184, 210, 231
floating point compare instructions, 182
floating point conversion instructions, 232, 234, 237
floating point instructions, 171
floating point integer conversion, 186
floating point load instructions, 253, 256, 260, 275
floating point move instructions, 195, 196
floating point negate instructions, 212
flush memory instruction, 187
jump-and-link instruction, 242
load instructions, 244, 267, 269
logical operation instructions, 148, 389
move instructions, 283, 285
POPC, 295
PREFETCH, 297
*RSTVADDR*, 469, 481, 482, 483, 503, 504, 505, 506, 507, 508, 631, 680
RTO, **17**
RTS, **17**

## S

savable windows register, *See* CANSAVE register
SAVE instruction, 51, **319**
actions, 127
after RESTORE instruction, 317
*clean_window* exception, 509, 523
and current window, 54
decrementing CWP register, 51

effect on privileged state, 320
leaf procedure, 242
and *local*/*out* registers of register window, 52
managing register windows, 32
no clean window available, 84
number of usable windows, 84
operation, 319
performance trade-off, 319
role in register state partitioning, 86
and savable windows (CANSAVE) register, 83
spill trap, 520, 522, 524
`save` synthetic instruction, 703
SAVED instruction, 128, **321**
creating inconsistent window state, 321
register window management, 32
spill handler, 320, 321
spill trap handler, 128, 524
scaling of the coefficient, 205
scratchpad registers
hyperprivileged, **454**
privileged, **453**
state after reset, 633
SDIV instruction, 71, **323**
SDIVcc instruction, 71, **323**
SDIVX instruction, **289**
Secondary Context ID 0, 567
Secondary Context ID 1, 567
Secondary Context register, 569
self-consistency, processor, 416
self-modifying code, 187, 189, 424
sequencing MEMBAR instructions, 120
sequential consistency, 410, 418, 419
sequential consistency memory model, **419**
service processor, **17**
SETHI instruction, **121**, **325**
creating 32-bit constant in R register, 29
and NOP instruction, 290
with rd = 0, 325
`set`*n* synthetic instructions, 703
SFSR register
error handling, 644
shall (keyword), **17**
shared memory, 407
shift count encodings, 329
shift instructions, 30
shift instructions, 120, **329**
short floating-point load and store instructions, 452
short floating-point load instructions, 262
short floating-point store instructions, 355
should (keyword), **17**
SHUTDOWN instruction, 326
SIAM instruction, 327
side effect
accesses, 409
definition, **17**
I/O locations, 408
instruction prefetching, 410
real memory storage, 408
visible, 409
side-effect page, illegal access to, 510
signalling NaN (not-a-number), 61, 234
signed integer data type, 35
`signx` synthetic instructions, 703
SIMD, **17**
instruction data formats, 43–45
simm10 instruction field
move instructions, 285
simm11 instruction field
move instructions, 283
simm13 instruction field
floating point
load instructions, 253, 275
simm13 instruction field
arithmetic instructions, 287, 289, 292, 323, 331, 377, 379
floating point load instructions, 256, 260
flush memory instruction, 187
jump-and-link instruction, 242
load instructions, 244, 264, 265, 267, 269
logical operation instructions, 148, 389
POPC, 295
PREFETCH, 297
RETURN, 317
single instruction/multiple data, *See* SIMD
SIR, **17**
SIR (*software_initiated_reset*), **630**
SIR instruction, **328**
affecting virtual processor, 630
causing *software_initiated_reset* exception, 476, 520
and trap priority, 496
use by supervisor software, 506
SIR, *See* *software_initiated_reset (SIR)*
SLL instruction, 329
SLLX instruction, 329
SMUL instruction, 70, **331**
SMULcc instruction, 70, **331**
snooping, **17**

SOFTINT register, 70, **77**
clearing, 527
clearing of selected bits, 79
communication from nucleus code to kernel code, 526
scheduling interrupt vectors, 525, 526
setting, 526
state after reset, 632
SOFTINT register fields
int_level, **78**
sm (stick_int), **78**
tm (tick_int), **78**, 80
SOFTINT_CLR pseudo-register, 70, **79**
SOFTINT_SET pseudo-register, 70, **78**, 79
software
nucleus, **14**
software tablewalk, 589
software translation table, 535
software trap, 372, 480, 483
software trap number (SWTN), **372**
software, nonprivileged, 75
*software_initiated_reset* (SIR), **506**, **520**, **630**
entering `error_state`, 466
entering `RED_state`, 467
and *MAXTL*, 469
per-strand reset, 614
`RED_state` trap processing, 501
`RED_state` trap vector, 482
SIR instruction, **328**, 476
and virtual processor, 630
virtual processor trap processing, 497
when TL = *MAXTL*, 497
*software_trap_number*, **701**
source operands, 219, 224
SPA
`ASI_TWIN_DW_NUCLEUS`, 455
SPARC V8 compatibility
LD, LDUW instructions, 244
operations to I/O locations, 410
read state register instructions, 306
STA instruction renamed, 337
STBAR instruction, 279
STD instruction, 358
STDA instruction, 360
tagged subtract instructions, 376
UNIMP instruction renamed, 238
*window_overflow* exception superseded, 512
write state register instructions, 383
SPARC V9
compliance, 14, 565
features, 22
SPARC V9 Application Binary Interface (ABI), 24
special trap, renamed, 467
special traps, **467**, 483
speculative load, **17**
spill register window, 520
FLUSH instruction, 129
overflow/underflow, 52
RESTORE instruction, 127
SAVE instruction, 86, 127, 319, 522
SAVED instruction, 128, 321, 524
selection of, 523
trap handling, 524
trap vectors, 320, 524
window state, 86
*spill_n_normal* exception, **320**, **520**
and FLUSHW instruction, 191
*spill_n_other* exception, 320, **520**
and FLUSHW instruction, 191
SRA instruction, 329
SRAX instruction, 329
SRL instruction, 329
SRLX instruction, 329
stack frame, 319
state registers (ASRs), 69–82
STB instruction, 333
STBA instruction, 335
STBAR instruction, 306, 382, 416, 424
STBLOCKF instruction, **338**, 451
STDF instruction, 112, **342**, 520
*STDF_mem_address_not_aligned* exception, **520**
and store instructions, 343, 347
STDF/STDFA instruction, 112
STDFA instruction, **345**
alignment, 112
ASIs for fp store operations, 452
causing *DAE_invalid_ASI* exception, 452
causing *mem_address_not_aligned* or *illegal_instruction* exception, 452
causing *STDF_mem_address_not_aligned* exception, 112, 520
for block load operations, 451
for partial store operations, 452
used with ASIs, 451
STF instruction, **342**
STFA instruction, **345**
STFSR instruction, 60, 62, 63, 513
STH instruction, 333

STHA instruction, 335
STICK register, 70, **73**, **80**
and *hstick_match* exception, 512
counter field, **80**, **81**
fields after power-on reset trap, 81
npt field, **73**, **80**
RDSTICK instruction, 305
state after reset, 632
while virtual processor is parked, 608
STICK_CMPR register, 70, **81**
and HINTP, 105
int_dis field, 78, **82**
RDSTICK_CMPR instruction, 305
state after reset, 633
stick_cmpr field, **82**
store
block, *See* block store instructions
partial, *See* partial store instructions
short floating-point, *See* short floating-point store instructions
store buffer
merging, 409
store floating-point into alternate space instructions, **345**
store instructions, 17, 111, 511
*store_error* exception, **520**
StoreLoad MEMBAR relationship, 278, 425
StoreLoad predefined constant, 700
stores to alternate space, 29, 73, 117
StoreStore MEMBAR relationship, 278, 425
StoreStore predefined constant, 700
STPARTIALF instruction, 351
STPARTIALF instruction, *DAE_nc_page* exception, 510
STQF instruction, 113, **342**, 522
*STQF_mem_address_not_aligned* exception, **522**
STQF/STQFA instruction, 113
STQFA instruction, 113, **345**
strand, **18**
STRAND_AVAILABLE register, 601, **605**, 605, 607, 608
state after reset, 634
STRAND_ENABLE register, **606**
state after reset, 634
STRAND_ENABLE_STATUS register, **606**
state after reset, 634
STRAND_ID register, **601**
state after reset, 635
STRAND_INTR_ID register, 530, **602**, 624
state after reset, 635
STRAND_RUNNING register, 608, **609**
simultaneous updates, 610
state after reset, 634
STRAND_RUNNING_RW pseudo-register, **609**, **610**
STRAND_RUNNING_STATUS register, 608, **612**
Parked or Unparked status, 613
state after reset, 634
STRAND_RUNNING_W1C pseudo-register, **609**, **610**
STRAND_RUNNING_W1S pseudo-register, **609**, **610**
strong consistency memory model, 419
strong ordering, 419
Strong Sequential Order, 420
STSHORTF instruction, 355
STTW instruction, 54, 112
STTW instruction (deprecated), **357**
STTWA instruction, 54, 112
STTWA instruction (deprecated), **359**
stuck-at fault, 637
STW instruction, 333
STWA instruction, 335
STX instruction, 333
STXA instruction, **335**
accessing CMP-specific registers, 598
accessing nontranslating ASIs, 360
initiating demap operation, 586
*mem_address_not_aligned* exception, 335
referencing internal ASIs, 420
writing to a CMP register, 600
STXFSR instruction, 60, 62, 63, **362**, 513
SUB instruction, **364**, 364
SUBC instruction, **364**, 364
SUBcc instruction, 120, **364**, 364
SUBCcc instruction, **364**, 364
subnormal number, **18**
subtract instructions, **364**
superscalar, **18**
supervisor software
accessing special protected registers, 28
definition, **18**
forcing processing into RED_state, 497
use of SIR trap, 506
suspend, **18**
suspended, **18**
*sw_recoverable_error* exception, **520**, 521
*sw_recoverable_error* trap, **639**
clearing in L2 cache, 652
disrupting traps, 650
handler routine, 640
reporting, 643

SWAP instruction, 27, **365**
accessing doubleword simultaneously with other instructions, 366
and *DAE_nc_page* exception, 510
hardware primitive for mutual exclusion, 422, 423
identification of R register to be exchanged, 111
in multiprocessor system, 264, 265
memory accessing, 365
ordering by MEMBAR, 424
swap R register
bit contents, 162
with alternate space memory instructions, **366**
with memory instructions, **365**
SWAPA instruction, **366**
accessing doubleword simultaneously with other instructions, 366
alternate space addressing, 29
and *DAE_nc_page* exception, 510
hardware primitive for mutual exclusion, 422
in multiprocessor system, 264, 265
ordering by MEMBAR, 424
SWTN (software trap number), **372**
`Sync` predefined constant, 700
synchonization, **280**
synchronization, **18**
Synchronous Fault Address register (SFAR), **9**, **575**
Synchronous Fault Address Register (SFAR),, *See Data Synchronous Fault Address Register (D-SFAR)*
synthetic instructions
mapping to SPARC V9 instructions, **702–704**
for assembly language programmers, 702
mapping
`bclrg`, 704
`bset`, 704
`btog`, 704
`btst`, 704
`call`, 702
`cas`*n*, 704
`clr`*n*, 704
`cmp`, 702
`dec`, 704
`deccc`, 704
`inc`, 704
`inccc`, 704
`iprefetch`, 702
`jmp`, 702
`mov`*n*, 704
`neg`, 704
`not`, 703
`restore`, 703
`ret/ret1`, 702
`save`, 703
`set`*n*, 703
`signx`, 703
`tst`, 702
vs. pseudo ops, 702
system clock-tick register (STICK), **80**
system software, 520
accessing memory space by server program, 413
ASIs allowing access to memory space, 414
FLUSH instruction, 189, 427
processing exceptions, 413
trap types from which software must recover, 63
System Tick Compare register, *See* STICK_CMPR register
System Tick register, *See* STICK register

## T

TA instruction, 371, 663
Tablewalk Pending Control register, 589
Tablewalk Pending Status register, 589
TADDcc instruction, 121, 368
TADDccTV instruction, 121, 520
Tag Access registers, **579**
effect of loads and stores, 577
and formation of pointer address, 549
Lower Tag Access, 580
updating, 579
Upper Tag Access register, 579
tag overflow, 121
*tag_overflow* exception, 121, 368, 369, 370, 374, 376
*tag_overflow* exception (deprecated), **520**
tagged arithmetic, 121
tagged arithmetic instructions, 30
tagged word data format, **35**
tagged words, 35
TBA (trap base address) register, **91**, 465
establishing table address, 32, 463
initialization, 479
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 631
trap behavior, 19
TBR register (SPARC V8), 383
TCC instruction, 371

Tcc instructions, **371**
at TL > 0, 480
causing trap, 463
causing trap to privileged trap handler, 483
CCR register bits, 72
generating *htrap_instruction* exception, 512
generating *illegal_instruction* exception, 513
generating *trap_instruction* exception, 520
opcode maps, 659, 663, 664
programming uses, 373
trap table space, 33
vector through trap table, 463
TCS instruction, 371, 663
TE instruction, 371, 663
terminating deferred trap, 650
termination deferred trap, **471**
test-and-set instruction, 423
TG instruction, 371, 663
TGE instruction, 371, 663
TGU instruction, 371, 663
thread, **18**
TICK register, 69
controlling access to timing information, 74
counter field, **73**, 677, 693
fields after power-on reset trap, 74
inaccuracies between two readings of, 677, 693
npt field, 74
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 632
while virtual processor is parked, 608, 691
TICK_CMPR register, 70, **79**
int_dis field, 78, **80**
state after reset, 632
tick_cmpr field, **80**
timer registers, *See* TICK register *and* STICK register
timing of instructions, 144
tininess (floating-point), 68
TL (trap level) register, **97**, 465
affect on privilege level to which a trap is delivered, 479
and implicit ASIs, 118
displacement in trap table, 463
executing RESTORED instruction, 313
executing SAVED instruction, 321
indexing for WRHPR instruction, 384
indexing for WRPR instruction, 386
indexing hyperprivileged register after RDHPR, 308
indexing privileged register after RDPR, 309
setting register value after WRHPR, 384
setting register value after WRPR, 386
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 632
and TBA register, 479
and TPC register, 87
and TSTATE register, 89, 103
and TT register, 90
use in calculating privileged trap vector address, 479
and VER.maxtl, 106
and WSTATE register, 85
TL instruction, 371, 663
TLB, **18**
and 3-dimensional arrays, 152
bypass operation, 591
Data Access register, 581
Data In register, 549
definition, **18**
demap operation, 591
error checking, 647
errors, **647–648**
hardware, 590
hit, **18**
Lower Tag Access register fields, 580
miss, **18**
handler, 535, 546
MMU behavior, 535
reloading TLB, 535, 543
miss/refill sequence, 548
operations, 590
page loading, 578
partition IDs, 537
read operation, 591
replacement attempts, 578
software-corrected errors, 648
specialized miss handler code, 565
Tag Access registers, 578, 579, 580, 583, 584
translation operation, 590
v field, 578
write operation, 591
TLB Upper Tag Access register fields, 579
TLE instruction, 371, 663
TLEU instruction, 371, 663
TN instruction, 371, 663
TNE instruction, 371, 663
TNEG instruction, 371, 663

TNPC (trap next program counter) register, **88**
after *async_data_error* disrupting trap, 520
saving NPC, 471
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 632
TNPC (trap-saved next program counter) register, **18**
total order, **418**
total store order (TSO) memory model, 93, 279, 409, **418**, 419, **420**
TPC (trap program counter) register, **18**, **87**
address of trapping instruction, 310
after *async_data_error* disrupting trap, 520
number of instances, 87
specification for RDPR instructions, 309
specification for WRPR instruction, 386
state after reset, 632
TPOS instruction, 371, 663
translating ASI, **430**
Translation Lookaside Buffer, *See* TLB
Translation Table Entry, *See* TTE
trap
*See also* exceptions *and* traps
noncacheable accesses, 410
when taken, 18
trap enable mask (tem) field of FSR register, 477, 478, 673
trap handler
for global registers, 101
hyperprivileged mode, 482
privileged mode, 482
regular/nonfaulting loads, 13
returning from, 165, 315
user, 64, 393
trap level register, *See* TL register
trap next program counter register, *See* TNPC register
Trap on Control Transfer
and instructions
Bicc, 154
BPcc, 158
BPr, 160
CALL, 161
DONE, 166, 316, 373
FBfcc, 175, 177
JMPL, 242
tct field of PSTATE register, **92**
trap on integer condition codes instructions, **371**
Trap Overflow Enable bit, **460**
trap program counter register, *See* TPC register
trap state register, *See* TSTATE register
trap type (TT) register, **482**
trap type register, *See* TT register
*trap_instruction* (ISA) exception, 372, 373, **520**
*trap_level_zero* exception, 103, **520**
state after reset, 631
with WRHPR instructions, 385
with write instructions, 388
trap_little_endian (tle) field of PSTATE register, **92**
traps, **19**
*See also* exceptions *and* individual trap names
categories
deferred, 471, **471**, 473
disrupting, 471, **473**, 476
precise, 471, **471**, 473
priority, 478, 496
reset, 90, 471, 473, **476**, 476, 497, 674
restartable
implementation dependency, 473
restartable deferred, **472**
termination deferred, **471**
caused by undefined feature/behavior, 19
causes, **33**, 33
deferred, 638
definition, 32, 464
disrupting, 639
*ECC_error*, 513
hardware, 483
hardware stack, 22
level specification, 97
model stipulations, **477**
nested, 22
normal, **13**, **467**, 482, 498, 501
precise, **638**
processing, 497
software, 372, 480, 483
*software_initiated_reset* (SIR), 501
special, **467**, 483
stack, 499
vector address, specifying, 91, 105
vector, `RED_state`, 481
TSB, **19**, **543**
cacheability, 544
caching, 544
configuration, 545
demap operation, 586
Direct Pointer registers, 573
I/D Translation Storage Buffer register, **572**
indexing support, 544

miss handler, 549
organization, 544
pointer generation for TTE, 575
Pointer register, 574
ra_not_pa field, 548
range checking, 548
TSB Config
e field, 573
page_size field, **573**
ra_not_pa field, 541, 548, 570, 571, **573**, 573
tsb_base field, **573**
tsb_size field, **573**, 575
use_cid<0|1> fields, 573
TSO, **19**
TSO, *See* total store order (TSO) memory model
tst synthetic instruction, 702
TSTATE (trap state) register, **89**
DONE instruction, 165, 315
registers saved after trap, 33
restoring GL value, 101
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 632
tstate, *See* trap state (TSTATE) register
TSUBcc instruction, 121, 374
TSUBccTV instruction, 121, 520
TT (trap type) register, **90**
and privileged trap vector address, 479, 480
reserved values, 674
specification for RDPR instruction, 309
specification for WRPR instruction, 386
state after reset, 631
and Tcc instructions, 373
transferring trap control, 482
trap type recorded after *RED_state_exception*, 520
window spill/fill exceptions, 85
WRHPR instruction, 384
WRPR instruction, 386
TTE, **19**
context ID field, **540**, 546
cp (cacheability) field, 408
cp field, 510, 542, **542**
cv field, 542, **542**
e field, 409, 427, 510, **542**
ie field, **541**
indexing support, 544
nfo field, 427, 510, **540**, 542
p field, 510, **542**
privileged code numbers, 546
ra field, 548
size field, **543**
soft2 field, **540**
SPARC V8 equivalence, 539
taddr field, **541**, 581
v field, **540**
va_tag field, **540**
w field, **543**
TVC instruction, 371, 663
TVS instruction, 371, 663
typewriter font, in assembly language syntax, 695


## U


UDIV instruction, 71, **377**
UDIVcc instruction, 71, **377**
UDIVX instruction, **289**
ufm (underflow mask) field of FSR.tem, **67**
UltraSPARC, previous ASIs
ASI_NUCLEUS_QUAD_LDD (deprecated), 455
ASI_NUCLEUS_QUAD_LDD_L (deprecated), 455
ASI_NUCLEUS_QUAD_LDD_LITTLE (deprecated), 455
ASI_PHY_BYPASS_EC_WITH_EBIT_L, 455
ASI_PHYS_BYPASS_EC_WITH_EBIT, 455
ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE, 455
ASI_PHYS_USE_EC, 455
ASI_PHYS_USE_EC_L, 455
ASI_PHYS_USE_EC_LITTLE, 455
ASI_QUAD_LDD_L (deprecated), 455
ASI_QUAD_LDD_LITTLE (deprecated), 455
ASI_QUAD_LDD_PHYS (deprecated), 455
UMMU demap, 586
UMMU TLB Tag Access register, 578
UMMU, ASIs, 579
UMMU_TLB_DATA_ACCESS, 581
UMMU_TLB_DATA_IN, 580
UMMU_TLB_LOWER_TAG_READ, 582
UMMU_TLB_TAG_READ, 582
UMMU_TLB_TAG_TARGET, 585
UMMU_TLB_UPPER_TAG_READ, 582
UMUL instruction, 70
UMUL instruction (deprecated), **379**
UMULcc instruction, 70
UMULcc instruction **(deprecated)**, 379
unassigned, **19**
unconditional branches, 153, 157, 174, 177

uncorrectable, **19**
undefined, **19**
underflow
  bits of FSR register
    accrued (ufa) bit of aexc field, **68**, 393
    current (ufc) bit of cexc, **68**
    current (ufc) bit of cexc field, 393
    mask (ufm) bit of FSR.tem, **68**
    mask (ufm) bit of tem field, 393
  detection, 52
  occurrence, 522
underflow mask (ufm) field of FSR.tem, **67**
unfinished_FPop floating-point trap type, **64**, 172, 184, 211, 231, 235, 236, 392
  handling, 69
  in normal computation, 63
  results after recovery, 64
UNIMP instruction (SPARC V8), 238
unimplemented, **19**
unimplemented_FPop floating-point trap type, 392
  handling, 69
*unimplemented_LDTW* exception, 268, **521**
*unimplemented_STTW* exception, 358, **521**
uniprocessor system, **19**
unpark, **19**
unparking CMP core, **608**
unrecoverable, **20**
unrestricted, **20**
unrestricted ASI, 429
unsigned integer data type, 35
*unsupported_page_size* exception, 578
user application program, **20**
user trap handler, 64, 393

## V

VA, **20**
*VA_watchpoint* exception, **521**
VA_WATCHPOINT register, 516, 521
value clipping, *See* FPACK instructions
value semantics of input/output (I/O) locations, 408
VER (version) register (SPARC V9), 106
virtual
  address, 408
  address 0, 428
virtual address, **20**
virtual core, **20**
virtual memory, 303
virtual-translating ASI, **430**
VIS, **20**
VIS instructions
  encoding, 665, 666
  implicitly referencing GSR register, 76
Visual Instruction Set, *See* VIS instructions

## W

W-A-R, *See* write-after-read memory hazard
*warm_reset* (WMR), **629**
  and STRAND_ENABLE register, 607
  enabling/disabling virtual processors, 605, 606
  machine state changes, 630
*watchdog_reset* (POR)
  and RED_state, 469
*watchdog_reset* (WDR), **521**
  entering error_state, 470
  exiting error_state, 630, 685
  full-processor reset, 614
  invoking RED_state trap processing, 501
  per-strand reset, 614
  and XIR traps, 505
*watchdog_reset* (WDR), and *guest_watchdog*, 465
*watchdog_reset* (WMR), 630
watchpoint comparator, 95
watchpoints
  trap, 551
W-A-W, *See* write-after-write memory hazard
WDR, **20**
WDR (*watchdog_reset*), **630**
WDR, *See watchdog_reset* (WDR)
WIM register (SPARC V8), 383
window fill exception, *See also fill_n_normal* exception
window fill trap handler, 32
window overflow, 52, 522
window spill exception, *See also spill_n_normal* exception
window spill trap handler, 32
window state register, *See* WSTATE register
window underflow, 522
window, clean, 319
*window_fill* exception, 85, 127, 483
  RETURN, 317
*window_spill* exception, 85, 483
WMR (*warm_reset*), **629**
  machine state changes, 630
word, **20**

alignment, 28, **112**, 411
data format, **35**
WRASI instruction, 69, 73, 381
WRasr instruction, **381**
accessing I/O registers, 29
attempt to write to ASR 5 (PC), 75
cannot write to PC register, 75
implementation dependencies, 675
writing ASRs, 69
WRCCR instruction, 69, 71, 72, 381
WRFPRS instruction, 69, 75, 381
WRGSR instruction, 70, 76, 381
WRHPR instruction, 101, 102, 104, **384**
WRIER instruction (SPARC V8), 383
write ancillary state register (WRasr) instructions, **381**
write ancillary state register instructions, *See* WRasr instruction
write hyperprivileged register instruction, **384**
write privileged register instruction, **386**
write-after-read memory hazard, 416
write-after-write memory hazard, 416
WRPR instruction, 468
accessing non-register-window PR state registers, 87
accessing register-window PR state registers, 82
and register-window PR state registers, 82
effect on TNPC register, 88
effect on TPC register, 88
effect on TSTATE register, 90
effect on TT register, 91
writing the TICK register, 74
writing to GL register, 100
writing to PSTATE register, 91
writing to TICK register, 73, 74
WRPSR instruction (SPARC V8), 383
WRSOFTINT instruction, 70, 77, 381
WRSOFTINT_CLR instruction, 70, 77, 79, 381, 527
WRSOFTINT_SET instruction, 70, 77, 78, 381, 526
WRSTICK instruction, 70, 80, 381
WRSTICK_CMPR instruction, 70, 381
WRTBR instruction (SPARC V8), 383
WRTICK_CMP instruction, 70, 381
WRWIM instruction (SPARC V8), 383
WRY instruction, 69, 71, 381
WSTATE (window state) register
description, 85
and fill/spill exceptions, 523
normal field, 523
other field, 523
overview, 82
reading with RDPR instruction, 309
spill exception, 191
spill trap, 320
state after reset, 632
writing with WRPR instruction, 386

## X

XIR, **20**
XIR (*externally_initiated_reset*), **629**
XIR reset, 615
XIR, *See* *externally_initiated_reset* (XIR)
XIR_STEERING register, **616**
state after reset, 634
XNOR instruction, **389**
XNORcc instruction, **389**
XOR instruction, **389**
XORcc instruction, **389**

## Y

Y register, 69, **70**
after multiplication completed, 287
content after divide operation, 323, 377
divide operation, 323, 377
multiplication, 287
state after reset, 631
unsigned multiply results, 331, 379
WRY instruction, 382
Y register (deprecated), **70**

## Z

zero virtual address, 428