UltraSPARC Architecture 2007

One Architecture
... Multiple Innovative Implementations

Draft D0.9.3b, 20 Oct 2009

Privilege Levels: Privileged and Nonprivileged

Distribution: Public
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>1</td>
<td>Document Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Navigating UltraSPARC Architecture 2007</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Fonts and Notational Conventions</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Implementation Dependencies</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Notation for Numbers</td>
<td>3</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Informational Notes</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Reporting Errors in this Specification</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Definitions</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Architecture Overview</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>The UltraSPARC Architecture 2007</td>
<td>13</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Features</td>
<td>13</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Attributes</td>
<td>14</td>
</tr>
<tr>
<td>3.1.2.1</td>
<td>Design Goals</td>
<td>15</td>
</tr>
<tr>
<td>3.1.2.2</td>
<td>Register Windows</td>
<td>15</td>
</tr>
<tr>
<td>3.1.3</td>
<td>System Components</td>
<td>15</td>
</tr>
<tr>
<td>3.1.3.1</td>
<td>Binary Compatibility</td>
<td>15</td>
</tr>
<tr>
<td>3.1.3.2</td>
<td>UltraSPARC Architecture 2007 MMU</td>
<td>15</td>
</tr>
<tr>
<td>3.1.3.3</td>
<td>Privileged Software</td>
<td>15</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Architectural Definition</td>
<td>16</td>
</tr>
<tr>
<td>3.1.5</td>
<td>UltraSPARC Architecture 2007 Compliance with SPARC V9 Architecture</td>
<td>16</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Implementation Compliance with UltraSPARC Architecture 2007</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Processor Architecture</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Integer Unit (IU)</td>
<td>16</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Floating-Point Unit (FPU)</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Instructions</td>
<td>17</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Memory Access</td>
<td>17</td>
</tr>
<tr>
<td>3.3.1.1</td>
<td>Memory Alignment Restrictions</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1.2</td>
<td>Addressing Conventions</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1.3</td>
<td>Addressing Range</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1.4</td>
<td>Load/Store Alternate</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1.5</td>
<td>Separate Instruction and Data Memories</td>
<td>19</td>
</tr>
<tr>
<td>3.3.1.6</td>
<td>Input/Output (I/O)</td>
<td>19</td>
</tr>
<tr>
<td>3.3.1.7</td>
<td>Memory Synchronization</td>
<td>19</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Integer Arithmetic / Logical / Shift Instructions</td>
<td>19</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Control Transfer</td>
<td>20</td>
</tr>
<tr>
<td>3.3.4</td>
<td>State Register Access</td>
<td>20</td>
</tr>
<tr>
<td>3.3.4.1</td>
<td>Ancillary State Registers</td>
<td>20</td>
</tr>
<tr>
<td>3.3.4.2</td>
<td>PR State Registers</td>
<td>20</td>
</tr>
</tbody>
</table>
5.5.8 SOFTINT$^P$ Register (ASRs 20, 21, 22) ............................................. 54
  5.5.8.1 SOFTINT_SET$^P$ Pseudo-Register (ASR 20) ........................ 55
  5.5.8.2 SOFTINT_CLR$^P$ Pseudo-Register (ASR 21) ......................... 56
5.5.9 Tick Compare (TICK_CMPR$^P$) Register (ASR 23) ......................... 56
5.5.10 System Tick (STICK$^P$) Register (ASR 24) ................................. 57
5.5.11 System Tick Compare (STICK_CMPR$^P$) Register (ASR 25) .......... 57

5.6 Register-Window PR State Registers ............................................. 58
  5.6.1 Current Window Pointer (CWP$^P$) Register (PR 9) ....................... 59
  5.6.2 Savable Windows (CANSAVE$^P$) Register (PR 10) ....................... 59
  5.6.3 Restorable Windows (CANRESTORE$^P$) Register (PR 11) ............. 59
  5.6.4 Clean Windows (CLEANWIN$^P$) Register (PR 12) ..................... 59
  5.6.5 Other Windows (OTHERWIN$^P$) Register (PR 13) ....................... 60
  5.6.6 Window State (WSTATE$^P$) Register (PR 14) .......................... 60
  5.6.7 Register Window Management ............................................... 60
    5.6.7.1 Register Window State Definition ..................................... 60
    5.6.7.2 Register Window Traps ............................................... 61

5.7 Non-Register-Window PR State Registers ..................................... 61
  5.7.1 Trap Program Counter (TPC$^P$) Register (PR 0) ......................... 61
  5.7.2 Trap Next PC (TNPC$^P$) Register (PR 1) .................................. 62
  5.7.3 Trap State (TSTATE$^P$) Register (PR 2) .................................. 63
  5.7.4 Trap Type (TT$^P$) Register (PR 3) ....................................... 64
  5.7.5 Trap Base Address (TBA$^P$) Register (PR 5) ............................ 64
  5.7.6 Processor State (PSTATE$^P$) Register (PR 6) ............................ 64
  5.7.7 Trap Level Register (TL$^P$) (PR 7) ....................................... 68
  5.7.8 Processor Interrupt Level (PIL$^P$) Register (PR 8) .................... 69
  5.7.9 Global Level Register (GL$^P$) (PR 16) .................................. 69

6 Instruction Set Overview ................................................................. 71
  6.1 Instruction Execution ............................................................... 71
  6.2 Instruction Formats ................................................................. 72
  6.3 Instruction Categories .............................................................. 72
    6.3.1 Memory Access Instructions ............................................... 73
      6.3.1.1 Memory Alignment Restrictions ...................................... 73
      6.3.1.2 Addressing Conventions .............................................. 74
      6.3.1.3 Address Space Identifiers (ASIs) .................................. 76
      6.3.1.4 Separate Instruction Memory ....................................... 78
    6.3.2 Memory Synchronization Instructions ..................................... 78
    6.3.3 Integer Arithmetic and Logical Instructions ............................ 79
      6.3.3.1 Setting Condition Codes ............................................. 79
      6.3.3.2 Shift Instructions .................................................... 79
      6.3.3.3 Set High 22 Bits of Low Word ...................................... 79
      6.3.3.4 Integer Multiply/Divide ............................................. 79
      6.3.3.5 Tagged Add/Subtract ................................................ 79
    6.3.4 Control-Transfer Instructions (CTIs) .................................... 79
      6.3.4.1 Conditional Branches ............................................... 81
      6.3.4.2 Unconditional Branches .......................................... 81
      6.3.4.3 CALL and JMPL Instructions ...................................... 81
      6.3.4.4 RETURN Instruction ............................................... 82
      6.3.4.5 DONE and RETRY Instructions .................................... 82
      6.3.4.6 Trap Instruction (Tcc) .......................................... 82
      6.3.4.7 DCTI Couples .................................................... 82
    6.3.5 Conditional Move Instructions ............................................. 83
    6.3.6 Register Window Management Instructions ............................... 83
      6.3.6.1 SAVE Instruction ................................................ 84
      6.3.6.2 RESTORE Instruction ............................................. 84
      6.3.6.3 SAVED Instruction ............................................... 84
      6.3.6.4 RESTORED Instruction ............................................ 85
6.3.6.5 Flush Windows Instruction ........................................ 85
6.3.7 Ancillary State Register (ASR) Access .............................. 85
6.3.8 Privileged Register Access ........................................... 85
6.3.9 Floating-Point Operate (FPop) Instructions ......................... 85
6.3.10 Implementation-Dependent Instructions ............................. 86
6.3.11 Reserved Opcodes and Instruction Fields ........................... 86

7 Instructions ........................................................................ 87
7.31.1 FMUL8x16 Instruction ............................................... 147
7.31.2 FMUL8x16AU Instruction ............................................. 147
7.31.3 FMUL8x16AL Instruction ............................................ 148
7.31.4 FMUL8SUx16 Instruction ............................................. 148
7.31.5 FMUL8ULx16 Instruction ............................................. 148
7.31.6 FMULD8SUx16 Instruction ........................................... 149
7.31.7 FMULD8ULx16 Instruction ........................................... 150
7.34.1 FPACK16 ................................................................. 154
7.34.2 FPACK32 ................................................................. 155
7.34.3 FPACKFIX ............................................................... 156

7.62.1 Memory Synchronization .............................................. 202
7.62.2 Synchronization of the Virtual Processor ......................... 203
7.62.3 TSO Ordering Rules affecting Use of MEMBAR ................. 203
7.73.1 Exceptions ............................................................... 220
7.73.2 Weak versus Strong Prefetches .................................... 221
7.73.3 Prefetch Variants ....................................................... 222
7.73.3.1 Prefetch for Several Reads (fcn = 0, 20(1416)) .............. 222
7.73.3.2 Prefetch for One Read (fcn = 1, 21(1516)) ................. 222
7.73.3.3 Prefetch for Several Writes (and Possibly Reads) (fcn = 2, 22(1616)) ........................................ 223
7.73.3.4 Prefetch for One Write (fcn = 3, 23(1716)) ............... 223
7.73.3.5 Prefetch Page (fcn = 4) ........................................... 223
7.73.3.6 Prefetch to Nearest Unified Cache (fcn = 17(11 16)) .... 223
7.73.4 Implementation-Dependent Prefetch Variants (fcn = 16, 18, 19, and 24–31) 224
7.73.5 Additional Notes ........................................................ 224

8 IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007 .... 291
8.1 Traps Inhibiting Results .................................................. 291
8.2 Underflow Behavior ......................................................... 292
8.2.1 Trapped Underflow Definition (ufm = 1) .......................... 293
8.2.2 Untrapped Underflow Definition (ufm = 0) ....................... 293
8.3 Integer Overflow Definition ............................................. 293
8.4 Floating-Point Nonstandard Mode .................................... 293
8.5 Arithmetic Result Tables .................................................. 294
8.5.1 Floating-Point Add (FADD) .......................................... 295
8.5.2 Floating-Point Subtract (FSUB) ..................................... 295
8.5.3 Floating-Point Multiply ............................................... 296
8.5.4 Floating-Point Multiply-Add (FMADD) ............................ 296
8.5.5 Floating-Point Negative Multiply-Add (FMNADD) ......... 297
8.5.6 Floating-Point Multiply-Subtract (FMSUB) ....................... 298
8.5.7 Floating-Point Negative Multiply-Subtract (FNMSUB) .... 299
8.5.8 Floating-Point Divide (FDIV) ......................................... 301
8.5.9 Floating-Point Square Root (FSQRT) ............................... 301
8.5.10 Floating-Point Compare (FCMP, FCMPE) ....................... 302
8.5.11 Floating-Point to Floating-Point Conversions (F<s|d|q>TO<s|d|q>) 302
8.5.12 Floating-Point to Integer Conversions (F<s|d|q>TO<i|x>) .... 303
8.5.13 Integer to Floating-Point Conversions (F<i|x>TO<s|d|q>) .... 304

9 Memory .............................................................................. 305
9.1 Memory Location Identification ....................................... 305
9.2 Memory Accesses and Cacheability .................................................. 306
  9.2.1 Coherence Domains ............................................................... 306
    9.2.1.1 Cacheable Accesses ...................................................... 306
    9.2.1.2 Noncacheable Accesses ................................................ 306
    9.2.1.3 Noncacheable Accesses with Side-Effect ............................ 307
  9.3 Memory Addressing and Alternate Address Spaces .......................... 308
      9.3.1 Memory Addressing Types ................................................ 308
      9.3.2 Memory Address Spaces ................................................... 308
      9.3.3 Address Space Identifiers ................................................. 309
  9.4 SPARC V9 Memory Model .......................................................... 310
      9.4.1 SPARC V9 Program Execution Model .................................... 310
      9.4.2 Virtual Processor/Memory Interface Model .............................. 312
  9.5 The UltraSPARC Architecture Memory Model — TSO ......................... 313
      9.5.1 Memory Model Selection .................................................... 313
      9.5.2 Programmer-Visible Properties of the UltraSPARC Architecture TSO Model 314
      9.5.3 TSO Ordering Rules .......................................................... 315
      9.5.4 Hardware Primitives for Mutual Exclusion .............................. 316
        9.5.4.1 Compare-and-Swap (CASA, CASXA) .................................. 316
        9.5.4.2 Swap (SWAP) ............................................................. 316
        9.5.4.3 Load Store Unsigned Byte (LDSTUB) ................................. 316
      9.5.5 Memory Ordering and Synchronization .................................... 316
        9.5.5.1 Ordering MEMBAR Instructions ....................................... 317
        9.5.5.2 Sequencing MEMBAR Instructions .................................... 318
        9.5.5.3 Synchronizing Instruction and Data Memory ......................... 318
  9.6 Nonfaulting Load ................................................................. 319
  9.7 Store Coalescing ................................................................. 320
10 Address Space Identifiers (ASIs) .................................................. 321
  10.1 Address Space Identifiers and Address Spaces ............................ 321
  10.2 ASI Values ............................................................................ 321
  10.3 ASI Assignments .................................................................... 322
    10.3.1 Supported ASIs ............................................................... 322
  10.4 Special Memory Access ASIs .................................................... 329
    10.4.1 ASIs 1016, 1116, 1616, 1f16 and 1816 (ASI_*AS_IF_USER_*) ....... 329
    10.4.2 ASIs 1816, 1916, 1E16, and 1F16 (ASI_*AS_IF_USER_*_LITTLE) .... 329
    10.4.3 ASI 1416 (ASI_REAL) .......................................................... 330
    10.4.4 ASI 1516 (ASI_REAL_IO) ..................................................... 330
    10.4.5 ASI 1C16 (ASI_REAL_LITTLE) .............................................. 330
    10.4.6 ASI 1D16 (ASI_REAL_IO_LITTLE) ......................................... 331
    10.4.7 ASIs 2216, 2316, 2716, 2A16, 2B16, 2F16 (Privileged Load Integer Twin Extended Word) 331
    10.4.8 ASIs 2616 and 2E16 (Privileged Load Integer Twin Extended Word, Real Addressing) 331
    10.4.9 ASIs E216, E316, EA16, EB16 (Nonprivileged Load Integer Twin Extended Word) 332
    10.4.10 Block Load and Store ASIs .................................................. 333
    10.4.11 Partial Store ASIs ............................................................. 333
    10.4.12 Short Floating-Point Load and Store ASIs .............................. 333
  10.5 ASI-Accessible Registers ........................................................ 334
    10.5.1 Privileged Scratchpad Registers (ASI_SCRATCHPAD) .................. 334
    10.5.2 ASI Changes in the UltraSPARC Architecture ............................ 334
11 Performance Instrumentation .......................................................... 337
  11.1 High-Level Requirements ........................................................ 337
    11.1.1 Usage Scenarios ............................................................... 337
    11.1.2 Metrics ............................................................................ 338
11.3 Accuracy Requirements 338
11.2 Performance Counters and Controls 339
11.2.1 Counter Overflow 339

12 Traps 341
12.1 Virtual Processor Privilege Modes 342
12.2 Virtual Processor States and Traps 343
  12.2.1 Usage of Trap Levels 343
12.3 Trap Categories 343
  12.3.1 Precise Traps 344
  12.3.2 Deferred Traps 344
  12.3.3 Disrupting Traps 345
    12.3.3.1 Disrupting versus Precise and Deferred Traps 345
    12.3.3.2 Causes of Disrupting Traps 346
    12.3.3.3 Conditioning of Disrupting Traps 346
    12.3.3.4 Trap Handler Actions for Disrupting Traps 347
  12.3.4 Uses of the Trap Categories 347
12.4 Trap Control 347
  12.4.1 PIL Control 348
  12.4.2 FSR.tem Control 348
12.5 Trap-Table Entry Addresses 348
  12.5.1 Trap-Table Entry Address to Privileged Mode 348
  12.5.2 Privileged Trap Table Organization 349
  12.5.3 Trap Type (TT) 349
    12.5.3.1 Trap Type for Fill/Fill Traps 355
  12.5.4 Trap Priorities 356
12.6 Trap Processing 356
  12.6.1 Normal Trap Processing 356
12.7 Exception and Interrupt Descriptions 358
  12.7.1 SPARC V9 Traps Not Used in UltraSPARC Architecture 2007 362
12.8 Register Window Traps 362
  12.8.1 Window Spill and Fill Traps 363
  12.8.2 clean_window Trap 363
  12.8.3 Vectoring of Fill/Spill Traps 363
  12.8.4 CWP on Window Traps 363
  12.8.5 Window Trap Handlers 364

13 Interrupt Handling 365
13.1 Interrupt Packets 365
13.2 Software Interrupt Register (SOFTINT) 366
  13.2.1 Setting the Software Interrupt Register 366
  13.2.2 Clearing the Software Interrupt Register 366
13.3 Interrupt Queues 366
  13.3.1 Interrupt Queue Registers 367
13.4 Interrupt Traps 368

14 Memory Management 369
14.1 Virtual Address Translation 369
14.2 Context ID 372
14.3 TSB Translation Table Entry (TTE) 373
14.4 Translation Storage Buffer (TSB) 376
  14.4.1 TSB Indexing Support 376
  14.4.2 TSB Cacheability and Consistency 377
  14.4.3 TSB Organization 377
14.5 ASI Value, Context ID, and Endianness Selection for Translation 377
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
- the SIR instruction is now hyperprivileged

UltraSPARC Architecture 2007 includes the following changes since :

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

In addition, architectural features are now tagged with Software Classes and Implementation Classes. 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

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.
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.

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 several chapters and appendices (including Memory, Memory Management, Performance Instrumentation, 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 specification
- 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 Principal Engineer 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:

- Fonts and Notational Conventions on page 2.
- Reporting Errors in this Specification on page 4.

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 VIS instructions.

This specification 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
- 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, “fccn”, “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 HEL VETICA 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, 10012, FFFF 000016). 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.
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
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.

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.

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.

cross-call  An interprocessor call in a system containing 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.

cycle  The atomic unit of time in a physical implementation of a processor core. The duration of a cycle is its period, and the inverse of the period is the physical processor core’s operating frequency (typically measured in gigahertz, in contemporary implementations). The physical processor core divides the work of managing instructions and data and executing instructions into multiple cycles. This division of processing steps into cycles is implementation-dependent. The operating frequency is implementation-dependent and potentially varying in time for a given implementation.

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

DCTI  Delayed control transfer instruction.

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.

Deprecation features should not be used in new UltraSPARC Architecture software and may not be supported in future versions of the architecture.

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

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 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.

fccn  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 85. 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 the processor is in hyperprivileged mode;
2) processor state that is only accessible to software while the processor is in hyperprivileged mode


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 hardware addresses. The MMU is composed of the ASRs and ASI registers used to manage address translation. See also context 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, 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 (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. 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.
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.
PC  Program counter.
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.
See also processor, strand, thread, and virtual processor.
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,
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 privileged mode; for example, privileged registers,privileged ASRs,
or, in general, privileged state;
(3) an instruction that can be executed only when the virtual processor is in
privileged mode.
privileged mode  The mode in which a processor is operating when PSTATE.priv = 1. 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.
See also pipeline, 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. See also virtual address.

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 86.
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 32).
Throughout this specification, figures and tables illustrating registers and instruction encodings indicate reserved fields and reserved bit combinations with a wide (“em”) dash (—).

restricted Describes an address space identifier (ASI) that may be accessed only while the virtual processor is operating in privileged 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).
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.
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. See also pipeline, 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.

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 hardware memory 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. See also pipeline, processor, strand, and virtual processor.

TNPC Trap-saved next program counter.

TPC Trap-saved program counter.

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. 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 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.
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.

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

Synonym for undefined.

A system containing a single virtual processor.

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

Synonym for application program.

Abbreviation for 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 real addresses, which can be used to access real memory. See also real address.

Synonyms for 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. See also pipeline, processor, strand, and thread.

Abbreviation for VIST™ Instruction Set.

Abbreviation for virtual processor.

A 4-byte datum. Note: The definition of this term is architecture dependent and may differ from that used in other processor architectures.
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 Microsystems’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, and supports the ability to run multiple simultaneous guest operating systems. (hyperprivileged mode is described in detail in the Hyperprivileged version of this specification)

- **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** — UltraSPARC Architecture 2007 provides a blueprint for the software view of the UltraSPARC MMU (TTEs and TSBs).

### 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 through 1987. 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.

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). 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 in mode(s) of greater privilege. 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.

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.
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 MAXPGL + 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 (single instruction, multiple data) instructions

1. No UltraSPARC Architecture processor currently implements the LDQF instruction in hardware; it generates an exception and is emulated in software running at a higher privilege level.
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.

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 73.

### 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 75 and FIGURE 6-3 on page 77.

### 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 $2^n - 1$ and $2^{64} - 2^{64-n} - 1$.

### 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 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 software can use the implementation-dependent ASIs to access special protected registers, such as cache control registers, virtual processor state registers, and other processor-dependent or system-dependent values. See Address Space Identifiers (ASIs) on page 76 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.

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 133 and *Memory Barrier* on page 201, 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.

---

\(^1\) 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
3.3.3 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 unconditional 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 (±231 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 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 software (TPC, TNPC, TSTATE, TT, TICK, TBA, PSTATE, TL, PIL, CWP, CANSAVE, CANRESTORE, CLEANWIN, OTHERWIN, and WSTATE).
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 85.

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 software through a trap table that may contain the first 8 instructions (32 for some frequently used traps) of each trap handler. The base address of the table is established by software in a state register (the Trap Base Address register, TBA). The displacement within the table is encoded in the type number of each trap and the level of the trap. 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 also sets bits in the PSTATE 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.
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 24.
- Floating-Point Data Formats on page 27.
- SIMD Data Formats on page 29.

Names are assigned to individual subwords of the multiword data formats as described in these sections:
- Signed Integer Doubleword (64 bits) on page 25.
- Unsigned Integer Doubleword (64 bits) on page 26.
- Floating Point, Double Precision (64 bits) on page 27.
- Floating Point, Quad Precision (128 bits) on page 28.
4.1 Integer Data Formats

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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Width (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed integer byte</td>
<td>8</td>
<td>$-2^7$ to $2^7 - 1$</td>
</tr>
<tr>
<td>Signed integer halfword</td>
<td>16</td>
<td>$-2^{15}$ to $2^{15} - 1$</td>
</tr>
<tr>
<td>Signed integer word</td>
<td>32</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
</tr>
<tr>
<td>Signed integer doubleword/extended-word</td>
<td>64</td>
<td>$-2^{63}$ to $2^{63} - 1$</td>
</tr>
<tr>
<td>Unsigned integer byte</td>
<td>8</td>
<td>0 to $2^8 - 1$</td>
</tr>
<tr>
<td>Unsigned integer halfword</td>
<td>16</td>
<td>0 to $2^{16} - 1$</td>
</tr>
<tr>
<td>Unsigned integer word</td>
<td>32</td>
<td>0 to $2^{32} - 1$</td>
</tr>
<tr>
<td>Unsigned integer doubleword/extended-word</td>
<td>64</td>
<td>0 to $2^{64} - 1$</td>
</tr>
<tr>
<td>Integer tagged word</td>
<td>32</td>
<td>0 to $2^{30} - 1$</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Subformat Name</th>
<th>Subformat Field</th>
<th>Memory Address</th>
<th>Register Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD-0</td>
<td>signed_dbl_integer[63:32]</td>
<td>$n \mod 8 = 0$</td>
<td>$r \mod 2 = 0$</td>
</tr>
<tr>
<td>SD-1</td>
<td>signed_dbl_integer[31:0]</td>
<td>$(n + 4) \mod 8 = 4$</td>
<td>$(r + 1) \mod 2 = 1$</td>
</tr>
<tr>
<td>SX</td>
<td>signed_ext_integer[63:0]</td>
<td>$n \mod 8 = 0$</td>
<td>$r$</td>
</tr>
<tr>
<td>UD-0</td>
<td>unsigned_dbl_integer[63:32]</td>
<td>$n \mod 8 = 0$</td>
<td>$r \mod 2 = 0$</td>
</tr>
<tr>
<td>UD-1</td>
<td>unsigned_dbl_integer[31:0]</td>
<td>$(n + 4) \mod 8 = 4$</td>
<td>$(r + 1) \mod 2 = 1$</td>
</tr>
<tr>
<td>UX</td>
<td>unsigned_ext_integer[63:0]</td>
<td>$n \mod 8 = 0$</td>
<td>$r$</td>
</tr>
</tbody>
</table>

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.

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.

4.1.1.3 Signed Integer Extended-Word (64 bits)

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

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.

4.1.2.3  Unsigned Extended Integer (64 bits)

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

4.1.3  Tagged Word (32 bits)

FIGURE 4-7 illustrates the 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.

<table>
<thead>
<tr>
<th>FS</th>
<th>exp [7:0]</th>
<th>fraction [22:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>23</td>
</tr>
</tbody>
</table>

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

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

- **s** = sign (1 bit)
- **e** = biased exponent (8 bits)
- **f** = fraction (23 bits)
- **u** = undefined

- Normalized value (0 < e < 255): \((-1)^s \times 2^{e-127} \times 1.f\)
- Subnormal value (e = 0): \((-1)^s \times 2^{-126} \times 0.f\)
- Zero (e = 0, f = 0): \((-1)^s \times 0\)
- Signalling NaN: s = u; e = 255 (max); f = 0.uuuuu
  (At least one bit of the fraction must be nonzero)
- Quiet NaN: s = u; e = 255 (max); f = 1.uuuuu
- \(-\infty\) (negative infinity): s = 1; e = 255 (max); f = 0.0000
- \(+\infty\) (positive infinity): s = 0; e = 255 (max); f = 0.0000

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.

<table>
<thead>
<tr>
<th>FD-0</th>
<th>exp [10:0]</th>
<th>fraction [51:32]</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FD-1</th>
<th>fraction [31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

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

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

<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>f</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign (1 bit)</td>
<td>biased exponent (15 bits)</td>
<td>fraction (112 bits)</td>
<td>undefined</td>
</tr>
<tr>
<td>Normalized value (0 &lt; e &lt; 32767):</td>
<td>$(−1)^s \times 2^{e−16383} \times 1.f$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subnormal value (e = 0):</td>
<td>$(−1)^s \times 2^{−16382} \times 0.f$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero (e = 0, f = 0)</td>
<td>$(−1)^s \times 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signalling NaN</td>
<td>s = u; e = 32767 (max); f = .0uu--uu (At least one bit of the fraction must be nonzero)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Subformat Name</th>
<th>Subformat Field</th>
<th>Memory Address</th>
<th>Register Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD-0</td>
<td>s:exp[10:0]:fraction[51:32]</td>
<td>0 mod 4  \‡</td>
<td>0 mod 2  \f</td>
</tr>
<tr>
<td>FD-1</td>
<td>fraction[31:0]</td>
<td>0 mod 4  \‡</td>
<td>1 mod 2  \f + 1(\ö)</td>
</tr>
<tr>
<td>FQ-0</td>
<td>s:exp[14:0]:fraction[111:96]</td>
<td>0 mod 4  \‡</td>
<td>0 mod 4  \f</td>
</tr>
<tr>
<td>FQ-1</td>
<td>fraction[95:64]</td>
<td>0 mod 4  \‡</td>
<td>1 mod 4  \f + 1(\ö)</td>
</tr>
<tr>
<td>FQ-2</td>
<td>fraction[63:32]</td>
<td>0 mod 4  \‡</td>
<td>2 mod 4  \f + 2</td>
</tr>
<tr>
<td>FQ-3</td>
<td>fraction[31:0]</td>
<td>0 mod 4  \‡</td>
<td>3 mod 4  \f + 3(\ö)</td>
</tr>
</tbody>
</table>

* 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).

\ö 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).

\f 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 \(\leq 31\)).

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.
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).

![Uint8 SIMD Data Format](image)

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).

![Int16 SIMD Data Format](image)

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).

![Int32 SIMD Data Format](image)

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.
The following registers are described in this chapter:

- **General-Purpose R Registers** on page 32.
- **Floating-Point Registers** on page 38.
- **Floating-Point State Register (FSR)** on page 42.
- **Ancillary State Registers** on page 48. The following registers are included in this category:
  - 32-bit Multiply/Divide Register (y) (ASR 0) on page 50.
  - Integer Condition Codes Register (ccr) (ASR 2) on page 50.
  - Address Space Identifier (asi) Register (ASR 3) on page 51.
  - Tick (tick) Register (ASR 4) on page 52.
  - Program Counters (pc, npc) (ASR 5) on page 52.
  - Floating-Point Registers State (fprs) Register (ASR 6) on page 53.
  - General Status Register (gsr) (ASR 19) on page 54.
  - softintP Register (ASRs 20, 21, 22) on page 54.
  - softint_setP Pseudo-Register (ASR 20) on page 55.
  - softint_clrP Pseudo-Register (ASR 21) on page 56.
  - Tick Compare (tick_cmprP) Register (ASR 23) on page 56.
  - System Tick (tick) Register (ASR 24) on page 57.
  - System Tick Compare (tick_cmprP) Register (ASR 25) on page 57.

- **Register-Window PR State Registers** on page 58. The following registers are included in this subcategory:
  - Current Window Pointer (cwpP) Register (PR 9) on page 59.
  - Savable Windows (cansaveP) Register (PR 10) on page 59.
  - Restorable Windows (canrestoreP) Register (PR 11) on page 59.
  - Clean Windows (cleanwinP) Register (PR 12) on page 59.
  - Other Windows (otherwinP) Register (PR 13) on page 60.
  - Window State (wstateP) Register (PR 14) on page 60.

- **Non-Register-Window PR State Registers** on page 61. The following registers are included in this subcategory:
  - Trap Program Counter (tpcP) Register (PR 0) on page 61.
  - Trap Next PC (tnpcP) Register (PR 1) on page 62.
  - Trap State (tstateP) Register (PR 2) on page 63.
  - Trap Type (ttP) Register (PR 3) on page 64.
  - Trap Base Address (tbaP) Register (PR 5) on page 64.
  - Processor State (pstateP) Register (PR 6) on page 64.
  - Trap Level Register (tlP) (PR 7) on page 68.
  - Processor Interrupt Level (pilP) Register (PR 8) on page 69.
  - Global Level Register (glP) (PR 16) on page 69.

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

Some register bit fields in this specification are explicitly marked as "reserved". In addition, for convenience, some registers in this chapter are illustrated as fewer than 64 bits wide. Any bits not illustrated are implicitly reserved and treated as if they were explicitly marked as reserved.

Reserved bits, whether explicitly or implicitly reserved, may be assigned meaning in future versions of the architecture.

To ensure that existing software will continue to operate correctly, software must take into account that reserved register bits may be used in the future. The following Programming and Implementation Notes support that intent.

Programming Notes

Software should ensure that when a reserved register field is written, it is only written with (1) the value zero or (2) a value previously read from that field.

If software writes a reserved register field to any value other than (1) zero or (2) a value previously read from that field, it is considered a software error. Such an error:

- may or may not be detected or reported (for example, by a trap) by UltraSPARC Architecture 2007 processors (and software should not expect that it will be)
- may cause a trap or cause other unintended behavior when executed on future UltraSPARC Architecture processors

When a register is read, software should not assume that register fields reserved in UltraSPARC Architecture 2007 will read as 0 or any other particular value, either now or in the future.

Implementation Notes

When a register is read by software, an UltraSPARC Architecture 2007 virtual processor should return a value of zero for any bits reserved in UltraSPARC Architecture 2007.

When software attempts to change the contents of a register field that is reserved in UltraSPARC Architecture 200x by writing a value to that field that differs from the current contents of that field, an UltraSPARC Architecture 200x virtual processor will either ignore the write to that field or cause an exception. "Current contents" means the contents that software would observe if it read that field (nominally zero).

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 MAXPGL + 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.

5.2.1 Global R Registers

Registers R[0]–R[7] refer to a set of eight registers called the global registers (labelled g0 through g7). At any time, one of MAXPGL +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 (glP) (PR 16) on page 69.

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

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

<table>
<thead>
<tr>
<th>Windowed Register Address</th>
<th>R Register Address</th>
</tr>
</thead>
</table>

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, MAXPGL, ranges from 2 to 15. The number of implemented global register sets is MAXPGL + 1. The total number of R registers in a given UltraSPARC Architecture implementation is:

\[(N_{\text{REG WINDOWS}} \times 16) + ((\text{MAXPGL} + 1) \times 8)\]

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.

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

Register address o, where 8 ≤ o ≤ 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 ≤ i ≤ 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 35 and FIGURE 5-3 on page 37.

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 $N_{\text{REG\_WINDOWS}}$, the highest-numbered implemented window overlaps with window 0. The outs of window $N_{\text{REG\_WINDOWS}} - 1$ are the ins of window 0. Implemented windows are numbered contiguously from 0 through $N_{\text{REG\_WINDOWS}} - 1$.

Because the windows overlap, the number of windows available to software is 1 less than the number of implemented windows; that is, $N_{\text{REG\_WINDOWS}} - 1$. 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. 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 (cleanwinP) Register (PR 12) on page 59, Savable Windows (cansaveP) Register (PR 10) on page 59, and Restorable Windows (canrestoreP) Register (PR 11) on page 59.

**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.

Register Window Management Instructions on page 83 describes how the windowed integer registers are managed.

---

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.
 CHAPTER 5 • Registers

5.2.3 Special \( \textbf{R} \) Registers

The use of two of the \( \textbf{R} \) 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 \( R[15] \) (\textit{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 an LDTW, LDTWA, STTW, or STTWA instruction that refers to a misaligned (odd) destination register number causes an illegal_instruction trap.

### 5.3 Floating-Point Registers

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 FQ[0], FQ[4], …, FQ[60]
- Thirty-two 64-bit double-precision registers, referenced as FD[0], FD[2], …, FD[62]
- Thirty-two 32-bit single-precision registers, referenced as FS[0], FS[1], …, FS[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)

<table>
<thead>
<tr>
<th>Single Precision (32-bit)</th>
<th>Double Precision (64-bit)</th>
<th>Quad Precision (128-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Language</td>
<td>Bits</td>
<td>Assembly Language</td>
</tr>
<tr>
<td>FS[0]</td>
<td>%f0</td>
<td>63:32</td>
</tr>
<tr>
<td>FS[7]</td>
<td>%f7</td>
<td>31:0</td>
</tr>
<tr>
<td>FS[9]</td>
<td>%f9</td>
<td>31:0</td>
</tr>
</tbody>
</table>

AT
<table>
<thead>
<tr>
<th>Single Precision (32-bit)</th>
<th>Double Precision (64-bit)</th>
<th>Quad Precision (128-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Assembly Language</td>
<td>Bits</td>
</tr>
<tr>
<td>$F_S[15]$</td>
<td>%f15</td>
<td>31:0</td>
</tr>
<tr>
<td>$F_S[17]$</td>
<td>%f17</td>
<td>31:0</td>
</tr>
<tr>
<td>$F_S[18]$</td>
<td>%f18</td>
<td>63:32</td>
</tr>
<tr>
<td>$F_S[21]$</td>
<td>%f21</td>
<td>31:0</td>
</tr>
<tr>
<td>$F_S[23]$</td>
<td>%f23</td>
<td>31:0</td>
</tr>
<tr>
<td>$F_S[24]$</td>
<td>%f24</td>
<td>63:32</td>
</tr>
<tr>
<td>$F_S[25]$</td>
<td>%f25</td>
<td>31:0</td>
</tr>
<tr>
<td>$F_S[26]$</td>
<td>%f26</td>
<td>63:32</td>
</tr>
<tr>
<td>$F_S[27]$</td>
<td>%f27</td>
<td>31:0</td>
</tr>
<tr>
<td>$F_S[28]$</td>
<td>%f28</td>
<td>63:32</td>
</tr>
<tr>
<td>$F_S[29]$</td>
<td>%f29</td>
<td>31:0</td>
</tr>
</tbody>
</table>
### 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 labelled \( b^4 \) \( \ldots \) \( b^0 \) (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

<table>
<thead>
<tr>
<th>Register Operand Type</th>
<th>Full 6-bit Register Number</th>
<th>Encoding in a 5-bit Register Field in an Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>0 ( b^4 ) ( b^3 ) ( b^2 ) ( b^1 ) ( b^0 )</td>
<td>( b^4 ) ( b^3 ) ( b^2 ) ( b^1 ) ( b^0 )</td>
</tr>
<tr>
<td>Double</td>
<td>( b^5 ) ( b^4 ) ( b^3 ) ( b^2 ) ( b^1 )</td>
<td>( b^4 ) ( b^3 ) ( b^2 ) ( b^1 ) ( b^0 )</td>
</tr>
<tr>
<td>Quad</td>
<td>( b^5 ) ( b^4 ) ( b^3 ) ( b^2 ) ( b^1 ) ( b^0 )</td>
<td>( b^4 ) ( b^3 ) ( b^2 ) ( b^1 ) ( b^0 )</td>
</tr>
</tbody>
</table>

**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.

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.

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.

UltraSPARC Architecture 2011 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.

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 42 through 48 describe the remaining fields in the FSR.

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

The four sets of floating-point condition code fields are labelled fcc0, fcc1, fcc2, and fcc3 (fccn 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 fccn fields in the FSR, as selected by the compare instruction. The fccn 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 5-4**, \( f_{rs1} \) 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 fccn is unchanged.

**TABLE 5-4** Floating-Point Condition Codes (fccn) Fields of FSR

<table>
<thead>
<tr>
<th>Indicated Relation (FCMP*, FCMPE*)</th>
<th>Content of fccn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( F[rs1] = F[rs2] )</td>
</tr>
<tr>
<td>1</td>
<td>( F[rs1] &lt; F[rs2] )</td>
</tr>
<tr>
<td>2</td>
<td>( F[rs1] &gt; F[rs2] )</td>
</tr>
<tr>
<td>3</td>
<td>( F[rs1] \neq F[rs2] ) (unordered)</td>
</tr>
</tbody>
</table>
5.4.2 Rounding Direction (rd)

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

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

<table>
<thead>
<tr>
<th>rd</th>
<th>Round Toward</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nearest (even, if tie)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>+ \infty</td>
</tr>
<tr>
<td>3</td>
<td>- \infty</td>
</tr>
</tbody>
</table>

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 54 for further details.

5.4.3 Trap Enable Mask (tem)

Bits 27 through 23 are enable bits for each of the five IEEE-754 floating-point exceptions that can be indicated in the current_exception field (cexc). See FIGURE 5-6 on page 47. 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 \texttt{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)

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 293.

5.4.5 FPU Version (ver)

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

For each SPARC V9 IU implementation, 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.
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 LDFSRR 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.

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 5-6. 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 5-6** FSR Floating-Point Trap Type (ftt) Field

<table>
<thead>
<tr>
<th>Condition Detected During Execution of an FPop</th>
<th>Relative Priority (1 = highest)</th>
<th>FSR.ftt Set to Value</th>
<th>Exception Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid_fp_register</td>
<td>20</td>
<td>6</td>
<td>fp_exception_other</td>
</tr>
<tr>
<td>unfinished_FPop</td>
<td>30</td>
<td>2</td>
<td>fp_exception_other</td>
</tr>
<tr>
<td>IEEE_754_exception</td>
<td>40</td>
<td>1</td>
<td>fp_exception_ieee_754</td>
</tr>
<tr>
<td>Reserved</td>
<td>—</td>
<td>3, 4, 5, 7</td>
<td>—</td>
</tr>
<tr>
<td>(none detected)</td>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
</tbody>
</table>

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 fcss 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 \texttt{fp\_exception\_other} exception with a floating-point trap type of \texttt{unfinished\_FPop} that does not subsequently generate an IEEE trap, the recovery software should set \texttt{cexc}, \texttt{aexc}, and the destination register or \texttt{fccn}, as appropriate.

\textbf{ftt = 1 (IEEE\_754\_exception).} The \texttt{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 \texttt{cexc} field. The \texttt{aexc} and \texttt{fccn} fields and the destination F register are unchanged.

\textbf{ftt = 2 (unfinished\_FPop).} The \texttt{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 \texttt{cexc} field is unchanged.

\begin{verbatim}
<table>
<thead>
<tr>
<th>Implementation Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementations are encouraged to support standard IEEE 754 floating-point arithmetic with reasonable performance (that is, without generating \texttt{fp_exception_other} with FSR.ftt=\texttt{unfinished_FPop}) in all cases, even if some cases are slower than others.</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{IMPL. DEP. \#248-U3:} The conditions under which an \texttt{fp\_exception\_other} exception with floating-point trap type of \texttt{unfinished\_FPop} can occur are implementation dependent. An implementation may cause \texttt{fp\_exception\_other} with FSR.ftt = \texttt{unfinished\_FPop} under a different (but specified) set of conditions.

\textbf{ftt = 3 (Reserved).}

\begin{verbatim}
<table>
<thead>
<tr>
<th>SPARC V9 Compatibility Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>In SPARC V9, FSR.ftt = 3 was defined to be &quot;unimplemented_FPop&quot;. All conditions which used to cause \texttt{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.</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{ftt = 4 (Reserved).}

\begin{verbatim}
<table>
<thead>
<tr>
<th>SPARC V9 Compatibility Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the SPARC V9 architecture, FSR.ftt = 4 was defined to be &quot;sequence_error&quot;, 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.</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{ftt = 5 (Reserved).}

\begin{verbatim}
<table>
<thead>
<tr>
<th>SPARC V9 Compatibility Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the SPARC V9 architecture, FSR.ftt = 5 was defined to be &quot;hardware_error&quot;, 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.</td>
</tr>
</tbody>
</table>
\end{verbatim}
**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 \( \text{mod} \) 4. An implementation generates an \( \text{fp\_exception\_other} \) 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 \( \text{illegal\_instruction} \) exception instead of \( \text{fp\_exception\_other} \) with FSR.ftt = 6 (invalid_fp_register), regardless of the specified F registers.

### 5.4.7 Accrued Exceptions (aexc)

Bits 9 through 5 accumulate IEEE_754 floating-point exceptions as long as floating-point exception traps are disabled through the tem field. See FIGURE 5-7 on page 47.

After an FPop completes with \( ftt = 0 \), the tem and cexc fields are logically \( \text{and} \)ed together. If the result is nonzero, aexc is left unchanged and an \( \text{fp\_exception\_ieee\_754} \) trap is generated; otherwise, the new cexc field is \( \text{or} \)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 LDFS or LDXFSR instruction is executed.

### 5.4.8 Current Exception (cexc)

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 47.

**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 47, by the execution of an FPop that either does not cause a trap or causes an \( \text{fp\_exception\_ieee\_754} \) 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 \( \text{fp\_exception\_ieee\_754} \) 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 \( \text{fp\_exception\_ieee\_754} \) 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 \( \text{fp\_exception\_ieee\_754} \) 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 \( \text{fp\_exception\_ieee\_754} \) 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 \( \text{fp\_exception\_ieee\_754} \) 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 5-7 (where “✔” indicates “exception was detected” and “x” indicates “don’t care”):

**TABLE 5-7** Setting of FSR.cexc Bits

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR.tem.ufm</td>
<td>FSR.cexc.ufm</td>
</tr>
<tr>
<td>Exception(s) Detected in F.p. operation</td>
<td>Trap Enable Mask bits (in FSR.tem)</td>
</tr>
<tr>
<td>of uf nx ofm ufm nxm</td>
<td>ofc ufc nxc</td>
</tr>
<tr>
<td>(-) - - x x x</td>
<td>no</td>
</tr>
<tr>
<td>(-) - ✔ x x 0</td>
<td>no</td>
</tr>
<tr>
<td>(-) ✔ ( ^1 ) ✔ ( ^1 ) x 0 0</td>
<td>no</td>
</tr>
<tr>
<td>✔ ( ^2 ) - ✔ ( ^2 ) 0 x 0</td>
<td>no</td>
</tr>
<tr>
<td>(-) - ✔ x x 1</td>
<td>yes</td>
</tr>
<tr>
<td>(-) ✔ ( ^1 ) ✔ ( ^1 ) x 0 1</td>
<td>yes</td>
</tr>
<tr>
<td>(-) ✔ - x 1 x</td>
<td>yes</td>
</tr>
<tr>
<td>(-) ✔ ✔ x 1 x</td>
<td>yes</td>
</tr>
<tr>
<td>✔ ( ^2 ) - ✔ ( ^2 ) 1 x x</td>
<td>yes</td>
</tr>
<tr>
<td>✔ ( ^2 ) - ✔ ( ^2 ) 0 x 1</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.

### 5.4.9 Floating-Point Exception Fields

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

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

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

**FIGURE 5-8** Current Exception Bits (cexc) Fields of FSR
Invalid (nvc, nva). An operand is improper for the operation to be performed. For example, 0.0 ÷ 0.0 and $\infty - \infty$ 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 293 and Untrapped Underflow Definition (ufm = 0) on page 293 for additional details.

Division by zero (dzc, dza). An infinite result is produced exactly from finite operands. For example, $X \div 0.0$, 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.

### 5.4.10 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 5-8 and defined in the following subsections.
Each virtual processor contains its own set of ASRs; ASRs are not shared among virtual processors.

<table>
<thead>
<tr>
<th>ASR number</th>
<th>ASR name</th>
<th>Register</th>
<th>Read by Instruction(s)</th>
<th>Written by Instruction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>Y register (deprecated)</td>
<td>RDY&lt;sup&gt;D&lt;/sup&gt;</td>
<td>WRY&lt;sup&gt;D&lt;/sup&gt;</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>CCR</td>
<td>Condition Codes register</td>
<td>RDCCR</td>
<td>WRCCR</td>
</tr>
<tr>
<td>3</td>
<td>ASI</td>
<td>ASI register</td>
<td>RDASI</td>
<td>WRASI</td>
</tr>
<tr>
<td>4</td>
<td>TICK&lt;sup&gt;P&lt;/sup&gt;&lt;sub&gt;opt&lt;/sub&gt;</td>
<td>TICK register</td>
<td>RDTICK&lt;sup&gt;P&lt;/sup&gt;&lt;sub&gt;opt&lt;/sub&gt;, RDPR&lt;sup&gt;P&lt;/sup&gt; (TICK)</td>
<td>WRPR&lt;sup&gt;P&lt;/sup&gt; (TICK)</td>
</tr>
<tr>
<td>5</td>
<td>PC</td>
<td>Program Counter (PC)</td>
<td>RDPC</td>
<td>(all instructions)</td>
</tr>
<tr>
<td>6</td>
<td>FPRS</td>
<td>Floating-Point Registers Status register</td>
<td>RDFPRS</td>
<td>WRFPRS</td>
</tr>
<tr>
<td>7–14 (7–0E&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>15 (0F&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16–31 (10&lt;sub&gt;16&lt;/sub&gt;–1F&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>non-SPARC V9 ASRs</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16–18 (10&lt;sub&gt;16&lt;/sub&gt;–12&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>19 (13&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>GSR</td>
<td>General Status register (GSR)</td>
<td>RDGSR, FALIGNDATA, many VIS and floating-point instructions</td>
<td>WRGSR, BMASK, SIAM</td>
</tr>
<tr>
<td>20 (14&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>SOFTINT_SET&lt;sup&gt;P&lt;/sup&gt;</td>
<td>(pseudo-register, for &quot;Write 1s Set&quot; to SOFTINT register, ASR 22)</td>
<td>—</td>
<td>WRSOFTINT_SET&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>21 (15&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>SOFTINT_CLR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>(pseudo-register, for &quot;Write 1s Clear&quot; to SOFTINT register, ASR 22)</td>
<td>—</td>
<td>WRSOFTINT_CLR&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>22 (16&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>SOFTINT&lt;sup&gt;P&lt;/sup&gt;</td>
<td>per-virtual processor Soft Interrupt register</td>
<td>RDSOFTINT&lt;sup&gt;P&lt;/sup&gt;</td>
<td>WRSOFTINT&lt;sup&gt;I&lt;/sup&gt;</td>
</tr>
<tr>
<td>23 (17&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>TICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>Tick Compare register</td>
<td>RDTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>WRTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>24 (18&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>STICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>System Tick register</td>
<td>RDSTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>WRSTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>25 (19&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>STICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>System Tick Compare register</td>
<td>RDSTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>WRSTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>26 (1A&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>27 (1B&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>28–29 (1C&lt;sub&gt;16&lt;/sub&gt;–1D&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>30 (1E&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>31 (1F&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>—</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
5.5.1 32-bit Multiply/Divide Register (Y) (ASR 0)

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 246 and page 283; for the multiply step instruction, see page 209; for division instructions, see pages 240 and 281; for the read instruction, see page 226; and for the write instruction, see page 286.

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.

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)

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.

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 FFFF16, 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.

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 5-9).

<table>
<thead>
<tr>
<th>TABLE 5-9 Setting of Carry (Borrow) bits for Subtraction That Sets CCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnsIGNED COMPARISON OF OPerAND VALUES</td>
</tr>
<tr>
<td>R[rs1]{31:0} ≥ R[rs2]{31:0}</td>
</tr>
<tr>
<td>R[rs1]{31:0} &lt; R[rs2]{31:0}</td>
</tr>
<tr>
<td>R[rs1]{63:0} ≥ R[rs2]{63:0}</td>
</tr>
<tr>
<td>R[rs1]{63:0} &lt; R[rs2]{63:0}</td>
</tr>
</tbody>
</table>

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)

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 00₁₆ to 7F₁₆ 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 00₁₆ to 7F₁₆ or an instruction executing in privileged mode attempts an access using an ASI the range 30₁₆ to 7F₁₆, a privileged_action exception is generated. See Chapter 10, Address Space Identifiers (ASIs) for details.
5.5.4 Tick (TICK) Register (ASR 4)

FIGURE 5-13 illustrates the 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 is the nonprivileged trap (npt) bit, which controls access to the TICK register by nonprivileged software.

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

Privileged software 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 by hyperprivileged software. If nonprivileged access is disabled, 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.

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.

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

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.

<table>
<thead>
<tr>
<th>FPRS</th>
<th>fef</th>
<th>du</th>
<th>dl</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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)  

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 225) and written by the WRGSR instruction (see *Write Ancillary State Register* on page 285).

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 5-10.

![FIGURE 5-15 General Status Register (GSR) (ASR 19)](image)

<table>
<thead>
<tr>
<th>Bit Field Description</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32 mask</td>
<td>This 32-bit field specifies the mask used by the BSUMMELLE instruction. The field contents are set by the BMASK instruction.</td>
</tr>
<tr>
<td>31:28 —</td>
<td>Reserved.</td>
</tr>
<tr>
<td>27 im</td>
<td>Interval Mode: If GSR.im = 0, rounding is performed according to FSR.rd; if GSR.im = 1, rounding is performed according to GSR.irnd.</td>
</tr>
<tr>
<td>26:25 irnd</td>
<td>IEEE Std 754-1985 rounding direction to use in Interval Mode (GSR.im = 1), as follows:</td>
</tr>
<tr>
<td></td>
<td>irnd</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>24:8 —</td>
<td>Reserved.</td>
</tr>
<tr>
<td>7:3 scale</td>
<td>5-bit shift count in the range 0–31, used by the FPACK instructions for formatting.</td>
</tr>
<tr>
<td>2:0 align</td>
<td>Least three significant bits of the address computed by the last-executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction.</td>
</tr>
</tbody>
</table>

5.5.8 SOFTINT\(^P\) Register (ASRs 20, 21, 22)  

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

SOFTINT\(^A\) can be read with a RDSOFTINT instruction (see *Read Ancillary State Register* on page 225) and written with a WRSOFTINT, WRSOFTINT_SET, or WRSOFTINT_CLR instruction (see *Write Ancillary State Register* on page 285). An attempt to access 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 5-11.

---

1. This register was (inaccurately) referred to as the “Graphics Status Register” in early UltraSPARC implementations.
Setting any of SOFTINT.sm, SOFTINT.tm, or SOFTINT.int_level 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 366 for additional information regarding the SOFTINT register.

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

A Write State register instruction to ASR 20 (WRSOFTINT_SET) atomically sets selected bits in the privileged SOFTINT Register (ASR 22) (see page 54). That is, bits 16:0 of the write data are ORed into SOFTINT; any ‘1’ bit in the write data causes the corresponding bit of SOFTINT to be set to 1. Bits 63:17 of the write data are ignored.

Access to ASR 20 is privileged 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.

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.
5.5.8.2 SOFTINT_CLR P Pseudo-Register (ASR 21)

A Write State register instruction to ASR 21 (WRSOFTINT_CLR) atomically clears selected bits in the privileged SOFTINT register (ASR 22) (see page 54). That is, bits 16:0 of the write data are inverted and ANDed into SOFTINT; any ‘1’ bit in the write data causes the corresponding bit of SOFTINT to be set to 0. Bits 63:17 of the write data are ignored.

Access to ASR 21 is privileged 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.

5.5.9 Tick Compare (TICK_CMPR P) Register (ASR 23)

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 358).

The TICK_CMPR register is illustrated in FIGURE 5-19 and described in TABLE 5-12.

TABLE 5-12 TICK_CMPR Register Description

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 int_dis</td>
<td>Interrupt Disable. If int_dis = 0, TICK compare interrupts are enabled and if int_dis = 1, TICK compare interrupts are disabled.</td>
</tr>
<tr>
<td>62:0 tick_cmpr</td>
<td>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 &lt; 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.</td>
</tr>
</tbody>
</table>
5.5.10 System Tick (STICK) Register (ASR 24)

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 is the nonprivileged trap (npt) bit, which controls access to the STICK register by nonprivileged software.

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

```
+-----------------+-----------------+
|                 |                 |
|  npt            |  counter        |
+-----------------+-----------------+
    63             62
```

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 by hyperprivileged software. If nonprivileged access is disabled, 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.

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.

5.5.11 System Tick Compare (STICK_CMPRP) Register (ASR 25)

The privileged STICK_CMPRP register allows system software to cause a trap when the STICK register reaches a specified value. An attempt to accesses to this register while in nonprivileged mode causes a privileged Opcode exception (see Exception and Interrupt Descriptions on page 358).

The System Tick Compare Register is illustrated in FIGURE 5-21 and described in TABLE 5-13.

```
+-----------------+-----------------+
|                  |                  |
|  int_dis         |  stick_cmpr     |
+-----------------+-----------------+
    63             62
```

```
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 privilegedOpcode 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 $N_{\text{REG WINDOWS}} - 1$. An attempt to write a value greater than $N_{\text{REG WINDOWS}} - 1$ to any of these registers causes an implementation-dependent value between 0 and $N_{\text{REG WINDOWS}} - 1$ (inclusive) to be written to the register. Furthermore, an attempt to write a value greater than $N_{\text{REG WINDOWS}} - 2$ violates the register window state definition in Register Window State Definition on page 60.

Although the width of each of these five registers is architecturally 5 bits, the width is implementation dependent and shall be between $\lceil \log_2(N_{\text{REG WINDOWS}}) \rceil$ 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, $N_{\text{REG WINDOWS}} = 8$. 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 83.

**Programming Note**: CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN must never be set to a value greater than $N_{\text{REG WINDOWS}} - 2$ on an UltraSPARC Architecture virtual processor. Setting any of these to a value greater than $N_{\text{REG WINDOWS}} - 2$ violates the register window state definition in Register Window State Definition on page 60. 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\textsuperscript{P}) Register (PR 9) \(\text{A1}\)

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 83 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\textsuperscript{P}) Register (PR 10) \(\text{A1}\)

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\textsuperscript{P}) Register (PR 11) \(\text{A1}\)

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\textsuperscript{P}) Register (PR 12) \(\text{A1}\)

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.

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\textsuperscript{P}) Register (PR 13) \textsuperscript{A1}

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\textsubscript{other}. If OTHERWIN is zero, register windows are spilled/filled by use of trap vectors based on the contents of WSTATE\textsubscript{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.

\begin{figure}
\centering
\includegraphics[width=0.3\textwidth]{OTHERWIN Register}
\caption{OTHERWIN Register}
\end{figure}

5.6.6 Window State (WSTATE\textsuperscript{P}) Register (PR 14) \textsuperscript{A1}

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\textsubscript{normal} bits are inserted into TT[TL]. Otherwise, the WSTATE\textsubscript{other} bits are inserted into TT[TL]. See Register Window State Definition, below, for details of the semantics of OTHERWIN.

\begin{figure}
\centering
\includegraphics[width=0.3\textwidth]{WSTATE Register}
\caption{WSTATE Register}
\end{figure}

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 58. Those registers are affected by the instructions described in Register Window Management Instructions on page 83. 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:

\begin{equation}
\text{CANSAVE} + \text{CANRESTORE} + \text{OTHERWIN} = N_{\text{REG WINDOWS}} - 2
\end{equation}

FIGURE 5-3 on page 37 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 \textit{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 \(\text{N\_REG\_WINDOWS} - 1\).

### 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 362 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 mode (that is, when \(\text{PSTATE}.\text{priv} = 1\)), 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)

The privileged Trap Program Counter register (TPC; FIGURE 5-28) contains the program counter (PC) from the previous trap level. There are \(\text{MAXPTL}\) 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.

During normal operation, the value of TPC\([n]\), where \(n\) is greater than the current trap level \((n > \text{TL})\), is undefined.

TABLE 5-14 lists the events that cause TPC to be read or written.
5.7.2 Trap Next PC (TNPC\(^P\)) Register (PR 1)

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

TABLE 5-14  Events that involve TPC, when executing with TL = \(n\).

<table>
<thead>
<tr>
<th>Event</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap</td>
<td>TPC([n+1] \leftarrow PC)</td>
</tr>
<tr>
<td>RETRY instruction</td>
<td>PC \leftarrow TPC([n])</td>
</tr>
<tr>
<td>RDPR (TPC)</td>
<td>R[rd] \leftarrow TPC([n])</td>
</tr>
<tr>
<td>WRPR (TPC)</td>
<td>TPC([n]) \leftarrow value</td>
</tr>
</tbody>
</table>

During normal operation, the value of TNPC\([n]\), where \(n\) is greater than the current trap level (\(n > TL\)), is undefined.

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

TABLE 5-15  Events that involve TNPC, when executing with TL = \(n\).

<table>
<thead>
<tr>
<th>Event</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap</td>
<td>TNPC([n+1] \leftarrow NPC)</td>
</tr>
<tr>
<td>DONE instruction</td>
<td>PC \leftarrow TNPC([n]); NPC \leftarrow TNPC([n]) + 4</td>
</tr>
<tr>
<td>RETRY instruction</td>
<td>NPC \leftarrow TNPC([n])</td>
</tr>
<tr>
<td>RDPR (TNPC)</td>
<td>R[rd] \leftarrow TNPC([n])</td>
</tr>
<tr>
<td>WRPR (TNPO)</td>
<td>TNPC([n]) \leftarrow value</td>
</tr>
</tbody>
</table>
5.7.3 Trap State ($TSTATESP$) Register (PR 2)

The privileged Trap State register ($TSTATESP$; 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 $MAXPTL$ instances of the $TSTATESP$ register, but only one is accessible at a time. The current value in the TL register determines which instance of $TSTATESP$ is accessible. An attempt to read or write the $TSTATESP$ register when $TL = 0$ causes an illegal_instruction exception.

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 there are more bits in the UltraSPARC Architecture’s PSTATE register than in a SPARC V9 PSTATE register, a 13-bit PSTATE value is stored in $TSTATESP$ instead of the 10-bit value specified in the SPARC V9 architecture.

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

### TABLE 5-16 Events That Involve $TSTATESP$, When Executing with $TL = n$

<table>
<thead>
<tr>
<th>Event</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap</td>
<td>$TSTATE[n + 1] \leftarrow$ (registers)</td>
</tr>
<tr>
<td>DONE instruction</td>
<td>(registers) $\leftarrow$ $TSTATE[n]$</td>
</tr>
<tr>
<td>RETRY instruction</td>
<td>(registers) $\leftarrow$ $TSTATE[n]$</td>
</tr>
<tr>
<td>RDPR ($TSTATESP$)</td>
<td>$R[rd] \leftarrow$ $TSTATE[n]$</td>
</tr>
<tr>
<td>WRPR ($TSTATESP$)</td>
<td>$TSTATE[n] \leftarrow$ value</td>
</tr>
</tbody>
</table>
5.7.4 Trap Type (TT\textsuperscript{P}) Register (PR 3)

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. There are $MAXPTL$ 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.

<table>
<thead>
<tr>
<th>TT\textsuperscript{P}</th>
<th>Trap type from trap while TL = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT\textsubscript{1}\textsuperscript{P}</td>
<td>Trap type from trap while TL = 1</td>
</tr>
<tr>
<td>TT\textsubscript{2}\textsuperscript{P}</td>
<td>Trap type from trap while TL = (MAXPTL - 1)</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

FIGURE 5-31 Trap Type Register Stack

During normal operation, the value of TT\([n]\), where \(n\) is greater than the current trap level (\(n > TL\)), is undefined.

TABLE 5-17 lists the events that cause TT to be read or written.

<table>
<thead>
<tr>
<th>Event</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap</td>
<td>TT([n + 1]) ← (trap type)</td>
</tr>
<tr>
<td>RDPR (TT)</td>
<td>R[rd] ← TT([n])</td>
</tr>
<tr>
<td>WRPR (TT)</td>
<td>TT([n]) ← value</td>
</tr>
</tbody>
</table>

5.7.5 Trap Base Address (TBA\textsuperscript{P}) Register (PR 5)

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](image)

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 348.

5.7.6 Processor State (PSTATE\textsuperscript{P}) Register (PR 6)

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](image)
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 074₁₆ 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 67) 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 76. 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.
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 5-18.

<table>
<thead>
<tr>
<th>mm Value</th>
<th>Selected Memory Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Total Store Order (TSO)</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Implementation dependent (impl. dep. #113-V9-Ms10)</td>
</tr>
<tr>
<td>11</td>
<td>Implementation dependent (impl. dep. #113-V9-Ms10)</td>
</tr>
</tbody>
</table>

The current memory model is determined by the value of PSTATE.mm. Software should refrain from writing the values 012, 102, or 112 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 = 102 or 112 are supported in an UltraSPARC Architecture processor is implementation dependent. If the 102 model is supported, then when PSTATE.mm = 102 the implementation must correctly execute software that adheres to the RMO model described in The SPARC Architecture Manual-Version 9. If the 112 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.pef 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 processor. When PSTATE.am = 1, bits 63:32 of virtual addresses are masked out (treated as 0). PSTATE.am does not affect real addresses.

When PSTATE.am = 0, the full 64 bits of all instruction and data addresses are preserved at all points in the virtual processor.

When an MMU is disabled, PSTATE.am has no effect on (does not cause masking of) addresses.
Instances in which the more-significant 32 bits of a virtual address are masked when \texttt{PSTATE.am} = 1 include:

- Before any data virtual address is sent out of the virtual processor (notably, to the memory system, which includes MMU, internal caches, and external caches).
- Before any instruction virtual address is sent out of the virtual processor (notably, to the memory system, which includes MMU, internal caches, and external caches).
- When the value of \texttt{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 \texttt{PC} and \texttt{NPC} are written to \texttt{TPC[TL]} and \texttt{TNPC[TL]} (respectively) during a trap (closed impl.dep. #125-V9-Cs10).
- Before any virtual address is sent to a watchpoint comparator.

When \texttt{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 \texttt{NPC} by a control transfer instruction.
  
  \textbf{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 \texttt{NPC} is incremented to \texttt{NPC + 4} during execution of an instruction that is not a taken control transfer.
  
  \textbf{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 \texttt{TPC[TL]} or \texttt{TNPC[TL]}

  \textbf{Programming Note} Since writes to \texttt{PSTATE} are nondelayed (see page 65), a change to \texttt{PSTATE.am} can affect which instruction is executed immediately after the write to \texttt{PSTATE.am}. Specifically, if a WRPR to the \texttt{PSTATE} register changes the value of \texttt{PSTATE.am} from ‘0’ to ‘1’, and \texttt{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 \texttt{NPC}[31:0] (with the more-significant 32 address bits set to zero).

- When a RDPR instruction reads from \texttt{TPC[TL]} or \texttt{TNPC[TL]}
If (1) \text{TSTATE}[\text{TL}].\text{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).

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

**Programming Note** \text{PSTATE}.am affects the operation of the edge-handling instructions, EDGE<8|16|32>[L]*. See Edge Handling Instructions on page 116 and Edge Handling Instructions (no CC) on page 118.

### Privileged Mode (priv).
When \text{PSTATE}.priv = 1, the virtual processor is operating in privileged mode.

When \text{PSTATE}.priv = 0, the processor is operating in nonprivileged mode.

**PSTATE**\_interrupt\_enable (ie). \text{PSTATE}.ie controls when the virtual processor can take traps due to disrupting exceptions (such as interrupts or errors unrelated to instruction processing).

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 \text{PSTATE}.ie = 1. At all other times, they are held pending. For more details, see Conditioning of Disrupting Traps on page 346.

**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 \text{PSTATE}.ag bit.

### 5.7.7 Trap Level Register (TL\(^P\)) (PR 7)\(^\text{A1}\)

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 MAXPTL, 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). In a typical implementation MAXPTL = MAXPGL (see impl. dep. #401-S10). Architecturally, MAXPTL must be \(\geq 2\).

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

\(^1\) which sets \text{PSTATE}.am to ‘1’, by restoring the value from \text{TSTATE}[\text{TL}].\text{pstate.am} to \text{PSTATE}.am
The effect of writing to TL with a WRPR instruction is summarized in TABLE 5-19.

### TABLE 5-19  Effect of WRPR of Value x to Register TL

<table>
<thead>
<tr>
<th>Value x Written with WRPR</th>
<th>Privilege Level when Executing WRPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \leq \text{MAXPTL} )</td>
<td>Nonprivileged ( \text{TL} \leftarrow x )</td>
</tr>
<tr>
<td></td>
<td>Privileged</td>
</tr>
<tr>
<td>( x &gt; \text{MAXPTL} )</td>
<td>privileged_opcode exception</td>
</tr>
<tr>
<td></td>
<td>( \text{TL} \leftarrow \text{MAXPTL} ) (no exception generated)</td>
</tr>
</tbody>
</table>

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 \( \text{MAXPTL} \leq 3 \), bits 63:2 of data written to the TL register using the WRPR instruction are ignored; only the least-significant two bits (bits 1:0) of TL are actually written. For example, if \( \text{MAXPTL} = 2 \), writing a value of 05 16 to the TL register causes a value of 1 16 to actually be stored in TL.

**Implementation Note** \( \text{MAXPTL} = 2 \) 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 privileged software to set \( \text{TL} > 0 \) for nonprivileged software\(^*\), an UltraSPARC Architecture virtual processor’s behavior when executing with \( \text{TL} > 0 \) in nonprivileged mode is undefined.

\(^*\) by executing a WRPR to TSTATE followed by DONE instruction or RETRY instruction.

### 5.7.8 Processor Interrupt Level (\( \text{PIL}^P \)) Register (PR 8) \(^{A1}\)

The privileged Processor Interrupt Level register (\( \text{PIL} \); see FIGURE 5-35) specifies the interrupt level above which the virtual processor will accept an \( \text{interrupt_level_}n \) 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 351 for a list of exception and interrupt priorities.

\[
\text{PIL}^P = \begin{array}{c}
\text{RW} \\
\text{pil} \\
0
\end{array}
\]

**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 (\( \text{GL}^P \)) (PR 16) \(^{A1}\)

The privileged Global Level (\( \text{GL} \)) 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](image)

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 0 to \( \text{MAXPGL} \), where \( \text{MAXPGL} \) is one fewer than the number of global register sets available to the virtual processor.

**IMPL. DEP. #401-S10:** The architectural parameter \( \text{MAXPGL} \) is a constant for each implementation; its legal values are from 2 to 7 (supporting from 3 to 8 sets of global registers). In a typical implementation, \( \text{MAXPGL} = \text{MAXPTL} \) (see impl. dep. #101-V9-CS10). Architecturally, \( \text{MAXPGL} \) must be \( \geq 2 \).

In all UltraSPARC Architecture 2007 implementations, \( \text{MAXPGL} = 2 \) (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 \( \text{MAXPGL} \) 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, \( \text{TL} = n \) does not imply that \( \text{GL} = n \), and \( \text{GL} = n \) does not imply that \( \text{TL} = n \). Furthermore, there may be a different total number of global levels (register sets) than there are trap levels; that is, \( \text{MAXPTL} \) and \( \text{MAXPGL} \) 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 \( \text{MAXPGL} \) to the GL register causes \( \text{MAXPGL} \) to be written to GL.

The effect of writing to GL with a WRPR instruction is summarized in **TABLE 5-20**.

**TABLE 5-20** Effect of WRPR to Register GL

<table>
<thead>
<tr>
<th>Value ( x ) Written with WRPR</th>
<th>Privilege Level when WRPR is Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \leq \text{MAXPGL} )</td>
<td>Nonprivileged</td>
</tr>
<tr>
<td>( x &gt; \text{MAXPGL} )</td>
<td>Privileged</td>
</tr>
</tbody>
</table>

\( \text{privilege_opcode} \) exception

\( \text{GL} \leftarrow x \)

\( \text{GL} \leftarrow \text{MAXPGL} \)

(no exception generated)

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.
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 71.
- Instruction Formats on page 72.
- Instruction Categories on page 72.

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 76) 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. The most typical 32-bit formats are shown in FIGURE 6-1. For detailed formats for specific instructions, see individual instruction descriptions in the Instructions chapter.

**op = 00<sub>2</sub>: SETHI, Branches, and ILLTRAP**

<table>
<thead>
<tr>
<th>00</th>
<th>rd</th>
<th>op2</th>
<th>imm22</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>a</td>
<td>cond</td>
<td>op2</td>
</tr>
<tr>
<td>00</td>
<td>a</td>
<td>cond</td>
<td>op2</td>
</tr>
<tr>
<td>00</td>
<td>a</td>
<td>0</td>
<td>rs1</td>
</tr>
<tr>
<td>00</td>
<td>a</td>
<td>0</td>
<td>rs1</td>
</tr>
</tbody>
</table>

**op = 01<sub>2</sub>: CALL**

<table>
<thead>
<tr>
<th>01</th>
<th>disp30</th>
</tr>
</thead>
</table>

**op = 10<sub>2</sub> or 11<sub>2</sub>: Arithmetic, Logical, Moves, Tcc, Loads, Stores, Prefetch, and Misc**

<table>
<thead>
<tr>
<th>1x</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>=0</th>
<th>imm_asi</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>rd</td>
<td>op3</td>
<td>rs1</td>
<td>=1</td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>

**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. LDFS/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.

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, STITWA) 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 \texttt{LDQF\_mem\_address\_not\_aligned} exception (impl. dep. \#111-V9-Cs10a).

\textbf{Implementation Note} Although the architecture provides for the \texttt{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 \texttt{STQF\_mem\_address\_not\_aligned} exception (impl. dep. \#112-V9-Cs10a).

\textbf{Implementation Note} Although the architecture provides for the \texttt{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 \textit{Processor State (pstateP) Register (PR 6)} on page 64 for more information.\footnote{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,” \textit{Computer} 14:10 (October 1981), pp. 48-54.}

**Big-endian Addressing Convention.** Within a multiple-byte integer, the byte with the smallest address is the most significant; a byte’s significance decreases as its address increases. The big-endian addressing conventions are described in TABLE 6-1 and illustrated in FIGURE 6-2.

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.</td>
</tr>
<tr>
<td>halfword</td>
<td>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.</td>
</tr>
<tr>
<td>word</td>
<td>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.</td>
</tr>
<tr>
<td>doubleword or</td>
<td>For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The most significant byte (bits 63:56) is accessed at the address specified in the instruction; the least significant byte (bits 7:0) is accessed at the address + 7. For the deprecated integer load/store twin word instructions (LDTW, LDTWA\footnote{Note that the LDTXA instruction, which is not an LDTWA operation but does share LDTWA’s opcode, is \textit{not} deprecated.}, 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.</td>
</tr>
<tr>
<td>extended word</td>
<td></td>
</tr>
<tr>
<td>quadword</td>
<td>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.</td>
</tr>
<tr>
<td>Byte</td>
<td>Address</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>Halfword</td>
<td>Address ( { 0 } ) =</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Address ( { 1:0 } ) =</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Doubleword / Extended word</td>
<td>Address ( { 2:0 } ) =</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Address ( { 2:0 } ) =</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Quadword</td>
<td>Address ( { 3:0 } ) =</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Address ( { 3:0 } ) =</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Address ( { 3:0 } ) =</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Address ( { 3:0 } ) =</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.</td>
</tr>
<tr>
<td>halfword</td>
<td>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.</td>
</tr>
<tr>
<td>word</td>
<td>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.</td>
</tr>
<tr>
<td>doubleword or extended word</td>
<td>For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 63–56) is accessed at the address + 7. For the deprecated integer load/store 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).</td>
</tr>
<tr>
<td>quadword</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Access Type</th>
<th>TL</th>
<th>PSTATE.cle</th>
<th>ASI Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>= 0</td>
<td>any</td>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td></td>
<td>&gt; 0</td>
<td>any</td>
<td>ASI_NUCLEUS*</td>
</tr>
</tbody>
</table>

†Note that the LDTXA instruction, which is not an LDTWA operation but does share LDTWA’s opcode, is not deprecated.
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.

Non-alternate-space Load, Store, or Load-Store = 0 0 ASI PRIMARY

<table>
<thead>
<tr>
<th>Access Type</th>
<th>TL</th>
<th>PSTATE.cle</th>
<th>ASI Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-alternate-space</td>
<td>0</td>
<td>0</td>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>Load, Store, or Load-Store</td>
<td>1</td>
<td>1</td>
<td>ASI PRIMARY_LITTLE</td>
</tr>
<tr>
<td></td>
<td>&gt; 0</td>
<td>0</td>
<td>ASI_NUCLEUS*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>ASI_NUCLEUS_LITTLE**</td>
</tr>
</tbody>
</table>

Alternate-space Load, any Store, or Load-Store

<table>
<thead>
<tr>
<th>Access Type</th>
<th>TL</th>
<th>PSTATE.cle</th>
<th>ASI Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>Any</td>
<td>Any</td>
<td>ASI explicitly specified in the instruction (subject to privilege-level restrictions)</td>
</tr>
</tbody>
</table>

*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.

**TABLE 6-3** ASIs Used for Data Accesses and Instruction Fetches

**FIGURE 6-3** Little-endian Addressing Conventions
See also Memory Addressing and Alternate Address Spaces on page 308.

ASIs $0_{16}$-$7_{16}$ are restricted; only software with sufficient privilege is allowed to access them. 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>
<thead>
<tr>
<th>Value</th>
<th>Access Type</th>
<th>Processor Mode (PSTATE.priv)</th>
<th>Result of ASI Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00_{16}$-$7_{16}$</td>
<td>Restricted</td>
<td>Nonprivileged (0)</td>
<td><em>privileged_action</em> exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Privileged (1)</td>
<td>Valid access</td>
</tr>
<tr>
<td>$80_{16}$-$FF_{16}$</td>
<td>Unrestricted</td>
<td>Nonprivileged (0)</td>
<td>Valid access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Privileged (1)</td>
<td>Valid access</td>
</tr>
</tbody>
</table>

**IMPL. DEP. #29-V8**: Some UltraSPARC Architecture 2007 ASIs are implementation dependent. See TABLE 10-1 on page 323 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 414 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 274 and Tagged Subtract on page 279 for details.

6.3.4 Control-Transfer Instructions (CTIs)

The basic control-transfer instruction types are as follows:

- Conditional branch (Bicc, BPcc, BPr, FBfcc, FBfcc)
- Unconditional branch
- Call and link (CALL)
- Jump and link (JMPI, 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>
<thead>
<tr>
<th>Instruction Group</th>
<th>Address Form</th>
<th>Delayed?</th>
<th>Taken?</th>
<th>Annul Bit?</th>
<th>New PC</th>
<th>New NPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-CTIs</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>NPC</td>
<td>NPC + 4</td>
</tr>
<tr>
<td>Bcc</td>
<td>PC-relative</td>
<td>Yes</td>
<td>Yes</td>
<td>0</td>
<td>NPC</td>
<td>EA</td>
</tr>
<tr>
<td>Bcc</td>
<td>PC-relative</td>
<td>Yes</td>
<td>No</td>
<td>0</td>
<td>NPC</td>
<td>NPC + 4</td>
</tr>
<tr>
<td>Bcc</td>
<td>PC-relative</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>NPC</td>
<td>EA</td>
</tr>
<tr>
<td>Bcc</td>
<td>PC-relative</td>
<td>Yes</td>
<td>No</td>
<td>1</td>
<td>NPC + 4</td>
<td>NPC + 8</td>
</tr>
<tr>
<td>BA</td>
<td>PC-relative</td>
<td>Yes</td>
<td>Yes</td>
<td>0</td>
<td>NPC</td>
<td>EA</td>
</tr>
<tr>
<td>BA</td>
<td>PC-relative</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>EA</td>
<td>EA + 4</td>
</tr>
<tr>
<td>BN</td>
<td>PC-relative</td>
<td>Yes</td>
<td>No</td>
<td>0</td>
<td>NPC</td>
<td>NPC + 4</td>
</tr>
<tr>
<td>BN</td>
<td>PC-relative</td>
<td>Yes</td>
<td>No</td>
<td>1</td>
<td>NPC + 4</td>
<td>NPC + 8</td>
</tr>
<tr>
<td>CALL</td>
<td>PC-relative</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>NPC</td>
<td>EA</td>
</tr>
</tbody>
</table>
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** — If \( i = 0 \), a register-indirect effective target address is \( R[rs1] + R[rs2] \). If \( i = 1 \), a register-indirect effective target address is \( R[rs1] + \text{sign_ext}(\text{simm13}) \).

- **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 \( R[rs1] + \text{imm_trap#} \) 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 mode. 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 356 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 annulling 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 annull 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.

---

**TABLE 6-5 Control-Transfer Characteristics (Continued) (2 of 2)**

<table>
<thead>
<tr>
<th>Instruction Group</th>
<th>Address Form</th>
<th>Delayed?</th>
<th>Taken?</th>
<th>Annul Bit?</th>
<th>New PC</th>
<th>New NPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMPL, RETURN</td>
<td>Register-indirect</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>NPC</td>
<td>EA</td>
</tr>
<tr>
<td>DONE</td>
<td>Trap state</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>TNPC[TL]</td>
<td>TNPC[TL] + 4</td>
</tr>
<tr>
<td>RETRY</td>
<td>Trap state</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>TPC[TL]</td>
<td>TNPC[TL]</td>
</tr>
<tr>
<td>Tcc</td>
<td>Trap vector</td>
<td>No</td>
<td>Yes</td>
<td>—</td>
<td>EA</td>
<td>EA + 4</td>
</tr>
<tr>
<td>Tcc</td>
<td>Trap vector</td>
<td>No</td>
<td>No</td>
<td>—</td>
<td>NPC</td>
<td>NPC + 4</td>
</tr>
</tbody>
</table>
The JMPL instruction writes the contents of the \texttt{PC}, which points to the JMPL instruction itself, into \texttt{R[rd]} and then causes a register-indirect delayed transfer of control to the address given by \texttt{“R[rs1] + R[rs2]”} or \texttt{“R[rs1] + a signed immediate value.”} The value written into \texttt{R[rd]} is visible to the instruction in the delay slot.

When \texttt{PSTATE.am = 1}, the value of the high-order 32 bits transmitted to \texttt{R[15]} by the CALL instruction or to \texttt{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 \texttt{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 \texttt{cond} field matches the current state of the condition code specified in its \texttt{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 \texttt{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 mode) least significant bits of the Tcc’s second source operand are written to \texttt{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, \textit{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

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, \textit{use of DCTI couples was deprecated.}
### 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:

```
movd %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>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NZ</td>
<td>Nonzero</td>
</tr>
<tr>
<td>Z</td>
<td>Zero</td>
</tr>
<tr>
<td>GEZ</td>
<td>Greater than or equal to zero</td>
</tr>
<tr>
<td>LZ</td>
<td>Less than zero</td>
</tr>
<tr>
<td>LEZ</td>
<td>Less than or equal to zero</td>
</tr>
<tr>
<td>GZ</td>
<td>Greater than zero</td>
</tr>
</tbody>
</table>

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, %i4, %i6
```

moves integer register $%i4$ to integer register $%i6$ 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.

### 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 58.
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).

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 239 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 232 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:

\[ N = \text{REG\_WINDOWS} - 2 - \text{CANSAVE} \]

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 48.

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 cond fields of the FSR.

The term “FPop” refers to instructions in the FPop1, FMAf, 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 83.

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 83.

If no floating-point unit is present or if \( \text{PSTATE}.\text{pef} = 0 \) or \( \text{FPRS}.\text{fef} = 0 \), then any instruction, including an FPop instruction, that attempts to access an FPU register generates an \( \text{fp\_disabled} \) 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 fccn 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 45 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 many VIS instructions. The remaining opcodes in IMPDEP1 and IMPDEP2 are now marked as reserved opcodes.

### 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” ("—").
UltraSPARC Architecture 2007 extends the standard SPARC V9 instruction set with additional classes of instructions:

- **Enhanced functionality:**
  - Instructions for alignment (*Align Address* on page 98)
  - Array handling (*Three-Dimensional Array Addressing* on page 101)
  - Byte-permutation instructions (*Byte Mask and Shuffle* on page 106)
  - Edge handling (*Edge Handling Instructions* on pages 116 and 118)
  - Logical operations on floating-point registers (*f Register Logical Operate (1 operand)* on page 163)
  - Partitioned arithmetic (*Fixed-point Partitioned Add* on page 158, *Fixed-point Partitioned Subtract (64-bit)* on page 161)
  - Pixel manipulation (*FEXPAND* on page 131, *FPACK* on page 153, and *FPMERGE* on page 160)

- **Efficient memory access**
  - Partial store (*Store Partial Floating-Point* on page 260)
  - Short floating-point loads and stores (*Store Short Floating-Point* on page 263)
  - Block load and store (*Block Load* on page 178 and *Block Store* on page 250)
  - Efficient interval arithmetic: SIAM (*Set Interval Arithmetic Mode* on page 243) 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 137)

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

<table>
<thead>
<tr>
<th>Superscript</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Deprecated instruction (do not use in new software)</td>
</tr>
<tr>
<td>N</td>
<td>Nonportable instruction</td>
</tr>
<tr>
<td>P</td>
<td>Privileged instruction</td>
</tr>
<tr>
<td>P&lt;sub&gt;ASl&lt;/sub&gt;</td>
<td>Privileged action if bit 7 of the referenced ASI is 0</td>
</tr>
<tr>
<td>P&lt;sub&gt;ASR&lt;/sub&gt;</td>
<td>Privileged instruction if the referenced ASR register is privileged</td>
</tr>
<tr>
<td>P&lt;sub&gt;opt&lt;/sub&gt;</td>
<td>Privileged action if in nonprivileged mode (<em>PSTATE.priv = 0</em>) and nonprivileged access is disabled</td>
</tr>
<tr>
<td>Page</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>97</td>
<td>ADD (ADDcc)</td>
</tr>
<tr>
<td>97</td>
<td>ADDC (ADDCcc)</td>
</tr>
<tr>
<td>97</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>ALIGNEDADDRESS[LITTLE]</td>
</tr>
<tr>
<td>99</td>
<td>ALLCLEAN</td>
</tr>
<tr>
<td>100</td>
<td>AND (ANDcc)</td>
</tr>
<tr>
<td>101</td>
<td>ARRAY&lt;8</td>
</tr>
<tr>
<td>104</td>
<td>Bicc</td>
</tr>
<tr>
<td>106</td>
<td>BMASK</td>
</tr>
<tr>
<td>107</td>
<td>BPcc</td>
</tr>
<tr>
<td>109</td>
<td>BPri</td>
</tr>
<tr>
<td>111</td>
<td>CALL</td>
</tr>
<tr>
<td>112</td>
<td>CASA&lt;Asi&gt;</td>
</tr>
<tr>
<td>112</td>
<td>CASXAsi</td>
</tr>
<tr>
<td>114</td>
<td>DONEP</td>
</tr>
<tr>
<td>116</td>
<td>EDGE&lt;8</td>
</tr>
<tr>
<td>118</td>
<td>EDGE&lt;8</td>
</tr>
<tr>
<td>168</td>
<td>F&lt;s</td>
</tr>
<tr>
<td>167</td>
<td>F&lt;s</td>
</tr>
<tr>
<td>167</td>
<td>F&lt;s</td>
</tr>
<tr>
<td>119</td>
<td>FABS&lt;s</td>
</tr>
<tr>
<td>120</td>
<td>FADD&lt;s</td>
</tr>
<tr>
<td>121</td>
<td>FALIGNDATA</td>
</tr>
<tr>
<td>165</td>
<td>FANDNOT&lt;1</td>
</tr>
<tr>
<td>165</td>
<td>FAND[s]</td>
</tr>
<tr>
<td>122</td>
<td>FBfccD</td>
</tr>
<tr>
<td>124</td>
<td>FBPfcc</td>
</tr>
<tr>
<td>128</td>
<td>FCMP&lt;s</td>
</tr>
<tr>
<td>126</td>
<td>FCMP&lt;16,32&gt;</td>
</tr>
<tr>
<td>128</td>
<td>FCMP&lt;s</td>
</tr>
<tr>
<td>130</td>
<td>FDIV&lt;s</td>
</tr>
<tr>
<td>Page</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>192</td>
<td>LDTWAD, PASI</td>
</tr>
<tr>
<td>194</td>
<td>LDTWA_D, PAsi</td>
</tr>
<tr>
<td>190</td>
<td>LDUB</td>
</tr>
<tr>
<td>176</td>
<td>LDUBA_PAsi</td>
</tr>
<tr>
<td>175</td>
<td>LDUH</td>
</tr>
<tr>
<td>176</td>
<td>LDUHA_PAsi</td>
</tr>
<tr>
<td>176</td>
<td>LDUWA_PAsi</td>
</tr>
<tr>
<td>175</td>
<td>LDUH</td>
</tr>
<tr>
<td>175</td>
<td>LDX</td>
</tr>
<tr>
<td>176</td>
<td>LDXA_PAsi</td>
</tr>
<tr>
<td>175</td>
<td>LDX</td>
</tr>
<tr>
<td>176</td>
<td>LDXFSR</td>
</tr>
<tr>
<td>201</td>
<td>MEMBAR</td>
</tr>
<tr>
<td>204</td>
<td>MOVcc</td>
</tr>
<tr>
<td>204</td>
<td>MOVr</td>
</tr>
<tr>
<td>207</td>
<td>MOVr</td>
</tr>
<tr>
<td>209</td>
<td>MULScD</td>
</tr>
<tr>
<td>211</td>
<td>MULX</td>
</tr>
<tr>
<td>212</td>
<td>NOP</td>
</tr>
<tr>
<td>213</td>
<td>NORMALW</td>
</tr>
<tr>
<td>214</td>
<td>OR (ORcc)</td>
</tr>
<tr>
<td>214</td>
<td>ORN (ORNcc)</td>
</tr>
<tr>
<td>215</td>
<td>OTHERW</td>
</tr>
<tr>
<td>216</td>
<td>PDIST</td>
</tr>
<tr>
<td>217</td>
<td>POPC</td>
</tr>
<tr>
<td>219</td>
<td>PREFETCH</td>
</tr>
<tr>
<td>219</td>
<td>PREFETCHA_PAsi</td>
</tr>
<tr>
<td>225</td>
<td>RDAI</td>
</tr>
<tr>
<td>225</td>
<td>RDasr_PAsr</td>
</tr>
<tr>
<td>225</td>
<td>RDCCR</td>
</tr>
<tr>
<td>225</td>
<td>RDFPRS</td>
</tr>
<tr>
<td>225</td>
<td>RDGSR</td>
</tr>
</tbody>
</table>
TABLE 7-2  UltraSPARC Architecture 2007 Instruction Set - Alphabetical  (3 of 3)

<table>
<thead>
<tr>
<th>Page</th>
<th>Instruction</th>
<th>Page</th>
<th>Instruction</th>
<th>Page</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>285</td>
<td>WRSOFTINT_CLR(^p)</td>
<td>285</td>
<td>WRSTICK(^p)</td>
<td>290</td>
<td>XNOR (XNORcc)</td>
</tr>
<tr>
<td>285</td>
<td>WRSOFTINT_SET(^p)</td>
<td>285</td>
<td>WRTICK_CMPRP(^p)</td>
<td>290</td>
<td>XOR (XORcc)</td>
</tr>
<tr>
<td>285</td>
<td>WRSOFTINT(^p)</td>
<td>285</td>
<td>WRY(^D)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>285</td>
<td>WRSTICK_CMPRP(^p)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Category and Function</td>
<td>Page</td>
<td>Ext. to V8?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------</td>
<td>------</td>
<td>------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVcc</td>
<td>Move integer register if condition is satisfied</td>
<td>204</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVr</td>
<td>Move integer register on contents of integer register</td>
<td>207</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMOV&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point move</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>FMOV&lt;s</td>
<td>d</td>
<td>q&gt;cc</td>
<td>Move floating-point register if condition is satisfied</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>FMOV&lt;s</td>
<td>d</td>
<td>q&gt;R</td>
<td>Move f-p reg. if integer reg. contents satisfy condition</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>FSRC&lt;1</td>
<td>2&gt;[s]</td>
<td>Copy source</td>
<td>164</td>
<td>VIS 1</td>
<td></td>
</tr>
<tr>
<td>FiTO&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Convert 32-bit integer to floating-point</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>F&lt;s</td>
<td>d</td>
<td>q&gt;TOi</td>
<td>Convert floating point to integer</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>F&lt;s</td>
<td>d</td>
<td>q&gt;TOx</td>
<td>Convert floating point to 64-bit integer</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>F&lt;s</td>
<td>d</td>
<td>q&gt;TO&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Convert between floating-point formats</td>
</tr>
<tr>
<td>FxTO&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Convert 64-bit integer to floating-point</td>
<td>171</td>
<td></td>
</tr>
<tr>
<td>AND (ANDcc)</td>
<td>Logical and (and modify condition codes)</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR (ORcc)</td>
<td>Inclusive-or (and modify condition codes)</td>
<td>214</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORN (ORNcc)</td>
<td>Inclusive-or not (and modify condition codes)</td>
<td>214</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XNOR (XNORcc)</td>
<td>Exclusive-nor (and modify condition codes)</td>
<td>290</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR (XORcc)</td>
<td>Exclusive-or (and modify condition codes)</td>
<td>290</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAND[s]</td>
<td>Logical and operation</td>
<td>165</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FANDNOT&lt;1</td>
<td>2&gt;[s]</td>
<td>Logical and operation with one inverted source</td>
<td>165</td>
<td>VIS 1</td>
<td></td>
</tr>
<tr>
<td>FNAND[s]</td>
<td>Logical and operation</td>
<td>165</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FNOR[s]</td>
<td>Logical nor operation</td>
<td>165</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FNOT&lt;1</td>
<td>2&gt;[s]</td>
<td>Copy negated source</td>
<td>164</td>
<td>VIS 1</td>
<td></td>
</tr>
<tr>
<td>FONE[s]</td>
<td>One fill</td>
<td>163</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR[s]</td>
<td>Logical or operation</td>
<td>165</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORNOT&lt;1</td>
<td>2&gt;[s]</td>
<td>Logical or operation with one inverted source</td>
<td>165</td>
<td>VIS 1</td>
<td></td>
</tr>
<tr>
<td>FXNOR[s]</td>
<td>Logical xnor operation</td>
<td>165</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FXOR[s]</td>
<td>Logical xor operation</td>
<td>165</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FZERO[s]</td>
<td>Zero fill</td>
<td>163</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLL</td>
<td>Shift left logical</td>
<td>244</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLLX</td>
<td>Shift left logical, extended</td>
<td>244</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRA</td>
<td>Shift right arithmetic</td>
<td>244</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRAX</td>
<td>Shift right arithmetic, extended</td>
<td>244</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRL</td>
<td>Shift right logical</td>
<td>244</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRLX</td>
<td>Shift right logical, extended</td>
<td>244</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIGNEDADDRESS_[LITTLE]</td>
<td>Calculate address for misaligned data</td>
<td>98</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAY&lt;8</td>
<td>16</td>
<td>32&gt;</td>
<td>3-D array addressing instructions</td>
<td>101</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FALIGNDATA</td>
<td>Perform data alignment for misaligned data</td>
<td>121</td>
<td>VIS 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bicc</td>
<td>Branch on integer condition codes</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPcc</td>
<td>Branch on integer condition codes with prediction</td>
<td>107</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Category and Function</td>
<td>Page</td>
<td>Ext. to V9?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------</td>
<td>------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPr</td>
<td>Branch on contents of integer register with prediction</td>
<td>109</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>Call and link</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DONEp</td>
<td>Return from trap</td>
<td>114</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FBfccD</td>
<td>Branch on floating-point condition codes</td>
<td>122</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FBPfcc</td>
<td>Branch on floating-point condition codes with prediction</td>
<td>124</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILLTRAP</td>
<td>Illegal instruction</td>
<td>172</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMPL</td>
<td>Jump and link</td>
<td>174</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETRYp</td>
<td>Return from trap and retry</td>
<td>233</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>Return</td>
<td>235</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcc</td>
<td>Trap on integer condition codes</td>
<td>276</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Byte Permutation**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMASK</td>
<td>Set the GSR.mask field</td>
<td>106</td>
<td>VIS 2</td>
</tr>
<tr>
<td>BSHUFFLE</td>
<td>Permute bytes as specified by GSR.mask</td>
<td>106</td>
<td>VIS 2</td>
</tr>
</tbody>
</table>

**Data Formatting Operations on F Registers**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEXPAND</td>
<td>Pixel expansion</td>
<td>131</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPACK&lt;16</td>
<td>32</td>
<td>FIX&gt;</td>
<td>Pixel packing</td>
</tr>
<tr>
<td>FPMERGE</td>
<td>Pixel merge</td>
<td>160</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

**Memory Operations to/from F Registers**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBLOCKF</td>
<td>Block loads</td>
<td>178</td>
<td>VIS 1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>Block stores</td>
<td>250</td>
<td>VIS 1</td>
</tr>
<tr>
<td>LDDF</td>
<td>Load double floating-point</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>LDDFApAsi</td>
<td>Load double floating-point from alternate space</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>LDF</td>
<td>Load floating-point</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>LDFApAsi</td>
<td>Load floating-point from alternate space</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>LDQF</td>
<td>Load quad floating-point</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>LDQFApAsi</td>
<td>Load quad floating-point from alternate space</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>Short floating-point loads</td>
<td>188</td>
<td>VIS 1</td>
</tr>
<tr>
<td>STDF</td>
<td>Store double floating-point</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>STDFApAsi</td>
<td>Store double floating-point into alternate space</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>STF</td>
<td>Store floating-point</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>STFApAsi</td>
<td>Store floating-point into alternate space</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>STPARITALF</td>
<td>Partial Store instructions</td>
<td>260</td>
<td>VIS 1</td>
</tr>
<tr>
<td>STQF</td>
<td>Store quad floating point</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>STQFApAsi</td>
<td>Store quad floating-point into alternate space</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>STSHORTF</td>
<td>Short floating-point stores</td>
<td>263</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

**Memory Operations — Miscellaneous**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDFSrD</td>
<td>Load floating-point state register (lower)</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>LDXFSR</td>
<td>Load floating-point state register</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>MEMBAR</td>
<td>Memory barrier</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>PREFETCH</td>
<td>Prefetch data</td>
<td>219</td>
<td></td>
</tr>
<tr>
<td>PREFETCHApAsi</td>
<td>Prefetch data from alternate space</td>
<td>219</td>
<td></td>
</tr>
<tr>
<td>STFSrD</td>
<td>Store floating-point state register (lower)</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>STXFSR</td>
<td>Store floating-point state register</td>
<td>269</td>
<td></td>
</tr>
</tbody>
</table>

**Atomic (Load-Store) Memory Operations to/from R Registers**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASApAsi</td>
<td>Compare and swap word in alternate space</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>CASXApAsi</td>
<td>Compare and swap doubleword in alternate space</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Category and Function</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>LDSTUB</td>
<td>Load-store unsigned byte</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>LDSTUBPASI</td>
<td>Load-store unsigned byte in alternate space</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>SWAPD</td>
<td>Swap integer register with memory</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>SWAPD, PASI</td>
<td>Swap integer register with memory in alternate space</td>
<td>272</td>
<td></td>
</tr>
</tbody>
</table>

**Memory Operations to/from R Registers**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDSB</td>
<td>Load signed byte</td>
<td>175</td>
</tr>
<tr>
<td>LDSBAPASI</td>
<td>Load signed byte from alternate space</td>
<td>176</td>
</tr>
<tr>
<td>LDSH</td>
<td>Load signed halfword</td>
<td>175</td>
</tr>
<tr>
<td>LDSHAPASI</td>
<td>Load signed halfword from alternate space</td>
<td>176</td>
</tr>
<tr>
<td>LSDW</td>
<td>Load signed word</td>
<td>175</td>
</tr>
<tr>
<td>LDSWAAPASI</td>
<td>Load signed word from alternate space</td>
<td>176</td>
</tr>
<tr>
<td>LDTXAN</td>
<td>Load integer twin extended word from alternate space</td>
<td>197</td>
</tr>
<tr>
<td>LDTWD, PASI</td>
<td>Load integer twin word</td>
<td>192</td>
</tr>
<tr>
<td>LDTWDA, PASI</td>
<td>Load integer twin word from alternate space</td>
<td>194</td>
</tr>
<tr>
<td>LDUB</td>
<td>Load unsigned byte</td>
<td>190</td>
</tr>
<tr>
<td>LDUBA, PASI</td>
<td>Load unsigned byte from alternate space</td>
<td>176</td>
</tr>
<tr>
<td>LDUH</td>
<td>Load unsigned halfword</td>
<td>175</td>
</tr>
<tr>
<td>LDUHAPASI</td>
<td>Load unsigned halfword from alternate space</td>
<td>176</td>
</tr>
<tr>
<td>LDUW</td>
<td>Load unsigned word</td>
<td>175</td>
</tr>
<tr>
<td>LDUWAAPASI</td>
<td>Load unsigned word from alternate space</td>
<td>176</td>
</tr>
<tr>
<td>LDX</td>
<td>Load extended</td>
<td>175</td>
</tr>
<tr>
<td>LDXA, PASI</td>
<td>Load extended from alternate space</td>
<td>176</td>
</tr>
<tr>
<td>STB</td>
<td>Store byte</td>
<td>247</td>
</tr>
<tr>
<td>STBAPASI</td>
<td>Store byte into alternate space</td>
<td>248</td>
</tr>
<tr>
<td>STTWD</td>
<td>Store twin word</td>
<td>265</td>
</tr>
<tr>
<td>STTWA, PASI</td>
<td>Store twin word into alternate space</td>
<td>267</td>
</tr>
<tr>
<td>STH</td>
<td>Store halfword</td>
<td>247</td>
</tr>
<tr>
<td>STA, PASI</td>
<td>Store halfword into alternate space</td>
<td>248</td>
</tr>
<tr>
<td>STW</td>
<td>Store word</td>
<td>247</td>
</tr>
<tr>
<td>STWA, PASI</td>
<td>Store word into alternate space</td>
<td>248</td>
</tr>
<tr>
<td>STX</td>
<td>Store extended</td>
<td>247</td>
</tr>
<tr>
<td>STXA, PASI</td>
<td>Store extended into alternate space</td>
<td>248</td>
</tr>
</tbody>
</table>

**Floating-Point Arithmetic Operations**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABS&lt;s</td>
<td>d</td>
<td>q&gt;</td>
</tr>
<tr>
<td>FADD&lt;s</td>
<td>d</td>
<td>q&gt;</td>
</tr>
<tr>
<td>FDIV&lt;s</td>
<td>d</td>
<td>q&gt;</td>
</tr>
<tr>
<td>FdMULq</td>
<td>Floating-point multiply double to quad</td>
<td>151</td>
</tr>
<tr>
<td>FMADD(s,d)</td>
<td>Floating-point multiply-add single/double (fused)</td>
<td>137</td>
</tr>
<tr>
<td>FMSUB(s,d)</td>
<td>Floating-point multiply-subtract single/double (fused)</td>
<td>137</td>
</tr>
<tr>
<td>FMUL&lt;s</td>
<td>d</td>
<td>q&gt;</td>
</tr>
<tr>
<td>FNMADD(s,d)</td>
<td>Floating-point negative multiply-add single/double (fused)</td>
<td>137</td>
</tr>
<tr>
<td>FNEG&lt;s</td>
<td>d</td>
<td>q&gt;</td>
</tr>
<tr>
<td>FNMSUB(s,d)</td>
<td>Floating-point negative multiply-subtract single/double (fused)</td>
<td>137</td>
</tr>
<tr>
<td>FsMULD</td>
<td>Floating-point multiply single to double</td>
<td>151</td>
</tr>
<tr>
<td>FSQRT&lt;s</td>
<td>d</td>
<td>q&gt;</td>
</tr>
<tr>
<td>FSUB&lt;s</td>
<td>d</td>
<td>q&gt;</td>
</tr>
</tbody>
</table>
### TABLE 7-3 Instruction Set - by Functional Category (4 of 5)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Floating-Point Comparison Operations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMP&lt;16,32&gt;</td>
<td>Compare four 16-bit signed values or two 32-bit signed values</td>
<td>126</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FCMP&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point compare</td>
</tr>
<tr>
<td>FCMPE&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point compare (exception if unordered)</td>
</tr>
<tr>
<td><strong>Register-Window Control Operations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALLCLEAN</td>
<td>Mark all register window sets as “clean”</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>INVALW</td>
<td>Mark all register window sets as “invalid”</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>FLUSHW</td>
<td>Flush register windows</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>NORMALW</td>
<td>“Normal” register windows become “normal” register windows</td>
<td>213</td>
<td></td>
</tr>
<tr>
<td>OTHERW</td>
<td>“Other” register windows become “other” register windows</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>RESTORE</td>
<td>Restore caller’s window</td>
<td>230</td>
<td></td>
</tr>
<tr>
<td>RESTOREd</td>
<td>Window has been restored</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>SAVE</td>
<td>Save caller’s window</td>
<td>237</td>
<td></td>
</tr>
<tr>
<td>SAVED</td>
<td>Window has been saved</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous Operations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td>Flush instruction memory</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>212</td>
<td></td>
</tr>
<tr>
<td><strong>Integer SIMD Operations on F Registers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPADD&lt;16,32&gt;[S]</td>
<td>Fixed-point partitioned add</td>
<td>158</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPSUB&lt;16,32&gt;[S]</td>
<td>Fixed-point partitioned subtract</td>
<td>161</td>
<td>VIS 1</td>
</tr>
<tr>
<td><strong>Integer Arithmetic Operations on R Registers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD (ADDcc)</td>
<td>Add (and modify condition codes)</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>ADDC (ADDCcc)</td>
<td>Add with carry (and modify condition codes)</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>MULSccD</td>
<td>Multiply step (and modify condition codes)</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>MULX</td>
<td>Multiply 64-bit integers</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>SDIVD (SDIVccD)</td>
<td>32-bit signed integer divide (and modify condition codes)</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>SDIVX</td>
<td>64-bit signed integer divide</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>SMULD (SMULccD)</td>
<td>Signed integer multiply (and modify condition codes)</td>
<td>246</td>
<td></td>
</tr>
<tr>
<td>SUB (SUBcc)</td>
<td>Subtract (and modify condition codes)</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>SUBC (SUBcc)</td>
<td>Subtract with carry (and modify condition codes)</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>TADDcc</td>
<td>Tagged add and modify condition codes (trap on overflow)</td>
<td>274</td>
<td></td>
</tr>
<tr>
<td>TADDccTVD</td>
<td>Tagged add and modify condition codes (trap on overflow)</td>
<td>275</td>
<td></td>
</tr>
<tr>
<td>TSUBcc</td>
<td>Tagged subtract and modify condition codes (trap on overflow)</td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>TSUBccTVD</td>
<td>Tagged subtract and modify condition codes (trap on overflow)</td>
<td>280</td>
<td></td>
</tr>
<tr>
<td>UDIVD (UDIVccD)</td>
<td>Unsigned integer divide (and modify condition codes)</td>
<td>281</td>
<td></td>
</tr>
<tr>
<td>UDIVX</td>
<td>64-bit unsigned integer divide</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>UMULD (UMULccD)</td>
<td>Unsigned integer multiply (and modify condition codes)</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td><strong>Integer Arithmetic Operations on F Registers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMULx16</td>
<td>8x16 partitioned product</td>
<td>146</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FMUL8x16[AU</td>
<td>AL]</td>
<td>8x16 upper/lower or partitioned product</td>
<td>146</td>
</tr>
<tr>
<td>FMUL8[xUL]x16</td>
<td>8x16 upper/lower partitioned product</td>
<td>146</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FMULD8[xUL]x16</td>
<td>8x16 upper/lower partitioned product</td>
<td>146</td>
<td>VIS 1</td>
</tr>
<tr>
<td><strong>Miscellaneous Operations on R Registers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPC</td>
<td>Population count</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>SETHI</td>
<td>Set high 22 bits of low word of integer register</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous Operations on F Registers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 7-3  Instruction Set - by Functional Category (5 of 5)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDGE&lt;8</td>
<td>16</td>
<td>32&gt;-[L]cc</td>
<td>Edge handling instructions (and modify condition codes)</td>
</tr>
<tr>
<td>EDGE&lt;8</td>
<td>16</td>
<td>32&gt;-[L]N</td>
<td>Edge handling instructions</td>
</tr>
<tr>
<td>PDIST</td>
<td>Pixel component distance</td>
<td>216</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

#### Control and Status Register Access

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDASI</td>
<td>Read ASI register</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDasrPASR</td>
<td>Read ancillary state register</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDCCR</td>
<td>Read Condition Codes register (CCR)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDFPRS</td>
<td>Read Floating-Point Registers State register (FPRS)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDGSR</td>
<td>Read General Status register (GSR)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDPC</td>
<td>Read Program Counter register (PC)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDPRP</td>
<td>Read privileged register</td>
<td>228</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDSOFTINTP</td>
<td>Read per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDSTICK</td>
<td>Read System Tick register (STICK)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDSTICK_CMPRP</td>
<td>Read System Tick Compare register (STICK_CMPR)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDTICK</td>
<td>Read Tick register (TICK)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>RDTICK_CMPRP</td>
<td>Read Tick Compare register (TICK_CMPR)</td>
<td>225</td>
<td>VIS 1</td>
</tr>
<tr>
<td>SIAM</td>
<td>Set interval arithmetic mode</td>
<td>243</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRASI</td>
<td>Write ASI register</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRasrPASR</td>
<td>Write ancillary state register</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WrCCR</td>
<td>Write Condition Codes register (CCR)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRFPFRS</td>
<td>Write Floating-Point Registers State register (FPRS)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRGSR</td>
<td>Write General Status register (GSR)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRPRP</td>
<td>Write privileged register</td>
<td>288</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRSOFTINTP</td>
<td>Write per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRSOFTINT_CLR</td>
<td>Clear bits of per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRSOFTINT_SET</td>
<td>Set bits of per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRTICK_CMPRP</td>
<td>Write Tick Compare register (TICK_CMPR)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRTICK</td>
<td>Write System Tick register (STICK)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRTICK_CMPRP</td>
<td>Write System Tick Compare register (TICK_CMPR)</td>
<td>285</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRYD</td>
<td>Write Y register</td>
<td>285</td>
<td>VIS 2</td>
</tr>
</tbody>
</table>
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 86.

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_*`, 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 356), 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>00000</td>
<td>Add</td>
<td>add rs[rs1], reg_or_imm, reg[rd]</td>
<td>A1</td>
</tr>
<tr>
<td>ADDcc</td>
<td>01000</td>
<td>Add and modify cc’s</td>
<td>addcc rs[rs1], reg_or_imm, reg[rd]</td>
<td>A1</td>
</tr>
<tr>
<td>ADDC</td>
<td>00100</td>
<td>Add with 32-bit Carry</td>
<td>addc rs[rs1], reg_or_imm, reg[rd]</td>
<td>A1</td>
</tr>
<tr>
<td>ADDCc</td>
<td>01100</td>
<td>Add with 32-bit Carry and modify cc’s</td>
<td>addccc rs[rs1], reg_or_imm, reg[rd]</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

If \( i = 0 \), ADD and ADDcc compute “\( R[rs1] + R[rs2] \)”. If \( i = 1 \), they compute “\( R[rs1] + \text{sign_ext}(\text{simm13}) \)”. In either case, the sum is written to \( R[rd] \).

ADDC and ADDCc (“ADD with carry”) also add the CCR register’s 32-bit carry (icc.c) bit. That is, if \( i = 0 \), they compute “\( R[rs1] + R[rs2] + \text{icc.c} \)” and if \( i = 1 \), they compute “\( R[rs1] + \text{sign_ext}(\text{simm13}) + \text{icc.c} \)”.

In either case, the sum is written to \( R[rd] \).

ADDC and ADDCc 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 ADDCc read the 32-bit condition codes’ carry bit (CCR.icc), not the 64-bit condition codes’ carry bit (CCR.xcc).

**SPARC V8 Compatibility Note**

ADDC and ADDCc were previously named ADDX and ADDXcc, respectively, in SPARC V8.

An attempt to execute an ADD, ADDcc, ADDC or ADDCc instruction when \( i = 0 \) and reserved instruction bits 12:5 are nonzero causes an illegal_instruction exception.

**Exceptions**

illegal_instruction
7.2 Align Address

**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.

```assembly
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 121
7.3 Mark All Register Window Sets “Clean”

The ALLCLEAN instruction marks all register window sets as “clean”; specifically, it performs the following operation:

\[
\text{CLEANWIN} \leftarrow (N_{\text{REG\_WINDOWS}} - 1)
\]

**Description**

The ALLCLEAN instruction marks all register window sets as “clean”; specifically, it performs the following operation:

\[
\text{CLEANWIN} \leftarrow (N_{\text{REG\_WINDOWS}} - 1)
\]

**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 \(N_{\text{REG\_WINDOWS}}\) is not known to privileged software.

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 \((\text{PSTATE}.\text{priv} = 0)\) causes a *privileged_opcode* exception.

**Exceptions**

- illegal_instruction
- privileged_opcode

**See Also**

INVALW on page 173
NORMALW on page 213
OTHERW on page 215
RESTORED on page 232
SAVED on page 239
7.4 AND Logical Operation

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

**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 × 64, M = m × 32, 0 ≤ n ≤ 5, 1 ≤ m ≤ 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.

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.

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.
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

<table>
<thead>
<tr>
<th>Result (R[rd]) Bits</th>
<th>Source (R[rs1]) Bits</th>
<th>Field Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>12:11</td>
<td>X_integer[1:0]</td>
</tr>
<tr>
<td>3:2</td>
<td>34:33</td>
<td>Y_integer[1:0]</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
<td>Z_integer[0]</td>
</tr>
<tr>
<td>8:5</td>
<td>16:13</td>
<td>X_integer[5:2]</td>
</tr>
<tr>
<td>12:9</td>
<td>38:35</td>
<td>Y_integer[5:2]</td>
</tr>
<tr>
<td>16:13</td>
<td>59:56</td>
<td>Z_integer[4:1]</td>
</tr>
<tr>
<td>17+n-1:17</td>
<td>17+n-1:17</td>
<td>X_integer[6+n-1:6]</td>
</tr>
<tr>
<td>17+2n-1:17+n</td>
<td>39+n-1:39</td>
<td>Y_integer[6+n-1:6]</td>
</tr>
<tr>
<td>20+2n:17+2n</td>
<td>63:60</td>
<td>Z_integer[8:5]</td>
</tr>
<tr>
<td>63:20+2n+1</td>
<td>n/a</td>
<td>0</td>
</tr>
</tbody>
</table>

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.

```assembly
add     Addr, DeltaAddr, Addr
array8  Addr, %g0, bAddr
lda     (bAddr) #ASI_FL8_PRIMARY, data
faligndata  data, accum, accum
```

**Exceptions**  None
7.6 Branch on Integer Condition Codes (Bicc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>cond</th>
<th>Operation</th>
<th>icc Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA</td>
<td>0000</td>
<td>Branch Always</td>
<td>1</td>
<td>ba{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BN</td>
<td>0000</td>
<td>Branch Never</td>
<td>0</td>
<td>bn{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BNE</td>
<td>0001</td>
<td>Branch on Not Equal</td>
<td>not Z</td>
<td>bne{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BE</td>
<td>0001</td>
<td>Branch on Equal</td>
<td>Z</td>
<td>be{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BG</td>
<td>0101</td>
<td>Branch on Greater</td>
<td>(Z or (N xor V))</td>
<td>bg{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BLE</td>
<td>0010</td>
<td>Branch on Less or Equal</td>
<td>Z or (N xor V)</td>
<td>ble{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BGE</td>
<td>0111</td>
<td>Branch on Greater or Equal</td>
<td>not (N xor V)</td>
<td>bge{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BL</td>
<td>0011</td>
<td>Branch on Less</td>
<td>N xor V</td>
<td>bl{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BGU</td>
<td>1100</td>
<td>Branch on Greater Unsigned</td>
<td>not (C or Z)</td>
<td>bgu{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BLEU</td>
<td>0100</td>
<td>Branch on Less or Equal Unsigned</td>
<td>C or Z</td>
<td>bleu{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BCC</td>
<td>1101</td>
<td>Branch on Carry Clear (Greater Than, Unsigned)</td>
<td>not C</td>
<td>bcc{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BCS</td>
<td>0101</td>
<td>Branch on Carry Set (Less Than, Unsigned)</td>
<td>C</td>
<td>bcs{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BPOS</td>
<td>1110</td>
<td>Branch on Positive</td>
<td>not N</td>
<td>bpos{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BNEG</td>
<td>0110</td>
<td>Branch on Negative</td>
<td>N</td>
<td>bneg{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BVC</td>
<td>1111</td>
<td>Branch on Overflow Clear</td>
<td>not V</td>
<td>bvc{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BVS</td>
<td>0111</td>
<td>Branch on Overflow Set</td>
<td>V</td>
<td>bvs{,a} label</td>
<td>A1</td>
</tr>
</tbody>
</table>

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.
Bicc

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), 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

**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].

<table>
<thead>
<tr>
<th>Destination Byte (in F_D[rd])</th>
<th>Source Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (most significant)</td>
<td>(F_D[rs1] :: F_D[rs2]) [GSR.mask[31:28]]</td>
</tr>
<tr>
<td>1</td>
<td>(F_D[rs1] :: F_D[rs2]) [GSR.mask[27:24]]</td>
</tr>
<tr>
<td>2</td>
<td>(F_D[rs1] :: F_D[rs2]) [GSR.mask[23:20]]</td>
</tr>
<tr>
<td>3</td>
<td>(F_D[rs1] :: F_D[rs2]) [GSR.mask[19:16]]</td>
</tr>
<tr>
<td>4</td>
<td>(F_D[rs1] :: F_D[rs2]) [GSR.mask[15:12]]</td>
</tr>
<tr>
<td>5</td>
<td>(F_D[rs1] :: F_D[rs2]) [GSR.mask[11:8]]</td>
</tr>
<tr>
<td>6</td>
<td>(F_D[rs1] :: F_D[rs2]) [GSR.mask[7:4]]</td>
</tr>
<tr>
<td>7 (least significant)</td>
<td>(F_D[rs1] :: F_D[rs2]) [GSR.mask[3:0]]</td>
</tr>
</tbody>
</table>

If the floating-point unit is not enabled (FPRS.lef = 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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>cc Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPA 1000</td>
<td>Branch Always</td>
<td>1</td>
<td>ba,[a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPN 0000</td>
<td>Branch Never</td>
<td>0</td>
<td>bn,[a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPNE 1001</td>
<td>Branch on Not Equal</td>
<td>not Z</td>
<td>bne,[a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPE 0001</td>
<td>Branch on Equal</td>
<td>Z</td>
<td>be,[a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPG 1010</td>
<td>Branch on Greater</td>
<td>not (Z or (N xor V))</td>
<td>bg[ ]a,pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPLE 0010</td>
<td>Branch on Less or Equal</td>
<td>Z or (N xor V)</td>
<td>ble,[a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPGE 1011</td>
<td>Branch on Greater or Equal</td>
<td>not (N xor V)</td>
<td>bg[e],[a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPE 0011</td>
<td>Branch on Equal</td>
<td>N xor V</td>
<td>ble,[a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPGE 1100</td>
<td>Branch on Greater Unsigned</td>
<td>not (C or Z)</td>
<td>bgu,[a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPL 0001</td>
<td>Branch on Less</td>
<td>N or (N xor V)</td>
<td>ble[,a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPGU 1101</td>
<td>Branch on Greater Unsigned</td>
<td>not C</td>
<td>bgu[ ]a,pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPCC 1110</td>
<td>Branch on Carry Clear</td>
<td>C</td>
<td>bcs[ ]a,pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPC 0110</td>
<td>Branch on Carry Set</td>
<td>C</td>
<td>bcs[ ]a,pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPPOS 1110</td>
<td>Branch on Positive</td>
<td>not N</td>
<td>bpos[,a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPNEG 0110</td>
<td>Branch on Negative</td>
<td>N</td>
<td>bneg[,a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPVC 1111</td>
<td>Branch on Overflow Clear</td>
<td>not V</td>
<td>bvc[,a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPCS 1111</td>
<td>Branch on Carry Set (Less than, Unsigned)</td>
<td>C</td>
<td>bcs[,a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
<tr>
<td>BPVC 1111</td>
<td>Branch on Overflow Set</td>
<td>V</td>
<td>bvs[,a],pt, pn</td>
<td>i_or_x_cc, label</td>
</tr>
</tbody>
</table>

† synonym: bnz  ‡ synonym: bz  ◊ synonym: bgeu  ∇ synonym: blu

<table>
<thead>
<tr>
<th>00</th>
<th>a</th>
<th>cond</th>
<th>001</th>
<th>cc1</th>
<th>cc0</th>
<th>p</th>
<th>disp19</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>25</td>
<td>24</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>cc1</td>
<td>cc0</td>
<td>Condition Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>icc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>xcc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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), 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 109
7.9 Branch on Integer Register with Prediction (BPr)

Although SPARC V9 implementations should cause an illegal_instruction exception when bit 28 = 1, some early implementations ignored the value of this bit and executed the opcode as a BPr instruction even if bit 28 = 1.

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 (0002 or 1002) causes an illegal_instruction exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), 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.

### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rcond</th>
<th>Operation</th>
<th>Register Contents Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRZ</td>
<td>001</td>
<td>Branch on Register Zero</td>
<td>R[rs1] = 0</td>
<td>brz {a},pt, label</td>
<td>A1</td>
</tr>
<tr>
<td>BRLEZ</td>
<td>010</td>
<td>Branch on Register Less Than or Equal to Zero</td>
<td>R[rs1] ≤ 0</td>
<td>brelz {a},pt, label</td>
<td>A1</td>
</tr>
<tr>
<td>BRLZ</td>
<td>011</td>
<td>Branch on Register Less Than Zero</td>
<td>R[rs1] &lt; 0</td>
<td>brlz {a},pt, label</td>
<td>A1</td>
</tr>
<tr>
<td>BRNZ</td>
<td>101</td>
<td>Branch on Register Not Zero</td>
<td>R[rs1] ≠ 0</td>
<td>brnz {a},pt, label</td>
<td>A1</td>
</tr>
<tr>
<td>BRGZ</td>
<td>110</td>
<td>Branch on Register Greater Than Zero</td>
<td>R[rs1] &gt; 0</td>
<td>brgz {a},pt, label</td>
<td>A1</td>
</tr>
<tr>
<td>BRGEZ</td>
<td>111</td>
<td>Branch on Register Greater Than or Equal to Zero</td>
<td>R[rs1] ≥ 0</td>
<td>brgez {a},pt,pn, label</td>
<td>A1</td>
</tr>
</tbody>
</table>

* Although SPARC V9 implementations should cause an illegal_instruction exception when bit 28 = 1, some 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, %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 (0002 or 1002) causes an illegal_instruction exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), 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 \( r_{\text{cond}} \) is TRUE:

<table>
<thead>
<tr>
<th>Branch</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRNZ</td>
<td>not Z</td>
</tr>
<tr>
<td>BRZ</td>
<td>Z</td>
</tr>
<tr>
<td>BRGEZ</td>
<td>not N</td>
</tr>
<tr>
<td>BRLZ</td>
<td>N</td>
</tr>
<tr>
<td>BRLEZ</td>
<td>N or Z</td>
</tr>
<tr>
<td>BRGZ</td>
<td>not (N or Z)</td>
</tr>
</tbody>
</table>

**Exceptions**

illegal_instruction

control_transfer_instruction (impl. dep. #450-S20)

**See Also**

Branch on Integer Condition Codes with Prediction (BPcc) on page 107
7.10 Call and Link

### Description

The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address \( PC + (4 \times \text{sign_ext}(\text{disp30})) \). Since the word displacement (\( \text{disp30} \)) 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 \( \text{R[15]} \) (\textit{out} register 7).

When \text{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 \( \text{R[15]} \). (closed impl. dep. #125-V9-Cs10)

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and \text{PSTATE}.tct = 1, then CALL generates a \textit{control_transfer_instruction} exception instead of causing a control transfer. When a \textit{control_transfer_instruction} trap occurs, \( PC \) (the address of the CALL instruction) is stored in \( \text{TPC}[\text{TL}] \) and the value of \( \text{NPC} \) from before the CALL was executed is stored in \( \text{TNPC}[\text{TL}] \). The full 64-bit (nonmasked) \( PC \) and \( NPC \) values are stored in \( \text{TPC}[\text{TL}] \) and \( \text{TNPC}[\text{TL}] \), regardless of the value of \text{PSTATE}.am.

### Exceptions

- \textit{control_transfer_instruction} (impl. dep. #450-S20)

### See Also

JMPL on page 174
7.11 Compare and Swap

Description
Concurrent processes use Compare-and-Swap 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 CASX 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 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), if bit 7 of the ASI is 0, CASXA and CASA cause a privileged_action exception. In privileged mode (PSTATE.priv = 1), if the ASI is in the range 30\textsubscript{16} to 7F\textsubscript{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.

<table>
<thead>
<tr>
<th>ASIs valid for CASA and CASXA instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_REAL</td>
</tr>
<tr>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
</tr>
</tbody>
</table>

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)
## 7.12 DONE

### Description

The DONE instruction restores the saved state from TSTATE[TL] (GL, CCR, ASI, PSTATE, and CWP), 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.

If software writes invalid or inconsistent state to TSTATE before executing DONE, virtual processor behavior during and after execution of the DONE instruction is undefined.

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), 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), 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.

Exceptions

-illegal_instruction
-privileged_opcode
-control_transfer_instruction* (impl. dep. #450-S20)

See Also
RETRY on page 233
7.13 Edge Handling Instructions

**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.

\( \text{EDGE8Lcc, EDGE16Lcc, and EDGE32Lcc} \) are little-endian versions of \( \text{EDGE8cc, EDGE16cc, and EDGE32cc} \), respectively. 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 32-bit address masking is disabled (\( \text{PSTATE.am} = 0 \)) so 64-bit addressing is in use, and the most significant 61 bits of \( R[rs1] \) are equal to the corresponding bits in \( R[rs2] \), \( R[rd] \) is set to the right edge mask and masked with the left edge mask.

3. If 32-bit address masking is enabled (\( \text{PSTATE.am} = 1 \)) so 32-bit addressing is in use, and bits 31:3 of \( R[rs1] \) match bits 31:3 of \( R[rs2] \), \( R[rd] \) is set to the right edge mask and masked 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 \( \text{SUBcc} \) instruction with the same operands (see Subtract on page 303).

The EDGE instructions set the integer condition codes as follows:

- EDGExcc \( x = 8, 16, 32 \)
- EDGExLcc \( x = 8, 16, 32 \)

\( \text{EDGE8cc, EDGE16cc, and EDGE32cc} \) are the edge handling instructions for 8-bit, 16-bit, and 32-bit pixel data, respectively. They operate on the left and right edge masks to produce a mask that is consistent with the mask produced by the Partial Store instruction. The mask is stored in the least significant bits of \( R[rd] \).

**TABLE 7-6**

<table>
<thead>
<tr>
<th>Edge Size</th>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( R[rs1] )</td>
<td>Left Edge</td>
</tr>
<tr>
<td>8 000</td>
<td>1111 1111</td>
<td>1000 0000</td>
</tr>
<tr>
<td>8 001</td>
<td>0111 1111</td>
<td>1100 0000</td>
</tr>
<tr>
<td>8 010</td>
<td>0011 1111</td>
<td>1110 0000</td>
</tr>
<tr>
<td>8 011</td>
<td>0001 1111</td>
<td>1111 0000</td>
</tr>
</tbody>
</table>

† 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”.
### TABLE 7-6  Edge Mask Specification  (Continued)

<table>
<thead>
<tr>
<th>Edge Size</th>
<th>R[rs] (2:0)</th>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Left Edge</td>
<td>Right Edge</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>0000 1111</td>
<td>1111 1000</td>
</tr>
<tr>
<td>8</td>
<td>101</td>
<td>0000 0111</td>
<td>1111 1100</td>
</tr>
<tr>
<td>8</td>
<td>110</td>
<td>0000 0011</td>
<td>1111 1110</td>
</tr>
<tr>
<td>8</td>
<td>111</td>
<td>0000 0001</td>
<td>1111 1111</td>
</tr>
<tr>
<td>16</td>
<td>00x</td>
<td>1111</td>
<td>1000</td>
</tr>
<tr>
<td>16</td>
<td>01x</td>
<td>0111</td>
<td>1100</td>
</tr>
<tr>
<td>16</td>
<td>10x</td>
<td>0011</td>
<td>1110</td>
</tr>
<tr>
<td>16</td>
<td>11x</td>
<td>0001</td>
<td>1111</td>
</tr>
<tr>
<td>32</td>
<td>0xx</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>1xx</td>
<td>01</td>
<td>11</td>
</tr>
</tbody>
</table>

**Exceptions**  None

**See Also**  EDGE<8|16|32>[L]N on page 118
7.14 Edge Handling Instructions (no CC)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDGE8N</td>
<td>0000 0001</td>
<td>Eight 8-bit edge boundary processing, no CC</td>
<td>edge8n rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE8LN</td>
<td>0000 0011</td>
<td>Eight 8-bit edge boundary processing, little-endian, no CC</td>
<td>edge8ln rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE16N</td>
<td>0000 0101</td>
<td>Four 16-bit edge boundary processing, no CC</td>
<td>edge16n rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE16LN</td>
<td>0000 0111</td>
<td>Four 16-bit edge boundary processing, little-endian, no CC</td>
<td>edge16ln rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE32N</td>
<td>0000 1001</td>
<td>Two 32-bit edge boundary processing, no CC</td>
<td>edge32n rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE32LN</td>
<td>0000 1011</td>
<td>Two 32-bit edge boundary processing, little-endian, no CC</td>
<td>edge32ln rs1, rs2, rd</td>
<td>B1</td>
</tr>
</tbody>
</table>

Description

See Edge Handling Instructions on page 116 for details.

Exceptions
None

See Also
EDGE<8,16,32>[L]cc on page 116
### 7.15 Floating-Point Absolute Value

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABSs</td>
<td>11</td>
<td>0</td>
<td>00000 1001</td>
<td>fabss</td>
<td>A1</td>
</tr>
<tr>
<td>FABSd</td>
<td>11</td>
<td>0</td>
<td>00000 1010</td>
<td>fabsd</td>
<td>A1</td>
</tr>
<tr>
<td>FABSq</td>
<td>11</td>
<td>0</td>
<td>00000 1011</td>
<td>fabsq</td>
<td>C3</td>
</tr>
</tbody>
</table>

**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.pcf = 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

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 (FPRSfef = 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 \((rs1[1] \neq 0)\) or \((rs2[1] \neq 0)\) or \((rd[1:0] \neq 0)\) 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)

### See Also
- FMAf on page 137
7.17 Align Data [VIS 1]

### Description
FALIGNDATA concatenates the two 64-bit floating-point registers specified by \( rs_1 \) and \( rs_2 \) 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.

### Exceptions
- **fp_disabled**

### See Also
- Align Address on page 98
### 7.18 Branch on Floating-Point Condition Codes (FBfcc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>cond</th>
<th>Operation</th>
<th>fcc Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBA</td>
<td>1000</td>
<td>Branch Always</td>
<td>1</td>
<td>fba[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBN</td>
<td>0000</td>
<td>Branch Never</td>
<td>0</td>
<td>fbn[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBU</td>
<td>0111</td>
<td>Branch on Unordered</td>
<td>U</td>
<td>fbu[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBG</td>
<td>0110</td>
<td>Branch on Greater</td>
<td>G</td>
<td>fbg[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUG</td>
<td>0101</td>
<td>Branch on Unordered or Greater</td>
<td>G or U</td>
<td>fbug[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBL</td>
<td>0100</td>
<td>Branch on Less</td>
<td>L</td>
<td>fbl[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUL</td>
<td>0011</td>
<td>Branch on Unordered or Less</td>
<td>L or U</td>
<td>fbul[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBNE</td>
<td>0001</td>
<td>Branch on Not Equal</td>
<td>L or G or U</td>
<td>fnne[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBE</td>
<td>1001</td>
<td>Branch on Equal</td>
<td>E</td>
<td>fbe[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBU</td>
<td>1010</td>
<td>Branch on Unordered or Equal</td>
<td>E or U</td>
<td>fbuie[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBGE</td>
<td>1011</td>
<td>Branch on Greater or Equal</td>
<td>E or G</td>
<td>fbg[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUGE</td>
<td>1100</td>
<td>Branch on Unordered or Greater or Equal</td>
<td>E or G or U</td>
<td>fbug[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBE</td>
<td>1101</td>
<td>Branch on Less or Equal</td>
<td>E or L</td>
<td>fbe[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBO</td>
<td>1110</td>
<td>Branch on Ordered</td>
<td>E or L or G</td>
<td>fbo[,a] label</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonym: fnz  ‡ synonym: fbz

<table>
<thead>
<tr>
<th>00</th>
<th>a</th>
<th>cond</th>
<th>110</th>
<th>disp22</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>25</td>
</tr>
</tbody>
</table>

### 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 \times \text{sign_ext}(\text{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 \times \text{sign_ext}(\text{disp22}))”. If FALSE, the branch is not taken.

**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.
FBfcc

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 \((\text{FPRS}.\text{fref} = 0 \text{ or } \text{PSTATE}.\text{pef} = 0)\) 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), \(\text{PSTATE}.\text{tct} = 1\), 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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>cond</th>
<th>Operation</th>
<th>fcc Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBPfcc</td>
<td>1000</td>
<td>Branch Always</td>
<td>1</td>
<td>fba[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>0000</td>
<td>Branch Never</td>
<td>0</td>
<td>fbn[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>0111</td>
<td>Branch on Unordered</td>
<td>U</td>
<td>fbu[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>0110</td>
<td>Branch on Greater</td>
<td>G</td>
<td>fbg[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>0101</td>
<td>Branch on Unordered or Greater</td>
<td></td>
<td>fbug[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>0100</td>
<td>Branch on Less</td>
<td>L</td>
<td>fbl[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>0011</td>
<td>Branch on Unordered or Less</td>
<td>L or U</td>
<td>fbu[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>0010</td>
<td>Branch on Less or Greater</td>
<td>L or G</td>
<td>fbg[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>0001</td>
<td>Branch on Not Equal</td>
<td>L or G or U</td>
<td>fn{\text{ne}}\text{\textdagger}\text{\textbullet}[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>1001</td>
<td>Branch on Equal</td>
<td>E</td>
<td>fbe[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>1010</td>
<td>Branch on Unordered or Equal</td>
<td>E or U</td>
<td>fbue[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>1011</td>
<td>Branch on Greater or Equal</td>
<td>E or G</td>
<td>fbg[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>1100</td>
<td>Branch on Unordered or Greater or Equal</td>
<td>E or G or U</td>
<td>fbue[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>1110</td>
<td>Branch on Less or Equal</td>
<td>E or L</td>
<td>fble[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>1110</td>
<td>Branch on Unordered or Less or Equal</td>
<td>E or L or U</td>
<td>fbule[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
<tr>
<td>FBPfcc</td>
<td>1111</td>
<td>Branch on Ordered</td>
<td>E or L or G</td>
<td>fbo[\text{a}],[\text{pt}],[\text{pn}] %fcc, label</td>
<td>A1</td>
</tr>
</tbody>
</table>

\text{\textdagger} synonym: fbnz \text{\textbullet} synonym: fbz

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.

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.
FBPfcc

- **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), PSTATE.tct = 1, and the FBPfcc 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>s1</th>
<th>s2</th>
<th>d</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMPLE16</td>
<td>0</td>
<td>Four 16-bit compare; set R[rd] if src1 ≤ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmlple16 freg&lt;rs1&gt;, freg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>B1</td>
</tr>
<tr>
<td>FCMPNE16</td>
<td>0</td>
<td>Four 16-bit compare; set R[rd] if src1 ≠ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpne16 freg&lt;rs1&gt;, freg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>B1</td>
</tr>
<tr>
<td>FCMPLE32</td>
<td>0</td>
<td>Two 32-bit compare; set R[rd] if src1 ≤ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmlple32 freg&lt;rs1&gt;, freg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>B1</td>
</tr>
<tr>
<td>FCMPNE32</td>
<td>0</td>
<td>Two 32-bit compare; set R[rd] if src1 ≠ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpne32 freg&lt;rs1&gt;, freg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>B1</td>
</tr>
<tr>
<td>FCMPGT16</td>
<td>0</td>
<td>Four 16-bit compare; set R[rd] if src1 &gt; src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmgt16 freg&lt;rs1&gt;, freg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>B1</td>
</tr>
<tr>
<td>FCMPGT32</td>
<td>0</td>
<td>Two 32-bit compare; set R[rd] if src1 &gt; src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmgt32 freg&lt;rs1&gt;, freg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>B1</td>
</tr>
<tr>
<td>FCMPEQ16</td>
<td>0</td>
<td>Four 16-bit compare; set R[rd] if src1 = src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpeq16 freg&lt;rs1&gt;, freg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>B1</td>
</tr>
<tr>
<td>FCMPEQ32</td>
<td>0</td>
<td>Two 32-bit compare; set R[rd] if src1 = src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpeq32 freg&lt;rs1&gt;, freg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>B1</td>
</tr>
</tbody>
</table>

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.
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.paf = 0) or if no FPU is present, an attempt to execute a SIMD signed compare instruction causes an *fp_disabled* exception.

**Exception** *fp_disabled*

**See Also** Floating-Point Compare on page 128
STPARTIALF on page 260
### 7.21 Floating-Point Compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMPs</td>
<td>0010 0001</td>
<td>Compare Single</td>
<td>fcmps %fcc, fregrs1, fregrs2</td>
<td>A1</td>
</tr>
<tr>
<td>FCMPd</td>
<td>0010 0010</td>
<td>Compare Double</td>
<td>fcmpd %fcc, fregrs1, fregrs2</td>
<td>A1</td>
</tr>
<tr>
<td>FCMPq</td>
<td>0010 0011</td>
<td>Compare Quad</td>
<td>fcmpq %fcc, fregrs1, fregrs2</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPEs</td>
<td>0010 0101</td>
<td>Compare Single and Exception if Unordered</td>
<td>fcmpes %fcc, fregrs1, fregrs2</td>
<td>A1</td>
</tr>
<tr>
<td>FCMPEd</td>
<td>0010 0110</td>
<td>Compare Double and Exception if Unordered</td>
<td>fcmped %fcc, fregrs1, fregrs2</td>
<td>A1</td>
</tr>
<tr>
<td>FCMPEq</td>
<td>0010 0111</td>
<td>Compare Quad and Exception if Unordered</td>
<td>fcmpeq %fcc, fregrs1, fregrs2</td>
<td>C3</td>
</tr>
</tbody>
</table>

#### Description

These instructions compare \( F[rs1] \) with \( F[rs2] \), and set the selected floating-point condition code \( (fcc) \) as follows:

<table>
<thead>
<tr>
<th>Relation</th>
<th>Resulting fcc value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( freg_{rs1} = freg_{rs2} )</td>
<td>0</td>
</tr>
<tr>
<td>( freg_{rs1} &lt; freg_{rs2} )</td>
<td>1</td>
</tr>
<tr>
<td>( freg_{rs1} &gt; freg_{rs2} )</td>
<td>2</td>
</tr>
<tr>
<td>( freg_{rs1} ? freg_{rs2} ) (unordered)</td>
<td>3</td>
</tr>
</tbody>
</table>

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, FBfPcc).

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 an illegal_instruction exception, 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))

See Also
SIMD Signed Compare on page 126
7.22 Floating-Point Divide

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 (FPRSfef = 0 or PSTATEpef = 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 [VIS1]

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.

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.

Exceptions
illegal_instruction
fp_disabled

See Also
FPMERGE on page 160
FPACK on page 153
7.24 Convert 32-bit Integer to Floating Point

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>s1</th>
<th>s2</th>
<th>d</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FiTOs</td>
<td>11 0100 0 1100 0100</td>
<td>Convert 32-bit Integer to Single</td>
<td>—</td>
<td>f32</td>
<td>f32</td>
<td>fitos freg[rs2], freg[rd]</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>FiTOd</td>
<td>11 0100 0 1100 1000</td>
<td>Convert 32-bit Integer to Double</td>
<td>—</td>
<td>f32</td>
<td>f64</td>
<td>fitod freg[rs2], freg[rd]</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>FiTOq</td>
<td>11 0100 0 1100 1100</td>
<td>Convert 32-bit Integer to Quad</td>
<td>—</td>
<td>f32</td>
<td>f128</td>
<td>fitoq freg[rs2], freg[rd]</td>
<td>C3</td>
<td></td>
</tr>
</tbody>
</table>

**Description**  
FiTOs, FiTOd, and FiTOq convert the 32-bit signed integer operand in floating-point register F_{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 (FiTOq)) 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))
### Flush Instruction Memory

**Description**

Flush Instruction Memory (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 to a memory location that may be executed as an instruction (self-modifying code), 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 201).

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 preceding 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 \( \text{R}[rs1] + \text{R}[rs2] \); if \( i = 1 \), it is \( \text{R}[rs1] + \text{sign_ext}(\text{simm13}) \). The three least-significant bits of the effective address are ignored; that is, the effective address always refers to an aligned doubleword.

---

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUSH</td>
<td>11</td>
<td>1011</td>
<td>flush [address]</td>
<td>A1</td>
</tr>
</tbody>
</table>

† 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.

---

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

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**: 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

<table>
<thead>
<tr>
<th>Programming Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Typically, FLUSH is used in self-modifying code. The use of self-modifying code is discouraged.</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>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.</td>
</tr>
<tr>
<td>5. FLUSH may be a time-consuming operation. (see the Implementation Note below)</td>
</tr>
<tr>
<td>6. In a multiprocessor system, the effects of a FLUSH operation will be globally visible before any subsequent store becomes globally visible.</td>
</tr>
</tbody>
</table>
FLUSH

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 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.

An attempt to execute a FLUSH instruction when i = 0 and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

Exceptions
illegal_instruction
DAE_nfo_page
## 7.26 Flush Register Windows

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUSHW</td>
<td>10 1011</td>
<td>Flush Register Windows</td>
<td>flushw</td>
<td>A1</td>
</tr>
</tbody>
</table>

### 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.

FLUSHW acts as a NOP if $\text{CANSAVE} = \text{N_REG_WINDOWS} - 2$. 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 $\text{OTHERWIN}$ and $\text{WSTATE}$. The spill trap handler is invoked with the $\text{CWP}$ set to the window to be spilled (that is, $(\text{CWP} + \text{CANSAVE} + 2) \mod \text{N_REG_WINDOWS}$). See Register Window Management Instructions on page 83.

### 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.

- Typically, the spill handler saves a window on a memory stack and returns to reexecute the FLUSHW instruction. Thus, FLUSHW traps and reexecutes until all active windows other than the current window have been spilled.

### Exceptions

- illegal_instruction
- spill_n_normal
- spill_n_other

---

FLUSHW
Floating-Point Multiply-Add and Multiply-Subtract (fused)

### Description

The fused floating-point multiply-add instructions, FMADD<\|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<\|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<\|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.

The fused floating-point negative multiply-subtract instructions, FNMSUB<\|d>, multiply the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, 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
FMAf

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.ife = 0 or PSTATE.ife = 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
- ∞ x 0
- ∞ − ∞

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

FMAADD:

(1) \( \text{tmp} \leftarrow F[\text{rs1}] \times F[\text{rs2}] \)
(2) \( \text{tmp} \leftarrow \text{tmp} + F[\text{rs3}] \)
(3) \( F[\text{rd}] \leftarrow \text{round}(\text{tmp}) \)

FNMAADD:

(1) \( \text{tmp} \leftarrow F[\text{rs1}] \times F[\text{rs2}] \)
(2) \( \text{tmp} \leftarrow \text{tmp} + F[\text{rs3}] \)
(3) \( \text{tmp} \leftarrow - \text{tmp} \)
(4) \( F[\text{rd}] \leftarrow \text{round}(\text{tmp}) \)

FMSUB:

(1) \( \text{tmp} \leftarrow F[\text{rs1}] \times F[\text{rs2}] \)
(2) \( \text{tmp} \leftarrow \text{tmp} - F[\text{rs3}] \)
(3) \( F[\text{rd}] \leftarrow \text{round}(\text{tmp}) \)

FNMSUB:

(1) \( \text{tmp} \leftarrow F[\text{rs1}] \times F[\text{rs2}] \)
(2) \( \text{tmp} \leftarrow \text{tmp} - F[\text{rs3}] \)
(3) \( \text{tmp} \leftarrow - \text{tmp} \)
(4) \( F[\text{rd}] \leftarrow \text{round}(\text{tmp}) \)

Exceptions

fp_disabled
fp_exception_ieee_754 (OF, UF, NX, NV)
fp_exception_other (FSR.ftt = unfinished_FPop)

See Also

FMUL on page 151
FADD on page 120
FSUB on page 161
7.28 Floating-Point Move

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 \textit{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 \textit{illegal\_instruction} exception.

If the FPU is not enabled (FPRS.ef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMOV instruction causes an \textit{fp\_disabled} exception.

An attempt to execute an FMOVq instruction when rs2[1] ≠ 0 or rd[1] ≠ 0 causes an \textit{fp\_exception\_other} (FSR.ftt = invalid_fp_register) exception.

For more details regarding floating-point exceptions, see Chapter 8, \textit{IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007}.

**Exceptions**

- \textit{illegal\_instruction}
- \textit{fp\_disabled}
- \textit{fp\_exception\_other} (FSR.ftt = invalid_fp_register (FMOVq only))

**See Also**

- \textit{f Register Logical Operate (2 operand)} on page 164
7.29 Move Floating-Point Register on Condition (FMOVcc)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf_low</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOVSicc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on 32-bit integer condition codes</td>
<td>fmovsicc %icc, fregrs2, fregrd</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVDicc</td>
<td>00 0010</td>
<td>Move Floating-Point Double, based on 32-bit integer condition codes</td>
<td>fmovdicc %icc, fregrs2, fregrd</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVQicc</td>
<td>00 0011</td>
<td>Move Floating-Point Quad, based on 32-bit integer condition codes</td>
<td>fmovqicc %icc, fregrs2, fregrd</td>
<td>C3</td>
</tr>
<tr>
<td>FMOVSxcc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on 64-bit integer condition codes</td>
<td>fmovsxcc %xcc, fregrs2, fregrd</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVDxcc</td>
<td>00 0010</td>
<td>Move Floating-Point Double, based on 64-bit integer condition codes</td>
<td>fmovdxcc %xcc, fregrs2, fregrd</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVQxcc</td>
<td>00 0011</td>
<td>Move Floating-Point Quad, based on 64-bit integer condition codes</td>
<td>fmovqxcc %xcc, fregrs2, fregrd</td>
<td>C3</td>
</tr>
<tr>
<td>FMOVSfcc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on floating-point condition codes</td>
<td>fmovsfcc %fccn, fregrs2, fregrd</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVDfcc</td>
<td>00 0010</td>
<td>Move Floating-Point Double, based on floating-point condition codes</td>
<td>fmovdfcc %fccn, fregrs2, fregrd</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVQfcc</td>
<td>00 0011</td>
<td>Move Floating-Point Quad, based on floating-point condition codes</td>
<td>fmovqfcc %fccn, fregrs2, fregrd</td>
<td>C3</td>
</tr>
</tbody>
</table>
### FMOVcc

**Encoding of the cond Field for F.P. Moves Based on Integer Condition Codes (icc or xcc)**

<table>
<thead>
<tr>
<th>cond</th>
<th>Operation</th>
<th>icc / xcc Test</th>
<th>icc/xcc name(s) in Assembly Language Mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Move Always</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>0000</td>
<td>Move Never</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>1001</td>
<td>Move if Not Equal</td>
<td>not Z</td>
<td>ne (or nz)</td>
</tr>
<tr>
<td>0001</td>
<td>Move if Equal</td>
<td>Z</td>
<td>e (or z)</td>
</tr>
<tr>
<td>1010</td>
<td>Move if Greater</td>
<td>not (Z or (N xor V))</td>
<td>g</td>
</tr>
<tr>
<td>0010</td>
<td>Move if Less or Equal</td>
<td>Z or (N xor V)</td>
<td>le</td>
</tr>
<tr>
<td>1011</td>
<td>Move if Greater or Equal</td>
<td>not (N xor V)</td>
<td>ge</td>
</tr>
<tr>
<td>0011</td>
<td>Move if Less</td>
<td>N xor V</td>
<td>l</td>
</tr>
<tr>
<td>1100</td>
<td>Move if Greater Unsigned</td>
<td>not (C or Z)</td>
<td>gu</td>
</tr>
<tr>
<td>0100</td>
<td>Move if Less or Equal Unsigned</td>
<td>(C or Z)</td>
<td>leu</td>
</tr>
<tr>
<td>1101</td>
<td>Move if Carry Clear (Greater or Equal, Unsigned)</td>
<td>not C</td>
<td>cc (or geu)</td>
</tr>
<tr>
<td>0101</td>
<td>Move if Carry Set (Less than, Unsigned)</td>
<td>C</td>
<td>cs (or lu)</td>
</tr>
<tr>
<td>1110</td>
<td>Move if Positive</td>
<td>not N</td>
<td>pos</td>
</tr>
<tr>
<td>0110</td>
<td>Move if Negative</td>
<td>N</td>
<td>neg</td>
</tr>
<tr>
<td>1111</td>
<td>Move if Overflow Clear</td>
<td>not V</td>
<td>vc</td>
</tr>
<tr>
<td>0111</td>
<td>Move if Overflow Set</td>
<td>V</td>
<td>vs</td>
</tr>
</tbody>
</table>

**Encoding of the cond Field for F.P. Moves Based on Floating-Point Condition Codes (fccn)**

<table>
<thead>
<tr>
<th>cond</th>
<th>Operation</th>
<th>fccn Test</th>
<th>fcc name(s) in Assembly Language Mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Move Always</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>0000</td>
<td>Move Never</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>0111</td>
<td>Move if Unordered</td>
<td>U</td>
<td>u</td>
</tr>
<tr>
<td>0110</td>
<td>Move if Greater</td>
<td>G</td>
<td>g</td>
</tr>
<tr>
<td>0101</td>
<td>Move if Unordered or Greater</td>
<td>G or U</td>
<td>ug</td>
</tr>
<tr>
<td>0100</td>
<td>Move if Less</td>
<td>L</td>
<td>l</td>
</tr>
<tr>
<td>0011</td>
<td>Move if Unordered or Less</td>
<td>L or U</td>
<td>ul</td>
</tr>
<tr>
<td>0010</td>
<td>Move if Less or Greater</td>
<td>L or G</td>
<td>lg</td>
</tr>
<tr>
<td>0001</td>
<td>Move if Not Equal</td>
<td>L or G or U</td>
<td>ne (or nz)</td>
</tr>
<tr>
<td>1001</td>
<td>Move if Equal</td>
<td>E</td>
<td>e (or z)</td>
</tr>
<tr>
<td>1010</td>
<td>Move if Unordered or Equal</td>
<td>E or U</td>
<td>ue</td>
</tr>
<tr>
<td>1011</td>
<td>Move if Greater or Equal</td>
<td>E or G</td>
<td>ge</td>
</tr>
<tr>
<td>1100</td>
<td>Move if Unordered or Greater or Equal</td>
<td>E or G or U</td>
<td>uge</td>
</tr>
<tr>
<td>1101</td>
<td>Move if Less or Equal</td>
<td>E or L</td>
<td>le</td>
</tr>
<tr>
<td>1110</td>
<td>Move if Unordered or Less or Equal</td>
<td>E or L or U</td>
<td>ule</td>
</tr>
<tr>
<td>1111</td>
<td>Move if Ordered</td>
<td>E or L or G</td>
<td>o</td>
</tr>
</tbody>
</table>
Encoding of opf_cc Field (also see TABLE E-10 on page 484)

<table>
<thead>
<tr>
<th>opf_cc</th>
<th>Instruction</th>
<th>Condition Code to be Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002</td>
<td>FMOV&lt;s</td>
<td>d</td>
</tr>
<tr>
<td>0012</td>
<td>FMOV&lt;s</td>
<td>d</td>
</tr>
<tr>
<td>0102</td>
<td>FMOV&lt;s</td>
<td>d</td>
</tr>
<tr>
<td>0112</td>
<td></td>
<td>fcc1</td>
</tr>
<tr>
<td>1012</td>
<td></td>
<td>fcc2</td>
</tr>
<tr>
<td>1112</td>
<td></td>
<td>fcc3</td>
</tr>
</tbody>
</table>

1012  (illegal_instruction exception)

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 = 1012 or 1112 causes an illegal_instruction exception.

If the FPU is not enabled (FPRS.lef = 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, FMOVQxCCc, or FMOVQfcc instruction when rs2[1] ≠ 0 or rd[1] ≠ 0 causes an fp_exception_other (FSR.ftt = invalid_fy_register) exception.
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:

```c
double A, B, X;
if (A > B) then X = 1.03; else X = 0.0;
```

can be coded as

```assembly
! assume A is in %{f0}; B is in %{f2}; %xx points to ! constant area
ldd [%xx+C_1.03],&%f4 ! X = 1.03
fcmpd %fccc3,%f0,%f2 ! A > B
fble,a %fccc3,label
! following instruction only 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

```assembly
ldd [%xx+C_1.03],&%f4 ! X = 1.03
fsubd &%f4,%f4,%f6 ! X' = 0.0
fcmpd %fccc3,%f0,%f2 ! A > B
fmovdle %fccc3,%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.flt = invalid_fp_register (FMOVQ instructions))
7.30 Move Floating-Point Register on Integer Register Condition (FMOVR)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rcond</th>
<th>opf_low</th>
<th>Operation</th>
<th>Test</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>000</td>
<td>0 0101</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FMOVRsZ</td>
<td>001</td>
<td>0 0101</td>
<td>Move Single if Register = 0</td>
<td>R[rs1] = 0</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVRsLEZ</td>
<td>010</td>
<td>0 0101</td>
<td>Move Single if Register ≤ 0</td>
<td>R[rs1] ≤ 0</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVRsLZ</td>
<td>011</td>
<td>0 0101</td>
<td>Move Single if Register &lt; 0</td>
<td>R[rs1] &lt; 0</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>100</td>
<td>0 0101</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FMOVRsNZ</td>
<td>101</td>
<td>0 0101</td>
<td>Move Single if Register ≠ 0</td>
<td>R[rs1] ≠ 0</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVRsGZ</td>
<td>110</td>
<td>0 0101</td>
<td>Move Single if Register &gt; 0</td>
<td>R[rs1] &gt; 0</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVRsGEZ</td>
<td>111</td>
<td>0 0101</td>
<td>Move Single if Register ≥ 0</td>
<td>R[rs1] ≥ 0</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>000</td>
<td>0 0110</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FMOVRdZ</td>
<td>001</td>
<td>0 0110</td>
<td>Move Double if Register = 0</td>
<td>R[rs1] = 0</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVRdLEZ</td>
<td>010</td>
<td>0 0110</td>
<td>Move Double if Register ≤ 0</td>
<td>R[rs1] ≤ 0</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVRdLZ</td>
<td>011</td>
<td>0 0110</td>
<td>Move Double if Register &lt; 0</td>
<td>R[rs1] &lt; 0</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>100</td>
<td>0 0110</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FMOVRdNZ</td>
<td>101</td>
<td>0 0110</td>
<td>Move Double if Register ≠ 0</td>
<td>R[rs1] ≠ 0</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVRdGZ</td>
<td>110</td>
<td>0 0110</td>
<td>Move Double if Register &gt; 0</td>
<td>R[rs1] &gt; 0</td>
<td>A1</td>
</tr>
<tr>
<td>FMOVRdGEZ</td>
<td>111</td>
<td>0 0110</td>
<td>Move Double if Register ≥ 0</td>
<td>R[rs1] ≥ 0</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>000</td>
<td>0 0111</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FMOVRqZ</td>
<td>001</td>
<td>0 0111</td>
<td>Move Quad if Register = 0</td>
<td>R[rs1] = 0</td>
<td>C3</td>
</tr>
<tr>
<td>FMOVRqLEZ</td>
<td>010</td>
<td>0 0111</td>
<td>Move Quad if Register ≤ 0</td>
<td>R[rs1] ≤ 0</td>
<td>C3</td>
</tr>
<tr>
<td>FMOVRqLZ</td>
<td>011</td>
<td>0 0111</td>
<td>Move Quad if Register &lt; 0</td>
<td>R[rs1] &lt; 0</td>
<td>C3</td>
</tr>
<tr>
<td>—</td>
<td>100</td>
<td>0 0111</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FMOVRqNZ</td>
<td>101</td>
<td>0 0111</td>
<td>Move Quad if Register ≠ 0</td>
<td>R[rs1] ≠ 0</td>
<td>C3</td>
</tr>
<tr>
<td>FMOVRqGZ</td>
<td>110</td>
<td>0 0111</td>
<td>Move Quad if Register &gt; 0</td>
<td>R[rs1] &gt; 0</td>
<td>C3</td>
</tr>
<tr>
<td>FMOVRqGEZ</td>
<td>111</td>
<td>0 0111</td>
<td>Move Quad if Register ≥ 0</td>
<td>R[rs1] ≥ 0</td>
<td>C3</td>
</tr>
</tbody>
</table>

Assembly Language Syntax

\[
\begin{align*}
\text{fmovr}[s,d,q]z & \quad r_{S_{rs1}}, f_{S_{rs2}}, f_{S_{rd}} \\
\text{fmovr}[s,d,q]le & \quad r_{S_{rs1}}, f_{S_{rs2}}, f_{S_{rd}} \\
\text{fmovr}[s,d,q]l & \quad r_{S_{rs1}}, f_{S_{rs2}}, f_{S_{rd}} \\
\text{fmovr}[s,d,q]n & \quad r_{S_{rs1}}, f_{S_{rs2}}, f_{S_{rd}} \\
\text{fmovr}[s,d,q]g & \quad r_{S_{rs1}}, f_{S_{rs2}}, f_{S_{rd}} \\
\text{fmovr}[s,d,q]ge & \quad r_{S_{rs1}}, f_{S_{rs2}}, f_{S_{rd}}
\end{align*}
\]
**FMOVR**

**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.flt. 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 \( rcond = 000 \) or \( 100 \) causes an *illegal_instruction* exception.

If the FPU is not enabled (\( FPRS.lef = 0 \) 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 \( rs2[1] \neq 0 \) or \( rd[1] \neq 0 \) 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:

<table>
<thead>
<tr>
<th>Branch</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOVRNZ</td>
<td>not Z</td>
</tr>
<tr>
<td>FMOVRZ</td>
<td>Z</td>
</tr>
<tr>
<td>FMOVRGEZ</td>
<td>not N</td>
</tr>
<tr>
<td>FMOVRLZ</td>
<td>N</td>
</tr>
<tr>
<td>FMOVRLEZ</td>
<td>N or Z</td>
</tr>
<tr>
<td>FMOVRGZ</td>
<td>N nor Z</td>
</tr>
</tbody>
</table>

**Exceptions**
*illegal_instruction*
*fp_disabled*  *fp_exception_other* (\( FSR.ftt = invalid_fp_register \) (FMOVRq instructions))
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](image)

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 \( \alpha \) pixel component value). FIGURE 7-11 illustrates the operation.

![FIGURE 7-11 FMUL8x16AU Operation](image)
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](image)

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](image)

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.
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.
FMUL (partitioned)

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

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.fe = 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*.

### 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)

### See Also
- FMAf on page 137

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMULs</td>
<td>11 0100</td>
<td>0 0100 1001</td>
<td>Multiply Single</td>
<td>fmuls freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FMULd</td>
<td>11 0100</td>
<td>0 0100 1010</td>
<td>Multiply Double</td>
<td>fmuld freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FMULq</td>
<td>11 0100</td>
<td>0 0100 1011</td>
<td>Multiply Quad</td>
<td>fmulq freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C3</td>
</tr>
<tr>
<td>FsMULd</td>
<td>11 0100</td>
<td>0 0110 1001</td>
<td>Multiply Single to Double</td>
<td>fsmuld freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FdMULq</td>
<td>11 0100</td>
<td>0 0110 1110</td>
<td>Multiply Double to Quad</td>
<td>fdmulq freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C3</td>
</tr>
</tbody>
</table>
7.33 Floating-Point Negate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNEGs</td>
<td>11 0100</td>
<td>0 0000 0101</td>
<td>Negate Single</td>
<td>fnegs freg_{rs2}, freg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>FNEGd</td>
<td>11 0100</td>
<td>0 0000 0110</td>
<td>Negate Double</td>
<td>fnegd freg_{rs2}, freg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>FNEGq</td>
<td>11 0100</td>
<td>0 0000 0111</td>
<td>Negate Quad</td>
<td>fnegq freg_{rs2}, freg_{rd}</td>
<td>C3</td>
</tr>
</tbody>
</table>

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.

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

### 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.cef = 0 or PSTATE.cef = 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 131
- FPMERGE on page 160

### Table: FPACK Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>s1</th>
<th>s2</th>
<th>d</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPACK16</td>
<td>0 0011 1011</td>
<td>Four 16-bit packs into 8 unsigned bits</td>
<td>—</td>
<td>f64</td>
<td>f32</td>
<td>fpack16 fregrs2, fregrd</td>
<td>B1</td>
</tr>
<tr>
<td>FPACK32</td>
<td>0 0011 1010</td>
<td>Two 32-bit packs into 8 unsigned bits</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpack32 fregrs1, fregrs2, fregrd</td>
<td>B1</td>
</tr>
<tr>
<td>FPACKFIX</td>
<td>0 0011 1101</td>
<td>Four 16-bit packs into 16 signed bits</td>
<td>—</td>
<td>f64</td>
<td>f32</td>
<td>fpackfix fregrs2, fregrd</td>
<td>B1</td>
</tr>
</tbody>
</table>

---

**Note:**
- VIS 1 indicates Video Instruction Set 1.
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:

```c
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.

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:

```plaintext
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) |
(unsighned_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](image)

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;

156 UltraSPARC Architecture 2007 • Draft D0.9.3b, 20 Oct 2009
else
    signed_16bit_result ← trunc_signed_value[31:16];
7.35 Fixed-point Partitioned Add

**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.
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**  \[ fp\_disabled \]
FPMERGE

7.36 FPMERGE

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, \( R1G1B1A1,R3G3B3A3 \rightarrow R1R3G1G3A1A3 \rightarrow R1R2R3R4G1G2G3G4 \).

FPMERGE also converts from planar to packed when it is applied twice in succession; for example, \( R1R2R3R4,B1B2B3B4 \rightarrow R1B1R2B2R3B3R4B4 \rightarrow R1G1B1A1R2G2B2A2 \).

FIGURE 7-24 illustrates the operation.

```
%d0  R1  G1  B1  A1  R2  G2  B2  A2 \} packed representation
%d2  R3  G3  B3  A3  R4  G4  B4  A4 \} intermediate

fpmerge %f0, %f2, %d4 \} intermediate
fpmerge %f1, %f3, %d6 \} intermediate
fpmerge %f4, %f6, %d0 \} intermediate
fpmerge %f5, %f7, %d2 \} intermediate

fpmerge %f0, %f2, %d4 \} planar representation
fpmerge %f1, %f3, %d6 \} planar representation
fpmerge %f4, %f6, %d0 \} planar representation
fpmerge %f5, %f7, %d2 \} planar representation
```

Exceptions
\( fp_{\text{disabled}} \)

See Also
FPACK on page 153
FEXPAND on page 131
7.37 Fixed-point Partitioned Subtract (64-bit)

**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](image-url1)

![FIGURE 7-26 FPSUB32 Operation](image-url2)
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 \textit{fp\_disabled} exception.

\textbf{Exceptions} \hspace{1cm} \textit{fp\_disabled}
### 7.38 F Register Logical Operate (1 operand)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FZEROd</td>
<td>0 0110 0000</td>
<td>Zero fill</td>
<td>fzero freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FZEROs</td>
<td>0 0110 0001</td>
<td>Zero fill, 32-bit</td>
<td>fzeros freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FONEd</td>
<td>0 0111 1110</td>
<td>One fill</td>
<td>one freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FONEs</td>
<td>0 0111 1111</td>
<td>One fill, 32-bit</td>
<td>fones freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

FZERO and FONE fill the 64-bit destination register, F<sub>D</sub>[rd], with all ‘0’ bits or all ‘1’ bits (respectively).

FZEROs and FONEs fill the 32-bit destination register, F<sub>D</sub>[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.<sub>ef</sub> = 0 or PSTATE.<sub>ef</sub> = 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 164
- F Register 3-operand Logical Operations on page 165
7.39 F Register Logical Operate (2 operand)

Description
The standard 64-bit versions of these instructions perform one of four 64-bit logical operations on the
data from the 64-bit floating-point source 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.pef = 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 139). Programmers are encouraged to use FMOVs
(FMOVd) instead of FSRC1s (FSRC1) whenever practical.

Exceptions
illegal_instruction
fp_disabled

See Also
Floating-Point Move on page 139
F Register 1-operand Logical Operations on page 163
F Register 3-operand Logical Operations on page 165
## F Register 3-operand Logical Ops

### 7.40 F Register Logical Operate (3 operand)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORD</td>
<td>0 0111 1100</td>
<td>Logical or</td>
<td>for $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FORs</td>
<td>0 0111 1101</td>
<td>Logical or, 32-bit</td>
<td>for $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FNORd</td>
<td>0 0110 0010</td>
<td>Logical nor</td>
<td>fnor $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FNORs</td>
<td>0 0110 0011</td>
<td>Logical nor, 32-bit</td>
<td>fnors $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FANDd</td>
<td>0 0111 0000</td>
<td>Logical and</td>
<td>fand $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FANDs</td>
<td>0 0111 0001</td>
<td>Logical and, 32-bit</td>
<td>fands $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FNANDd</td>
<td>0 0110 1110</td>
<td>Logical nand</td>
<td>fnand $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FNANDs</td>
<td>0 0110 1111</td>
<td>Logical nand, 32-bit</td>
<td>fnands $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FXORd</td>
<td>0 0110 1100</td>
<td>Logical xor</td>
<td>fxor $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FXORs</td>
<td>0 0110 1101</td>
<td>Logical xor, 32-bit</td>
<td>fxors $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FXNORd</td>
<td>0 0111 0010</td>
<td>Logical xnor</td>
<td>fxnor $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FXNORs</td>
<td>0 0111 0011</td>
<td>Logical xnor, 32-bit</td>
<td>fxnors $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FORNOT1d</td>
<td>0 0111 1010</td>
<td>(not $F_D[rs1]$) or $F_D[rs2]$</td>
<td>fornot1 $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FORNOT1s</td>
<td>0 0111 1011</td>
<td>(not $F_S[rs1]$) or $F_S[rs2]$, 32-bit</td>
<td>fornot1s $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FORNOT2d</td>
<td>0 0111 0110</td>
<td>$F_D[rs1]$ or (not $F_D[rs2]$)</td>
<td>fornot2 $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FORNOT2s</td>
<td>0 0111 0111</td>
<td>$F_S[rs1]$ or (not $F_S[rs2]$), 32-bit</td>
<td>fornot2s $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FANDNOT1d</td>
<td>0 0110 1000</td>
<td>(not $F_D[rs1]$) and $F_D[rs2]$</td>
<td>fandnot1 $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FANDNOT1s</td>
<td>0 0110 1001</td>
<td>(not $F_S[rs1]$) and $F_S[rs2]$, 32-bit</td>
<td>fandnot1s $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FANDNOT2d</td>
<td>0 0110 0100</td>
<td>$F_D[rs1]$ and (not $F_D[rs2]$)</td>
<td>fandnot2 $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FANDNOT2s</td>
<td>0 0110 0101</td>
<td>$F_S[rs1]$ and (not $F_S[rs2]$), 32-bit</td>
<td>fandnot2s $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
</tbody>
</table>

### 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 ($FPRS.fe = 0$ or $PSTATE.pe = 0$) 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 163
F Register 2-operand Logical Operations on page 164

---

**CHAPTER 7 • Instructions 165**
7.41 Floating-Point Square Root

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 \textit{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 \textit{illegal\_instruction} exception.

If the FPU is not enabled (\( FPRS.\text{r}ef = 0 \) or \( \text{PSTATE.}\text{r}ef = 0 \)) or if no FPU is present, an attempt to execute an FSQRT instruction causes an \textit{fp\_disabled} exception.

An attempt to execute an FSQRTq instruction when \( rs2[1] \neq 0 \) or \( rd[1] \neq 0 \) causes an \textit{fp\_exception\_other} (\( FSR.\text{ftt} = \text{invalid\_fp\_register} \)) exception.

An \textit{fp\_exception\_other} (with \( FSR.\text{ftt} = \text{unfinished\_FPop} \)) can occur if the operand to the square root is positive and subnormal. See \textit{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
- \textit{illegal\_instruction}
- \textit{fp\_disabled}
- \textit{fp\_exception\_other} (\( FSR.\text{ftt} = \text{invalid\_fp\_register} \) (FSQRTq only))
- \textit{fp\_exception\_other} (\( FSR.\text{ftt} = \text{unfinished\_FPop} \))
- \textit{fp\_exception\_ieee\_754} (IEEE_754_exception (NV, NX))
7.42 Convert Floating-Point to Integer

### 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.\textbf{f}ef = 0 or PSTATE.\textbf{p}ef = 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 \( rs2 \neq 0 \) causes an *fp_exception_other* (FSR.\textbf{f}tt = 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 293.

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.\textbf{f}tt = invalid_fp_register (FqTOx and FqTOi only))
- *fp_exception_other* (FSR.\textbf{f}tt = unfinished_FPop)
- *fp_exception_ieee_754* (NV, NX)
7.43  Convert Between Floating-Point Formats

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.ief = 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.flt = invalid_fp_register) exception. An attempt to execute an FqTOS or FqTOd instruction when rs2[1] ≠ 0 causes an fp_exception_other (FSR.flt = 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.flt = unfinished_FPpop 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.flt = invalid_fp_register (FsTOq, FqTOS, FdTOq, and FqTOd only))
- fp_exception_other (FSR.flt = unfinished_FPpop)
F<s|d|q>TO<s|d|q>

fp_exception_ieee_754 (NV)
fp_exception_ieee_754 (OF, UF, NX (FqTOd, FqTOs, and FdTOs))
7.44 Floating-Point Subtract

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.flt = invalid_fp_register) exception.

Note | An fp_exception_other with FSR.flt = 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.flt = invalid_fp_register (FSUBq only))
fp_exception_other (FSR.flt = unfinished_FPop)
fp_exception_ieee_754 (OF, UF, NX, NV)

See Also | FMAf on page 137
### 7.45 Convert 64-bit Integer to Floating Point

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op3</th>
<th>Opf</th>
<th>Operation</th>
<th>S1</th>
<th>S2</th>
<th>D</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FxTOs</td>
<td>110100</td>
<td>01000100</td>
<td>Convert 64-bit Integer to Single</td>
<td>—</td>
<td>i64</td>
<td>f32</td>
<td>fxtos frcRs2, frcRsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FxTOd</td>
<td>110100</td>
<td>01001000</td>
<td>Convert 64-bit Integer to Double</td>
<td>—</td>
<td>i64</td>
<td>f64</td>
<td>fxtod frcRs2, frcRsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FxTOq</td>
<td>110100</td>
<td>01001100</td>
<td>Convert 64-bit Integer to Quad</td>
<td>—</td>
<td>i64</td>
<td>f128</td>
<td>fxtoq frcRs2, frcRsrd</td>
<td>C3</td>
</tr>
</tbody>
</table>

**Description**

FxTOs, FxTOd, and FxTOq convert the 64-bit signed integer operand in the floating-point register FSD[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

**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**
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 Mark Register Window Sets as “Invalid”

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALWP</td>
<td>Mark all register window sets as “invalid”</td>
<td>invalw</td>
<td>A1</td>
</tr>
</tbody>
</table>

#### Description
The INVALW instruction marks all register window sets as “invalid”; specifically, it atomically performs the following operations:

- CANSAVE ← \((N \_ \text{REG} \_ \text{WINDOWS} - 2)\)
- CANRESTORE ← 0
- OTHERWIN ← 0

#### Programming Notes
INVALW marks all windows as invalid; after executing INVALW, \(N\_\text{REG}\_\text{WINDOWS}\)-2 SAVEs can be performed without generating a spill trap.

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) causes a privileged_opcode exception.

#### Exceptions
- illegal_instruction
- privileged_opcode

#### See Also
- ALLCLEAN on page 99
- NORMALW on page 213
- OTHERW on page 215
- RESTORED on page 232
- SAVED on page 239
7.48 Jump and Link

**Description**

The JMPL instruction causes a register-indirect delayed control transfer to the address given by “R[rs1] + R[rs2]” if \( i = 0 \), 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)

**Exceptions**

*illegal_instruction*

*mem_address_not_aligned*

*control_transfer_instruction* (impl. dep. #450-S20)

**See Also**

CALL on page 111

Bicc on page 104

BPcc on page 109
### 7.49 Load Integer

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 76). The effective address is "$R[rs1] + R[rs2]$" if $i = 0$, or "$R[rs1] + \text{sign\_ext}(\text{simm13})$" 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.

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 192 for details.

### Exceptions
- illegal_instruction
- mem_address_not_aligned (all except LDSB, LDUB)
- VA_watchpoint
- DAE_privilege_violation
- DAE_nfo_page
7.50 Load Integer from Alternate Space

### 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 $"R[rs1] + \text{sign ext}(simm13)"$ 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 194 for details.

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), if bit 7 of the ASI is 0, these instructions cause a **privileged_action** exception. In privileged mode (PSTATE.priv = 1), if the ASI is in the range $30_{16}$ to $7F_{16}$, these instructions cause a **privileged_action** exception.

### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDSBA_PASI</td>
<td>01 1001</td>
<td>Load Signed Byte from Alternate Space</td>
<td>ldsba [regaddr] imm_asr, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDSHA_PASI</td>
<td>01 1010</td>
<td>Load Signed Halfword from Alternate Space</td>
<td>ldsha [regaddr] imm_asr, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDSWA_PASI</td>
<td>01 1000</td>
<td>Load Signed Word from Alternate Space</td>
<td>ldswa [regaddr] imm_asr, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUBA_PASI</td>
<td>01 0001</td>
<td>Load Unsigned Byte from Alternate Space</td>
<td>lduba [regaddr] imm_asr, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUHA_PASI</td>
<td>01 0010</td>
<td>Load Unsigned Halfword from Alternate Space</td>
<td>lduha [regaddr] imm_asr, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUWA_PASI</td>
<td>01 0000</td>
<td>Load Unsigned Word from Alternate Space</td>
<td>lduwa [regaddr] imm_asr, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDXAPASI</td>
<td>01 1011</td>
<td>Load Extended Word from Alternate Space</td>
<td>ldxa [regaddr] imm_asr, regrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonym: lda
LDA

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.

<table>
<thead>
<tr>
<th>ASIs valid for LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_REAL</td>
</tr>
<tr>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
</tr>
<tr>
<td>ASI_PRIMARY_NO_FAULT</td>
</tr>
<tr>
<td>ASI_SECONDARY_NO_FAULT</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>ASIs invalid for LDXA (cause <strong>DAE_invalid_asi</strong> exception)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22_16 (ASI_TWINX_AIUP)</td>
</tr>
<tr>
<td>23_16 (ASI_TWINX_AIUS)</td>
</tr>
<tr>
<td>26_16 (ASI_TWINX_REAL)</td>
</tr>
<tr>
<td>27_16 (ASI_TWINX_N)</td>
</tr>
<tr>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_PST8_PRIMARY</td>
</tr>
<tr>
<td>ASI_PST8_SECONDARY</td>
</tr>
<tr>
<td>ASI_PST16_PRIMARY</td>
</tr>
<tr>
<td>ASI_PST16_SECONDARY</td>
</tr>
<tr>
<td>ASI_PST32_PRIMARY</td>
</tr>
<tr>
<td>ASI_PST32_SECONDARY</td>
</tr>
<tr>
<td>ASI_FL8_PRIMARY</td>
</tr>
<tr>
<td>ASI_FL8_SECONDARY</td>
</tr>
<tr>
<td>ASI_FL16_PRIMARY</td>
</tr>
<tr>
<td>ASI_FL16_SECONDARY</td>
</tr>
<tr>
<td>ASI_BLOCK_COMMIT_PRIMARY</td>
</tr>
<tr>
<td>E2_16 (ASI_TWINX_P)</td>
</tr>
<tr>
<td>E3_16 (ASI_TWINX_S)</td>
</tr>
<tr>
<td>ASI_BLOCK_PRIMARY</td>
</tr>
<tr>
<td>ASI_BLOCK_SECONDARY</td>
</tr>
</tbody>
</table>

**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**

**See Also**

LD on page 175
STA on page 248
7.51 Block Load [VIS1]

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 or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASI Value</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBLOCKFD</td>
<td>1616</td>
<td>64-byte block load from primary address space, user privilege</td>
<td>ldda [regaddr] #ASI_BLK_AIUP, freg</td>
<td>D2</td>
</tr>
<tr>
<td>LDBLOCKFD</td>
<td>1716</td>
<td>64-byte block load from secondary address space, user privilege</td>
<td>ldda [regaddr] #ASI_BLK_AIUS, freg</td>
<td>D2</td>
</tr>
<tr>
<td>LDBLOCKFD</td>
<td>1E16</td>
<td>64-byte block load from primary address space, little-endian, user privilege</td>
<td>ldda [regaddr] #ASI_BLK_AIUPL, freg</td>
<td>D2</td>
</tr>
<tr>
<td>LDBLOCKFD</td>
<td>1F16</td>
<td>64-byte block load from secondary address space, little-endian, user privilege</td>
<td>ldda [regaddr] #ASI_BLK_AIUSL, freg</td>
<td>D2</td>
</tr>
<tr>
<td>LDBLOCKFD</td>
<td>F016</td>
<td>64-byte block load from primary address space</td>
<td>ldda [regaddr] #ASI_BLK_P, freg</td>
<td>D2</td>
</tr>
<tr>
<td>LDBLOCKFD</td>
<td>F116</td>
<td>64-byte block load from secondary address space</td>
<td>ldda [regaddr] #ASI_BLK_S, freg</td>
<td>D2</td>
</tr>
<tr>
<td>LDBLOCKFD</td>
<td>F816</td>
<td>64-byte block load from primary address space, little-endian</td>
<td>ldda [regaddr] #ASI_BLK_P, freg</td>
<td>D2</td>
</tr>
<tr>
<td>LDBLOCKFD</td>
<td>F916</td>
<td>64-byte block load from secondary address space, little-endian</td>
<td>ldda [regaddr] #ASI_BLK_S, freg</td>
<td>D2</td>
</tr>
</tbody>
</table>

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.
LDBLOCKF

Programming Note

LDBLOCKF is intended to be a processor-specific instruction (see the warning at the top of page 178). 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 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.cef = 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), if bit 7 of the ASI is 0 (ASIs 16₁₆, 17₁₆, 1E₁₆, and 1F₁₆), LDBLOCKF causes a privileged_action exception.

An access caused by LDBLOCKF may trigger a VA_watchpoint exception (impl. dep. #410-S10).

An attempted access by an LDBLOCKF instruction to noncacheable memory causes an a DAE_nc_page exception.
LDBLOCKF

**Implementation**

LDBLOCKF shares an opcode with LDDFA and LDSHORTF; it is distinguished by the ASI used.

**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)

**See Also**

STBLOCKF on page 250
7.52  Load Floating-Point Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>rd</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDF</td>
<td>10 0000</td>
<td>0–31</td>
<td>Load Floating-Point Register</td>
<td>ld [address], freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>LDDF</td>
<td>10 0011</td>
<td>‡</td>
<td>Load Double Floating-Point Register</td>
<td>ldd [address], freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>LDQF</td>
<td>10 0010</td>
<td>‡</td>
<td>Load Quad Floating-Point Register</td>
<td>ldq [address], freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>C3</td>
</tr>
</tbody>
</table>

‡ 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<sub>S</sub>[rd].

The load doubleword floating-point instruction (LDDF) copies a word-aligned doubleword from memory into a 64-bit floating-point destination register, F<sub>D</sub>[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<sub>Q</sub>[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 76).

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 (FPRE.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.
LDF / LDDF / LDQF

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\ 

**See Also**
- Load Floating-Point from Alternate Space on page 183
- Load Floating-Point State Register (Lower) on page 186
- Store Floating-Point on page 253
7.53 Load Floating-Point from Alternate Space

**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 “$R[rs1] + \text{sign_ext}(simm13)$”.

**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** | 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)).
LDFA / LDDFA / LDQFA

An attempt to execute an LDQFA instruction when rd[1] ≠ 0 causes a *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*), if bit 7 of the ASI is 0, this instruction causes a *privileged_action* exception. In privileged mode (*PSTATE.priv = 1*), if the ASI is in the range 30₁₆ to 7F₁₆, 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.

<table>
<thead>
<tr>
<th>ASIs valid for LDFA and LDQFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_REAL</td>
</tr>
<tr>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
</tr>
<tr>
<td>ASI_PRIMARY_NO_FAULT</td>
</tr>
<tr>
<td>ASI_SECONDARY_NO_FAULT</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>ASIs valid for LDDFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_REAL</td>
</tr>
<tr>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>ASIPRIMARY</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
</tr>
<tr>
<td>ASI_PRIMARY_NO_FAULT</td>
</tr>
<tr>
<td>ASI_SECONDARY_NO_FAULT</td>
</tr>
</tbody>
</table>

**Behavior with Block-Store-with-Commit ASIs**

ASIs E₀₁₆ and E₁₁₆ are only defined for use in Block Store with Commit operations (see page 250). Neither ASI E₀₁₆ nor E₁₁₆ should be used with LDDFA; however, if it is used, the LDDFA behaves as follows:
LDFA / LDDFA / LDQFA

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 260). 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_nlo_page
- DAE_side_effect_page

**See Also**

- Load Floating-Point Register on page 181
- Block Load on page 178
- Store Short Floating-Point on page 263
- Store Floating-Point into Alternate Space on page 255
7.54 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.

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 42).

LDFSR accesses memory using the implicit ASI (see page 76).

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.fef = 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
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.

Implementation
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 \( \text{op} = 112, \text{op3} = 1000012 \) opcode with an invalid rd value causes an illegal_instruction exception.

Exceptions
illegal_instruction
fp_disabled
mem_address_not_aligned
VA_watchpoint
LDFSR (Deprecated)

DAE_privilege_violation
DAE_nfo_page

See Also
Load Floating-Point Register on page 181
Load Floating-Point State Register on page 199
Store Floating-Point on page 253
7.55 Short Floating-Point Load [VIS1]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASI Value</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDSHORTF D016</td>
<td>8-bit load from primary address space</td>
<td>ldda [regaddr] #ASI_FL8_P, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D116</td>
<td>8-bit load from secondary address space</td>
<td>ldda [regaddr] #ASI_FL8_S, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D816</td>
<td>8-bit load from primary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL8_PL, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D916</td>
<td>8-bit load from secondary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL8_SL, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D216</td>
<td>16-bit load from primary address space</td>
<td>ldda [regaddr] #ASI_FL16_P, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D316</td>
<td>16-bit load from secondary address space</td>
<td>ldda [regaddr] #ASI_FL16_S, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF DA16</td>
<td>16-bit load from primary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL16_PL, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF DB16</td>
<td>16-bit load from secondary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL16_SL, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
</tbody>
</table>

**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.<sup>ef</sup> = 0 or PSTATE.<sup>pef</sup> = 0) or if no FPU is present, an attempt to execute an LDSHORTF instruction causes an \textit{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 \textit{mem_address_not_aligned} exception.

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 \textit{Align Address} on page 98) 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**

\textit{fp_disabled}

\textit{mem_address_not_aligned}

\textit{VA_watchpoint}
DAE_privilege_violation
DAE_nfo_page
7.56  Load-Store Unsigned Byte

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDSTUB</td>
<td>00 1101</td>
<td>Load-Store Unsigned Byte</td>
<td>ldstub [address], regrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

**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 76). 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
7.57 Load-Store Unsigned Byte to Alternate Space

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), if bit 7 of the ASI is 0, this instruction causes a privileged_action exception. In privileged mode (PSTATE.priv = 1), 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.

<table>
<thead>
<tr>
<th>ASIs valid for LDSTUBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_REAL</td>
</tr>
<tr>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
</tr>
</tbody>
</table>

Exceptions
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nc_page
DAE_nfo_page
7.58 Load Integer Twin Word

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.

Load integer twin word instructions access memory using the implicit ASI (see page 76). If \( i = 0 \), the effective address for these instructions is "\( R[rs1] + R[rs2] \)" and if \( i = 0 \), the effective address is "\( R[rs1] + \text{sign_ext}(\text{simm13}) \)".

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.

**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.
LDTW (Deprecated)

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`

**See Also**
- LDW/LDX on page 175
- STTW on page 265
7.59 Load Integer Twin Word from Alternate Space

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.

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 “\( R[rs1] + \text{sign_ext}(\text{simm13}) \)”. 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.

**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 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.
LDTWA (Deprecated)

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), if bit 7 of the ASI is 0, these instructions cause a privileged_action exception. In privileged mode (PSTATE.priv = 1), if the ASI is in the range 30\textsubscript{16} to 7F\textsubscript{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).

- ASI_NUCLEUS
- ASI_REAL
- ASI_REAL_IO
- 22\textsubscript{16}‡ (ASI_TWINX_AIUP)
- 23\textsubscript{16}‡ (ASI_TWINX_AIUS)
- 26\textsubscript{16}‡ (ASI_TWINX_REAL)
- 27\textsubscript{16}‡ (ASI_TWINX_N)
- ASI_PRIMARY
- ASI_SECONDARY
- ASI_PRIMARY_NO_FAULT
- ASI_SECONDARY_NO_FAULT
- EA\textsubscript{16}‡ (ASI_TWINX_P)
- EB\textsubscript{16}‡ (ASI_TWINX_S)
- E2\textsubscript{16}‡ (ASI_TWINX_P)
- EB\textsubscript{16}‡ (ASI_TWINX_S)

‡ 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 197.

Programming Note

Nontranslating ASIs (see page 321) 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 197.

Exceptions

- unimplemented_LDTW (not used in UltraSPARC Architecture 2007)
- illegal_instruction
- mem_address_not_aligned
- privileged_action
- VA_watchpoint

CHAPTER 7 • Instructions 195
LDTWA (Deprecated)

- DAE_invalid asi
- DAE_privilege_violation
- DAE_nfo_page
- DAE_side_effect_page

See Also

- LDWA/LDXA on page 176
- LDTXA on page 197
- STTWA on page 267
7.60 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 or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASI Value</th>
<th>Operation</th>
<th>Assembly Language Syntax †</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDTXAN</td>
<td>2216</td>
<td>Load Integer Twin Extended Word, as if user (nonprivileged), Primary address space</td>
<td>ldtxa [regaddr] #ASI_TWINX_AIUP, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>2316</td>
<td>Load Integer Twin Extended Word, as if user (nonprivileged), Secondary address space</td>
<td>ldtxa [regaddr] #ASI_TWINX_AIUS, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>2616</td>
<td>Load Integer Twin Extended Word, real address</td>
<td>ldtxa [regaddr] #ASI_TWINX_REAL, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>2716</td>
<td>Load Integer Twin Extended Word, nucleus context</td>
<td>ldtxa [regaddr] #ASI_TWINX_N, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>2A16</td>
<td>Load Integer Twin Extended Word, as if user (nonprivileged), Primary address space, little endian</td>
<td>ldtxa [regaddr] #ASI_TWINX_AIUP_L, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>2B16</td>
<td>Load Integer Twin Extended Word, as if user (nonprivileged), Secondary address space, little endian</td>
<td>ldtxa [regaddr] #ASI_TWINX_AIUS_L, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>2E16</td>
<td>Load Integer Twin Extended Word, real address, little endian</td>
<td>ldtxa [regaddr] #ASI_TWINX_REAL_L, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>2F16</td>
<td>Load Integer Twin Extended Word, nucleus context, little-endian</td>
<td>ldtxa [regaddr] #ASI_TWINX_NL, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td>LDTXAN</td>
<td>E216</td>
<td>Load Integer Twin Extended Word, Primary address space</td>
<td>ldtxa [regaddr] #ASI_TWINX_P, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>E316</td>
<td>Load Integer Twin Extended Word, Secondary address space</td>
<td>ldtxa [regaddr] #ASI_TWINX_S, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>EA16</td>
<td>Load Integer Twin Extended Word, Primary address space, little endian</td>
<td>ldtxa [regaddr] #ASI_TWINX_PL, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>EB16</td>
<td>Load Integer Twin Extended Word, Secondary address space, little-endian</td>
<td>ldtxa [regaddr] #ASI_TWINX_SL, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N–</td>
</tr>
</tbody>
</table>

† 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.

![Instruction Format Diagram](image-url)
Description  ASIs 26₁₆, 2E₁₆, E₂₁₆, E₃₁₆, F₀₁₆, and F₁₁₆ 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 E₂₁₆, E₃₁₆, F₀₁₆, and F₁₁₆ perform an access using a virtual address, while ASIs 2₆₁₆ and 2E₁₆ 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 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 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.

The virtual processor MMU does not provide virtual-to-real translation for ASIs 2₆₁₆ and 2E₁₆; the effective address provided with either of those ASIs is interpreted directly as a real address.

Compatibility Note  ASIs 2₇₁₆, 2F₁₆, 2₆₁₆, and 2E₁₆ are now standard ASIs that replace (respectively) ASIs 2₄₁₆, 2C₁₆, 3₄₁₆, and 3C₁₆ 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 194.

Exceptions  illegal_instruction
mem_address_not_aligned
privileged_action
VA_watchpoint (impl. dep. #413-S10)
DAE_nc_page
DAE_nfo_page

See Also  LDTWA on page 194
7.61 Load Floating-Point State Register

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 42).

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 76).

If \( i = 0 \), the effective address for these instructions is \( R[rs1] + R[rs2] \) and if \( i = 0 \), the effective address is \( R[rs1] + \text{sign_ext}(\text{simm}13) \).

Exceptions. An attempt to execute an instruction encoded as \( \text{op} = 2 \) and \( \text{op}3 = 21_{16} \) 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 \( \text{op} = 11_{2}, \text{op}3 = 10001_{2} \) 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

See Also

Load Floating-Point Register on page 181
Load Floating-Point State Register (Lower) on page 186
Store Floating-Point State Register on page 269
7.62 Memory Barrier

**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**

<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Assembly Language Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mmask[3]</code></td>
<td><code>#StoreStore</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>mmask[2]</code></td>
<td><code>#LoadStore</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>mmask[1]</code></td>
<td><code>#StoreLoad</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>mmask[0]</code></td>
<td><code>#LoadLoad</code></td>
<td>All loads appearing prior to the MEMBAR instruction must have been performed before any loads following the MEMBAR may be performed.</td>
</tr>
</tbody>
</table>
The \texttt{cmask} field is encoded in bits 6 through 4 of the instruction. Bits in the \texttt{cmask} field, described in \textbf{TABLE 7-8}, specify additional constraints on the order of memory references and the processing of instructions. If \texttt{cmask} is zero, then MEMBAR enforces the partial ordering specified by the \texttt{mmask} field; if \texttt{cmask} is nonzero, then completion and partial order constraints are applied.

\begin{table}[ht]
\centering
\begin{tabular}{|c|c|c|}
\hline
\text{Mask Bit} & \text{Function} & \text{Assembly Language Name} \\
\hline
\text{cmask}[2] & \text{Synchronization barrier} & \texttt{#Sync} \\
\hline
\text{cmask}[1] & \text{Memory issue barrier} & \texttt{#MemIssue} \\
\hline
\text{cmask}[0] & \text{Lookaside barrier} & \texttt{#Lookaside} \\
\hline
\end{tabular}
\caption{MEMBAR \texttt{cmask} Encodings}
\end{table}

A MEMBAR instruction with both \texttt{mmask} = 0 and \texttt{cmask} = 0 is functionally a NOP.

For information on the use of MEMBAR, see Memory Ordering and Synchronization on page 316 and Programming with the Memory Models contained in the separate volume \textit{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).

\textbf{V9 Compatibility Note} \quad MEMBAR with \texttt{mmask} = 8_{16} and \texttt{cmask} = 0_{16} (MEMBAR \texttt{#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 \texttt{illegal_instruction} exception.

\textbf{Implementation Note} \quad MEMBAR shares an opcode with RDasr; it is distinguished by \texttt{rs1} = 15, \texttt{rd} = 0, \texttt{i} = 1, and bit 12 = 0.

\section*{7.62.1 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.

\textbf{IMPL. DEP. #412-S10}: An UltraSPARC Architecture implementation may define the operation of each MEMBAR variant in any manner that provides the required semantics.
7.62.2 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.

Completion of a MEMBAR \#Sync instruction does not guarantee that data previously stored has been written all the way out to external memory. 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.

7.62.3 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 315.

Exceptions illegal_instruction
7.63 Move Integer Register on Condition (MOVcc)

For Integer Condition Codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>cond</th>
<th>Operation</th>
<th>icc / xcc Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVA</td>
<td>10 1100</td>
<td>1000</td>
<td>Move Always</td>
<td>1</td>
<td>mova i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVN</td>
<td>10 1100</td>
<td>0000</td>
<td>Move Never</td>
<td>0</td>
<td>movn i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVNE</td>
<td>10 1100</td>
<td>1001</td>
<td>Move if Not Equal</td>
<td>not Z</td>
<td>movne† i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVE</td>
<td>10 1100</td>
<td>0001</td>
<td>Move if Equal</td>
<td>Z</td>
<td>move‡ i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVG</td>
<td>10 1100</td>
<td>1010</td>
<td>Move if Greater</td>
<td>not (Z or N xor V))</td>
<td>movg i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVLE</td>
<td>10 1100</td>
<td>0010</td>
<td>Move if Less or Equal</td>
<td>Z or (N xor V)</td>
<td>movle i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVGE</td>
<td>10 1100</td>
<td>1011</td>
<td>Move if Greater or Equal</td>
<td>not (N xor V)</td>
<td>movge i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVL</td>
<td>10 1100</td>
<td>0011</td>
<td>Move if Less</td>
<td>N xor V</td>
<td>movl i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVGU</td>
<td>10 1100</td>
<td>1100</td>
<td>Move if Greater, Unsigned</td>
<td>not (C or Z)</td>
<td>movgu i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVLEU</td>
<td>10 1100</td>
<td>0100</td>
<td>Move if Less or Equal, Unsigned</td>
<td>(C or Z)</td>
<td>movleu i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVCC</td>
<td>10 1100</td>
<td>1101</td>
<td>Move if Carry Clear (Greater or Equal, Unsigned)</td>
<td>not C</td>
<td>movcc◊ i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVCS</td>
<td>10 1100</td>
<td>0101</td>
<td>Move if Carry Set (Less than, Unsigned)</td>
<td>C</td>
<td>movcs V i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVPOS</td>
<td>10 1100</td>
<td>1110</td>
<td>Move if Positive</td>
<td>not N</td>
<td>movpos i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVNEG</td>
<td>10 1100</td>
<td>0110</td>
<td>Move if Negative</td>
<td>N</td>
<td>movneg i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVVC</td>
<td>10 1100</td>
<td>1111</td>
<td>Move if Overflow Clear</td>
<td>not V</td>
<td>movvc i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVVS</td>
<td>10 1100</td>
<td>0111</td>
<td>Move if Overflow Set</td>
<td>V</td>
<td>movvs i_or_x_cc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonym: movnz ‡ synonym: movz ◊ synonym: movgeu V synonym: movlu

Programming Note: In assembly language, to select the appropriate condition code, include %icc or %xcc before the reg_or_imm11 field.
MOVcc

For Floating-Point Condition Codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>cond</th>
<th>Operation</th>
<th>fcc Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVFA</td>
<td>10</td>
<td>1100</td>
<td>Move Always</td>
<td>1</td>
<td>mova %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFN</td>
<td>10</td>
<td>1100</td>
<td>Move Never</td>
<td>0</td>
<td>movn %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFU</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered</td>
<td>U</td>
<td>movu %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFG</td>
<td>10</td>
<td>1110</td>
<td>Move if Greater</td>
<td>G</td>
<td>movg %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFUG</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Greater</td>
<td>G or U</td>
<td>movug %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFL</td>
<td>10</td>
<td>1100</td>
<td>Move if Less</td>
<td>L</td>
<td>movl %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFUL</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Less</td>
<td>L or U</td>
<td>movul %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFLG</td>
<td>10</td>
<td>1100</td>
<td>Move if Less or Greater</td>
<td>L or G</td>
<td>movlg %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFNE</td>
<td>10</td>
<td>1100</td>
<td>Move if Not Equal</td>
<td>L or G or U</td>
<td>movne† %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFE</td>
<td>10</td>
<td>1100</td>
<td>Move if Equal</td>
<td>E</td>
<td>move† %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFUE</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Equal</td>
<td>E or U</td>
<td>movue %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFGE</td>
<td>10</td>
<td>1100</td>
<td>Move if Greater or Equal</td>
<td>E or G</td>
<td>movge %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFUGE</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Greater or Equal</td>
<td>E or G or U</td>
<td>movuge %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFLE</td>
<td>10</td>
<td>1100</td>
<td>Move if Less or Equal</td>
<td>E or L</td>
<td>movle %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFULE</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Less or Equal</td>
<td>E or L or U</td>
<td>movule %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFO</td>
<td>10</td>
<td>1110</td>
<td>Move if Ordered</td>
<td>E or L or G</td>
<td>movo %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

† 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.

<table>
<thead>
<tr>
<th>rd</th>
<th>op3</th>
<th>cc2</th>
<th>cond</th>
<th>i=0 cc1 cc0</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rd</th>
<th>op3</th>
<th>cc2</th>
<th>cond</th>
<th>i=1 cc1 cc0</th>
<th>simm11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cc2</th>
<th>cc1</th>
<th>cc0</th>
<th>Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>%fcc0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>%fcc1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>%fcc2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>%fcc3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>icc</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Reserved (illegal_instruction)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>xcc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Reserved (illegal_instruction)</td>
</tr>
</tbody>
</table>
**MOVcc**

**Description**

These instructions test to see if \texttt{cond} is \textsc{true} for the selected condition codes. If so, they copy the value in \texttt{R[rs2]} if \texttt{i field} = 0, or \texttt{\texttt{sign}_\texttt{ext}(simm11)} if \texttt{i} = 1 into \texttt{R[rd]}. The condition code used is specified by the \texttt{cc2}, \texttt{cc1}, and \texttt{cc0} fields of the instruction. If the condition is \textsc{false}, then \texttt{R[rd]} is not changed.

These instructions copy an integer register to another integer register if the condition is \textsc{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

\[
\text{if (} A > B \text{) then } X = 1; \text{ else } X = 0; 
\]

can be coded as

\begin{verbatim}
cmp %i0,%i2
bg,a %xcc,label
or %g0,1,%i3! X = 1
or %g0,0,%i3! X = 0
\end{verbatim}

\text{label:...}

The above sequence requires four instructions, including a branch. With MOVcc this could be coded as:

\begin{verbatim}
cmp %i0,%i2
or %g0,1,%i3! assume X = 1
movle %xcc,0,%i3! overwrite with X = 0
\end{verbatim}

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 \((\texttt{cc2}::\texttt{cc1}::\texttt{cc0}) = 101_2\) or \(111_2\) causes an \textsc{illegal\_instruction} exception.

If \texttt{cc2} = 0 (that is, a floating-point condition code is being referenced in the MOVcc instructions) and either the FPU is not enabled (\texttt{FPRS.fef = 0} or \texttt{PSTATE.pef} = 0) or if no FPU is present, an attempt to execute a MOVcc instruction causes an \textsc{fp\_disabled} exception.

**Exceptions**

- \textsc{illegal\_instruction}
- \textsc{fp\_disabled}
7.64 Move Integer Register on Register Condition (MOVr)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>rcond</th>
<th>Operation</th>
<th>Test</th>
<th>Assembly Language Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVNZ</td>
<td>10</td>
<td>1111</td>
<td>Move if Register Zero</td>
<td>(R[rs1] = 0)</td>
<td>movnz(\dagger) reg(\dagger)rs1, reg(\dagger)or(\dagger)imm10, reg(\dagger)rd</td>
</tr>
<tr>
<td>MOVZ</td>
<td>10</td>
<td>1111</td>
<td>Move if Register Less Than or Equal to Zero</td>
<td>(R[rs1] \leq 0)</td>
<td>movrzez reg(\dagger)rs1, reg(\dagger)or(\dagger)imm10, reg(\dagger)rd</td>
</tr>
<tr>
<td>MOVNZ</td>
<td>10</td>
<td>1111</td>
<td>Move if Register Less Than Zero</td>
<td>(R[rs1] &lt; 0)</td>
<td>movnznz reg(\dagger)rs1, reg(\dagger)or(\dagger)imm10, reg(\dagger)rd</td>
</tr>
<tr>
<td>MOVZ</td>
<td>10</td>
<td>1111</td>
<td>Move if Register Greater Than Zero</td>
<td>(R[rs1] &gt; 0)</td>
<td>movzregz reg(\dagger)rs1, reg(\dagger)or(\dagger)imm10, reg(\dagger)rd</td>
</tr>
<tr>
<td>MOVNZ</td>
<td>10</td>
<td>1111</td>
<td>Move if Register Greater Than or Equal to Zero</td>
<td>(R[rs1] \geq 0)</td>
<td>movnzngez reg(\dagger)rs1, reg(\dagger)or(\dagger)imm10, reg(\dagger)rd</td>
</tr>
</tbody>
</table>

\(\dagger\) synonym: movre
\(\dagger\) 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 \(i = 1\), \(\text{sign_ext}(\text{simm10})\)) 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.

<table>
<thead>
<tr>
<th>Move</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVNZ</td>
<td>not (Z)</td>
</tr>
<tr>
<td>MOVZ</td>
<td>(Z)</td>
</tr>
<tr>
<td>MOVREGZ</td>
<td>not (N)</td>
</tr>
<tr>
<td>MOVRLZ</td>
<td>(N)</td>
</tr>
<tr>
<td>MOVRELZ</td>
<td>(N) or (Z)</td>
</tr>
<tr>
<td>MOVREGZ</td>
<td>(N) or (Z)</td>
</tr>
</tbody>
</table>

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.
MOVr

Exceptions

illegal_instruction
7.65 Multiply Step

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 \texttt{sign\_ext}\texttt{(simm13)}.
2. A 32-bit value is computed by shifting the value from \( R[rs1] \) right by one bit with “\texttt{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:

<table>
<thead>
<tr>
<th>Register field</th>
<th>Value written by MULScc</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCR.icc</td>
<td>updated according to the result of the addition in step (3) above</td>
</tr>
<tr>
<td>( R[rd][63:33] )</td>
<td>0</td>
</tr>
<tr>
<td>( R[rd][32] )</td>
<td>CCR.icc.c</td>
</tr>
<tr>
<td>( R[rd][31:0] )</td>
<td>the least-significant 32 bits of the sum from step (3) above</td>
</tr>
<tr>
<td>( Y )</td>
<td>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)</td>
</tr>
<tr>
<td>CCR.xcc.n</td>
<td>0</td>
</tr>
<tr>
<td>CCR.xcc.v</td>
<td>0</td>
</tr>
<tr>
<td>CCR.xcc.c</td>
<td>0</td>
</tr>
<tr>
<td>CCR.xcc.z</td>
<td>if ( (R[rd][63:0] = 0) ) then 1 else 0</td>
</tr>
</tbody>
</table>
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 \textit{illegal_instruction} exception.

### Exceptions

- \textit{illegal_instruction}

### See Also

- RDY on page 225
- SDIV, SDIVcc on page 240
- SMUL, SMULcc on page 246
- UDIV, UDIVcc on page 281
- UMUL, UMULcc on page 283

---

**SPARC V9 Compatibility Note**

In SPARC V9, MULScc’s effect on \( R[rd][63:32] \) and \( CCR.xcc \) were explicitly left undefined.
7.66  Multiply and Divide (64-bit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULX</td>
<td>001001</td>
<td>Multiply (signed or unsigned)</td>
<td>mulx reg&lt;rs1&gt;, reg_or_imm, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>SDIVX</td>
<td>101101</td>
<td>Signed Divide</td>
<td>sdivx reg&lt;rs1&gt;, reg_or_imm, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>UDIVX</td>
<td>001101</td>
<td>Unsigned Divide</td>
<td>udivx reg&lt;rs1&gt;, reg_or_imm, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

MULX computes “$R[rs1] \times R[rs2]$” if $i = 0$ or “$R[rs1] \times \text{sign\_ext}(\text{simm13})$” 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 “$R[rs1] \div R[rs2]$” if $i = 0$ or “$R[rs1] \div \text{sign\_ext}(\text{simm13})$” 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:

$$8000\,0000\,0000\,0000_{16} \div \text{FFFF\,FFFF\,FFFF\,FFFF}_{16} = 8000\,0000\,0000\,0000_{16}.$$  

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`
### No Operation

**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
7.68 NORMALW

NORMALW

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORMALWP</td>
<td>“Other” register windows become “normal” register windows</td>
<td>normalw</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

NORMALWP 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.

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) causes a privileged_opcode exception.

**Exceptions**

illegal_instruction

privileged_opcode

**See Also**

ALLCLEAN on page 99
INVALW on page 173
OTHERW on page 215
RESTORED on page 232
SAVED on page 239
7.69   OR Logical Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>00 0010</td>
<td>Inclusive or</td>
<td><code>or rs1, reg_or_imm, regrd</code></td>
<td>A1</td>
</tr>
<tr>
<td>ORcc</td>
<td>01 0010</td>
<td>Inclusive or and modify cc’s</td>
<td><code>orcc rs1, reg_or_imm, regrd</code></td>
<td>A1</td>
</tr>
<tr>
<td>ORN</td>
<td>00 0110</td>
<td>Inclusive or not</td>
<td><code>orn rs1, reg_or_imm, regrd</code></td>
<td>A1</td>
</tr>
<tr>
<td>ORNcc</td>
<td>01 0110</td>
<td>Inclusive or not and modify cc’s</td>
<td><code>orncc rs1, reg_or_imm, regrd</code></td>
<td>A1</td>
</tr>
</tbody>
</table>

**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*
7.70 OTHERW

*Description*
OTHERWP 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.

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) causes a privileged_opcode exception.

*Exceptions*
illegal_instruction
privileged_opcode

*See Also*
ALLCLEAN on page 99
INVALW on page 173
NORMALW on page 213
RESTORED on page 232
SAVED on page 239
7.71 Pixel Component Distance (with Accumulation) [VIS 1]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDIST</td>
<td>00011110</td>
<td>Distance between eight 8-bit components, with accumulation</td>
<td>pdist freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt; C2</td>
<td></td>
</tr>
</tbody>
</table>

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 (FPRSfef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FPMERGE instruction causes an fp_disabled exception.
7.72 Population Count

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>POPC</td>
<td>10</td>
<td>Population Count</td>
<td>popc     reg_or_imm, reg_rd</td>
<td>C2</td>
</tr>
</tbody>
</table>

**Description**

POPC counts the number of ‘1’ bits in \( R[rs2] \) if \( i = 0 \), or the number of ‘1’ 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:

```c
int ffs(int) /* finds first 1 bit, counting from the LSB */
unsigned in;
{
  return popc(in ^ (¬(–in))); /* for nonzero zz */
}
```

Inline assembly language code for `ffs()` is:

```assembly
neg %IN, %NEG_IN, %0, %1 ! -zz(2’s complement)
xor %IN, %NEG_IN, %0, %1 ! exclusive nor
popc %0, %RESULT ! result = popc(zz ^ -zz)
movrz %IN, %RESULT, %0, %1 ! %RESULT should be 0 for %IN=0
```

where %IN, M_IN, TEMP, and RESULT are integer registers.

Example computation:

```plaintext
IN = ...00101000 ! 1st ‘1’ bit from right is 3 (4th bit)
¬IN = ...11011000
~¬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:

```assembly
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 !
movc %0, %DEST, %result ! 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
Prefetch

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
PREFETCH

If \( i = 0 \), the effective address operand for the PREFETCH instruction is \( R[rs1] + R[rs2] \); if \( i = 1 \), it is \( R[rs1] + \text{sign_ext} (\text{simm13}) \).

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, but with certain important differences. In particular, a PREFETCHA instruction is non-blocking; subsequent instructions can continue to execute while the prefetch is in progress.

<table>
<thead>
<tr>
<th>Implementation Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>A PREFETCHA 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.</td>
</tr>
</tbody>
</table>

When executed in nonprivileged or privileged mode, PREFETCHA 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))

**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.

<table>
<thead>
<tr>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software may prefetch 64 bytes beginning at an arbitrary address by issuing the instructions</td>
</tr>
<tr>
<td>\text{prefetch} [address], \text{prefetch} fcn</td>
</tr>
<tr>
<td>\text{prefetch} [address + 63], \text{prefetch} fcn</td>
</tr>
</tbody>
</table>

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 PREFETCHA 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.

### 7.73.1 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 PREFETCHA Instructions Under Exceptional Conditions (1 of 2)

<table>
<thead>
<tr>
<th>fcn</th>
<th>Instruction</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>PREFETCH</td>
<td>( i = 0 ) and instruction bits 12:5 are nonzero</td>
<td>illegal_instruction</td>
</tr>
<tr>
<td>any</td>
<td>PREFETCHA</td>
<td>reference to an ASI in the range ( 0_{16}^{16}\text{7F}_{16} ), while in nonprivileged mode (privileged_action condition)</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>any</td>
<td>PREFETCHA</td>
<td>reference to an ASI in range ( 30_{16}\text{7F}_{16} ), while in privileged mode (privileged_action condition)</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-3 (weak)</td>
<td>PREFETCHA</td>
<td>condition detected for MMU miss</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-4</td>
<td>PREFETCHA</td>
<td>variant unimplemented</td>
<td>executes as NOP</td>
</tr>
</tbody>
</table>
7.73.2 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 in order to complete. If a time-consuming operation is required, a Weak prefetch will abandon the operation and behave like a
PREFETCH

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Load</th>
<th>Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upon detection of privileged_action, DAE_* or VA_watchpoint exception…</td>
<td>Traps</td>
<td>NOP‡</td>
</tr>
<tr>
<td>If page table entry has cp = 0, e = 1, and cv = 0 for Prefetch for Several Reads</td>
<td>Traps</td>
<td>NOP‡</td>
</tr>
<tr>
<td>If page table entry has nfo = 1 for a non-NoFault access…</td>
<td>Traps</td>
<td>NOP‡</td>
</tr>
<tr>
<td>If page table entry has w = 0 for any prefetch for write access (fcn = 2, 3, 22, or 23)…</td>
<td>Traps</td>
<td>NOP‡</td>
</tr>
<tr>
<td>Instruction blocks until cache line filled?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

7.73.3 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 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).

7.73.3.1 Prefetch for Several Reads (fcn = 0, 20(1416))

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.

7.73.3.2 Prefetch for One Read (fcn = 1, 21(1516))

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.
PREFETCH

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.

7.73.3.3 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.

7.73.3.4 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.

7.73.3.5 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.

7.73.3.6 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.
PREFETCH

7.73.4 Implementation-Dependent Prefetch Variants ($fcn = 16, 18, 19, \text{ and } 24-31$)

**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.

7.73.5 Additional Notes

<table>
<thead>
<tr>
<th>Programming Note</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Note</td>
<td>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.</td>
</tr>
<tr>
<td>Implementation Note</td>
<td>Any effects of a data prefetch operation in privileged code should be reasonable (for example, no page prefetching is allowed within code that handles page faults). The benefits of prefetching should be available to most privileged code.</td>
</tr>
<tr>
<td>Implementation Note</td>
<td>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.</td>
</tr>
</tbody>
</table>

Exceptions illegal_instruction
### 7.74 Read Ancillary State Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rs1</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDY\textsuperscript{D}</td>
<td>0</td>
<td>Read Y register (deprecated)</td>
<td>rd %y, %reg\text{rd}</td>
<td>D2</td>
</tr>
<tr>
<td>—</td>
<td>1</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDCCR</td>
<td>2</td>
<td>Read Condition Codes register (CCR)</td>
<td>rd %ccr, %reg\text{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>RDASI</td>
<td>3</td>
<td>Read ASI register</td>
<td>rd %asi, %reg\text{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>RDTICK\textsuperscript{opt}</td>
<td>4</td>
<td>Read TICK register</td>
<td>rd %tick, %reg\text{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>RDPC</td>
<td>5</td>
<td>Read Program Counter (PC)</td>
<td>rd %pc, %reg\text{rd}</td>
<td>A2</td>
</tr>
<tr>
<td>RDFPRS</td>
<td>6</td>
<td>Read Floating-Point Registers Status (FPRS) register</td>
<td>rd %fprs, %reg\text{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>7–14</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>15 (F\text{16})</td>
<td>MEMBAR or Reserved; see text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>16–18</td>
<td>Reserved (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>19 (13\text{16})</td>
<td>Read General Status register (GSR)</td>
<td>rd %gsr, %reg\text{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>20–21</td>
<td>Reserved (impl. dep. #8-V8-Cs20, #9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>22 (16\text{16})</td>
<td>Read per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>rd %softint, %reg\text{rd}</td>
<td>A2</td>
</tr>
<tr>
<td>—</td>
<td>23 (17\text{16})</td>
<td>Read Tick Compare register (TICK_CMPR)</td>
<td>rd %tick_cmp\text{r}, %reg\text{rd}</td>
<td>N–</td>
</tr>
<tr>
<td>—</td>
<td>24 (18\text{16})</td>
<td>Read System Tick Register (STICK)</td>
<td>rd %stick\text{t}, %reg\text{rd}</td>
<td>A2</td>
</tr>
<tr>
<td>—</td>
<td>25 (19\text{16})</td>
<td>Read System Tick Compare register (STICK_CMPR)</td>
<td>rd %stick_cmp\text{r}, %reg\text{rd}</td>
<td>A2</td>
</tr>
<tr>
<td>—</td>
<td>26 (20\text{16})</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, #9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>27 (1B\text{16})</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>28 (1C\text{16})</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>29 (1D\text{16})</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>30 (1E\text{16})</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>31 (1F\text{16})</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\* The original assembly language names for \%tick and \%tick\_cmp\text{r} were, respectively, \%sys\_tick and \%sys\_tick\_cmp\text{r}, which are now deprecated. Over time, assemblers will support the new \%tick and \%tick\_cmp\text{r} names for these registers (which are consistent with \%tick and \%tick\_cmp\text{r}). 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 (impl. dep. #9-V8-Cs20), and
■ whether an attempt to execute the instruction causes an illegal_instruction exception.

Implementation

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

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 48 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 RDTICK_CMPR, RDSTICK_CMPR, or RDSOFTINT instruction in nonprivileged mode (PSTATE.priv = 0) causes a privileged_opcode exception (impl. dep. #250-U3-Cs10).

Nonprivileged software can read the TICK register by using the RDTICK instruction, but only when nonprivileged access to TICK is enabled. If nonprivileged access is disabled, an attempt by nonprivileged software to read the TICK register using the RDTICK instruction causes a privileged_action exception. See Tick (tick) Register (ASR 4) on page 52 for details.
Nonprivileged software can read the STICK register by using the RDSTICK instruction, but only when nonprivileged access to STICK is enabled. If nonprivileged access is disabled, an attempt by nonprivileged software to read the STICK register causes a privileged_action exception. See System Tick (stick) Register (ASR 24) on page 57 for details.

Privileged software can read the STICK register with the RDSTICK instruction, but only when privileged access to STICK is enabled by hyperprivileged software. An attempt by privileged software to read the STICK register when privileged access is disabled causes a privileged_action exception. See System Tick (stick) Register (ASR 24) on page 57 for details.

If the FPU is not enabled (FPRS.fef = 0 or STATE.pef = 0) or if no FPU is present, an attempt to execute a RDGSR instruction causes an fp_disabled exception.

In nonprivileged mode (STATE.priv = 0), the following cause a privileged_action exception:
- execution of RDTICK when nonprivileged access to TICK is disabled
- execution of RDSTICK when nonprivileged access to STICK is disabled

Implementation

Note

RDasr shares an opcode with MEMBAR; 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

RDPR on page 228
WRasr on page 285
7.75 Read Privileged Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>rs1</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDPR</td>
<td>10</td>
<td>rdpr</td>
<td>0</td>
<td>%tpc, rsrd</td>
<td>A2?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>%tnpc, rsrd</td>
<td>A1?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>%tstate, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>%tt, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>%tick, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>%tba, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>%pstate, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>%tl, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>%pil, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>%cwp, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>%cansave, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>%canrestore, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>%cleanwin, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>%otherwin, rsrd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>%wstate, rsrd</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL</td>
<td>16</td>
<td>rdpr</td>
<td></td>
<td>%gl, rsrd</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>17–31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The rs1 field in the instruction determines the privileged register that is read. There are MAXPTL 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 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.

An attempt to execute a RDPR instruction in nonprivileged mode (PSTATE.priv = 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 225
WRPR on page 288
7.76 RESTORE

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, \( N_{\text{REG\_WINDOWS}} \).

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 \( N_{\text{REG\_WINDOWS}} \)) 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 Spill/Fill Traps on page 355. 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.
RESTORE

Exceptions

illegal_instruction
fill_n_normal (n = 0–7)
fill_n_other (n = 0–7)

See Also
SAVE on page 237
7.77 RESTORED

Description

RESTORED adjusts the state of the register-windows control registers.

RESTORED increments CANRESTORE.

If CLEANWIN < \(N_{\text{REG\_WINDOWS}} - 1\), 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 ≥ \(N_{\text{REG\_WINDOWS}} - 2\) 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 60) 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) causes a privileged_opcode exception.

Exceptions

illegal_instruction
privileged_opcode

See Also

ALLCLEAN on page 99
INVALW on page 173
NORMALW on page 213
OTHERW on page 215
SAVED on page 239

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESTOREDp</td>
<td>Window has been restored</td>
<td>restored</td>
<td>A1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fcn = 0 0001</th>
<th>11 0001</th>
<th>—</th>
</tr>
</thead>
</table>

31 30 29 25 24 19 18 0
The RETRY instruction restores the saved state from TSTATE[TL] (GL, CCR, ASI, PSTATE, and CWP), 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).

Execution of a RETRY instruction in the delay slot of a control-transfer instruction produces undefined results.

If software writes invalid or inconsistent state to TSTATE before executing RETRY, virtual processor behavior during and after execution of the RETRY instruction is undefined.

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 (PSTATE.priv = 1)

An attempt to execute a RETRY instruction in nonprivileged mode (PSTATE.priv = 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, 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.
RETRY

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 114 and *Trap on Control Transfer (tct)* on page 65.

Exceptions
- `illegal_instruction`
- `privileged_opcode`
- `control_transfer_instruction` (impl. dep. #450-S20)

See Also | DONE on page 114
7.79 RETURN

Description
The RETURN instruction causes a register-indirect 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
```

A routine that uses a register window may be structured either as:
```
save %sp, -framesize, %sp
... 
ret     : "ret" is shorthand for "jmpl %i7 + 8,%g0"
restore : A useful instruction in the delay slot, such as
          : "restore %o2,%l2,%o0"
or as:
save %sp, -framesize, %sp
... 
return %i7 + 8
nop     : Instead of "nop", could do some useful work in the
          : caller's window, for example, "or %o1,%o2,%o0"
```

An attempt to execute a RETURN instruction when bits 29:25 are nonzero causes an illegal_instruction exception.

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)
7.80  SAVE

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE</td>
<td>11</td>
<td>1100</td>
<td>save     reg1r, reg_or_imm, regrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

**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.
SAVE

If $\text{CANSAVE} \neq 0$, 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, $(\text{CLEANWIN} - \text{CANRESTORE}) = 0$. 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 230
7.81 SAVED

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.

Exceptions
illegal_instruction
privileged_opcode

See Also
ALLCLEAN on page 99
INVALW on page 173
NORMALW on page 213
OTHERW on page 215
RESTORED on page 232
7.82  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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDIV</td>
<td>00 1111 Signed Integer Divide</td>
<td>sdiv {reg, reg_or_imm, reg}</td>
<td>D2</td>
</tr>
<tr>
<td>SDIVcc</td>
<td>01 1111 Signed Integer Divide and modify cc's</td>
<td>sdivcc {reg, reg_or_imm, reg}</td>
<td>D2</td>
</tr>
</tbody>
</table>

Description
The signed divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If \( i = 0 \), they compute \( (Y::R[rs1][31:0]) \div R[rs2][31:0] \). Otherwise (that is, if \( i = 1 \)), the divide instructions compute \( (Y::R[rs1][31:0]) \div (\text{sign_ext}(\text{simm13})[31:0]) \). 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 \((Y::\text{lower 32 bits of } R[rs1])\) and a signed integer word divisor (lower 32 bits of \( R[rs2] \) or lower 32 bits of \( \text{sign_ext}(\text{simm13}) \)) and computes a signed integer word quotient \( R[rd] \).

Signed division rounds an inexact quotient toward zero. For example, \(-7 \div 4\) 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>
<thead>
<tr>
<th>Condition Under Which Overflow Occurs</th>
<th>Value Returned in ( R[rd] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational quotient ( \geq 2^{31} )</td>
<td>( 2^{31} - 1 (0000 0000 7FFF FFFF_{16}) )</td>
</tr>
<tr>
<td>Rational quotient ( \leq -2^{31} - 1 )</td>
<td>( -2^{31} (FFFF FFFF 8000 0000_{16}) )</td>
</tr>
</tbody>
</table>

When no overflow occurs, the 32-bit result is sign-extended to 64 bits and written into register \( R[rd] \).
SDIV, SDIVcc (Deprecated)

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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Effect on bit of SDIVcc instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc.n</td>
<td>Set to 1 if R[rd][31] = 1; otherwise, set to 0</td>
</tr>
<tr>
<td>icc.z</td>
<td>Set to 1 if R[rd][31:0] = 0; otherwise, set to 0</td>
</tr>
<tr>
<td>icc.v</td>
<td>Set to 1 if overflow (per TABLE 7-12); otherwise set to 0</td>
</tr>
<tr>
<td>icc.c</td>
<td>Set to 0</td>
</tr>
<tr>
<td>xcc.n</td>
<td>Set to 1 if R[rd][63] = 1; otherwise, set to 0</td>
</tr>
<tr>
<td>xcc.z</td>
<td>Set to 1 if R[rd][63:0] = 0; otherwise, set to 0</td>
</tr>
<tr>
<td>xcc.v</td>
<td>Set to 0</td>
</tr>
<tr>
<td>xcc.c</td>
<td>Set to 0</td>
</tr>
</tbody>
</table>

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 209
- RDY on page 225
- UDIV[cc] on page 281
7.83 SETHI

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op2</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
</table>
| SETHI       | 100 | Set High 22 Bits of Low Word | sethi const22, reg<rd>  
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 1234₁₆:

```
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 1234₁₆:

```
sethi %hi(0x5432edcb),%o0! note 0x5432EDCB, not 0xABCD1234
xor %o0, 0x1e34, %o0! part of imm. overlaps upper bits
```

### Exceptions
None
7.84 Set Interval Arithmetic Mode

**Description**

The SIAM instruction sets the GSR.im and GSR.irnd fields as follows:

\[
\begin{align*}
GSR\cdot im & \leftarrow \text{mode}[2] \\
GSR\cdot irnd & \leftarrow \text{mode}[1:0]
\end{align*}
\]

**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*
7.85 Shift

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>x</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>10 0101</td>
<td>0</td>
<td>Shift Left Logical – 32 bits</td>
<td>sll reg&lt;rs1&gt;, reg_or_shcnt&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>SRL</td>
<td>10 0110</td>
<td>0</td>
<td>Shift Right Logical – 32 bits</td>
<td>srl reg&lt;rs1&gt;, reg_or_shcnt&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>SRA</td>
<td>10 0111</td>
<td>0</td>
<td>Shift Right Arithmetic– 32 bits</td>
<td>sra reg&lt;rs1&gt;, reg_or_shcnt&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>SLLX</td>
<td>10 0101</td>
<td>1</td>
<td>Shift Left Logical – 64 bits</td>
<td>sllx reg&lt;rs1&gt;, reg_or_shcnt&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>SRLX</td>
<td>10 0110</td>
<td>1</td>
<td>Shift Right Logical – 64 bits</td>
<td>srlx reg&lt;rs1&gt;, reg_or_shcnt&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>SRAX</td>
<td>10 0111</td>
<td>1</td>
<td>Shift Right Arithmetic – 64 bits</td>
<td>srax reg&lt;rs1&gt;, reg_or_shcnt&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
</tbody>
</table>

TABLE 7-14 Shift Count Encodings

<table>
<thead>
<tr>
<th>i</th>
<th>x</th>
<th>Shift Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>bits 4–0 of R[rs2]</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>bits 5–0 of R[rs2]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>bits 4–0 of instruction</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>bits 5–0 of instruction</td>
</tr>
</tbody>
</table>

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.

SLL and SLLX shift all 64 bits of the value in R[rs1] left by the number of bits specified by the shift count, replacing the vacated positions with zeroes, and write the shifted result to R[rd].

SRL shifts the low 32 bits of the value in R[rs1] right by the number of bits specified by the shift count. Zeroes are shifted into bit 31. The upper 32 bits are set to zero, and the result is written to R[rd].

SRLX shifts all 64 bits of the value in R[rs1] right by the number of bits specified by the shift count. Zeroes are shifted into the vacated high-order bit positions, and the shifted result is written to R[rd].

SRA shifts the low 32 bits of the value in R[rs1] right by the number of bits specified by the shift count and replaces the vacated positions with bit 31 of R[rs1]. The high-order 32 bits of the result are all set with bit 31 of R[rs1], and the result is written to R[rd].

SRAX shifts all 64 bits of the value in R[rs1] right by the number of bits specified by the shift count and replaces the vacated positions with bit 63 of R[rs1]. The shifted result is written to R[rd].
SLL / SRL / SRA

No shift occurs when the shift count is 0, but the high-order bits are affected by the 32-bit shifts as noted above.

These instructions do not modify the condition codes.

Programming Notes

"Arithmetic left shift by 1 (and calculate overflow)" can be effected with the ADDcc instruction.

The instruction “sra rs1, 0, rd” can be used to convert a 32-bit value to 64 bits, with sign extension into the upper word. “srl rs1, 0, rd” can be used to clear the upper 32 bits of R[rd].

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
7.86 Signed Multiply (32-bit)

The signed multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute "$R^{rs1}[31:0] \times R^{rs2}[31:0]$" if $i = 0$, or "$R^{rs1}[31:0] \times \text{signext}(simm13)[31:0]$" 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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Effect on bit by execution of SMULcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>$icc.n$</td>
<td>Set to 1 if product[31] = 1; otherwise, set to 0</td>
</tr>
<tr>
<td>$icc.z$</td>
<td>Set to 1 if product[31:0] = 0; otherwise, set to 0</td>
</tr>
<tr>
<td>$icc.v$</td>
<td>Set to 0</td>
</tr>
<tr>
<td>$icc.c$</td>
<td>Set to 0</td>
</tr>
<tr>
<td>$xcc.n$</td>
<td>Set to 1 if product[63] = 1; otherwise, set to 0</td>
</tr>
<tr>
<td>$xcc.z$</td>
<td>Set to 1 if product[63:0] = 0; otherwise, set to 0</td>
</tr>
<tr>
<td>$xcc.v$</td>
<td>Set to 0</td>
</tr>
<tr>
<td>$xcc.c$</td>
<td>Set to 0</td>
</tr>
</tbody>
</table>

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.

Programming Notes: 32-bit overflow after SMUL or SMULcc is indicated by $Y \neq (R[rd] >> 31)$, 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 283
7.87 Store Integer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STB</td>
<td>00 0101</td>
<td>Store Byte</td>
<td>stb† reg [address]</td>
<td>A1</td>
</tr>
<tr>
<td>STH</td>
<td>00 0110</td>
<td>Store Halfword</td>
<td>sth‡ reg [address]</td>
<td>A1</td>
</tr>
<tr>
<td>STW</td>
<td>00 0100</td>
<td>Store Word</td>
<td>stw◊ reg [address]</td>
<td>A1</td>
</tr>
<tr>
<td>STX</td>
<td>00 1110</td>
<td>Store Extended Word</td>
<td>stx reg [address]</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonyms: stub, stab  ‡ 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 76). 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

See Also

STTW on page 265
7.88  Store Integer into Alternate Space

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STBAPASI</td>
<td>01 0101</td>
<td>Store Byte into Alternate Space</td>
<td>stba^ [reg_addr] [imm_asi]</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>stba [reg_plus_imm] %asi</td>
<td></td>
</tr>
<tr>
<td>STHAPASI</td>
<td>01 0110</td>
<td>Store Halfword into Alternate Space</td>
<td>stha^ [reg_addr] [imm_asi]</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>stha [reg_plus_imm] %asi</td>
<td></td>
</tr>
<tr>
<td>STWAPASI</td>
<td>01 0100</td>
<td>Store Word into Alternate Space</td>
<td>stwa^ [reg_addr] [imm_asi]</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>stwa [reg_plus_imm] %asi</td>
<td></td>
</tr>
<tr>
<td>STXAPASI</td>
<td>01 1110</td>
<td>Store Extended Word into Alternate Space</td>
<td>stxa [reg_addr] [imm_asi]</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>stxa [reg_plus_imm] %asi</td>
<td></td>
</tr>
</tbody>
</table>

\^ synonyms: stuba, stba  \^ synonyms: stuha, stsha  \^ synonyms: sta, stua, stwa

**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), if bit 7 of the ASI is 0, these instructions cause a `privileged_action` exception. In privileged mode (PSTATE.priv = 1), 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, 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_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.

<table>
<thead>
<tr>
<th>ASIs invalid for STXA</th>
<th>(cause DAE_invalid_asi exception)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_BLOCK AS IF_USER_PRIMARY</td>
<td>ASI_BLOCK AS IF_USER_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_BLOCK AS IF_USER_SECONDARY</td>
<td>ASI_BLOCK AS IF_USER_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_BLOCK AS IF_USER_PRIMARY</td>
<td>ASI_BLOCK AS IF_USER_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_BLOCK AS IF_USER_SECONDARY</td>
<td>ASI_BLOCK AS IF_USER_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PST8_PRIMARY</td>
<td>ASI_PST8_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PST8_SECONDARY</td>
<td>ASI_PST8_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PRIMARY_NO_FAULT</td>
<td>ASI_PRIMARY_NO_FAULT_LITTLE</td>
</tr>
<tr>
<td>ASISECONDARY_NO_FAULT</td>
<td>ASISECONDARY_NO_FAULT_LITTLE</td>
</tr>
<tr>
<td>ASI_PST16_PRIMARY</td>
<td>ASI_PST16_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PST16_SECONDARY</td>
<td>ASI_PST16_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PST32_PRIMARY</td>
<td>ASI_PST32_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PST32_SECONDARY</td>
<td>ASI_PST32_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_FL8_PRIMARY</td>
<td>ASI_FL8_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_FL8_SECONDARY</td>
<td>ASI_FL8_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_FL16_PRIMARY</td>
<td>ASI_FL16_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_FL16_SECONDARY</td>
<td>ASI_FL16_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_BLOCK_COMMIT_PRIMARY</td>
<td>ASI_BLOCK_COMMIT_SECONDARY</td>
</tr>
<tr>
<td>ASI_BLOCK_PRIMARY</td>
<td>ASI_BLOCK_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_BLOCK_SECONDARY</td>
<td>ASI_BLOCK_SECONDARY_LITTLE</td>
</tr>
</tbody>
</table>

**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

**See Also**

- LDA on page 176
- STTWA on page 267
### 7.89 Block Store

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 or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASI Value</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STBLOCKF</td>
<td>1616</td>
<td>64-byte block store to primary address space, user privilege</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_AIUP</td>
<td>A1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>1716</td>
<td>64-byte block store to secondary address space, user privilege</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_AIUS</td>
<td>A1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>1E16</td>
<td>64-byte block store to primary address space, little-endian, user privilege</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_AIUPL</td>
<td>A1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>1F16</td>
<td>64-byte block store to secondary address space, little-endian, user privilege</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_AIUSL</td>
<td>A1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>F016</td>
<td>64-byte block store to primary address space</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_P</td>
<td>A1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>F116</td>
<td>64-byte block store to secondary address space</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_S</td>
<td>A1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>F816</td>
<td>64-byte block store to primary address space, little-endian</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_PL</td>
<td>A1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>F916</td>
<td>64-byte block store to secondary address space, little-endian</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_SL</td>
<td>A1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>E016</td>
<td>64-byte block commit store to primary address space</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_COMMIT_P</td>
<td>B1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>E116</td>
<td>64-byte block commit store to secondary address space</td>
<td>stda $freg$_addr [regaddr] #ASI_BLK_COMMIT_S</td>
<td>B1</td>
</tr>
</tbody>
</table>

**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 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. 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 318 for more information.)

ASIs E016 and E116 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 333 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 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 250). 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 exceptions are recognized on accesses to all 64 bytes of the STBLOCKF (the recommended behavior), or only on accesses to the first eight bytes.

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.
STBLOCKF

- 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.lef = 0 or PSTATE.preferred = 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), if bit 7 of the ASI is 0 (ASIs 16, 17, 1E, and 1F), STBLOCKF causes a `privileged_action` exception.

An access caused by STBLOCKF may trigger a `VA_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`

**See Also** LDBLOCKF on page 178
7.90 Store Floating-Point

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 76). The effective address for these instructions is "$R[rs1] + \text{sign\_ext}(\text{simm13})$" 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 ($\text{FPRS}.\text{fef} = 0$ or $\text{PSTATE}.\text{pef} = 0$) 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 \( \text{rd}[1] \neq 0 \) causes an \textit{fp\textunderscore exception\_other} (FSR.flt = 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 \texttt{STQF\_mem\_address\_not\_aligned} and \textit{fp\_exception\_other} (with FSR.flt = invalid\_fp\_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an \textit{illegal\_instruction} exception and subsequent trap.

### Exceptions
- \textit{illegal\_instruction}
- \textit{fp\_disabled}
- \texttt{STDF\_mem\_address\_not\_aligned}
- \texttt{STQF\_mem\_address\_not\_aligned} (not used in UltraSPARC Architecture 2007)
- \texttt{mem\_address\_not\_aligned}
- \textit{fp\_exception\_other} (FSR.flt = invalid\_fp\_register (STQF only))
- VA\_watchpoint
- DAE\_privilege\_violation
- DAE\_nfo\_page

### See Also
- Load Floating-Point Register on page 181
- Block Store on page 250
- Store Floating-Point into Alternate Space on page 255
- Store Floating-Point State Register (Lower) on page 258
- Store Short Floating-Point on page 263
- Store Partial Floating-Point on page 260
- Store Floating-Point State Register on page 269
7.91 Store Floating-Point into Alternate Space

### 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 \( \text{imm}_\text{asi} \) 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 \( R[rs1] + \text{sign_ext}(\text{simm13}) \) if \( i = 1 \).

### Exceptions

STFA causes a \textit{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 \textit{STDFA_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 \textit{STQFA_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.

---

**Instruction** | \text{op3} | \text{rd} | \text{Operation} | \text{Assembly Language Syntax} | \text{Class}
--- | --- | --- | --- | --- | ---
\text{STFA}^{\text{Asi}} | 11 0100 | 0–31 | Store Floating-Point Register to Alternate Space | \text{sta} \ freg\_{rd} [\text{regaddr}] \ \text{imm}_\text{asi} | \text{A1}
\text{STDFA}^{\text{Asi}} | 11 0111 | \dagger | Store Double Floating-Point Register to Alternate Space | \text{stda} \ freg\_{rd} [\text{regaddr}] \ \text{imm}_\text{asi} | \text{A1}
\text{STQFA}^{\text{Asi}} | 11 0110 | \dagger | Store Quad Floating-Point Register to Alternate Space | \text{stqa} \ freg\_{rd} [\text{regaddr}] \ \text{imm}_\text{asi} | \text{C3}

\dagger \text{Encoded floating-point register value, as described on page 51.}
An attempt to execute an STQFA instruction when \( \text{rd}[1] \neq 0 \) causes an \( \text{fp\_exception\_other} \) (FSR.flt = 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 \( \text{STQF\_mem\_address\_not\_aligned} \) and \( \text{fp\_exception\_other} \) (with FSR.flt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an \( \text{illegal\_instruction} \) exception and subsequent trap.

In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0, this instruction causes a \( \text{privileged\_action} \) exception. In privileged mode (PSTATE.priv = 1), if the ASI is in the range \( 30_{16} \) to \( 7F_{16} \), this instruction causes a \( \text{privileged\_action} \) exception.

STFA and STQFA can be used with any of the following ASIs, subject to the privilege mode rules described for the \( \text{privileged\_action} \) exception above. Use of any other ASI with these instructions causes a \( \text{DAE\_invalid\_asi} \) exception.

<table>
<thead>
<tr>
<th>ASIs valid for STFA and STQFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_REAL</td>
</tr>
<tr>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
</tr>
</tbody>
</table>

STDFA can be used with any of the following ASIs, subject to the privilege mode rules described for the \( \text{privileged\_action} \) exception above. Use of any other ASI with the STDFA instruction causes a \( \text{DAE\_invalid\_asi} \) exception.

<table>
<thead>
<tr>
<th>ASIs valid for STDFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_REAL</td>
</tr>
<tr>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
</tr>
<tr>
<td>ASI_BLOCK_IF_USER_PRIMARY†</td>
</tr>
<tr>
<td>ASI_BLOCK_IF_USER_SECONDARY†</td>
</tr>
<tr>
<td>ASI_BLOCK_PRIMARY†</td>
</tr>
<tr>
<td>ASI_BLOCK_SECONDARY†</td>
</tr>
<tr>
<td>ASI_BLOCK_COMMIT_PRIMARY†</td>
</tr>
<tr>
<td>ASI_BLOCK_COMMIT_SECONDARY†</td>
</tr>
<tr>
<td>ASI_FL8_PRIMARY‡</td>
</tr>
<tr>
<td>ASI_FL8_SECONDARY‡</td>
</tr>
<tr>
<td>ASI_FL16_PRIMARY‡</td>
</tr>
<tr>
<td>ASI_FL16_SECONDARY‡</td>
</tr>
<tr>
<td>ASI_PST8_PRIMARY*</td>
</tr>
<tr>
<td>ASI_PST8_SECONDARY*</td>
</tr>
<tr>
<td>ASI_PST16_PRIMARY*</td>
</tr>
<tr>
<td>ASI_PST16_SECONDARY*</td>
</tr>
<tr>
<td>ASI_PST32_PRIMARY*</td>
</tr>
</tbody>
</table>

\*\(\text{PST}\) instructions are only supported on UltraSPARC T1 processors.

\†\(\text{T}:\) indicates that the instruction is only supported on UltraSPARC T processors.
STFA / STDFA / STQFA

ASIs valid for STDFA

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 250.
‡ 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 263.
* 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 260.

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

See Also

Load Floating-Point from Alternate Space on page 183
Block Store on page 250
Store Floating-Point on page 253
Store Short Floating-Point on page 263
Store Partial Floating-Point on page 260
7.92 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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>rd</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STFSR</td>
<td>10 0101</td>
<td>0</td>
<td>Store Floating-Point State Register (Lower)</td>
<td>st %fsr, [address]</td>
<td>D2</td>
</tr>
<tr>
<td></td>
<td>10 0101</td>
<td>1-31</td>
<td>(see page 269)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 76). 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.ife = 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 causes a mem_address_notAligned 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 = 102, op3 = 10 01012 opcode with an invalid rd value causes an illegal_instruction exception.

Exceptions
illegal_instruction
fp_disabled
mem_address_not_aligned
VA_watchpoint
DAE_privilegeViolation
DAE_nfo_page
STFSR (Deprecated)

See Also

- Store Floating-Point on page 253
- Store Floating-Point State Register on page 269
7.93 Store Partial Floating-Point

STPARTIALF

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Operation</th>
<th>Assembly Language Syntax †</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STPARTIALF</td>
<td>C016</td>
<td>Eight 8-bit conditional stores to primary address space</td>
<td>stda freg, reg, [reg]   #ASI_PST8_P</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C116</td>
<td>Eight 8-bit conditional stores to secondary address space</td>
<td>stda freg, reg, [reg]   #ASI_PST8_S</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C816</td>
<td>Eight 8-bit conditional stores to primary address space, little-endian</td>
<td>stda freg, reg, [reg]   #ASI_PST8_PL</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C916</td>
<td>Eight 8-bit conditional stores to secondary address space, little-endian</td>
<td>stda freg, reg, [reg]   #ASI_PST8_SL</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C216</td>
<td>Four 16-bit conditional stores to primary address space</td>
<td>stda freg, reg, [reg]   #ASI_PST16_P</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C316</td>
<td>Four 16-bit conditional stores to secondary address space</td>
<td>stda freg, reg, [reg]   #ASI_PST16_S</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>CA16</td>
<td>Four 16-bit conditional stores to primary address space, little-endian</td>
<td>stda freg, reg, [reg]   #ASI_PST16_PL</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>CB16</td>
<td>Four 16-bit conditional stores to secondary address space, little-endian</td>
<td>stda freg, reg, [reg]   #ASI_PST16_SL</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C416</td>
<td>Two 32-bit conditional stores to primary address space</td>
<td>stda freg, reg, [reg]   #ASI_PST32_P</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C516</td>
<td>Two 32-bit conditional stores to secondary address space</td>
<td>stda freg, reg, [reg]   #ASI_PST32_S</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>CC16</td>
<td>Two 32-bit conditional stores to primary address space, little-endian</td>
<td>stda freg, reg, [reg]   #ASI_PST32_PL</td>
<td>B1</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>CD16</td>
<td>Two 32-bit conditional stores to secondary address space, little-endian</td>
<td>stda freg, reg, [reg]   #ASI_PST32_SL</td>
<td>B1</td>
</tr>
</tbody>
</table>

† The original assembly language syntax for a Partial Store instruction ("stda freg, reg, [reg]   #ASI_PST8_P") 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. Two 32-bit, four 16-bit, or eight 8-bit values from the 64-bit floating-point register F0[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, F0[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 126). 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.

### Exceptions.
A Partial Store instruction can cause a virtual watchpoint exception when the following conditions are met:
- The virtual address in $R[rs1]$ matches the address in the VA 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 ($FPRS.cef = 0$ 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.
**STPARTIALF**

**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 C0₁₆–C₅₁₆ and C₈₁₆–C₁₆ 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 185.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>STPARTIALF shares an opcode with the STBLOCKF, STDFA, and STSHORTF instructions; it is distinguished by the ASI used.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Note</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Exceptions**

- illegal_instruction
- fp_disabled
- mem_address_not_aligned
- VA_watchpoint (see text)
- DAE_privilege_violation
- DAE_nc_page
- DAE_nfo_page
7.94 Store Short Floating-Point

**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 121) to assemble or store 64 bits on noncontiguous components.

**Implementation**
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 causes a mem_address_not_aligned exception if the effective memory address is not halfword-aligned.

An 8-bit STSHORTF (using ASI D0₁₆, D1₁₆, D₈₁₆, or D₉₁₆) can be performed to an arbitrary memory address (no alignment requirement).

A 16-bit STSHORTF (using ASI D₂₁₆, D₃₁₆, DA₁₆, or DB₁₆) 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
7.95 Store Integer Twin Word

The STTW instruction is deprecated and should not be used in new software. The STX instruction should be used instead.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax †</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STTW</td>
<td>00 0111</td>
<td>Store Integer Twin Word</td>
<td>sttw reg, [address]</td>
<td>D2</td>
</tr>
</tbody>
</table>

† 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 76). 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 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.
STTW (Deprecated)

Exceptions

unimplemented_STTW (not used in UltraSPARC Architecture 2007)
illegal_instruction
mem_address_not_aligned
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

See Also
STW/STX on page 247
STTWA on page 267
7.96 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.
```

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STTWA</td>
<td>01 0111</td>
<td>Store Twin Word into Alternate Space</td>
<td>sttwa reg[regaddr] immasi</td>
<td>D2, Y3†</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sttwa reg[reg_plus_imm] %asi</td>
<td></td>
</tr>
</tbody>
</table>

† 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 (0016-7F16); D2 for unrestricted ASIs (8016-FF16)

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.

**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), if bit 7 of the ASI is 0, this instruction causes a privileged_action exception. In privileged mode (PSTATE.priv = 1), if the ASI is in the range 30_{16} to 7F_{16}, this instruction causes a privileged_action exception.
STTWA (Deprecated)

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).

<table>
<thead>
<tr>
<th>ASIs valid for STTWA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>ASI_NUCLEUS_LITTLE</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_REAL</td>
</tr>
<tr>
<td>ASI_REAL_LITTLE</td>
</tr>
<tr>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>ASI_REAL_IO_LITTLE</td>
</tr>
<tr>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>ASI_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
</tr>
<tr>
<td>ASI_SECONDARY_LITTLE</td>
</tr>
</tbody>
</table>

**Programming Note**

Nontranslating ASIs (see page 321) 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
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nfo_page

**See Also**

STWA/STXA on page 248
STTW on page 265
7.97 Store Floating-Point State Register

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.flt after writing the FSR to memory.

Implementation Note
FSR.flt 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 76). 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.pfs = 0 or PSTATE.pfa = 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 STXFSR instruction 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 = 10₂, op3 = 10 0101₂ 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

See Also
Load Floating-Point State Register on page 199
Store Floating-Point on page 253
Store Floating-Point State Register (Lower) on page 258
7.98 Subtract

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].

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
7.99 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWAP</td>
<td>00 1111</td>
<td>Swap Register with Memory</td>
<td>swap [address], reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>D2</td>
</tr>
</tbody>
</table>

#### 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 76). 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
7.100 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 | 01 1111 | Swap register with Alternate Space Memory | `swapa [regaddr] imm_asi, regrd` | D2, Y3‡

‡ Y3 for restricted ASIs (0016-7F16); D2 for unrestricted ASIs (8016-FF16)

---

### 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.

In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0, this instruction causes a **privileged_action** exception. In privileged mode (PSTATE.priv = 1), 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
7.101 Tagged Add

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>TADDcc</td>
<td>10 0000</td>
<td>Tagged Add and modify cc's</td>
<td>taddcc rs1, reg_or_imm, rd</td>
<td>A1</td>
</tr>
</tbody>
</table>

### Description

This instruction computes a sum that is \( R[rs1] + R[rs2] \) if \( i = 0 \), or \( R[rs1] + \text{sign_ext}(\text{simm13}) \) 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 275
TSUBcc on page 279
7.102 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.

**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.

**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 274
TSUBccTV on page 280
### Trap on Integer Condition Codes (Tcc)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>cond</th>
<th>Operation</th>
<th>cc Test Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA</td>
<td>11</td>
<td>0000</td>
<td>Trap Always</td>
<td>1 ta i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TN</td>
<td>11</td>
<td>0000</td>
<td>Trap Never</td>
<td>0 tn i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TNE</td>
<td>11</td>
<td>0001</td>
<td>Trap on Not Equal</td>
<td>not Z tne i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TE</td>
<td>11</td>
<td>0001</td>
<td>Trap on Equal</td>
<td>Z te i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TG</td>
<td>11</td>
<td>0101</td>
<td>Trap on Greater</td>
<td>not (Z or (N xor V)) tg i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TLE</td>
<td>11</td>
<td>0010</td>
<td>Trap on Less or Equal</td>
<td>Z or (N xor V) tle i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TGE</td>
<td>11</td>
<td>0111</td>
<td>Trap on Greater or Equal</td>
<td>not (N xor V) tge i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TL</td>
<td>11</td>
<td>0011</td>
<td>Trap on Less</td>
<td>N xor V tl i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TGU</td>
<td>11</td>
<td>1100</td>
<td>Trap on Greater, Unsigned</td>
<td>not (C or Z) tgu i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TLEU</td>
<td>11</td>
<td>0100</td>
<td>Trap on Less or Equal, Unsigned</td>
<td>(C or Z) tleu i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TCC</td>
<td>11</td>
<td>1101</td>
<td>Trap on Carry Clear (Greater than or Equal, Unsigned)</td>
<td>not C tcc i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TCS</td>
<td>11</td>
<td>0101</td>
<td>Trap on Carry Set (Less Than, Unsigned)</td>
<td>C tcs i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TPOS</td>
<td>11</td>
<td>1110</td>
<td>Trap on Positive or zero</td>
<td>not N tpos i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TNEG</td>
<td>11</td>
<td>0110</td>
<td>Trap on Negative</td>
<td>N tneg i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TVC</td>
<td>11</td>
<td>1111</td>
<td>Trap on Overflow Clear</td>
<td>not V tvc i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TVS</td>
<td>11</td>
<td>0111</td>
<td>Trap on Overflow Set Clear</td>
<td>V tvs i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
</tbody>
</table>

+ syn: tnz  ‡ syn: tz  ◊ syn: tgeu  ∇ syn: tlu

---

<table>
<thead>
<tr>
<th>cc1 :: cc0</th>
<th>Condition Codes Evaluated</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>CCR.icc</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>(illegal_instruction)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>CCR.xcc</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>(illegal_instruction)</td>
<td></td>
</tr>
</tbody>
</table>
Tcc

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 mode, 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 mode 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 ≤ SWTN ≤ 127 are used to trap to privileged-mode software and values of 128 ≤ SWTN ≤ 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:

<table>
<thead>
<tr>
<th>Privilege Mode in effect when Tcc is executed</th>
<th>0 ≤ SWTN ≤ 127</th>
<th>128 ≤ SWTN ≤ 255</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonprivileged (PSTATE.priv = 0)</td>
<td>trap_instruction exception (to privileged mode) (256 ≤ TT ≤ 383)</td>
<td>— (not possible, because SWTN is a 7-bit value in nonprivileged mode)</td>
</tr>
<tr>
<td>Privileged (PSTATE.priv = 1)</td>
<td>trap_instruction exception (to privileged mode) (256 ≤ TT ≤ 383)</td>
<td>htrap_instruction exception (to hyperprivileged mode) (384 ≤ TT ≤ 511)</td>
</tr>
</tbody>
</table>

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 10:5 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 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 356.
Exceptions

illegal_instruction
control_transfer_instruction (impl. dep. #450-S20)
trap_instruction \(0 \leq \text{SWTN} \leq 127\)
htrap_instruction \(128 \leq \text{SWTN} \leq 255\)
## 7.104 Tagged Subtract

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSUBcc</td>
<td>10 0001</td>
<td>Tagged Subtract and modify cc’s</td>
<td>tsubcc rs1, reg_or_imm, reg</td>
<td>A1</td>
</tr>
</tbody>
</table>

### 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 274
TSUBccTV\(^D\) on page 280
7.105 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.

Description

This instruction computes \( R[rs1] - R[rs2] \) if \( i = 0 \), or \( R[rs1] - \text{sign} \_\text{ext} (\text{simm}13) \) 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.

Exceptions

illegal\_instruction

tag\_overflow

See Also

TADDccTV\(^D\) on page 275
TSUBcc on page 279
7.106 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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDIV</td>
<td>00 1110</td>
<td>Unsigned Integer Divide</td>
<td>udiv regrs1, reg_or_imm, regrd</td>
<td>D2</td>
</tr>
<tr>
<td>UDIVcc</td>
<td>01 1110</td>
<td>Unsigned Integer Divide and modify cc’s</td>
<td>udivcc regrs1, reg_or_imm, regrd</td>
<td>D2</td>
</tr>
</tbody>
</table>

**Description**

The unsigned divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If \( i = 0 \), they compute \( (Y :: \text{R}[rs1][31:0]) \div \text{R}[rs2][31:0] \). Otherwise (that is, if \( i = 1 \)), the divide instructions compute \( (Y :: \text{R}[rs1][31:0]) \div (\text{sign ext}(\text{simm13})[31:0]) \). 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 \( Y :: \text{R}[rs1][31:0] \) and an unsigned integer word divisor \( \text{R}[rs2][31:0] \) or \( (\text{sign ext}(\text{simm13})[31:0]) \) and computes an unsigned integer word quotient \( \text{R}[rd] \). Immediate values in \text{simm13} \) are in the ranges 0 to \( 2^{12} - 1 \) and \( 2^{32} - 2^{12} \) to \( 2^{32} - 1 \) for unsigned divide instructions.

Unsigned division rounds an inexact rational quotient toward zero.

**Programming Note**

The rational quotient is the infinitely precise result quotient. It includes both the integer part and the fractional part of the result. For example, the rational quotient of \( 11/4 = 2.75 \) (integer part = 2, fractional part = .75).

The result of an unsigned divide instruction can overflow the less significant 32 bits of the destination register R[rd] under certain conditions. When overflow occurs, the largest appropriate unsigned integer is returned as the quotient in R[rd]. The condition under which overflow occurs and the value returned in R[rd] under this condition are specified in TABLE 7-15.

**TABLE 7-15** UDIV / UDIVcc Overflow Detection and Value Returned

<table>
<thead>
<tr>
<th>Condition Under Which Overflow Occurs</th>
<th>Value Returned in R[rd]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational quotient ≥ ( 2^{32} )</td>
<td>( 2^{32} - 1 )</td>
</tr>
<tr>
<td></td>
<td>(0000 0000 FFFF FFFF16)</td>
</tr>
</tbody>
</table>

When no overflow occurs, the 32-bit result is zero-extended to 64 bits and written into register R[rd].
UDIV, UDIVcc  (Deprecated)

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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Effect on bit of UDIVcc instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc.n</td>
<td>Set if R[rd][31] = 1</td>
</tr>
<tr>
<td>icc.z</td>
<td>Set if R[rd][31:0] = 0</td>
</tr>
<tr>
<td>icc.v</td>
<td>Set if overflow (per TABLE 7-15)</td>
</tr>
<tr>
<td>icc.c</td>
<td>Zero</td>
</tr>
<tr>
<td>xcc.n</td>
<td>Set if R[rd][63] = 1</td>
</tr>
<tr>
<td>xcc.z</td>
<td>Set if R[rd][63:0] = 0</td>
</tr>
<tr>
<td>xcc.v</td>
<td>Zero</td>
</tr>
<tr>
<td>xcc.c</td>
<td>Zero</td>
</tr>
</tbody>
</table>

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 225
- SDIV[cc] on page 240,
- UMUL[cc] on page 283
7.107 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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMULD</td>
<td>001010</td>
<td>Unsigned Integer Multiply</td>
<td>umul rs1, reg_or_imm, reg</td>
<td>D2</td>
</tr>
<tr>
<td>UMULccD</td>
<td>011010</td>
<td>Unsigned Integer Multiply and modify cc's</td>
<td>umulcc rs1, reg_or_imm, reg</td>
<td>D2</td>
</tr>
</tbody>
</table>

Description

The unsigned multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute \(R[rs1][31:0] \times R[rs2][31:0]\) if \(i = 0\), or \(R[rs1][31:0] \times \text{sign_ext}(\text{simm13})[31:0]\) 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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Effect on bit by execution of UMULcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc.n</td>
<td>Set to 1 if product[31] = 1; otherwise, set to 0</td>
</tr>
<tr>
<td>icc.z</td>
<td>Set to 1 if product[31:0] = 0; otherwise, set to 0</td>
</tr>
<tr>
<td>icc.v</td>
<td>Set to 0</td>
</tr>
<tr>
<td>icc.c</td>
<td>Set to 0</td>
</tr>
<tr>
<td>xcc.n</td>
<td>Set to 1 if product[63] = 1; otherwise, set to 0</td>
</tr>
<tr>
<td>xcc.z</td>
<td>Set to 1 if product[63:0] = 0; otherwise, set to 0</td>
</tr>
<tr>
<td>xcc.v</td>
<td>Set to 0</td>
</tr>
<tr>
<td>xcc.c</td>
<td>Set to 0</td>
</tr>
</tbody>
</table>

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.

Programming Notes

- 32-bit overflow after UMUL or UMULcc is indicated by \(Y \neq 0\).

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
UMUL, UMULcc (Deprecated)

See Also
MULScc on page 209
RDY on page 225
SMUL[cc] on page 246,
UDIV[cc] on page 281
### Write Ancillary State Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rd</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRYD</td>
<td>0</td>
<td>Write Y register (deprecated)</td>
<td>wr regset, reg_or_imm,%y</td>
<td>D2</td>
</tr>
<tr>
<td>—</td>
<td>1</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRCCR</td>
<td>2</td>
<td>Write Condition Codes register</td>
<td>wr regset, reg_or_imm,%ccr</td>
<td>A1</td>
</tr>
<tr>
<td>WRASI</td>
<td>3</td>
<td>Write ASI register</td>
<td>wr regset, reg_or_imm,%asi</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>4</td>
<td>Reserved (read-only ASR (TICK))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>5</td>
<td>Reserved (read-only ASR (PC))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRFPRS</td>
<td>6</td>
<td>Write Floating-Point Registers Status register</td>
<td>wr regset, reg_or_imm,%fprs</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>7–14</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>15</td>
<td>(0F16) used at higher privilege level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>16–18</td>
<td>Reserved (impl. dep. #8-V8-Cs20, #9-(10-1216) V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRGSR</td>
<td>19</td>
<td>(1316) Write General Status register (GSR)</td>
<td>wr regset, reg_or_imm,%gsr</td>
<td>A1</td>
</tr>
<tr>
<td>WRSOFTINT_SETP</td>
<td>20</td>
<td>(1416) Set bits of per-virtual processor Soft Interrupt register</td>
<td>wr regset, reg_or_imm,%softint_set</td>
<td>N–</td>
</tr>
<tr>
<td>WRSOFTINT_CLRp</td>
<td>21</td>
<td>(1516) Clear bits of per-virtual processor Soft Interrupt register</td>
<td>wr regset, reg_or_imm,%softint_clr</td>
<td>N–</td>
</tr>
<tr>
<td>WRSOFTINTp</td>
<td>22</td>
<td>(1616) Write per-virtual processor Soft Interrupt register</td>
<td>wr regset, reg_or_imm,%softint</td>
<td>N–</td>
</tr>
<tr>
<td>WRTICK_CMPRp</td>
<td>23</td>
<td>(1716) Write Tick Compare register</td>
<td>wr regset, reg_or_imm,%tick_cmpr</td>
<td>N–</td>
</tr>
<tr>
<td>—</td>
<td>24</td>
<td>(1816) used at higher privilege level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRSTICK_CMPRp</td>
<td>25</td>
<td>(1916) Write System Tick Compare register</td>
<td>wr regset, reg_or_imm,%tick_cmpr</td>
<td>N–</td>
</tr>
<tr>
<td>—</td>
<td>26</td>
<td>(1A16) Reserved (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>26</td>
<td>(1A16) Reserved (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>27</td>
<td>(1B16) Reserved (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>28</td>
<td>(1C16) Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>29</td>
<td>(1D16) Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>30</td>
<td>(1E16) Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>31</td>
<td>(1F16) Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† The original assembly language names for %tick and %tick_cmpr were, respectively, %sys_tick and %sys_tick_cmpr, which are now deprecated. Over time, assemblers will support the new %tick and %tick_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.

### Assembly Language Syntax

<table>
<thead>
<tr>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i=0</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>110000</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>110000</td>
<td>1</td>
<td>—</td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i=1</th>
<th>simm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>29</td>
<td>25</td>
<td>24</td>
<td>19</td>
</tr>
</tbody>
</table>
**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 \( "R[rs1] \text{xor} R[rs2]" \); if \( i = 1 \), store \( "R[rs1] \text{xor} \text{sign_ext}(\text{simm13})" \).

**Note** The operation is exclusive-or.

The WRasr instruction with \( rd = 0 \) 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 \( rd \) of 16-18, 28, 29, or 31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRasr instruction using one of those \( rd \) values, 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 (impl. dep. #9-V8-Cs20), and
- whether an attempt to execute the instruction causes an illegal_instruction exception.

**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 48 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 \( (rs1 \neq 0) \) or \( (i = 0) \)

An attempt to execute a WRSOFTINT_SET, WRSOFTINT_CLR, WRSOFTINT, WRTICK_CMPR, or WRSTICK_CMPR instruction in nonprivileged mode (\( \text{PSTATE}.priv = 0 \)) causes a privileged_opcode exception.

If the floating-point unit is not enabled (\( \text{FPRS}.fef = 0 \) or \( \text{PSTATE}.pef = 0 \)) or if the FPU is not present, then an attempt to execute a WRGSR instruction causes an fp_disabled exception.
### Exceptions
- `illegal_instruction`
- `privileged_opcode`
- `fp_disabled`

### See Also
- RDsr on page 225
- WRPR on page 288
7.109 Write Privileged Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>rd</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRPR</td>
<td>11 0010</td>
<td>Write Privileged register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPC</td>
<td>0</td>
<td>wrpr rs1, reg_or_imm, %tpc</td>
<td></td>
<td></td>
<td>A1</td>
</tr>
<tr>
<td>TNPC</td>
<td>1</td>
<td>wrpr rs1, reg_or_imm, %tnpc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSTATE</td>
<td>2</td>
<td>wrpr rs1, reg_or_imm, %tstate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT</td>
<td>3</td>
<td>wrpr rs1, reg_or_imm, %tt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(illegal_instruction)</td>
<td>4</td>
<td>wrpr rs1, reg_or_imm, %tba</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TBA</td>
<td>5</td>
<td>wrpr rs1, reg_or_imm, %tpc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSTATE</td>
<td>6</td>
<td>wrpr rs1, reg_or_imm, %tpstate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TL</td>
<td>7</td>
<td>wrpr rs1, reg_or_imm, %tl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIL</td>
<td>8</td>
<td>wrpr rs1, reg_or_imm, %pil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CWP</td>
<td>9</td>
<td>wrpr rs1, reg_or_imm, %cwp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANSAVE</td>
<td>10</td>
<td>wrpr rs1, reg_or_imm, %cansave</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANRESTORE</td>
<td>11</td>
<td>wrpr rs1, reg_or_imm, %canrestore</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLEANWIN</td>
<td>12</td>
<td>wrpr rs1, reg_or_imm, %cleanwin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTHERWIN</td>
<td>13</td>
<td>wrpr rs1, reg_or_imm, %otherwin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WSTATE</td>
<td>14</td>
<td>wrpr rs1, reg_or_imm, %wstate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL</td>
<td>16</td>
<td>wrpr rs1, reg_or_imm, %gl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>17–31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 \(MAXPTL\) 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.

\(MAXPTL\) 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. For details, see TABLE 5-19 on page 69.

\(MAXPGL\) 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. For details, see TABLE 5-20 on page 70.
**Exceptions.** An attempt to execute a WRPR instruction in nonprivileged mode (PSTATE.priv = 0) causes a *privilegedOpcode* exception.

An attempt to execute a WRPR instruction when any of the following conditions exist causes an *illegalInstruction* exception:

- i = 0 and instruction bits 12:5 are nonzero
- rd = 4
- 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 mode.

**Implementation Note**

In nonprivileged mode, *illegalInstruction* exception due to 0 ≤ rd ≤ 3 and TL = 0 does not occur; the *privilegedOpcode* exception occurs instead.

**Exceptions**

- privilegedOpcode
- illegalInstruction

**See Also**

- RDPR on page 228
- WRasr on page 285
### 7.110 XOR Logical Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>00</td>
<td>Exclusive or</td>
<td>xor rs1, reg_or_imm, reg</td>
<td>A1</td>
</tr>
<tr>
<td>XORcc</td>
<td>01</td>
<td>Exclusive or and modify cc's</td>
<td>xorcc rs1, reg_or_imm, reg</td>
<td>A1</td>
</tr>
<tr>
<td>XNOR</td>
<td>00</td>
<td>Exclusive nor</td>
<td>xnor rs1, reg_or_imm, reg</td>
<td>A1</td>
</tr>
<tr>
<td>XNORcc</td>
<td>01</td>
<td>Exclusive nor and modify cc's</td>
<td>xnorcc rs1, reg_or_imm, reg</td>
<td>A1</td>
</tr>
</tbody>
</table>

**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)

**Exceptions**

illegal_instruction

**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.

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 291.
- Underflow Behavior on page 292.
- Integer Overflow Definition on page 293.
- Floating-Point Nonstandard Mode on page 293.
- Arithmetic Result Tables on page 294.

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 42 and elsewhere, when a floating-point trap occurs, the following conditions are true:
- The destination floating-point register(s) (the F registers) are unchanged.
- The floating-point condition codes (fcc0, fcc1, fcc2, and fcc3) are unchanged.
- The FSR.aexc (accrued exceptions) field is unchanged.
- The FSR.cexc (current exceptions) field is unchanged except for IEEE_754_exceptions; in that case, cexc contains a bit set to 1, corresponding to the exception that caused the trap. Only one bit shall be set in cexc.

Instructions causing an fp_exception_other trap because of unfinished FPops execute as if by hardware; that is, such a trap is undetectable by application software, except that timing may be affected.
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

\[
0 \leq |u| \leq \text{smallest normalized number}
\]

would round, if no trap intervened, to a rounded value \( r \) which might be zero, subnormal, or the smallest normalized value.

<table>
<thead>
<tr>
<th>( u = r )</th>
<th>( r ) is minimum normal</th>
<th>( r ) is subnormal</th>
<th>( r ) is zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \neq r )</td>
<td>( r ) is minimum normal</td>
<td>( r ) is subnormal</td>
<td>( r ) is zero</td>
</tr>
<tr>
<td>( u = r )</td>
<td>None</td>
<td>UF</td>
<td>None</td>
</tr>
<tr>
<td>( u \neq r )</td>
<td>UF</td>
<td>UF</td>
<td>UF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( u = r )</th>
<th>( u ) is ( \text{minimum normal} )</th>
<th>( u ) is ( \text{subnormal} )</th>
<th>( u ) is ( \text{zero} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \neq r )</td>
<td>( u ) is ( \text{minimum normal} )</td>
<td>( u ) is ( \text{subnormal} )</td>
<td>( u ) is ( \text{zero} )</td>
</tr>
<tr>
<td>( u = r )</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>( u \neq r )</td>
<td>UF</td>
<td>UF</td>
<td>UF</td>
</tr>
</tbody>
</table>

**Programming Note** A user-mode trap handler invoked for an IEEE_754_exception, whether as a direct result of a hardware \( \text{fp_exception_ieee}_754 \) trap or as an indirect result of privileged software handling of an \( \text{fp_exception_other} \) 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.

**TABLE 8-1** Floating-Point Underflow Behavior (Tininess Detected Before Rounding)

- \( \text{ufm} = 1 \) and \( \text{nmx} = x \)
- \( \text{ufm} = 0 \) and \( \text{nmx} = 1 \)
- \( \text{ufm} = 0 \) and \( \text{nmx} = 0 \)

**Note**

\( \text{UF} = \text{fp_exception_ieee}_754 \) trap with cexc.ufc = 1

\( \text{NX} = \text{fp_exception_ieee}_754 \) trap with cexc.nxc = 1

\( \text{uf} = \text{cexc.ufc} = 1, \ aexc.ufa = 1, \) no \( \text{fp_exception_ieee}_754 \) trap

\( \text{nx} = \text{cexc.nxc} = 1, \ aexc.nxa = 1, \) no \( \text{fp_exception_ieee}_754 \) 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, 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 43).

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 \( \text{FSR.ns} = 1 \) are as follows:

- **IMPL. DEP. #18-V8-Ms10(a):** When \( \text{FSR.ns} = 1 \) 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 46). Whether the generated condition(s) trigger an \( \text{fp_exception_ieee_754} \) exception or not depends on the setting of \( \text{FSR.tem} \).

- **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 \( \text{fp_exception_ieee_754} \) exception or not depends on the setting of \( \text{FSR.tem} \).

- **IMPL. DEP. #18-V8-Ms10(c):** If an FPop generates an intermediate result value, the intermediate value is subnormal, and \( \text{FSR.ns} = 1 \), 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 \( \text{fp_exception_ieee_754} \) exception or not depends on the setting of \( \text{FSR.tem} \).

If \( \text{GSR.im} = 1 \), then the value of \( \text{FSR.ns} \) is ignored and the processor operates as if \( \text{FSR.ns} = 0 \) (see page 54).

### 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:
- \( \text{Nn} \) is a number in \( \text{F[rsn]} \), which may be normal or subnormal.
- \( \text{QNaNn} \) and \( \text{SNaNn} \) are Quiet and Signaling Not-a-Number values in \( \text{F[rsn]} \), respectively.

Notes on result types:
- \( \text{R} \): (rounded) result of operation, which may be normal, subnormal, zero, or infinity. May also cause OF, UF, NX, unfinished.
- \( \text{dQNaN} \) 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.
- \( \text{QSNaNn} \) is the Signalling NaN operand from \( \text{F[rsn]} \) with the Quiet bit asserted.
8.5.1 Floating-Point Add (FADD)

**TABLE 8-2**  Floating-Point Add operation (F[rs1] + F[rs2])

<table>
<thead>
<tr>
<th>F[rs2]</th>
<th>−∞</th>
<th>−N2</th>
<th>−0</th>
<th>+0</th>
<th>+N2</th>
<th>+∞</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>−∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dQNaN, NV</td>
</tr>
<tr>
<td>−N1</td>
<td>−R</td>
<td>−N1</td>
<td>±R*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
</tr>
<tr>
<td>−0</td>
<td>−N2</td>
<td>−0</td>
<td>±0**</td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
</tr>
<tr>
<td>+0</td>
<td>±0**</td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
</tr>
<tr>
<td>+N1</td>
<td>±R*</td>
<td>+N1</td>
<td>+R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
</tr>
<tr>
<td>+∞</td>
<td>dQNaN, NV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Q NaN2</td>
</tr>
<tr>
<td>QNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Q NaN1</td>
</tr>
<tr>
<td>SNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S NaN1</td>
</tr>
</tbody>
</table>

* 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])

<table>
<thead>
<tr>
<th>F[rs2]</th>
<th>−∞</th>
<th>−N2</th>
<th>−0</th>
<th>+0</th>
<th>+N2</th>
<th>+∞</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>dQNaN, NV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>−∞</td>
</tr>
<tr>
<td>−N1</td>
<td>±R*</td>
<td>−N1</td>
<td>−R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
</tr>
<tr>
<td>−0</td>
<td>+N2</td>
<td>±0**</td>
<td>−0</td>
<td>−N2</td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
</tr>
<tr>
<td>+0</td>
<td>±0**</td>
<td>±0**</td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
</tr>
<tr>
<td>+N1</td>
<td>+R</td>
<td>+N1</td>
<td>±R*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dQNaN, NV</td>
</tr>
<tr>
<td>QNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Q NaN1</td>
</tr>
<tr>
<td>SNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S NaN1</td>
</tr>
</tbody>
</table>

* 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 ≠ 0 − x when x is zero or NaN.
8.5.3 Floating-Point Multiply

TABLE 8-4 Floating-Point Multiply operation ($F[rs1] \times F[rs2]$)

<table>
<thead>
<tr>
<th>$F[rs2]$</th>
<th>$-\infty$</th>
<th>$-N2$</th>
<th>$-0$</th>
<th>$+0$</th>
<th>$+N2$</th>
<th>$+\infty$</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>+\infty</td>
<td>dQNaN, NV</td>
<td></td>
<td></td>
<td>+R</td>
<td>-R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-N1$</td>
<td>-R</td>
<td>+0</td>
<td>-0</td>
<td>+0</td>
<td></td>
<td>dQNaN, NV</td>
<td>QNaN2</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>dQNaN, NV</td>
<td>+0</td>
<td>-0</td>
<td>+0</td>
<td></td>
<td>dQNaN, NV</td>
<td>QNaN2</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>-0</td>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+N1$</td>
<td>+R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+R</td>
<td>QNaN2</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>-0</td>
<td>dQNaN, NV</td>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN1</td>
<td></td>
</tr>
<tr>
<td>SNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SNaN1</td>
</tr>
</tbody>
</table>

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 296) to select a row in the table below.
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_FP_pop for additional details.

### 8.5.5 Floating-Point Negative Multiply-Add (FN MADD)

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 296) to select a row in the table below.
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 296) to select a row in the table below.
TABLE 8-7 Floating-Point Multiply-Subtract ((F[rs1] × F[rs2])− F[rs3])

<table>
<thead>
<tr>
<th>F[rs3]</th>
<th>−∞</th>
<th>−N3</th>
<th>0</th>
<th>+N3</th>
<th>+∞</th>
<th>QNaN3</th>
<th>SNaN3</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>dQNaN, NV</td>
<td>±R*</td>
<td>−N</td>
<td>−R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−N</td>
<td>±0**</td>
<td>−0</td>
<td>−N3</td>
<td>QNaN3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>+0</td>
<td>±0**</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+N</td>
<td>+R</td>
<td>+N</td>
<td>±R*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>±∞</td>
<td>+∞</td>
<td>dQNaN, NV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F[rs1] × F[rs2]

QNaN1
QNaN2
QNaN (±0 × ±∞)
QNaN1
QNaN2

* 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 296) to select a row in the table below.
R may be any number; its generation may cause OE, 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 \( \pm \infty \).

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

<table>
<thead>
<tr>
<th>( F[\text{rs3}] )</th>
<th>(-\infty)</th>
<th>(-N)</th>
<th>(-0)</th>
<th>(+0)</th>
<th>(+N3)</th>
<th>(+\infty)</th>
<th>( \text{QNaN3} )</th>
<th>( \text{SNaN3} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F[\text{rs1}] \times F[\text{rs2}] )</td>
<td>( \text{dQNaN, NV} ) &amp; ( \pm R^* ) &amp; ( +N ) &amp; ( +R ) &amp; ( +N3 ) &amp; ( QNaN3 ) &amp; ( \text{QNaN3, NV} ) &amp; ( \text{QNaN3, NV} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{QNaN1} )</td>
<td>( \text{QNaN1} )</td>
<td>( \text{QNaN2} )</td>
<td>( \text{dQNaN, NV***} ) &amp; ( \text{QNaN3, NV***} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{QNaNN} ) &amp; ( \pm0 \times \pm0 )</td>
<td>( \text{dQNaN, NV***} )</td>
<td>( \text{QNaN3, NV***} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{QSNaN1} )</td>
<td>( \text{QSNaN1, NV***} )</td>
<td>( \text{QSNaN2, NV***} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{QSNaN2} )</td>
<td>( \text{QSNaN2, NV***} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* if \( N = N3 \), then **

** result is \(+0\) unless rounding mode is round to \(-\infty\), in which case the result is \(-0\)

*** if \( \text{FSR.nvm} = 1 \), \( \text{FSR.nvc} \leftarrow 1 \), the trap occurs, and \( \text{FSR.aexc} \) is left unchanged; otherwise, \( \text{FSR.nvm} = 0 \) so \( \text{FSR.nva} \leftarrow 1 \) and for FNMSUB \( \text{FSR.nve} \leftarrow 1 \).
### 8.5.8 Floating-Point Divide (FDIV)

**TABLE 8-9**  Floating-Point Divide operation ($F_{rs1} \div F_{rs2}$)

<table>
<thead>
<tr>
<th>$F_{rs1}$</th>
<th>$\infty$</th>
<th>$-N2$</th>
<th>-0</th>
<th>+0</th>
<th>$+N2$</th>
<th>$\pm\infty$</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>dQNaN, NV</td>
<td>+∞</td>
<td>-∞</td>
<td>+∞</td>
<td>-∞</td>
<td>dQNaN, NV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-N1</td>
<td>+R</td>
<td>$+\infty$, DZ</td>
<td>$-\infty$, DZ</td>
<td>-R</td>
<td></td>
<td></td>
<td>QNaN2</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>dQNaN, NV</td>
<td></td>
<td>-0</td>
<td></td>
<td></td>
<td>QNaN2</td>
<td>SNaN2, NV</td>
</tr>
<tr>
<td>+0</td>
<td>-0</td>
<td></td>
<td></td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td>NV</td>
</tr>
<tr>
<td>+N1</td>
<td>-R</td>
<td></td>
<td>-∞, DZ</td>
<td>$+\infty$, DZ</td>
<td>+R</td>
<td></td>
<td>QNaN1</td>
<td></td>
</tr>
<tr>
<td>$\pm\infty$</td>
<td>dQNaN, NV</td>
<td>-∞</td>
<td>+∞</td>
<td>+∞</td>
<td>dQNaN, NV</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 ($\sqrt{F_{rs2}}$)

<table>
<thead>
<tr>
<th>$F_{rs2}$</th>
<th>$\infty$</th>
<th>$-N2$</th>
<th>-0</th>
<th>+0</th>
<th>$+N2$</th>
<th>$\pm\infty$</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dQNaN, \ NV$</td>
<td>-0</td>
<td>+0</td>
<td>+R</td>
<td>$+\infty$</td>
<td>QNaN2</td>
<td>QNaN2, NV</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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[r1] ? F[r2])

<table>
<thead>
<tr>
<th>F[r2]</th>
<th>−∞</th>
<th>−N2</th>
<th>−0</th>
<th>+0</th>
<th>+N2</th>
<th>+∞</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−N1</td>
<td></td>
<td>0, 1, 2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td>0, 1, 2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+N1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* NV for FCMPE, but not for FCMP.

TABLE 8-12  FSR.fcc Encoding for Result of FCMP, FCMPE

<table>
<thead>
<tr>
<th>fcc result</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>=</td>
</tr>
<tr>
<td>1</td>
<td>&lt;</td>
</tr>
<tr>
<td>2</td>
<td>&gt;</td>
</tr>
<tr>
<td>3</td>
<td>unordered</td>
</tr>
</tbody>
</table>

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[r2]))

<table>
<thead>
<tr>
<th>F[r2]</th>
<th>−SNaN2</th>
<th>−QNaN2</th>
<th>−∞</th>
<th>−N2</th>
<th>−0</th>
<th>+0</th>
<th>+N2</th>
<th>+∞</th>
<th>+QNaN2</th>
<th>+SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>−QNaN2, NV</td>
<td>−QNaN2</td>
<td>−∞</td>
<td>−R</td>
<td>−0</td>
<td>+0</td>
<td>+R</td>
<td>+∞</td>
<td>+QNaN2</td>
<td>+QNaN2, NV</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>NV</th>
<th>SNaN operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF</td>
<td>FdTOs, FqTOs: the input is larger than can be expressed in single precision</td>
</tr>
<tr>
<td></td>
<td>FqTOd: the input is larger than can be expressed in double precision</td>
</tr>
<tr>
<td></td>
<td>does not occur during other conversion operations</td>
</tr>
<tr>
<td>UF</td>
<td>FdTOs, FqTOs: the input is smaller than can be expressed in single precision</td>
</tr>
<tr>
<td></td>
<td>FqTOd: the input is smaller than can be expressed in double precision</td>
</tr>
<tr>
<td></td>
<td>does not occur during other conversion operations</td>
</tr>
<tr>
<td>NX</td>
<td>FdTOs, FqTOs: the input fraction has more significant bits than can be held in a single precision fraction</td>
</tr>
<tr>
<td></td>
<td>FqTOd: the input fraction has more significant bits than can be held in a double precision fraction</td>
</tr>
<tr>
<td></td>
<td>does not occur during other conversion operations</td>
</tr>
</tbody>
</table>
```

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]))

<table>
<thead>
<tr>
<th>F[rs2]</th>
<th>−SNaN2</th>
<th>−QNaN2</th>
<th>−∞</th>
<th>−N2</th>
<th>0</th>
<th>+0</th>
<th>+∞</th>
<th>+QNaN2</th>
<th>+SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FdTOx</td>
<td>−2^63,</td>
<td>−2^63,</td>
<td>−R</td>
<td>0</td>
<td>+R</td>
<td>2^63−1,</td>
<td>2^63−1,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NV</td>
<td>NV</td>
<td></td>
<td></td>
<td></td>
<td>NV</td>
<td>NV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FsTOx</td>
<td>−2^31,</td>
<td>−2^31,</td>
<td></td>
<td></td>
<td></td>
<td>2^31−1,</td>
<td>2^31−1,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NV</td>
<td>NV</td>
<td></td>
<td></td>
<td></td>
<td>NV</td>
<td>NV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqTOx</td>
<td>−2^31,</td>
<td>−2^31,</td>
<td></td>
<td></td>
<td></td>
<td>2^31−1,</td>
<td>2^31−1,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NV</td>
<td>NV</td>
<td></td>
<td></td>
<td></td>
<td>NV</td>
<td>NV</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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 293.

```
Table 8-16  Floating-point to Integer Conversion Exception Conditions

<table>
<thead>
<tr>
<th>NV</th>
<th>SNaN operand</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QNaN operand</td>
</tr>
<tr>
<td></td>
<td>±∞ operand</td>
</tr>
<tr>
<td></td>
<td>integer overflow</td>
</tr>
<tr>
<td>NX</td>
<td>non-integer source (truncation occurred)</td>
</tr>
</tbody>
</table>
```
8.5.13 Integer to Floating-Point Conversions ($F_{i|x}TO_{s|d|q}$)

**TABLE 8-17** Integer to Floating-Point Conversions (convert($F_{rs2}$))

<table>
<thead>
<tr>
<th>$F_{rs2}$</th>
<th>$-\text{int}$</th>
<th>0</th>
<th>$+\text{int}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-R$</td>
<td>$+0$</td>
<td>$+R$</td>
<td></td>
</tr>
</tbody>
</table>

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)  
  • not applicable to $FiTOd$, $FxTOq$, or $FiTOq$ (FSR.cexc will always be cleared) |
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 305.
- Memory Accesses and Cacheability on page 306.
- Memory Addressing and Alternate Address Spaces on page 308.
- SPARC V9 Memory Model on page 310.
- The UltraSPARC Architecture Memory Model — TSO on page 313.
- Nonfaulting Load on page 319.
- Store Coalescing on page 320.

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 or real.
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).

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.

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 320).

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 317.

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 73, 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. 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 can be presented in nonprivileged mode and privileged mode.

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.

Nonprivileged software only uses virtual addresses. Privileged software uses virtual and real addresses.

9.3.2 Memory Address Spaces

The UltraSPARC Architecture supports accessing memory using virtual or real addresses. Multiple virtual address spaces within the same real address space are distinguished by a context identifier (context ID).

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.

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.
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.

The full representation of a real address is as follows:

\[
\text{real_address} = \text{context_ID} :: \text{virtual_address}
\]

### 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 and privileged modes is indicated in TABLE 10-1 on page 323.

Attempts by nonprivileged software (PSTATE.priv = 0) to access restricted ASIs (ASI bit 7 = 0) cause a privileged_action exception. Attempts by privileged software (PSTATE.priv = 1) 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 \(\leq\) 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).

SPARC V9 supports the PRIMARY[_LITTLE], SECONDARY[_LITTLE], and NUCLEUS[_LITTLE] address spaces.

Accesses to other address spaces use the load/store alternate instructions. For these accesses, the ASI is either contained in the instruction (for the register-register addressing mode) or taken from the ASI register (for register-immediate addressing).

ASIs are either nonrestricted or restricted-to-privileged:
A nonrestricted ASI (ASI range $80_{16}$ – $FF_{16}$) is one that may be used independently of the privilege level ($PSTATE.priv$) 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 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

<table>
<thead>
<tr>
<th>ASI Value</th>
<th>Type</th>
<th>Result of ASI Access in NP Mode</th>
<th>Result of ASI Access in P Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00_{16}$ – $2F_{16}$</td>
<td>Restricted-to-privileged</td>
<td>privileged action, exception</td>
<td>Valid Access</td>
</tr>
<tr>
<td>$80_{16}$ – $FF_{16}$</td>
<td>Nonrestricted</td>
<td>Valid Access</td>
<td>Valid Access</td>
</tr>
</tbody>
</table>

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.

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 316 and The UltraSPARC Architecture Memory Model — TSO on page 313 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](image)

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 318 for details. Memory is shared in terms of address space, but it may be nonhomogeneous and distributed in an implementation. 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.

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.

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.
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 \texttt{PSTATE.mm} = 10_2 or 11_2 are supported in an UltraSPARC Architecture processor is implementation dependent (impl. dep. #I13-V9-Ms10). If the 10_2 model is supported, then when \texttt{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 \texttt{PSTATE.mm}. The value 00_2 represents the TSO memory model; increasing values of \texttt{PSTATE.mm} indicate increasingly weaker (less strongly ordered) memory models.

Writing a new value into \texttt{PSTATE.mm} causes subsequent memory reference instructions to be performed with the order constraints of the specified memory model.
**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 311), 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 313) 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.
CHAPTER 9 • Memory

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>
<thead>
<tr>
<th>From Memory Operation R (row)</th>
<th>To Memory Operation C (column):</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td># # # S S # # # # S S</td>
</tr>
<tr>
<td>store</td>
<td>M # # M S M # M # M S</td>
</tr>
<tr>
<td>atomic</td>
<td># # # M S # # # # M S</td>
</tr>
<tr>
<td>bload</td>
<td>S S S S S S S S S S S S S S</td>
</tr>
<tr>
<td>load_nc_e</td>
<td># # # S S #1 #1 #1 #1 S S</td>
</tr>
<tr>
<td>store_nc_e</td>
<td>S # # S S #1 #1 M2 #1 M S</td>
</tr>
<tr>
<td>load_nc_ne</td>
<td># # # S S #1 #1 #1 #1 S S</td>
</tr>
<tr>
<td>store_nc_ne</td>
<td>S # # S S M2 #1 M2 #1 M S</td>
</tr>
<tr>
<td>bload_nc</td>
<td>S S S S S S S S S S S S</td>
</tr>
<tr>
<td>bstore_nc</td>
<td>S S S S S M S M S M S M S</td>
</tr>
</tbody>
</table>

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.

Note that transitivity applies; if operation X is always ordered before operation Y ("#" in TABLE 9-2) and operation Y is always ordered before operation Z (again, "#" in the table), then the sequence of operations X ... Y ... Z may safely be executed with no intervening MEMBAR, even if the table shows that a MEMBAR is normally needed between X and Z. For example, a MEMBAR is normally needed between a store and a load ("M" in TABLE 9-2); however, the sequence "store ... atomic ... load" may be executed safely with no intervening MEMBAR because stores are always ordered before atomics and atomics are always ordered before loads.
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. 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 (LDSTUBA, 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 0116, 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 0816 (#StoreStore) is equivalent to the STBAR instruction; it requires that the values stored by store instructions appearing in program order prior to the STBAR instruction be visible to other virtual processors before issuing any store operations that appear in program order following the STBAR.

In TABLE 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>
<thead>
<tr>
<th>Ordering Relation, Earlier &lt;m Later</th>
<th>Assembly Language Constant Mnemonic</th>
<th>Effective Behavior in TSO model</th>
<th>Mask Value</th>
<th>nmask Bit #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load &lt;m Load</td>
<td>#LoadLoad</td>
<td>nop</td>
<td>0116</td>
<td>0</td>
</tr>
<tr>
<td>Store &lt;m Load</td>
<td>#StoreLoad</td>
<td>#StoreLoad</td>
<td>0216</td>
<td>1</td>
</tr>
<tr>
<td>Load &lt;m Store</td>
<td>#LoadStore</td>
<td>nop</td>
<td>0416</td>
<td>2</td>
</tr>
<tr>
<td>Store &lt;m Store</td>
<td>#StoreStore</td>
<td>nop</td>
<td>0816</td>
<td>3</td>
</tr>
</tbody>
</table>

**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.

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.
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>
<thead>
<tr>
<th>Sequecing Function</th>
<th>Assembler Tag</th>
<th>Mask Value</th>
<th>cmask Bit #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookaside Barrier</td>
<td>#Lookaside</td>
<td>10₁₆</td>
<td>0</td>
</tr>
<tr>
<td>Memory Issue Barrier</td>
<td>#MemIssue</td>
<td>2₀₁₆</td>
<td>1</td>
</tr>
<tr>
<td>Synchronization Barrier</td>
<td>#Sync</td>
<td>4₀₁₆</td>
<td>2</td>
</tr>
</tbody>
</table>

**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 201 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.
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.

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.

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 133 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 016 can safely be accessed with no penalty. 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 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.
Address Space Identifiers (ASIs)

This appendix describes address space identifiers (ASIs) in the following sections:

- Address Space Identifiers and Address Spaces on page 321.
- ASI Values on page 321.
- ASI Assignments on page 322.
- Special Memory Access ASIs on page 329.

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.

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.

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 \textit{Translating} ASI (the most common type) causes the accompanying address to be treated as a virtual address (which is translated by the MMU).
- A \textit{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 \textit{Real} ASI causes the accompanying address to be treated as a real address. An access using a Real ASI can cause exception(s) only visible in hyperprivileged mode. Real ASIs are typically used by privileged software for directly accessing memory using real (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 implementation 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 supplied virtual address is decoded by the virtual processor.

The “V / non-T / R” column of the table indicates whether each ASI is a Translating ASI(translates from Virtual), non-Translating ASI, or Real ASI, respectively.

ASIs marked "Reserved" are set aside for use in future revisions to the architecture and are not to be used by implementations. 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>
<thead>
<tr>
<th>ASI Value</th>
<th>req'd(opt)</th>
<th>ASI Name (and Abbreviation)</th>
<th>Access Type(s)</th>
<th>Virtual Address (VA)</th>
<th>V/ non-T/R</th>
<th>Shared/strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0016–0316</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>0416</td>
<td>●</td>
<td>ASI_NUCLEUS (ASI_N)</td>
<td>RW²,4 (decoded)</td>
<td>v</td>
<td></td>
<td></td>
<td>Implicit address space, nucleus context, TL &gt; 0</td>
</tr>
<tr>
<td>0516–0B16</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>0C16</td>
<td>●</td>
<td>ASI_NUCLEUS_LITTLE (ASI_NL)</td>
<td>RW²,4 (decoded)</td>
<td>v</td>
<td></td>
<td></td>
<td>Implicit address space, nucleus context, TL &gt; 0, little-endian</td>
</tr>
<tr>
<td>0D16–0F16</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>1016</td>
<td>●</td>
<td>ASI_AS_IF_USER_PRIMARY (ASI_AIUP)</td>
<td>RW²,4,18 (decoded)</td>
<td>v</td>
<td></td>
<td>Primary address space, as if user (nonprivileged)</td>
<td></td>
</tr>
<tr>
<td>1116</td>
<td>●</td>
<td>ASI_AS_IF_USER_SECONDARY (ASI_AIUS)</td>
<td>RW²,4,18 (decoded)</td>
<td>v</td>
<td></td>
<td>Secondary address space, as if user (nonprivileged)</td>
<td></td>
</tr>
<tr>
<td>1216–1316</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>1416</td>
<td>○</td>
<td>ASI_REAL</td>
<td>RW²,4 (decoded)</td>
<td>r</td>
<td></td>
<td></td>
<td>Real address</td>
</tr>
<tr>
<td>1516</td>
<td>○</td>
<td>ASI_REAL_IO²⁵</td>
<td>RW²,4 (decoded)</td>
<td>r</td>
<td></td>
<td></td>
<td>Real address, noncacheable, with side effect (deprecated)</td>
</tr>
<tr>
<td>1616</td>
<td>○</td>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)</td>
<td>RW²,4,14,18 (decoded)</td>
<td>v</td>
<td></td>
<td>Primary address space, block load/store, as if user (nonprivileged)</td>
<td></td>
</tr>
<tr>
<td>1716</td>
<td>○</td>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS)</td>
<td>RW²,4,14,18 (decoded)</td>
<td>v</td>
<td></td>
<td>Secondary address space, block load/store, as if user (nonprivileged)</td>
<td></td>
</tr>
<tr>
<td>1816</td>
<td>●</td>
<td>ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)</td>
<td>RW²,4,18 (decoded)</td>
<td>v</td>
<td></td>
<td>Primary address space, as if user (nonprivileged), little-endian</td>
<td></td>
</tr>
<tr>
<td>1916</td>
<td>●</td>
<td>ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUPL)</td>
<td>RW²,4,18 (decoded)</td>
<td>v</td>
<td></td>
<td>Secondary address space, as if user (nonprivileged), little-endian</td>
<td></td>
</tr>
<tr>
<td>1A16–1B16</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>1C16</td>
<td>○</td>
<td>ASI_REAL_LITTLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Real address, little-endian</td>
</tr>
<tr>
<td>1D16</td>
<td>○</td>
<td>ASI_REAL_IO_LITTLE²⁷ (ASI_REAL_IO_L²⁷)</td>
<td>RW²,5 (decoded)</td>
<td>r</td>
<td></td>
<td>Real address, noncacheable, with side effect, little-endian (deprecated)</td>
<td></td>
</tr>
<tr>
<td>1E16</td>
<td>○</td>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUPL)</td>
<td>RW²,3,14,18 (decoded)</td>
<td>v</td>
<td></td>
<td>Primary address space, block load/store, as if user (nonprivileged), little-endian</td>
<td></td>
</tr>
<tr>
<td>1F16</td>
<td>○</td>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUS_L)</td>
<td>RW²,3,14,18 (decoded)</td>
<td>v</td>
<td></td>
<td>Secondary address space, block load/store, as if user (nonprivileged), little-endian</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 10-1  UltraSPARC Architecture ASIs  (2 of 5)

<table>
<thead>
<tr>
<th>ASI Value</th>
<th>req'd (●)</th>
<th>ASI Name (and Abbreviation)</th>
<th>Access Type(s)</th>
<th>Virtual Address [VA]</th>
<th>V/ non-T/ R</th>
<th>Shared per strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016</td>
<td>○</td>
<td>ASI_SCRATCHPAD</td>
<td>RW(^{2,6})</td>
<td>(decoded; see below)</td>
<td>non-T per strand</td>
<td>Privileged Scratchpad registers; implementation dependent(^1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0(_{16})</td>
<td>&quot;</td>
<td>0(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Scratchpad Register 0(^{1})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8(_{16})</td>
<td>&quot;</td>
<td>8(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Scratchpad Register 1(^{1})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10(_{16})</td>
<td>&quot;</td>
<td>10(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Scratchpad Register 2(^{1})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18(_{16})</td>
<td>&quot;</td>
<td>18(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Scratchpad Register 3(^{1})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20(_{16})</td>
<td>&quot;</td>
<td>20(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Scratchpad Register 4(^{1})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28(_{16})</td>
<td>&quot;</td>
<td>28(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Scratchpad Register 5(^{1})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30(_{16})</td>
<td>&quot;</td>
<td>30(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Scratchpad Register 6(^{1})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>38(_{16})</td>
<td>&quot;</td>
<td>38(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Scratchpad Register 7(^{1})</td>
</tr>
<tr>
<td>2116</td>
<td>○</td>
<td>ASI_MMU_CONTEXTID</td>
<td>RW(^{2,6})</td>
<td>(decoded; see below)</td>
<td>non-T per strand</td>
<td>MMU context registers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8(_{16})</td>
<td>&quot;</td>
<td>8(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>I/D MMU Primary Context ID register 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10(_{16})</td>
<td>&quot;</td>
<td>10(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>I/D MMU Secondary Context ID register 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108(_{16})</td>
<td>&quot;</td>
<td>108(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>I/D Primary Context ID register 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110(_{16})</td>
<td>&quot;</td>
<td>110(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>I/D MMU Secondary Context ID register 1</td>
</tr>
<tr>
<td>2216</td>
<td>○</td>
<td>ASI_TWINX_AS_IP_USER_, PRIMARY (ASI_TWINX_AIUP)</td>
<td>R(^{2,7,11})</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 128-bit atomic load twin extended word, as if user (nonprivileged)</td>
</tr>
<tr>
<td>2316</td>
<td>○</td>
<td>ASI_TWINX_AS_IP_USER_, SECONDARY (ASI_TWINX_AIUS)</td>
<td>R(^{2,7,11})</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 128-bit atomic load twin extended word, as if user (nonprivileged)</td>
</tr>
<tr>
<td>2416</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent(^1)</td>
</tr>
<tr>
<td>2516</td>
<td>○</td>
<td>ASI_QUEUE</td>
<td>(see below)</td>
<td>(decoded; see below)</td>
<td>non-T per strand</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RW(^{2,6}) 3C0(_{16})</td>
<td>&quot;</td>
<td>3C0(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>CPU Mondo Queue Head Pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RW(^{2,6,17}) 3C8(_{16})</td>
<td>&quot;</td>
<td>3C8(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>CPU Mondo Queue Tail Pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RW(^{2,6}) 3D0(_{16})</td>
<td>&quot;</td>
<td>3D0(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Device Mondo Queue Head Pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RW(^{2,6,17}) 3D8(_{16})</td>
<td>&quot;</td>
<td>3D8(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Device Mondo Queue Tail Pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RW(^{2,6}) 3E0(_{16})</td>
<td>&quot;</td>
<td>3E0(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Resumable Error Queue Head Pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RW(^{2,6,17}) 3E8(_{16})</td>
<td>&quot;</td>
<td>3E8(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Resumable Error Queue Tail Pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RW(^{2,6}) 3F0(_{16})</td>
<td>&quot;</td>
<td>3F0(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Nonresumable Error Queue Head Pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RW(^{2,6,17}) 3F8(_{16})</td>
<td>&quot;</td>
<td>3F8(_{16})</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Nonresumable Error Queue Tail Pointer</td>
</tr>
</tbody>
</table>
### CHAPTER 10 • Address Space Identifiers (ASIs)

#### Table 10-1: UltraSPARC Architecture ASIs (3 of 5)

<table>
<thead>
<tr>
<th>ASI Value</th>
<th>req’d (●)</th>
<th>ASI Name (and Abbreviation)</th>
<th>Access Type(s)</th>
<th>Virtual Address (VA)</th>
<th>V/ non-T/R</th>
<th>Shared/per strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2616</td>
<td>○</td>
<td>ASI_TWINX_REAL (ASI_TWINX_R)</td>
<td>R^2,7,11</td>
<td><em>decoded</em></td>
<td>R</td>
<td>—</td>
<td>128-bit atomic twin extended-word load from real address</td>
</tr>
<tr>
<td>2716</td>
<td>○</td>
<td>ASI_TWINX_NUCLEUS (ASI_TWINX_N)</td>
<td>R^2,7,11</td>
<td><em>decoded</em></td>
<td>V</td>
<td>—</td>
<td>Nucleus context, 128-bit atomic load twin extended-word</td>
</tr>
<tr>
<td>2816−</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent†</td>
</tr>
<tr>
<td>2916</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2A16</td>
<td>○</td>
<td>ASI_TWINX_AS_IF_USER_ PRIMARY_LITTLE (ASI_TWINXAIUPL)</td>
<td>R^2,7,11</td>
<td><em>decoded</em></td>
<td>V</td>
<td>—</td>
<td>Primary address space, 128-bit atomic load twin extended-word, as if user (nonprivileged), little-endian</td>
</tr>
<tr>
<td>2B16</td>
<td>○</td>
<td>ASI_TWINX_AS_IF_USER_ SECONDARY_LITTLE (ASI_TWINX_AIUS_L)</td>
<td>R^2,7,11</td>
<td><em>decoded</em></td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 128-bit atomic load twin extended-word, as if user (nonprivileged), little-endian</td>
</tr>
<tr>
<td>4C16</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent†</td>
</tr>
<tr>
<td>2D16</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent†</td>
</tr>
<tr>
<td>2E16</td>
<td>○</td>
<td>ASI_TWINX_REAL_LITTLE (ASI_TWINX_REAL_L)</td>
<td>R^2,7,11</td>
<td><em>decoded</em></td>
<td>R</td>
<td>—</td>
<td>128-bit atomic twin-extended-word load from real address, little-endian</td>
</tr>
<tr>
<td>2F16</td>
<td>○</td>
<td>ASI_TWINX_NUCLEUS_LITTLE (ASI_TWINX_NL)</td>
<td>R^2,7,11</td>
<td><em>decoded</em></td>
<td>V</td>
<td>—</td>
<td>Nucleus context, 128-bit atomic load twin extended-word, little-endian</td>
</tr>
<tr>
<td>3016−</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Reserved for use in hyperprivilege mode</td>
</tr>
<tr>
<td>7F16</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4516</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent†</td>
</tr>
<tr>
<td>4616−</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent†</td>
</tr>
<tr>
<td>4816</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4916</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent†</td>
</tr>
<tr>
<td>4A16−</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent†</td>
</tr>
<tr>
<td>4B16</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4C16</td>
<td>○</td>
<td>Error Status and Enable Registers</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent†</td>
</tr>
<tr>
<td>7516−</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>7F16</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8016</td>
<td>●</td>
<td>ASI_PRIMARY (ASI_P)</td>
<td>RW^4</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Implicit primary address space</td>
</tr>
<tr>
<td>8116</td>
<td>●</td>
<td>ASI_SECONDARY (ASI_S)</td>
<td>RW^4</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space</td>
</tr>
<tr>
<td>8216</td>
<td>●</td>
<td>ASI_PRIMARY_NO_FAULT (ASI_PNF)</td>
<td>R^2,7,11</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, no fault</td>
</tr>
<tr>
<td>8316</td>
<td>●</td>
<td>ASI_SECONDARY_NO_FAULT (ASI_SNF)</td>
<td>R^2,7,11</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, no fault</td>
</tr>
<tr>
<td>8416−</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>8716</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8816</td>
<td>●</td>
<td>ASI_PRIMARY_LITTLE (ASI_PL)</td>
<td>RW^4</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Implicit primary address space, little-endian</td>
</tr>
<tr>
<td>8916</td>
<td>●</td>
<td>ASI_SECONDARY_LITTLE (ASI_SL)</td>
<td>RW^4</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, little-endian</td>
</tr>
<tr>
<td>8A16</td>
<td>●</td>
<td>ASI_PRIMARY_NO_FAULT_LITTLE (ASI_PNFL)</td>
<td>R^2,7,11</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, no fault, little-endian</td>
</tr>
<tr>
<td>ASI Value</td>
<td>req’d (●)</td>
<td>ASI Name (and Abbreviation)</td>
<td>Access Type(s)</td>
<td>Virtual Address (VA)</td>
<td>V/ non-T/ R</td>
<td>Shared per strand</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8B16</td>
<td>●</td>
<td>ASI_SECONDARY_NO_FAULT_LITTLE (ASI_SNFL)</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, no fault, little-endian</td>
<td></td>
</tr>
<tr>
<td>C016</td>
<td>○</td>
<td>ASI_PST8_PRIMARY (ASI_PST8_P)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 8x8-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C116</td>
<td>○</td>
<td>ASI_PST8_SECONDARY (ASI_PST8_S)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 8x8-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C216</td>
<td>○</td>
<td>ASI_PST16_PRIMARY (ASI_PST16_P)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 4x16-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C316</td>
<td>○</td>
<td>ASI_PST16_SECONDARY (ASI_PST16_S)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 4x16-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C416</td>
<td>○</td>
<td>ASI_PST32_PRIMARY (ASI_PST32_P)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 2x32-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C516</td>
<td>○</td>
<td>ASI_PST32_SECONDARY (ASI_PST32_S)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 2x32-bit partial store</td>
<td></td>
</tr>
<tr>
<td>CA16</td>
<td>○</td>
<td>ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 4x16-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CB16</td>
<td>○</td>
<td>ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 4x16-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CC16</td>
<td>○</td>
<td>ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 2x32-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CD16</td>
<td>○</td>
<td>ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL)</td>
<td>W8,10,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, 2x32-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CE16</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent</td>
<td></td>
</tr>
<tr>
<td>D016</td>
<td>○</td>
<td>ASI_FL8_PRIMARY (ASI_FL8_P)</td>
<td>RW8,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, one 8-bit floating-point load/store</td>
<td></td>
</tr>
<tr>
<td>D116</td>
<td>○</td>
<td>ASI_FL8_SECONDARY (ASI_FL8_S)</td>
<td>RW8,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, one 8-bit floating-point load/store</td>
<td></td>
</tr>
<tr>
<td>D216</td>
<td>○</td>
<td>ASI_FL16_PRIMARY (ASI_FL16_P)</td>
<td>RW8,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, one 16-bit floating-point load/store</td>
<td></td>
</tr>
<tr>
<td>D316</td>
<td>○</td>
<td>ASI_FL16_SECONDARY (ASI_FL16_S)</td>
<td>RW8,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, one 16-bit floating-point load/store</td>
<td></td>
</tr>
<tr>
<td>D416</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent</td>
<td></td>
</tr>
<tr>
<td>D816</td>
<td>○</td>
<td>ASI_FL8_PRIMARY_LITTLE (ASI_FL8_PL)</td>
<td>RW8,14 (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, one 8-bit floating point load/store, little-endian</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 10-1  UltraSPARC Architecture ASIs  (5 of 5)

<table>
<thead>
<tr>
<th>ASI</th>
<th>Value</th>
<th>req’d (❍)</th>
<th>opt’l (●)</th>
<th>ASI Name (and Abbreviation)</th>
<th>Access Type(s)</th>
<th>Virtual Address (VA)</th>
<th>V/ non-T/R</th>
<th>Shared per strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_FL8_SECONDARY_LITTLE (ASI_FL8_SL)</td>
<td>RW^8,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, one 8-bit floating point load/store, little-endian</td>
</tr>
<tr>
<td>DA_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_FL16_PRIMARY_LITTLE (ASI_FL16_PL)</td>
<td>RW^8,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, one 16-bit floating-point load/store, little-endian</td>
</tr>
<tr>
<td>DB_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_FL16_SECONDARY_LITTLE (ASI_FL16_SL)</td>
<td>RW^8,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, one 16-bit floating point load/store, little-endian</td>
</tr>
<tr>
<td>DC_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>ED_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>E0_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_BLOCK_COMMIT_PRIMARY (ASI_BLK_COMMIT_P)</td>
<td>W^8,11,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 8x8-byte block store commit operation</td>
</tr>
<tr>
<td>E1_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_BLOCK_COMMIT_SECONDARY (ASI_BLK_COMMIT_S)</td>
<td>W^8,11,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 8x8-byte block store commit operation</td>
</tr>
<tr>
<td>E2_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_TWINX_PRIMARY (ASI_TWINX_P)</td>
<td>R^19</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 128-bit atomic load twin extended word</td>
</tr>
<tr>
<td>E3_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_TWINX_SECONDARY (ASI_TWINX_S)</td>
<td>R^19</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 128-bit atomic load twin extended word</td>
</tr>
<tr>
<td>E4_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>EA_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>EB_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_TWINX_SECONDARY_LITTLE (ASI_TWINX_SL)</td>
<td>R^19</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 128-bit atomic load twin extended word, little endian</td>
</tr>
<tr>
<td>EC_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>ED_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>F0_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_BLOCK_PRIMARY (ASI_BLK_P)</td>
<td>RW^8,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 8x8-byte block load/store</td>
</tr>
<tr>
<td>F1_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_BLOCK_SECONDARY (ASI_BLK_S)</td>
<td>RW^8,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 8x8-byte block load/store</td>
</tr>
<tr>
<td>F2_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>F3_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>F4_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>F5_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>F6_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>F7_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>F8_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_BLOCK_PRIMARY_LITTLE (ASI_BLK_PL)</td>
<td>RW^8,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 8x8-byte block load/store, little endian</td>
</tr>
<tr>
<td>F9_16</td>
<td>○</td>
<td></td>
<td></td>
<td>ASI_BLOCK_SECONDARY_LITTLE (ASI_BLK_SL)</td>
<td>RW^8,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 8x8-byte block load/store, little endian</td>
</tr>
<tr>
<td>FA_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>FD_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>FE_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
<tr>
<td>FF_16</td>
<td>●</td>
<td></td>
<td></td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>Implementation dependent^1</td>
</tr>
</tbody>
</table>
† 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.

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 privilegied_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 privilegied_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, STTTWA, 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, STTTWA, 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 LDTXAx 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 any mode causes a DAE_invalid_asi exception if this ASI is not implemented by the model dependent implementation.

13 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.

14 The Queue Tail Registers (ASI 25_16) are read-only. An attempted write to the Queue Tail Registers causes a DAE_invalid_asi exception.

15 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 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 U/DMMU TTE.ie bit. If U/DMMU TTE.ie = 0, the access is big-endian; otherwise, it is little-endian.

<table>
<thead>
<tr>
<th>ASI</th>
<th>Names</th>
<th>Addressing (Context)</th>
<th>Endianness of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10_{16}$</td>
<td>$ASI_AS_IF_USER_PRIMARY$ ($ASI_AIUP$)</td>
<td>Virtual (Primary)</td>
<td>Big-endian when U/DMMU TTE.ie = 0; little-endian when U/DMMU TTE.ie = 1</td>
</tr>
<tr>
<td>$11_{16}$</td>
<td>$ASI_AS_IF_USER_SECONDARY$ ($ASI_AIUS$)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
<tr>
<td>$16_{16}$</td>
<td>$ASI_BLOCK_AS_IF_USER_PRIMARY$ ($ASI_BLK_AIUP$)</td>
<td>Virtual (Primary)</td>
<td></td>
</tr>
<tr>
<td>$17_{16}$</td>
<td>$ASI_BLOCK_AS_IF_USER_SECONDARY$ ($ASI_BLK_AIUS$)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
</tbody>
</table>

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 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

<table>
<thead>
<tr>
<th>ASI</th>
<th>Names</th>
<th>Addressing (Context)</th>
<th>Endianness of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>1816</td>
<td>ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)</td>
<td>Virtual (Primary)</td>
<td>Little-endian when U/DMMU TTE.ie = 0; big-endian when U/DMMU TTE.ie = 1</td>
</tr>
<tr>
<td>1916</td>
<td>ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
<tr>
<td>1E16</td>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUP)</td>
<td>Virtual (Primary)</td>
<td></td>
</tr>
<tr>
<td>1F16</td>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
</tbody>
</table>

### 10.4.3 ASI 14\textsubscript{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
  - During the address translation, context values are disregarded.
  - 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.

Even if data address translation is disabled, an access with this ASI is still a cacheable access.

### 10.4.4 ASI 15\textsubscript{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 instruction, in privileged mode:
  - VA is passed through to RA
  - During the address translation, context values are disregarded.
  - 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.

### 10.4.5 ASI 1C\textsubscript{16} (ASI\_REAL\_LITTLE)

ASI\_REAL\_LITTLE is a little-endian version of ASI 14\textsubscript{16} (ASI\_REAL). It operates identically to ASI\_REAL, except if an access occurs, its endianness is the opposite of that for ASI\_REAL.
10.4.6 ASI 1D16 (ASI_REAL_IO_LITTLE)

ASI_REAL_IO_LITTLE is a little-endian version of ASI 1516 (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 2216, 2316, 2716, 2A16, 2B16, 2F16 (Privileged Load Integer Twin Extended Word)

ASIs 2216, 2316, 2716, 2A16, 2B16 and 2F16 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 197). 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.

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 2416 and 2C16 used in earlier UltraSPARC implementations; see the detailed Compatibility Note on page 335 for details.

10.4.8 ASIs 2616 and 2E16 (Privileged Load Integer Twin Extended Word, Real Addressing)

ASIs 2616 and 2E16 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 197). These two ASIs are distinguished by the endianness of the access, as described in TABLE 10-5.
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 3416 and 3C16 used in earlier UltraSPARC implementations; see the Compatibility Note on page 335 for details.

## 10.4.9 ASIs E216, E316, EA16, EB16

(Nonprivileged Load Integer Twin Extended Word)

ASIs `E216`, `E316`, `EA16`, and `EB16` 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 197). These ASIs are distinguished by the address space accessed (Primary or Secondary) and the endianness of the access, as described in Table 10-6.

### Table 10-5: Load Integer Twin Extended Word (Real) ASIs

<table>
<thead>
<tr>
<th>ASI</th>
<th>Name</th>
<th>Addressing (Context)</th>
<th>Endianness of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>2616</td>
<td><code>ASI_TWINX_REAL</code></td>
<td>Real</td>
<td>Big-endian when U/DMMU TTE.ι = 0; little-endian when U/DMMU TTE.ι = 1</td>
</tr>
<tr>
<td></td>
<td>(<code>ASI_TWINX_REAL</code>)</td>
<td>(--)</td>
<td></td>
</tr>
<tr>
<td>2E16</td>
<td><code>ASI_TWINX_REAL_LITTLE</code></td>
<td>Real</td>
<td>Little-endian when U/DMMU TTE.ι = 0; big-endian when U/DMMU TTE.ι = 1</td>
</tr>
<tr>
<td></td>
<td>(<code>ASI_TWINX_REAL_L</code>)</td>
<td>(--)</td>
<td></td>
</tr>
</tbody>
</table>

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.
10.4.10 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 178 and Block Store on page 250). 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 250). 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 184.

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.

10.4.11 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 260). 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 260). 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 185.

10.4.12 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 181 and Store Floating-Point on page 253). 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, and scratchpad registers are described.

A list of UltraSPARC Architecture 2007 ASIs is shown in TABLE 10-1 on page 323.
10.5.1 Privileged Scratchpad Registers (ASI_SCRATCHPAD)

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 10-7.

### TABLE 10-7 Scratchpad Registers

<table>
<thead>
<tr>
<th>Assembly Language ASI Name</th>
<th>ASI #</th>
<th>Virtual Address</th>
<th>Privileged Scratchpad Register #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00₁₆</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>08₁₆</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10₁₆</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18₁₆</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20₁₆</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28₁₆</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30₁₆</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>38₁₆</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

**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, 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 ASI Changes in the UltraSPARC Architecture

The following Compatibility Note 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.

<table>
<thead>
<tr>
<th>ASI#</th>
<th>Previous UltraSPARC</th>
<th>UltraSPARC Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>14₁₆</td>
<td>ASI_PHYS_USE_EC</td>
<td>ASI_REAL</td>
</tr>
<tr>
<td>15₁₆</td>
<td>ASI_PHYS_BYPASS_EC_WITH_EBIT</td>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>1C₁₆</td>
<td>ASI_PHYS_USE_EC_LITTLE</td>
<td>ASI_REAL_LITTLE</td>
</tr>
<tr>
<td></td>
<td>(ASI_PHYS_USE_EC_L)</td>
<td></td>
</tr>
<tr>
<td>1D₁₆</td>
<td>ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE</td>
<td>ASI_REAL_IO_LITTLE</td>
</tr>
<tr>
<td></td>
<td>(ASI_PHY_BYPASS_EC_WITH_EBIT_L)</td>
<td></td>
</tr>
</tbody>
</table>
Compatibility Note

The names and ASI assignments (but not functions) changed between earlier UltraSPARC implementations and UltraSPARC Architecture, for the following ASIs:

<table>
<thead>
<tr>
<th>Previous UltraSPARC</th>
<th>UltraSPARC Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI#</td>
<td>Name</td>
</tr>
<tr>
<td>2416</td>
<td>ASI_NUCLEUS_QUAD_LDD</td>
</tr>
<tr>
<td>2C16</td>
<td>ASI_NUCLEUS_QUAD_LDD_LITTLE (ASI_NUCLEUS_QUAD_LDD_L)</td>
</tr>
</tbody>
</table>
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 inaccuracies 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 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 instrumentation counter 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.
Traps

A trap is a vectored transfer of control to software running in a privilege mode (see page 342) 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.

The actual transfer of control occurs through a trap table that contains the first eight instructions (32 instructions for clean_window, 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 displacement within the 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.
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 342.
- Virtual Processor States and Traps on page 343.
- Trap Categories on page 343.
- Trap Control on page 347.
- Trap-Table Entry Addresses on page 348.
- Trap Processing on page 356.
- Exception and Interrupt Descriptions on page 358.
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 (not described in this document)

The virtual processor’s operating mode is determined by the state of two mode bits, as shown in TABLE 12-1.

<table>
<thead>
<tr>
<th>PSTATE.priv</th>
<th>Virtual Processor Privilege Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nonprivileged</td>
</tr>
<tr>
<td>1</td>
<td>Privileged</td>
</tr>
</tbody>
</table>

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.

TABLE 12-4 on page 351 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 348 for details.

The maximum trap level at which privileged software may execute is \( \text{MAXPTL} \) (which, on an UltraSPARC Architecture 2007 virtual processor, is 2).

**Notes** Execution in nonprivileged mode with TL > 0 is an invalid condition that privileged 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. 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 and Traps

The value of TL affects the generated trap vector address. TL also determines where (that is, into which element of the TSTATE array) the states are saved.

12.2.0.1 Usage of Trap Levels

If MAXPTL = 2 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

<table>
<thead>
<tr>
<th>TL</th>
<th>Corresponding Execution Mode</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nonprivileged</td>
<td>Normal execution</td>
</tr>
<tr>
<td>1</td>
<td>Privileged</td>
<td>System calls; interrupt handlers; instruction emulation</td>
</tr>
<tr>
<td>2</td>
<td>Privileged</td>
<td>Window spill/fill handler</td>
</tr>
</tbody>
</table>

12.3 Trap Categories

An exception, error, or interrupt request can cause any of the following trap types:
- Precise trap
- Deferred trap
- Disrupting 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.

<table>
<thead>
<tr>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

- **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.

<table>
<thead>
<tr>
<th>SPARC V9 Compatibility Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>A restartable deferred trap is the “deferred trap” defined in the SPARC V9 specification.</td>
</tr>
</tbody>
</table>

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).

**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.

The above is summarized in **TABLE 12-3**.

**TABLE 12-3 Conditioning of Disrupting Traps**

<table>
<thead>
<tr>
<th>Type of Disrupting Trap Condition</th>
<th>Current Virtual Processor Privilege Mode</th>
<th>Disposition of Disrupting Traps, based on privilege mode in which the trap is destined to be delivered</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt_level_n</td>
<td>Nonprivileged or Privileged</td>
<td>Held pending while PSTATE.ie = 0 or interrupt level ≤ PIL</td>
</tr>
<tr>
<td>All other disrupting traps</td>
<td>Nonprivileged or Privileged</td>
<td>Held pending while PSTATE.ie = 0</td>
</tr>
</tbody>
</table>
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.

12.3.4 Uses of the Trap Categories

The SPARC V9 trap model stipulates the following:

1. 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 358 for identification of which traps are precise.
3. In an UltraSPARC Architecture implementation, all exceptions that occur as the result of program execution 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 345 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 TL register, which contains the current level of trap nesting, affects whether the trap is processed in privileged mode or hyperprivileged mode.
- PSTATE.tle determines whether implicit data accesses in the trap handler routine will be performed using big-endian or little-endian byte order.

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 356.
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.fccn, and FSR.aexc fields remain unchanged. However, if an IEEE_754_exception does not result in a trap, then the F register, FSR.fccn, and FSR.aexec 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 351 and TABLE 12-5 on page 354 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 12-2 illustrates the trap vector address for a trap delivered to privileged mode. In FIGURE 12-2, 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.

<table>
<thead>
<tr>
<th>63</th>
<th>from TBA{63:15} (TBA.tba_high49)</th>
<th>TL&gt;0</th>
<th>TT[TL]</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>5 4</td>
<td>0</td>
</tr>
</tbody>
</table>

FIGURE 12-2 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 12-3.

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 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 the trap’s destination privilege mode:

- The TBA register, \( (TL > 0) \), and \( TT[TL] \) (see Trap-Table Entry Address to Privileged Mode on page 348)

\( TT \) values \( 000_{16} - 07F_{16} \) 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.

**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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>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.</td>
</tr>
<tr>
<td>⊙</td>
<td>This trap type is associated with a feature that is architecturally defined in UltraSPARC Architecture 2007, but its implementation is optional.</td>
</tr>
<tr>
<td>P</td>
<td>Trap is taken via the Privileged trap table, in Privileged mode (PSTATE.priv = 1)</td>
</tr>
<tr>
<td>H</td>
<td>Trap is taken in Hyperprivileged mode</td>
</tr>
<tr>
<td>-x-</td>
<td>Not possible. Hardware cannot generate this trap in the indicated running mode. For example, all privileged instructions can be executed in privileged mode, therefore a privileged_opcode trap cannot occur in privileged mode.</td>
</tr>
<tr>
<td>—</td>
<td>This trap is reserved for future use.</td>
</tr>
<tr>
<td>(ie)</td>
<td>When the outstanding disrupting trap condition occurs in this privilege mode, it may be conditioned (masked out) by PSTATE.ie = 0 (but remains pending).</td>
</tr>
<tr>
<td>(nm)</td>
<td>Never Masked — when the condition occurs in this running mode, it is never masked out and the trap is always taken.</td>
</tr>
<tr>
<td>(pend)</td>
<td>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.</td>
</tr>
<tr>
<td>UA-2007</td>
<td>Exception or Interrupt Request</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>● Req’d</td>
<td>Reserved</td>
</tr>
<tr>
<td>○ Opt’l</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>● Req’d</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>●</td>
<td>IAE_privilege_violation</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>IAE_unauth_access</td>
</tr>
<tr>
<td>●</td>
<td>IAE_nfo_page</td>
</tr>
<tr>
<td>●</td>
<td>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>illegal_instruction</td>
</tr>
<tr>
<td>●</td>
<td>privileged_opcode</td>
</tr>
<tr>
<td>●</td>
<td>unimplemented_LDTW</td>
</tr>
<tr>
<td>●</td>
<td>unimplemented_STTW</td>
</tr>
<tr>
<td>●</td>
<td>DAE_invalid_asi</td>
</tr>
<tr>
<td>●</td>
<td>DAE_privilege_violation</td>
</tr>
<tr>
<td>●</td>
<td>DAE_nc_page</td>
</tr>
<tr>
<td>●</td>
<td>DAE_nfo_page</td>
</tr>
<tr>
<td>●</td>
<td>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>fp_disabled</td>
</tr>
<tr>
<td>●</td>
<td>fp_exception_ieee_754</td>
</tr>
<tr>
<td>●</td>
<td>fp_exception_other</td>
</tr>
<tr>
<td>●</td>
<td>tag_overflow&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>●</td>
<td>clean_window</td>
</tr>
<tr>
<td>●</td>
<td>division_by_zero</td>
</tr>
<tr>
<td>●</td>
<td>DAE_side_effect_page</td>
</tr>
<tr>
<td>●</td>
<td>Reserved</td>
</tr>
<tr>
<td>UA-2007</td>
<td>Exception or Interrupt Request</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>●</td>
<td>mem_address_not_aligned</td>
</tr>
<tr>
<td>●</td>
<td>LDDF_mem_address_not_aligned</td>
</tr>
<tr>
<td>●</td>
<td>STDF_mem_address_not_aligned</td>
</tr>
<tr>
<td>●</td>
<td>privileged_action</td>
</tr>
<tr>
<td>●</td>
<td>LDQF_mem_address_not_aligned</td>
</tr>
<tr>
<td>●</td>
<td>STQF_mem_address_not_aligned</td>
</tr>
<tr>
<td>●</td>
<td>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>interrupt_level_n (n = 1–15)</td>
</tr>
<tr>
<td>●</td>
<td>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>VA_watchpoint</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
</tr>
<tr>
<td>●</td>
<td>control_transfer_instruction</td>
</tr>
<tr>
<td>●</td>
<td>instruction_VA_watchpoint</td>
</tr>
<tr>
<td>●</td>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
</tr>
<tr>
<td>●</td>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
</tr>
<tr>
<td>●</td>
<td>cpu_mondo</td>
</tr>
<tr>
<td>●</td>
<td>dev_mondo</td>
</tr>
</tbody>
</table>
TABLE 12-4  Exception and Interrupt Requests, by TT Value  (3 of 3)

<table>
<thead>
<tr>
<th>Exception or Interrupt Request</th>
<th>TT (Trap Type)</th>
<th>Trap Category</th>
<th>Priority (0 = Highest)</th>
<th>Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>● resumable_error</td>
<td>07E16</td>
<td>disrupting</td>
<td>33.3</td>
<td>P (ie) P (ie)</td>
</tr>
<tr>
<td>— nonresumable_error</td>
<td>07F16</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>● spill_n_normal (n = 0–7)</td>
<td>08016–09F16</td>
<td>precise</td>
<td>9</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>● spill_n_other (n = 0–7)</td>
<td>0A016–0BF16</td>
<td>precise</td>
<td>9</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>● fill_n_normal (n = 0–7)</td>
<td>0C016–0DF16</td>
<td>precise</td>
<td>9</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>● fill_n_other (n = 0–7)</td>
<td>0E016–0FF16</td>
<td>precise</td>
<td>9</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>● trap_instruction</td>
<td>10016–17F16</td>
<td>precise</td>
<td>16.02</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>● htrap_instruction</td>
<td>18016–1FF16</td>
<td>precise</td>
<td>16.02</td>
<td>-x-</td>
</tr>
</tbody>
</table>

* Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36-V8 on page 356), including relative priorities within a given priority level.

† 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.

‡ This exception is deprecated, because the only instructions that can generate it have been deprecated.
<table>
<thead>
<tr>
<th>Exception or Interrupt Request</th>
<th>TT Type</th>
<th>Trap Category</th>
<th>Priority</th>
<th>Mode in which Trap is Delivered and (and Conditioning Applied), based on Current Privilege Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction_VA_watchpoint</td>
<td>075_16</td>
<td>precise</td>
<td>2.05</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>IAE_privilege_violation</td>
<td>008_16</td>
<td>precise</td>
<td>3.1</td>
<td>H -x-</td>
</tr>
<tr>
<td>IAE_unauth_access</td>
<td>00B_16</td>
<td>precise</td>
<td>3.2</td>
<td>H H</td>
</tr>
<tr>
<td>IAE_nfo_page</td>
<td>00C_16</td>
<td>precise</td>
<td>3.3</td>
<td>H H</td>
</tr>
<tr>
<td>illegal_instruction</td>
<td>010_16</td>
<td>precise</td>
<td>6.2</td>
<td>H H</td>
</tr>
<tr>
<td>unimplemented_LDTW</td>
<td>012_16</td>
<td>precise</td>
<td>-</td>
<td>H H</td>
</tr>
<tr>
<td>unimplemented_STTW</td>
<td>013_16</td>
<td>precise</td>
<td>6.3</td>
<td>H H</td>
</tr>
<tr>
<td>privileged_opcode</td>
<td>011_16</td>
<td>precise</td>
<td>7</td>
<td>P -x-</td>
</tr>
<tr>
<td>fp_disabled</td>
<td>020_16</td>
<td>precise</td>
<td>8</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>spill_n_normal (n = 0)–7</td>
<td>080_16–09F_16</td>
<td>precise</td>
<td>-</td>
<td>P P (nm)</td>
</tr>
<tr>
<td>spill_n_other (n = 0)–7</td>
<td>0A0_16–0BF_16</td>
<td>precise</td>
<td>9</td>
<td>P P (nm)</td>
</tr>
<tr>
<td>fill_n_normal (n = 0)–7</td>
<td>0C0_16–0DF_16</td>
<td>precise</td>
<td>-</td>
<td>P P (nm)</td>
</tr>
<tr>
<td>fill_n_other (n = 0)–7</td>
<td>0E0_16–0FF_16</td>
<td>precise</td>
<td>-</td>
<td>P P (nm)</td>
</tr>
<tr>
<td>clean_window</td>
<td>024_16–027_16</td>
<td>precise</td>
<td>-</td>
<td>P P (nm)</td>
</tr>
<tr>
<td>LDDF_mem_address_not_aligned</td>
<td>035_16</td>
<td>precise</td>
<td>10.1</td>
<td>H H</td>
</tr>
<tr>
<td>STDF_mem_address_not_aligned</td>
<td>036_16</td>
<td>precise</td>
<td>-</td>
<td>H H</td>
</tr>
<tr>
<td>LDQF_mem_address_not_aligned</td>
<td>038_16</td>
<td>precise</td>
<td>-</td>
<td>H H</td>
</tr>
<tr>
<td>STQF_mem_address_not_aligned</td>
<td>039_16</td>
<td>precise</td>
<td>-</td>
<td>H H</td>
</tr>
<tr>
<td>mem_address_not_aligned</td>
<td>034_16</td>
<td>precise</td>
<td>10.2</td>
<td>H H</td>
</tr>
<tr>
<td>fp_exception_other</td>
<td>022_16</td>
<td>precise</td>
<td>-</td>
<td>P P (nm)</td>
</tr>
<tr>
<td>fp_exception_ieee_754</td>
<td>021_16</td>
<td>precise</td>
<td>11.1</td>
<td>P P (nm)</td>
</tr>
<tr>
<td>privileged_action</td>
<td>037_16</td>
<td>precise</td>
<td>-</td>
<td>H H</td>
</tr>
<tr>
<td>control_transfer_instruction</td>
<td>074_16</td>
<td>precise</td>
<td>-</td>
<td>P H</td>
</tr>
</tbody>
</table>
### 12.5.3.1 Trap Type for Spill/Fill Traps

The trap type for window spill/fill traps is determined on the basis of the contents of the OTHERWIN and WSTATE registers as described below and shown in FIGURE 12-4.

<table>
<thead>
<tr>
<th>Bit Field Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:6 spill_or_fill</td>
<td>010₂ for spill traps; 011₂ for fill traps</td>
</tr>
<tr>
<td>5 other</td>
<td>(OTHERWIN ≠ 0)</td>
</tr>
<tr>
<td>4:2 wtype</td>
<td>If (other) then WSTATE.other; else WSTATE.normal</td>
</tr>
</tbody>
</table>
12.5.4 Trap Priorities

TABLE 12-4 on page 351 and TABLE 12-5 on page 354 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.

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), 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.

During normal operation, the virtual processor is in execute_state. It processes traps in execute_state and continues.

TABLE 12-6 describes the virtual processor mode and trap-level transitions involved in handling traps.

**TABLE 12-6** Trap Received While in execute_state

<table>
<thead>
<tr>
<th>Original State</th>
<th>New State, After Receiving Trap or Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute_state</td>
<td>execute_state</td>
</tr>
<tr>
<td>TL &lt; MAXPTL - 1</td>
<td>TL ← TL + 1</td>
</tr>
</tbody>
</table>

12.6.1 Normal Trap Processing

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.
Existing state is preserved.

\[
\begin{align*}
\text{TSTATE}[\text{TL}].gl & \leftarrow \text{GL} \\
\text{TSTATE}[\text{TL}].ccr & \leftarrow \text{CCR} \\
\text{TSTATE}[\text{TL}].asi & \leftarrow \text{ASI} \\
\text{TSTATE}[\text{TL}].pstate & \leftarrow \text{PSTATE} \\
\text{TSTATE}[\text{TL}].cwp & \leftarrow \text{CWP} \\
\text{TPC}[\text{TL}] & \leftarrow \text{PC} /\!\!/ \text{ (upper 32 bits zeroed if PSTATE.am = 1)} \\
\text{TNPC}[\text{TL}] & \leftarrow \text{NPC} /\!\!/ \text{ (upper 32 bits zeroed if PSTATE.am = 1)}
\end{align*}
\]

The trap type is preserved.

\[
\begin{align*}
\text{TT}[\text{TL}] & \leftarrow \text{the trap type}
\end{align*}
\]

The Global Level register (GL) is updated. This normally provides access to a fresh set of global registers:

\[
\begin{align*}
\text{GL} & \leftarrow \text{min} (\text{GL} + 1, \text{MAXPGL})
\end{align*}
\]

The PSTATE register is updated to a predefined state:

\[
\begin{align*}
\text{PSTATE}.mm & \text{ is unchanged} \\
\text{PSTATE}.pef & \leftarrow 1 /\!\!/ \text{ if an FPU is present, it is enabled} \\
\text{PSTATE}.am & \leftarrow 0 /\!\!/ \text{ address masking is turned off} \\
\text{PSTATE}.priv & \leftarrow 1 /\!\!/ \text{ the virtual processor enters privileged mode} \\
\text{PSTATE}.cle & \leftarrow \text{PSTATE}.tle /\!\!/ \text{set endian mode for traps}
\end{align*}
\]

Control is transferred into the trap table:

\[
\begin{align*}
& \text{// Note that at this point, TL has already been incremented (above)} \\
& \text{if ( (trap is to privileged mode) \textbf{and} (TL \leq \text{MAXPTL}) )} \\
& \text{then} \\
& \text{// the trap is handled in privileged mode} \\
& \text{// Note: The expression \text{“(TL > 1)" below evaluates to the}} \\
& \text{// value 0\textsubscript{2} if TL was 0 just before the trap (in which}} \\
& \text{// case, TL = 1 now, since it was incremented above,} \\
& \text{// during trap entry). \text{“(TL > 1)” evaluates to 1\textsubscript{2} if}} \\
& \text{// TL was > 0 before the trap.} \\
& \text{PC} \leftarrow \text{TBA}[63:15] :: (\text{TL} \geq 1) :: \text{TT}[\text{TL}] :: 0\text{00002} \\
& \text{NPC} \leftarrow \text{TBA}[63:15] :: (\text{TL} \geq 1) :: \text{TT}[\text{TL}] :: 0\text{01002}
\end{align*}
\]

For non-register-window traps, CWP is not changed.
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 = 02416−02716] (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 = 07416] (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 = 7416) 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 always 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. If it occurs in hyperprivileged mode, the trap is delivered in hyperprivileged mode.

- **cpu_mondo** [TT = 07C16] (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. When the CPU mondo queue contains a valid entry, a cpu_mondo exception is sent to the target virtual processor.
**DAE_invalid_asi** [TT = 01416] (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 333 and Partial Store ASIs on page 333). This includes the following:
  - an attempt to use a (deprecated) atomic quad load ASI (24, 2C16, 34, or 3C16) 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 LDUXA, 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, E016 or E116)
  - an attempt to write to a read-only ASI-accessible register

**DAE_nc_page** [TT = 01616] (Precise) — An access to a noncacheable page (TTE.cp = 0) 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 = 01716] (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 = 01516] (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 = 03016] (Precise) — An attempt was made to access a page which may cause side effects (TTE.e = 1) by any type of load instruction with nonfaulting ASI.

**dev_mondo** [TT = 07D16] (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 = 02816] (Precise) — An integer divide instruction attempted to divide by zero.

fill_n_normal [TT = 0C016−0DF16] (Precise)

fill_n_other [TT = 0E016−0FF16] (Precise)

A RESTORE or RETURN instruction has determined that the contents of a register window must be restored from memory.

fp_disabled [TT = 02016] (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 = 02116] (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 = 02216] (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.

htrap_instruction [TT = 18016−1FF16] (Precise) — A Tcc instruction was executed in privileged mode, the trap condition evaluated to TRUE, and the software trap number was greater than 127. The trap is delivered in hyperprivileged mode. See also trap_instruction on page 362.

IAE_nfo_page [TT = 00C16] (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 = 00816] (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 = 00B16] (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 = 01016] (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 86.)
- 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, rcond, or opf_cc.
- Illegal register alignment (such as odd rd value in a doubleword load instruction).

Programming Note: It is possible that an implementation may occasionally cause a dev_mondo interrupt when the Device Mondo queue is empty (Device Mondo Queue Head pointer = Device Mondo Queue Tail pointer). A guest operating system running in privileged mode should handle this by ignoring any Device Mondo interrupt with an empty queue.
Illegal rd value for LDXFSR, STXFSR, or the deprecated instructions LDSFR 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.

**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_VA_watchpoint** [TT = 07516] (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 hardware address. If the PC is not being translated from a virtual address (for example, the PC is being treated as a hardware 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.

**interrupt_level_n** [TT = 04116–04F16] (Disrupting) — SOFTINT{[n]} was set to 1 or an external interrupt request of level n was presented to the virtual processor and n > PIL.

**mem_address_not_aligned** [TT = 03416] (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 329.)

**nonresumable_error** [TT = 07F16] (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.

**privileged_action** [TT = 03716] (Precise) — An action defined to be privileged has been attempted while in nonprivileged mode (PSTATE.priv = 0), or an action defined to be hyperprivileged has been attempted while in nonprivileged or privileged mode. Examples:
- A data access by nonprivileged software using a restricted (privileged or hyperprivileged) ASI, that is, an ASI in the range 0016 to 7F16 (inclusively)
- A data access by nonprivileged or privileged software using a hyperprivileged ASI, that is, an ASI in the range 3016 to 7F16 (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).

**privileged_opcode** [TT = 01116] (Precise) — An attempt was made to execute a privileged instruction while in nonprivileged mode (PSTATE.priv = 0).

**resumable_error** [TT = 07E16] (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.

**spill_n_normal** [TT = 08016–09F16] (Precise)

**spill_n_other** [TT = 0A016–0BF16] (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)
- **tag\_overflow** [TT = 023\_16] (Precise) (deprecated) — 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.
- **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).
- **VA\_watchpoint** [TT = 062\_16] (Precise) — The virtual processor has detected an attempt to access (load from or store to) 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 hardware 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.

### 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:

- **implementation\_dependent\_exception\_n** [TT = 077\_16 - 07B\_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 181. (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 253. (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 348, 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:

\[ \text{CWP} \leftarrow (\text{CWP} + 2) \mod \text{N\_REG\_WINDOWS} \]

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:

\[ \text{CWP} \leftarrow (\text{CWP} + \text{CANSAVE} + 2) \mod \text{N\_REG\_WINDOWS} \]

**Implementation Note**

All spill traps can set CWP by using the calculation:

\[ \text{CWP} \leftarrow (\text{CWP} + \text{CANSAVE} + 2) \mod \text{N\_REG\_WINDOWS} \]

since CANSAVE is 0 whenever a trap occurs because of a SAVE instruction.

- On a fill trap, the window preceding CWP must be filled:
  \[ \text{CWP} \leftarrow (\text{CWP} - 1) \mod \text{N\_REG\_WINDOWS} \]

- On a clean_window trap, the window following CWP must be cleaned. Then
  \[ \text{CWP} \leftarrow (\text{CWP} + 1) \mod \text{N\_REG\_WINDOWS} \]

### 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:
  \[ \text{CLEANWIN} \leftarrow (\text{CLEANWIN} + 1) \]
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 implementation-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 366).

**Programming Note**
An interrupt request packet is sent by an interrupt source 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 365.
- **Software Interrupt Register (softint)** on page 366.
- **Interrupt Queues** on page 366.
- **Interrupt Traps** on page 368.

### 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.

See softintP Register (ASRs 20, 21, 22) on page 54 for a detailed description of the SOFTINT register.

**Programming Note** The SOFTINT register (ASR 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 366. The two mechanisms do not interact.

13.2.1 Setting the Software Interrupt Register

SOFTINT[n] is set to 1 by executing a WRSOFTINT_SETP 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.


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 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 anded with the SOFTINT register. This approach allows the interrupt handler to clear one or more bits in the SOFTINT register with a single instruction.

**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_clrP Pseudo-Register (ASR 21) on page 56 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\(^1\)
New interrupt entries are appended to the tail of a queue 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. The interrupt queue registers are accessed through ASI ASI_QUEUE (2516). The ASI and address assignments for the interrupt queue registers are provided in TABLE 13-1.

<table>
<thead>
<tr>
<th>Register</th>
<th>ASI</th>
<th>Virtual Address</th>
<th>Privileged mode Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Mondo Queue Head</td>
<td>2516 (ASI_QUEUE)</td>
<td>3C016</td>
<td>RW</td>
</tr>
<tr>
<td>CPU Mondo Queue Tail</td>
<td>2516 (ASI_QUEUE)</td>
<td>3C816</td>
<td>R or RW†</td>
</tr>
<tr>
<td>Device Mondo Queue Head</td>
<td>2516 (ASI_QUEUE)</td>
<td>3D016</td>
<td>RW</td>
</tr>
<tr>
<td>Device Mondo Queue Tail</td>
<td>2516 (ASI_QUEUE)</td>
<td>3D816</td>
<td>R or RW†</td>
</tr>
<tr>
<td>Resumable Error Queue Head</td>
<td>2516 (ASI_QUEUE)</td>
<td>3E016</td>
<td>RW</td>
</tr>
<tr>
<td>Resumable Error Queue Tail</td>
<td>2516 (ASI_QUEUE)</td>
<td>3E816</td>
<td>R or RW†</td>
</tr>
<tr>
<td>Nonresumable Error Queue Head</td>
<td>2516 (ASI_QUEUE)</td>
<td>3F016</td>
<td>RW</td>
</tr>
<tr>
<td>Nonresumable Error Queue Tail</td>
<td>2516 (ASI_QUEUE)</td>
<td>3F816</td>
<td>R or RW†</td>
</tr>
</tbody>
</table>

† see IMPL. DEP.#422-S10

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.

---

1. “mondo” is a historical term, referring to the name of the original UltraSPARC 1 bus transaction in which these interrupts were introduced.
### 13.4 Interrupt Traps

The following interrupt traps are defined in the UltraSPARC Architecture 2007: cpu_mondo, dev_mondo, resumable_error, and nonresumable_error. 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.

---

**Programming Note**

By current convention, the format of a Queue Head or Tail register is as follows:

<table>
<thead>
<tr>
<th>head/tail offset</th>
<th>000000</th>
</tr>
</thead>
</table>

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.
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.

**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 privileged software, in these sections:

- Virtual Address Translation on page 369.
- Context ID on page 372.
- TSB Translation Table Entry (TTE) on page 373.
- Translation Storage Buffer (TSB) on page 376.

### 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) bits supported is implementation dependent. A minimum of 40 bits and maximum of 56 bits can be provided for real addresses (RA). 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 hardware 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.
FIGURE 14-1  Virtual-to–Real Address Translation for 8-Kbyte, 64-Kbyte, 512-Kbyte, and 4-Mbyte Page Sizes
Privileged software manages virtual-to-real 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.

A conceptual view of privileged-mode memory management the MMU is shown in FIGURE 14-3. 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.
14.2 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.

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.

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.
The above paragraphs are summarized in TABLE 14-1.

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Privilege Mode</th>
<th>Under What Conditions each Context is Used</th>
<th>Access Type</th>
<th>Privilege Mode</th>
<th>Under What Conditions each Context is Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Primary Context</td>
<td>Secondary Context</td>
<td>Nucleus Context</td>
<td>Nonprivileged or Privileged</td>
</tr>
<tr>
<td>Instruction Access</td>
<td>Nonprivileged or Privileged</td>
<td>(when TL = 0)</td>
<td>†</td>
<td>(when TL &gt; 0)</td>
<td></td>
</tr>
<tr>
<td>Data access using implicit</td>
<td>Nonprivileged or Privileged</td>
<td>(when TL = 0)</td>
<td>†</td>
<td>(when TL &gt; 0)</td>
<td></td>
</tr>
<tr>
<td>ASI</td>
<td>Nonprivileged</td>
<td>ASI_PRIMARY*</td>
<td>ASI_SECONDARY*</td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>Data access</td>
<td>Privileged</td>
<td>ASI_PRIMARY*</td>
<td>ASI_SECONDARY*</td>
<td>ASI_NUCLEUS*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASI_AS_IF_USER_PRIMARY*</td>
<td>ASI_AS_IF_USER_SECONDARY*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, are defined. See Context ID Registers on page 380 for details.

Programming Note: Privileged software (operating systems) 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.

14.3 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 3)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag– 63:48</td>
<td>context_id</td>
<td>The 16-bit context ID associated with the TTE.</td>
</tr>
<tr>
<td>Tag– 47:42</td>
<td>—</td>
<td>These bits must be zero for a tag match.</td>
</tr>
<tr>
<td>Tag– 41:0</td>
<td>va</td>
<td>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.</td>
</tr>
<tr>
<td>Data – 63</td>
<td>v</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Data – 62</td>
<td>nfo</td>
<td>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.</td>
</tr>
<tr>
<td>Data – 61:56</td>
<td>soft2</td>
<td>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.</td>
</tr>
<tr>
<td>Data – 55:13</td>
<td>taddr</td>
<td>Target address; the underlying address (Real Address {55:13}) to which the MMU will map the page. IMPL. DEP. # 238-U3: When page offset bits for larger page sizes are stored in the TSB, 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.</td>
</tr>
</tbody>
</table>
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 377 for details.

**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 to be issued by hand-written assembler code.

(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.

**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.

**IMPL. DEP. #226-U3:** 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.

**Data – 11**  
\( \theta \)  
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 \( \theta \)-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 \( \theta \) bit causes the prefetch instruction to be treated as a nop, but does not prevent normal (hardware) instruction prefetching.

**Note 1:** The \( \theta \) 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 \( \theta \) bit is set to 1. If both the \( cp \) and \( cv \) bits are set to 1 along with the \( \theta \) bit, the result is undefined.

**Note 2:** The \( \theta \) bit and the \( nfo \) bit are mutually exclusive; both bits should never be set to 1 in any TTE.

**Data – 10**  
\( cp, cv \)  
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:

<table>
<thead>
<tr>
<th>Cacheable (cp:cv)</th>
<th>Meaning of TTE when placed in:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I-TLB (Instruction Cache PA-indexed)</td>
</tr>
<tr>
<td>00, 01</td>
<td>Noncacheable</td>
</tr>
<tr>
<td>10</td>
<td>Cacheable L2-cache, I-cache</td>
</tr>
<tr>
<td>11</td>
<td>Cacheable L2-cache, I-cache</td>
</tr>
</tbody>
</table>

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.

**TABLE 14-2**  
TSB TTE Bit Description (2 of 3)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data – 12</td>
<td>ie</td>
<td>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 377 for details.</td>
</tr>
<tr>
<td><strong>Programming Notes</strong></td>
<td></td>
<td>(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 to be issued by hand-written assembler code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(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.</td>
</tr>
<tr>
<td><strong>Implementation Note</strong></td>
<td></td>
<td>Some implementations may require cacheable accesses to pages tagged with ( TTE.ie = 1 ) to bypass the data cache, adding latency to those accesses.</td>
</tr>
<tr>
<td>Data – 11</td>
<td>( \theta )</td>
<td>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 ( \theta )-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 ( \theta ) bit causes the prefetch instruction to be treated as a nop, but does not prevent normal (hardware) instruction prefetching.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note 1:</strong> The ( \theta ) 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 ( \theta ) bit is set to 1. If both the ( cp ) and ( cv ) bits are set to 1 along with the ( \theta ) bit, the result is undefined.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note 2:</strong> The ( \theta ) bit and the ( nfo ) bit are mutually exclusive; both bits should never be set to 1 in any TTE.</td>
</tr>
<tr>
<td>Data – 10</td>
<td>( cp, cv )</td>
<td>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:</td>
</tr>
<tr>
<td>Data – 9</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>IMPL. DEP. #226-U3:</strong> 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.</td>
</tr>
</tbody>
</table>
14.4 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.

14.4.1 TSB Indexing Support

Hardware TSB indexing support via TSB pointers should be provided for the TTEs.
14.4.2 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.

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.

14.4.3 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 \( \log_2 \) of the number of TTEs in the TSB.

The TSB organization is illustrated in FIGURE 14-5. The constant \( n \) can range from 512 to an implementation-dependent number.

![FIGURE 14-5 TSB Organization](image)

**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).

14.5 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 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, according to TABLE 14-3 through TABLE 14-4.

When using the Primary Context ID, the values stored in the Primary Context IDs are used by the Data (or Unified) MMU. The Secondary Context ID is never used for instruction accesses.
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 TTE “invert endianness” bit (ie). The TTE 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, 

```c
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 TTE.ie bit.

**TABLE 14-3** ASI Mapping for Instruction Access

<table>
<thead>
<tr>
<th>Mode</th>
<th>TL</th>
<th>PSTATE.cle</th>
<th>Endianness</th>
<th>ASI Used</th>
<th>Resulting Address Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonprivileged</td>
<td>0</td>
<td>—</td>
<td>Big</td>
<td>ASI_PRIMARY</td>
<td>VA</td>
</tr>
<tr>
<td>Privileged</td>
<td>0</td>
<td>—</td>
<td>Big</td>
<td>ASI_PRIMARY</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td>1–2</td>
<td>—</td>
<td>Big</td>
<td>ASI_NUCLEUS</td>
<td>VA</td>
</tr>
</tbody>
</table>

**TABLE 14-4** ASI Mapping for Data Accesses  
(1 of 2)

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Privilege Mode</th>
<th>TL</th>
<th>PSTATE.cle</th>
<th>TTE.ie</th>
<th>Endianness</th>
<th>ASI Used</th>
<th>Resulting Address Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load, Store, Atomic Load-Store, or Prefetch with implicit ASI</td>
<td>NP</td>
<td>0^1</td>
<td>0</td>
<td>0</td>
<td>Big</td>
<td>ASI_PRIMARY</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Little</td>
<td>ASI_PRIMARY</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Little</td>
<td>ASI_PRIMARY_LITTLE</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Big</td>
<td>ASI_PRIMARY_LITTLE</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1–21</td>
<td>0</td>
<td>0</td>
<td>Big</td>
<td>ASI_NUCLEUS</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1–21</td>
<td>0</td>
<td>1</td>
<td>Little</td>
<td>ASI_NUCLEUS</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1–21</td>
<td>1</td>
<td>0</td>
<td>Little</td>
<td>ASI_NUCLEUS_LITTLE</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1–21</td>
<td>1</td>
<td>1</td>
<td>Big</td>
<td>ASI_NUCLEUS_LITTLE</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td>NP</td>
<td>0^2</td>
<td>any</td>
<td>0</td>
<td>Big^2</td>
<td>Explicitly specified in instruction</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>any</td>
<td>1</td>
<td>Little^1</td>
<td>Explicitly specified in instruction</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>0–21</td>
<td>any</td>
<td>0</td>
<td>Big^1</td>
<td>ASA<em>_REAL</em> ASI</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0–21</td>
<td>any</td>
<td>1</td>
<td>Little</td>
<td>ASA<em>_REAL</em> ASI</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0–21</td>
<td>any</td>
<td>any</td>
<td>Big</td>
<td>Nontranslating ASIs</td>
<td>any</td>
</tr>
</tbody>
</table>
The Context ID used by the data and instruction MMUs is determined according to TABLE 14-5. 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 14-5 IMMU, DMMU and UMMU Context ID Usage

<table>
<thead>
<tr>
<th>ASI Value</th>
<th>Context ID Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_<em>NUCLEUS</em> (any ASI name containing the string “NUCLEUS”)</td>
<td>Nucleus (0000_16, hard-wired)</td>
</tr>
<tr>
<td>ASI_<em>PRIMARY</em> (any ASI name containing the string “PRIMARY”)</td>
<td>All Primary Context IDs</td>
</tr>
<tr>
<td>ASI_<em>SECONDARY</em> (any ASI name containing the string “SECONDARY”)</td>
<td>All Secondary Context IDs</td>
</tr>
<tr>
<td>All other ASI values</td>
<td>(Not applicable; no translation)</td>
</tr>
</tbody>
</table>

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 14-6 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 14-6  MMU SPARC V9 Appendix F.3.2 Protection Mode Compliance

<table>
<thead>
<tr>
<th>Condition</th>
<th>Resultant Protection Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes No Yes 0 0</td>
<td>Read-only¹</td>
</tr>
<tr>
<td>No Yes N/A 1 N/A</td>
<td>Execute-only¹</td>
</tr>
<tr>
<td>Yes No Yes 0 1</td>
<td>Read/Write¹</td>
</tr>
<tr>
<td>Yes Yes Yes 1 0</td>
<td>Read-only/Execute</td>
</tr>
<tr>
<td>Yes Yes Yes 1 1</td>
<td>Read/Write/Execute</td>
</tr>
<tr>
<td>No No No N/A N/A</td>
<td>No Access</td>
</tr>
</tbody>
</table>

¹ These protection modes are optional, according to SPARC V9.

#### 14.6.1 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 14-7 lists the MMU registers and provides references to sections with more details.

### TABLE 14-7  MMU Internal Registers and ASI Operations

<table>
<thead>
<tr>
<th>IMMU ASI</th>
<th>D/UMMU ASI</th>
<th>VA(63:0)</th>
<th>Access</th>
<th>Register or Operation Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>21₁₆</td>
<td>8₁₆</td>
<td>RW</td>
<td>Primary Context ID 0 register</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>21₁₆</td>
<td>10₁₆</td>
<td>RW</td>
<td>Secondary Context ID 0 register</td>
</tr>
<tr>
<td>21₁₆</td>
<td>108₁₆</td>
<td>RW</td>
<td>Primary Context ID 1 register</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>21₁₆</td>
<td>110₁₆</td>
<td>RW</td>
<td>Secondary Context ID 1 register</td>
</tr>
</tbody>
</table>

#### 14.6.2 Context ID Registers

The MMU architecture supports multiple primary and secondary context IDs. The address assignment of the context IDs is shown in TABLE 14-8.

### TABLE 14-8  Context ID ASI Assignments

<table>
<thead>
<tr>
<th>Register</th>
<th>ASI</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Context ID 0</td>
<td>21₁₆</td>
<td>008₁₆</td>
</tr>
<tr>
<td>Primary Context ID 1</td>
<td>21₁₆</td>
<td>108₁₆</td>
</tr>
<tr>
<td>Secondary Context ID 0</td>
<td>21₁₆</td>
<td>010₁₆</td>
</tr>
<tr>
<td>Secondary Context ID 1</td>
<td>21₁₆</td>
<td>110₁₆</td>
</tr>
</tbody>
</table>
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 exception for such an access.

The Primary Context ID register is illustrated in FIGURE 14-6, where pcontext is the context ID for the primary address space.

![Primary Context ID](image)

FIGURE 14-6 IMMU, DMMU, and UMMU Primary Context ID

The Secondary Context ID register is illustrated in FIGURE 14-7, where scontextid is the context ID for the secondary address space.

![Secondary Context ID](image)

FIGURE 14-7 D/UMMU Secondary Context ID

The Nucleus Context ID register is hardwired to zero, as illustrated in FIGURE 14-6.

![Nucleus Context ID](image)

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.

---

**Programming Note**

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 exception for such an access.

The Primary Context ID register is illustrated in FIGURE 14-6, where pcontext is the context ID for the primary address space.
 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 87. 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.

<table>
<thead>
<tr>
<th>Shading</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An attempt to execute opcode will cause an illegal_instruction exception.</td>
</tr>
</tbody>
</table>

An attempt to execute a reserved opcode behaves as defined in Reserved Opcodes and Instruction Fields on page 86.

**TABLE A-1**  
<table>
<thead>
<tr>
<th>op[1:0]</th>
<th>op (1:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Branches and SETHI (See TABLE A-2)</td>
<td>CALL</td>
</tr>
</tbody>
</table>

**TABLE A-2**  
<table>
<thead>
<tr>
<th>op2[2:0] (op = 0)</th>
<th>op2 (2:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ILLTRAP</td>
<td>BPcc (See TABLE A-7)</td>
</tr>
</tbody>
</table>

1. See the footnote regarding bit 28 on page 109.
2. rd = 0, imm22 = 0
<table>
<thead>
<tr>
<th>op3[5:4]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>ADDcc</td>
<td>TADDcc</td>
<td>WRY(^{13}) (rd = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRCCR (rd = 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRASI (rd = 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 4, 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 15, rs1 = 0, i = 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 15) and (rs1 ≠ 0 or i ≠ 1))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 7 – 14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRFPRS (rd = 6)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRa_{PA}SR (7 ≤ rd ≤ 14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRPRP (rd = 16)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRPIC (rd = 17)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 18)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRCGR (rd = 19)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRSOFTINT_SET(P) (rd = 20)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRSOFTINT_CLR(P) (rd = 21)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRSOFT(P) (rd = 22)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRTICK_CMPR(P) (rd = 23)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WRSTICK_CMPR(P) (rd = 25)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 26)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 27)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 28 – 31)</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>ANDcc</td>
<td>TSUBcc</td>
<td>SAVED(P) (fcn = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RESTORED(P) (fcn = 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ALLCLEAN(P) (fcn = 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OTHERWP (fcn = 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NORMALWP (fcn = 4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INVALWP (fcn = 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (fcn ≥ 6)</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>ORcc</td>
<td>TADDecTV(P)</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>ORcc</td>
<td>TADDecTV(P)</td>
<td>WRPR(P) (rd = 0-14 or 16)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rd = 15 or 17–31)</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>XORcc</td>
<td>TSUBccTV(P)</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>SUBcc</td>
<td>MUL(Scc)(^{D})</td>
<td>FPop1 (See TABLE A-5)</td>
<td></td>
</tr>
<tr>
<td>ANDN</td>
<td>ANDNcc</td>
<td>SLL (x = 0), SLLX (x = 1)</td>
<td>FPop2 (See TABLE A-6)</td>
<td></td>
</tr>
<tr>
<td>ORN</td>
<td>ORNcc</td>
<td>SRL (x = 0), SRLX (x = 1)</td>
<td>(VIS) (See TABLE A-12)</td>
<td></td>
</tr>
<tr>
<td>XNOR</td>
<td>XNORcc</td>
<td>SRA (x = 0), SRAX (x = 1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE A-3  \( \text{op3}[5:0] \) (op = 102)  (2 of 2)

<table>
<thead>
<tr>
<th>( \text{op3}[5:4] )</th>
<th>( \text{op3}[3:0] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td><strong>B</strong></td>
</tr>
<tr>
<td>UMUL(^D)</td>
<td>SMUL(^D)</td>
</tr>
<tr>
<td>UMULcc(^D)</td>
<td>SMULcc(^D)</td>
</tr>
<tr>
<td>RDPCR(^P)</td>
<td>FLUSHW</td>
</tr>
<tr>
<td>RDPR(^P)</td>
<td>—</td>
</tr>
<tr>
<td>(rs1 = 1 – 14 or 16)</td>
<td>—</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td><strong>F</strong></td>
</tr>
<tr>
<td>UDIV(^D)</td>
<td>SDIV(^D)</td>
</tr>
<tr>
<td>UDIVcc(^D)</td>
<td>SDIVcc(^D)</td>
</tr>
<tr>
<td>POPC (rs1 = 0)</td>
<td>—</td>
</tr>
<tr>
<td>— (rs1 &gt; 0)</td>
<td>—</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td><strong>J</strong></td>
</tr>
<tr>
<td>SDIV(^D)</td>
<td>SDIVcc(^D)</td>
</tr>
<tr>
<td><strong>M</strong></td>
<td><strong>N</strong></td>
</tr>
<tr>
<td>ADDC</td>
<td>ADDCcc</td>
</tr>
<tr>
<td>— (rs1 = 1, i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDCCCR (rs1 = 2, i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDASI (rs1 = 3, i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDTC(^D) (rs1 = 4, i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDPC (rs1 = 5, i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDFPRS (rs1 = 6, i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDs(^A) (7 \leq rd \leq 14, i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>MEMBAR (rs1 = 15, rd = 0, i = 1, instruction bit 12 = 0)</td>
<td>—</td>
</tr>
<tr>
<td>— (rs1 = 15, rd = 0, i = 1, instruction bit 12 = 1)</td>
<td>—</td>
</tr>
<tr>
<td>— (i = 1, (rs1 \neq 15 or rd \neq 0))</td>
<td>—</td>
</tr>
<tr>
<td>— (rs1 = 15, rd = 0, i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>— (rs1 = 15 AND rd &gt; 0 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDPC(^P) (rs1 = 16 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDPC(^P) (rs1 = 17 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>— (rs1 = 18 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDGSR (rs1 = 19 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>— (rs1 = 20 or 21 AND (i = 0))</td>
<td>—</td>
</tr>
<tr>
<td>RDSOFTINT(^D) (rs1 = 22 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDTC(^D) (rs1 = 23 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDSTICK (rs1 = 24 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>RDSTICK(^P) (rs1 = 25 AND i = 0)</td>
<td>—</td>
</tr>
<tr>
<td>— (rs1 = 26 AND (i = 0))</td>
<td>—</td>
</tr>
<tr>
<td>— (rs1 = 27 – 31 AND (i = 0))</td>
<td>—</td>
</tr>
<tr>
<td><strong>J</strong></td>
<td><strong>K</strong></td>
</tr>
<tr>
<td>SMULD</td>
<td>SMULcc(^D)</td>
</tr>
<tr>
<td>SMULcc(^D)</td>
<td>—</td>
</tr>
<tr>
<td><strong>O</strong></td>
<td><strong>P</strong></td>
</tr>
<tr>
<td>SUBC</td>
<td>SUBCc</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>R</strong></td>
<td><strong>S</strong></td>
</tr>
<tr>
<td>UDIVX</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td><strong>T</strong></td>
</tr>
<tr>
<td>SDIVD</td>
<td>SDIVcc(^D)</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
### TABLE A-4  \ op3[5:0] (op = 112)

<table>
<thead>
<tr>
<th>op3 (3:0)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUW</td>
<td>LDUWA\textsuperscript{PASI}</td>
<td>LDF</td>
<td>LDFA\textsuperscript{PASI}</td>
<td></td>
</tr>
<tr>
<td>LDUB</td>
<td>LDUBA\textsuperscript{PASI}</td>
<td>(rd = 0) LDFSR\textsuperscript{D}</td>
<td>(rd = 1) LDXFSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(rd &gt; 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDUH</td>
<td>LDUHA\textsuperscript{PASI}</td>
<td>LDQF</td>
<td>LDQFA\textsuperscript{PASI}</td>
<td></td>
</tr>
<tr>
<td>LDTWD</td>
<td>LDTWA\textsuperscript{D, PASI}</td>
<td>LDDF</td>
<td>LDDFA\textsuperscript{PASI}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>— (rd odd)</td>
<td>LDTX A</td>
<td>LDBLOCKF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>— (rd odd)</td>
<td></td>
<td>LDBLOCKF</td>
<td></td>
</tr>
<tr>
<td>STW</td>
<td>STWA\textsuperscript{PASI}</td>
<td>STF</td>
<td>STFA\textsuperscript{PASI}</td>
<td></td>
</tr>
<tr>
<td>STB</td>
<td>STBA\textsuperscript{PASI}</td>
<td>STFSR\textsuperscript{D}, STXFSR</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(rd &gt; 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STH</td>
<td>STHA\textsuperscript{PASI}</td>
<td>STQF</td>
<td>STQFA\textsuperscript{PASI}</td>
<td></td>
</tr>
<tr>
<td>LDTWD</td>
<td>LDTWA\textsuperscript{D, PASI}</td>
<td>STDF</td>
<td>STDEA\textsuperscript{PASI}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>— (rd odd)</td>
<td></td>
<td>STLBLOCKF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>— (rd odd)</td>
<td></td>
<td>STSHORTF</td>
<td></td>
</tr>
<tr>
<td>LDSW</td>
<td>LDSWA\textsuperscript{PASI}</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>LDSB</td>
<td>LDSBA\textsuperscript{PASI}</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>LDSH</td>
<td>LDSHA\textsuperscript{PASI}</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>LDX</td>
<td>LDXAPASI</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Reserved</td>
<td>Reserved</td>
<td>CASA\textsuperscript{PASI}</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>LDSTUB</td>
<td>LDSTUBA\textsuperscript{PASI}</td>
<td>PREFETCH</td>
<td>PREFETCHA\textsuperscript{PASI}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— (fen = 5 \text{-} 15)</td>
<td>— (fen = 5 \text{-} 15)</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>STX</td>
<td>STXA\textsuperscript{PASI}</td>
<td>Reserved</td>
<td>CASX\textsuperscript{PASI}</td>
</tr>
<tr>
<td>F</td>
<td>SWAP\textsuperscript{D}</td>
<td>SWAPA\textsuperscript{D, PASI}</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>opf{8:4}</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>----------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>00_{16}</td>
<td>—</td>
<td>FMOVs</td>
<td>FMOVd</td>
<td>FMOVq</td>
</tr>
<tr>
<td>01_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>02_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>03_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>04_{16}</td>
<td>—</td>
<td>FADDs</td>
<td>FADDd</td>
<td>FADDq</td>
</tr>
<tr>
<td>05_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>06_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>07_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>08_{16}</td>
<td>—</td>
<td>FtOx</td>
<td>FdTx</td>
<td>FqTx</td>
</tr>
<tr>
<td>09_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0A_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0B_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0C_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0D_{16}</td>
<td>—</td>
<td>FtOi</td>
<td>FdToi</td>
<td>FqToi</td>
</tr>
<tr>
<td>0E_{16}–1F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00_{16}</td>
<td>—</td>
<td>FABSs</td>
<td>FABSD</td>
<td>FABSQ</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>01_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>02_{16}</td>
<td>—</td>
<td>FSQRTs</td>
<td>FSQRTd</td>
<td>FSQRTq</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>03_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>04_{16}</td>
<td>—</td>
<td>FMULs</td>
<td>FMULD</td>
<td>FMULq</td>
<td>—</td>
<td>FDIVs</td>
<td>FDIVd</td>
</tr>
<tr>
<td>05_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>06_{16}</td>
<td>—</td>
<td>FsMULd</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>07_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>08_{16}</td>
<td>FsToD</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>FsToq</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>09_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0A_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0B_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0C_{16}</td>
<td>FtOD</td>
<td>FtOD</td>
<td>—</td>
<td>FtOq</td>
<td>FtOD</td>
<td>FtOD</td>
<td>FtOD</td>
</tr>
<tr>
<td>0D_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0E_{16}–1F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
**TABLE A-6**  \( \text{opf}[8:0] \)  \( \text{op = 10}_2, \text{op3 = 35}_{16} = \text{FPop2} \)

| \text{opf}(8:4) | 0016 | 0116 | 0216 | 0316 | 0416 | 0516 | 0616 | 0716 | 0816 | 0916 | 0A16 | 0B16 | 0C16 | 0D16 | 0E16 | 0F16 | 1016 | 1116-1716 | 1816 | 1916-1F16 |
|-----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| \text{opf}(3:0) | FMOVs \( \text{fcc0} \) | FMOVd \( \text{fcc0} \) | FMOVq \( \text{fcc0} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 1               | FMOVs \( \text{fcc1} \) | FMOVd \( \text{fcc1} \) | FMOVq \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc0} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| 0               | FCMPls \( \text{fcc1} \) | FCMPls \( \text{fcc1} \) | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |

† Reserved variation of FMOVR
‡ bit 13 of instruction = 0
**TABLE A-7**  \(\text{cond}[3:0]\)

<table>
<thead>
<tr>
<th>(\text{cond})</th>
<th>(\text{BPcc})</th>
<th>(\text{Bicc})</th>
<th>(\text{FBPcc})</th>
<th>(\text{FBfcc}^D)</th>
<th>(\text{Tcc})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>BPN</td>
<td>BN(^D)</td>
<td>FBPN</td>
<td>FBN(^D)</td>
<td>TN</td>
</tr>
<tr>
<td>(1)</td>
<td>BPE</td>
<td>BE(^D)</td>
<td>FBPNE</td>
<td>FBNE(^D)</td>
<td>TE</td>
</tr>
<tr>
<td>(2)</td>
<td>BLE</td>
<td>BLE(^D)</td>
<td>FBPLG</td>
<td>FBLG(^D)</td>
<td>TLE</td>
</tr>
<tr>
<td>(3)</td>
<td>BPL</td>
<td>BL(^D)</td>
<td>FBPUL</td>
<td>FBUL(^D)</td>
<td>TL</td>
</tr>
<tr>
<td>(4)</td>
<td>BPLEU</td>
<td>BLEU(^D)</td>
<td>FBPL</td>
<td>FBL(^D)</td>
<td>TLEU</td>
</tr>
<tr>
<td>(5)</td>
<td>BPCS</td>
<td>BCS(^D)</td>
<td>FBPUG</td>
<td>FBUG(^D)</td>
<td>TCS</td>
</tr>
<tr>
<td>(6)</td>
<td>BPNEG</td>
<td>BNEG(^D)</td>
<td>FBPG</td>
<td>FBC(^D)</td>
<td>TNEG</td>
</tr>
<tr>
<td>(7)</td>
<td>BPVS</td>
<td>BV(^D)</td>
<td>FBPU</td>
<td>FBU(^D)</td>
<td>TVS</td>
</tr>
<tr>
<td>(8)</td>
<td>BPA</td>
<td>BA(^D)</td>
<td>FBPA</td>
<td>FBA(^D)</td>
<td>TA</td>
</tr>
<tr>
<td>(9)</td>
<td>BPNE</td>
<td>BNE(^D)</td>
<td>FBPE</td>
<td>FBE(^D)</td>
<td>TNE</td>
</tr>
<tr>
<td>(A)</td>
<td>BPG</td>
<td>B(^G)</td>
<td>FBPU</td>
<td>FBUED(^D)</td>
<td>TG</td>
</tr>
<tr>
<td>(B)</td>
<td>BPGE</td>
<td>BGE(^D)</td>
<td>FBPG</td>
<td>FBGE(^D)</td>
<td>TGE</td>
</tr>
<tr>
<td>(C)</td>
<td>BPGU</td>
<td>B(^G)</td>
<td>FBPG(^E)</td>
<td>FBUGE(^D)</td>
<td>TGU</td>
</tr>
<tr>
<td>(D)</td>
<td>BPCC</td>
<td>BCC(^D)</td>
<td>FBPLE</td>
<td>FBE(^D)</td>
<td>TCC</td>
</tr>
<tr>
<td>(E)</td>
<td>BPPOS</td>
<td>BPOS(^D)</td>
<td>FBPULE</td>
<td>FBUED(^D)</td>
<td>TPOS</td>
</tr>
<tr>
<td>(F)</td>
<td>BPVC</td>
<td>BVC(^D)</td>
<td>FBPO</td>
<td>FBO(^D)</td>
<td>TVC</td>
</tr>
</tbody>
</table>

**TABLE A-8**  Encoding of \(r\text{cond}[2:0]\) Instruction Field

<table>
<thead>
<tr>
<th>(\text{rcond})</th>
<th>(\text{BPr})</th>
<th>(\text{MOVR})</th>
<th>(\text{FMOVR})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(1)</td>
<td>BRZ</td>
<td>MOVNZ</td>
<td>FMOVR&lt;\text{s}</td>
</tr>
<tr>
<td>(2)</td>
<td>BRLEZ</td>
<td>MOVRLLEZ</td>
<td>FMOVR&lt;\text{s}</td>
</tr>
<tr>
<td>(3)</td>
<td>BRLZ</td>
<td>MOVRLZ</td>
<td>FMOVR&lt;\text{s}</td>
</tr>
<tr>
<td>(4)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(5)</td>
<td>BRNZ</td>
<td>MOVNZ</td>
<td>FMOVR&lt;\text{s}</td>
</tr>
<tr>
<td>(6)</td>
<td>BRGZ</td>
<td>MOVRGZ</td>
<td>FMOVR&lt;\text{s}</td>
</tr>
<tr>
<td>(7)</td>
<td>BRGEZ</td>
<td>MOVGEZ</td>
<td>FMOVR&lt;\text{s}</td>
</tr>
</tbody>
</table>

**TABLE A-9**  \(cc / \text{opf\_cc}\) Fields (MOVcc and FMOVcc)

<table>
<thead>
<tr>
<th>\text{opf_cc}</th>
<th>(\text{Condition Code Selected})</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc2 cc1 cc0</td>
<td>fcc0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>fcc0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>fcc1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>fcc2</td>
</tr>
<tr>
<td>0 1 1</td>
<td>fcc3</td>
</tr>
<tr>
<td>1 0 0</td>
<td>icc</td>
</tr>
</tbody>
</table>
### TABLE A-9 cc / opf_cc Fields (MOVcc and FMOVcc)

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc0</th>
<th>Condition Code Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>xcc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

### TABLE A-10 cc Fields (FBPfcc, FCMP, and FCMPE)

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc0</th>
<th>Condition Code Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>fcc0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>fcc1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>fcc2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>fcc3</td>
</tr>
</tbody>
</table>

### TABLE A-11 cc Fields (BPcc and Tcc)

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc0</th>
<th>Condition Code Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>icc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>xcc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>
### TABLE A-12 \( \text{opf}[8:0] \) for VIS opcodes (\( \text{op} = 10_2, \text{op3} = 36_{16} \))

<table>
<thead>
<tr>
<th>( \text{opf}[8:4] )</th>
<th>( \text{opf}[3:0] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{EDGE8cc} )</td>
<td>ARRAY8</td>
</tr>
<tr>
<td>( \text{FCMPE16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FPADD16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FZERO} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FAND} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE8cc} )</td>
<td>ARRAY8</td>
</tr>
<tr>
<td>( \text{FCMPE16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FPADD16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FZERO} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FAND} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE8N} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FMUL} )</td>
<td>( 8x16 )</td>
</tr>
<tr>
<td>( \text{FPADD16S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FZEROS} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FANDS} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{SIAM} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE8Lcc} )</td>
<td>ARRAY16</td>
</tr>
<tr>
<td>( \text{FCMPNE16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FPADD32} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FNOR} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FXNOR} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE8LN} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FMUL} )</td>
<td>( 8x16\text{AU} )</td>
</tr>
<tr>
<td>( \text{FPADD32S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FNORS} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FXNORS} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE16cc} )</td>
<td>ARRAY32</td>
</tr>
<tr>
<td>( \text{FCMPE32} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FPSUB16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FANDNOT2} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FSRC1} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE16N} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FMUL} )</td>
<td>( 8x16\text{AL} )</td>
</tr>
<tr>
<td>( \text{FPADD16S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FANDNOT2S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FSRC1S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE16Lcc} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FMUL} )</td>
<td>( 8\text{SU}_x16 )</td>
</tr>
<tr>
<td>( \text{FPSUB32} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FNOT2} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FORNOT2} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE16LN} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FMUL} )</td>
<td>( 8\text{UL}_x16 )</td>
</tr>
<tr>
<td>( \text{FPSUB32S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FNOT2S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FORNOT2S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE32cc} )</td>
<td>( \text{ALIGN ADDRESS} )</td>
</tr>
<tr>
<td>( \text{FCMPGT16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FALIGN DATA} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FANDNOT1} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FSRC2} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE32N} )</td>
<td>( \text{BMASK} )</td>
</tr>
<tr>
<td>( \text{FMUL} )</td>
<td>( 8\text{UL}_x16 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FANDNOT1S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FSRC2S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE32Lcc} )</td>
<td>( \text{ALIGN ADDRESS, LITTLE} )</td>
</tr>
<tr>
<td>( \text{FCMPEQ16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FPACK32} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FNOT1} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FORNOT1} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{EDGE32LN} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FPACK16} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FNOT1S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FORNOT1S} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{C} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FCMPGT32} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{BSHUFFLE} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FXOR} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FOR} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{D} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FPACKFIX} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FEXPAND} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FXORS} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FORS} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{E} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FCMPEQ32} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{PDIST} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FNAND} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{FONE} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \text{F} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>
TABLE A-14 opf[8:0] for VIS opcodes (op = 102, op3 = 3616) (3 of 3)

<table>
<thead>
<tr>
<th>opf (3:0)</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18–1F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>7</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>A</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>D</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>E</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>F</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Reserved

opf (8:4)
**TABLE A-13** \( \text{op5[3:0]} \) \( \text{op} = 10_2, \text{op3} = 37_{16} = \text{FMAf} \\

<table>
<thead>
<tr>
<th>\text{op5[3:2]}</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{op5[1:0]}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>—</td>
<td>FMADDs</td>
<td>FMADDd</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>FMSUBs</td>
<td>FMSUBd</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>FNMSUBs</td>
<td>FNMSUBd</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>FNMADDs</td>
<td>FNMADDd</td>
<td>—</td>
</tr>
</tbody>
</table>
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 397.

The appendix contains these sections:
- Definition of an Implementation Dependency on page 395.
- Hardware Characteristics on page 396.
- Implementation Dependency Categories on page 396.
- List of Implementation Dependencies on page 397.

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 16.

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>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-V8</td>
<td>f</td>
<td>Software emulation of instructions</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whether an instruction complies with UltraSPARC Architecture 2007 by being implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>2-V8</td>
<td>v</td>
<td>Number of IU registers</td>
<td>17, 34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 MAXPGL + 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).</td>
<td></td>
</tr>
<tr>
<td>3-V8</td>
<td>f</td>
<td>Incorrect IEEE Std 754-1985 results</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>4, 5</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>6-V8</td>
<td>f</td>
<td>I/O registers privileged status</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whether I/O registers can be accessed by nonprivileged code is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>7-V8</td>
<td>t</td>
<td>I/O register definitions</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents and addresses of I/O registers are implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>8-V8</td>
<td>t</td>
<td>RDasr/WRasr target registers</td>
<td>20, 48, 225, 285</td>
</tr>
<tr>
<td>Cs20</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>9-V8</td>
<td>f</td>
<td>RDasr/WRasr privileged status</td>
<td>20, 48, 225, 285</td>
</tr>
<tr>
<td>Cs20</td>
<td></td>
<td>The privilege level required to execute each of the implementation-dependent read/write ancillary state register instructions (for ASRs 28–31) is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>10-V8–12-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>13-V8</td>
<td>a</td>
<td>(this implementation dependency applies to execution modes with greater privileges)</td>
<td></td>
</tr>
<tr>
<td>14-V8–15-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>16-V8-Cu3</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>17-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
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. 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 46). Whether the generated condition(s) trigger an $fp\textunderscore exception\_ieee\_754$ exception or not depends on the setting of $FSR.tem$.

**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 $fp\textunderscore exception\_ieee\_754$ exception or not depends on the setting of $FSR.tem$.

**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 $fp\textunderscore exception\_ieee\_754$ exception or not depends on the setting of $FSR.tem$.

**FPU version, FSR.ver**

Bits 19:17 of the FSR, FSR.ver, identify one or more implementations of the FPU architecture.

**Reserved.**

An UltraSPARC Architecture implementation implements the tem, cexc, and aexc fields in hardware, conformant to IEEE Std 754-1985.

Reserved.

Reserved.

Reserved.

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.

Reserved.

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 322 and Block Load and Store ASIs on page 333 for details.

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.
### APPENDIX B • Implementation Dependencies

#### TABLE B-1 SPARC V9 Implementation Dependencies (3 of 7)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-</td>
<td>f</td>
<td>This implementation dependency is no longer used in the UltraSPARC Architecture, since “catastrophic” errors are now handled using normal error-reporting mechanisms.</td>
<td>—</td>
</tr>
<tr>
<td>32-</td>
<td>t</td>
<td>Restartable deferred traps</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>V8 Ms10</td>
<td>Whether any restartable deferred traps (and associated deferred-trap queues) are present is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>33-</td>
<td>f</td>
<td>Trap precision</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>V8 Cs10</td>
<td>In an UltraSPARC Architecture implementation, all exceptions that occur as the result of program execution are precise.</td>
<td></td>
</tr>
<tr>
<td>34-V8</td>
<td>f</td>
<td>Interrupt clearing</td>
<td>366</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>The method by which an interrupt is removed is now defined in the UltraSPARC Architecture (see Clearing the Software Interrupt Register on page 366).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>How quickly a virtual processor responds to an interrupt request, like all timing-related issues, is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>35-</td>
<td>t</td>
<td>Implementation-dependent traps</td>
<td>349</td>
</tr>
<tr>
<td></td>
<td>V8 Cs20</td>
<td>Trap type (TT) values 060h–07Fh were reserved for implementation_dependent_exception exceptions in SPARC V9 but are now all defined as standard UltraSPARC Architecture exceptions.</td>
<td></td>
</tr>
<tr>
<td>36-V8</td>
<td>f</td>
<td>Trap priorities</td>
<td>356</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>41-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>42-</td>
<td>t, f, v</td>
<td>FLUSH instruction</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>V8 Cs10</td>
<td>FLUSH is implemented in hardware in all UltraSPARC Architecture 2007 implementations, so never causes a trap as an unimplemented instruction.</td>
<td></td>
</tr>
<tr>
<td>43-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>44-</td>
<td>f</td>
<td>Data access FPU trap</td>
<td>182, 199</td>
</tr>
<tr>
<td></td>
<td>V8 Cs10</td>
<td>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.</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>45-V8–46-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>47-</td>
<td>t</td>
<td>RDAsr</td>
<td>226</td>
</tr>
<tr>
<td></td>
<td>V8 Cs20</td>
<td>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:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the interpretation of bits 13:0 and 29:25 in the instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• whether an attempt to execute the instruction causes an illegal_instruction exception</td>
<td></td>
</tr>
<tr>
<td>48-</td>
<td>t</td>
<td>WRAsr</td>
<td>286</td>
</tr>
<tr>
<td></td>
<td>V8 Cs20</td>
<td>WRAsr instructions with rd of 16-18, 28, 29, or 31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRAsr instruction using one of those rd values, the following are implementation dependent:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the interpretation of bits 18:0 in the instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the operation(s) performed (for example, xor) to generate the value written to the ASR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• whether an attempt to execute the instruction causes an illegal_instruction exception</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE B-1 SPARC V9 Implementation Dependencies (4 of 7)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>49-V8–54-V8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55-V8</td>
<td>f</td>
<td>Tininess detection</td>
<td>48</td>
</tr>
<tr>
<td>V8</td>
<td>Cs10</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>56–100</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101-V9</td>
<td>v</td>
<td>Maximum trap level ((MAXPTL))</td>
<td>68, 70</td>
</tr>
<tr>
<td>V9</td>
<td>CS10</td>
<td>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). In a typical implementation (MAXPTL = MAXPGL) (see impl. dep. #401-S10). Architecturally, (MAXPTL) must be (\geq 2).</td>
<td></td>
</tr>
<tr>
<td>102-V9</td>
<td>f</td>
<td>Clean windows trap</td>
<td>358</td>
</tr>
<tr>
<td>V9</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>103-V9</td>
<td>f</td>
<td>Prefetch instructions</td>
<td></td>
</tr>
<tr>
<td>V9</td>
<td>Ms10</td>
<td>The following aspects of the PREFETCH and PREFETCHA instructions are implementation dependent:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(a): the attributes of the block of memory prefetched: its size (minimum = 64 bytes) and its alignment (minimum = 64-byte alignment)</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(b): whether each defined prefetch variant is implemented (1) as a NOP, (2) with its full semantics, or (3) with common-case prefetching semantics</td>
<td>220, 222</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(c): whether and how variants 16, 18, 19 and 24–31 are implemented; if not implemented, a variant must execute as a NOP</td>
<td>224C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The following aspects of the PREFETCH and PREFETCHA instructions used to be (but are no longer) implementation dependent:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(d): while in nonprivileged mode ((PSTATE.priv = 0)), an attempt to reference an ASI in the range (0_{16..7F16}) by a PREFETCHA instruction executes as a NOP; specifically, it does not cause a privileged_action exception.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(e): PREFETCH and PREFETCHA have no observable effect in privileged code</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(g): while in privileged mode ((PSTATE.priv = 1)), an attempt to reference an ASI in the range (30_{16..7F16}) by a PREFETCHA instruction executes as a NOP (specifically, it does not cause a privileged_action exception)</td>
<td></td>
</tr>
<tr>
<td>105-V9</td>
<td>f</td>
<td>TICK register</td>
<td>52</td>
</tr>
<tr>
<td>V9</td>
<td></td>
<td>(a): If an accurate count cannot always be returned when TICK is read, any inaccuracy should be small, bounded, and documented.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(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.</td>
<td></td>
</tr>
<tr>
<td>106-V9cS</td>
<td>f</td>
<td>IMPDEP2A instructions</td>
<td></td>
</tr>
<tr>
<td>V9cS</td>
<td>10</td>
<td>The IMPDEP2A instructions were defined to be completely implementation dependent in SPARC V9. The opcodes that have not been used in this space are now just documented as reserved opcodes.</td>
<td></td>
</tr>
<tr>
<td>107-V9</td>
<td>f</td>
<td>Unimplemented LDTW(A) trap</td>
<td>192</td>
</tr>
<tr>
<td>V9</td>
<td></td>
<td>(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.</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(b): It is implementation dependent whether LDTW is implemented in hardware. If not, an attempt to execute an LDTWA instruction will cause an unimplemented_LDTW exception.</td>
<td></td>
</tr>
</tbody>
</table>
## Unimplemented STTW(A) trap

- **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.
- **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.

## 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 \((i \neq 1 \text{ or instruction bits } 12:5 = 0)\) 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. (In an UltraSPARC Architecture processor, the *LDDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDDF instruction)

- **b:** LDDFA requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid \((i \neq 1 \text{ or instruction bits } 12:5 = 0)\) 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. (In an UltraSPARC Architecture processor, the *LDDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDDFA instruction)

## 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 \((i = 1 \text{ or instruction bits } 12:5 = 0)\) 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. (In an UltraSPARC Architecture processor, the *STDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STDF instruction)

- **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 \((i = 1 \text{ or instruction bits } 12:5 = 0)\) 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. (In an UltraSPARC Architecture processor, the *STDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STDFA instruction)
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.
(In an UltraSPARC Architecture processor, the LDQF_mem_address_not_aligned exception occurs in this case and trap handler software emulates the LDQF instruction)
(this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the LDQF instruction in hardware)

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.
(In an UltraSPARC Architecture processor, the LDQF_mem_address_not_aligned exception occurs in this case and trap handler software emulates the LDQF instruction)
(this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the LDQF instruction in hardware)

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.
(In an UltraSPARC Architecture processor, the STQF_mem_address_not_aligned exception occurs in this case and trap handler software emulates the STQF instruction)
(this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the STQF instruction in hardware)

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.
(In an UltraSPARC Architecture processor, the STQF_mem_address_not_aligned exception occurs in this case and trap handler software emulates the STQFA instruction)
(this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the STQFA instruction in hardware)

Implemented memory models
Whether memory models represented by PSTATE_mm = 102 or 112 are supported in an UltraSPARC Architecture processor is implementation dependent. If the 102 model is supported, then when PSTATE_mm = 102 the implementation must correctly execute software that adheres to the RMO model described in The SPARC Architecture Manual - Version 9. If the 112 model is supported, its definition is implementation dependent.

Identifying I/O locations
The manner in which I/O locations are identified is implementation dependent.

Unimplemented values for PSTATE_mm
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.
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>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>200–201</td>
<td>Reserved.</td>
<td>—</td>
</tr>
<tr>
<td>203-U3-Cs10</td>
<td>Dispatch Control register (DCR) bits 13:6 and 1</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>204-U3-Cs10</td>
<td>DCR bits 5:3 and 0</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>205-U3-Cs10</td>
<td>Instruction Trap Register</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>206-U3-Cs10</td>
<td>SHUTDOWN instruction</td>
<td>—</td>
</tr>
<tr>
<td>208-U3</td>
<td>Ordering of errors captured in instruction execution</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>The order in which errors are captured in instruction execution is implementation dependent. Ordering may be in program order or in order of detection.</td>
<td>—</td>
</tr>
<tr>
<td>209-U3</td>
<td>Software intervention after instruction-induced error</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Precision of the trap to signal an instruction-induced error of which recovery requires software intervention is implementation dependent.</td>
<td>—</td>
</tr>
<tr>
<td>211-U3</td>
<td>Error logging registers’ information</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>The information that the error logging registers preserves beyond the reset induced by an ERROR signal is implementation dependent.</td>
<td>—</td>
</tr>
<tr>
<td>212-U3-Cs10</td>
<td>Trap with fatal error</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>213-U3</td>
<td>AFSR_.priv</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td>—</td>
</tr>
<tr>
<td>226-U3</td>
<td>TTE support for cv bit</td>
<td>375</td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td>—</td>
</tr>
<tr>
<td>227-U3</td>
<td>TSB number of entries</td>
<td>377</td>
</tr>
<tr>
<td></td>
<td>The maximum number of entries in a TSB is implementation dependent in the UltraSPARC Architecture (to a maximum of 16 million).</td>
<td>—</td>
</tr>
<tr>
<td>228-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>229-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005. TSB</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Base address generation</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Whether the implementation generates the TSB Base address by exclusive-ORing 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.</td>
<td>—</td>
</tr>
<tr>
<td>230</td>
<td>Reserved.</td>
<td>—</td>
</tr>
<tr>
<td>230-U3-Cs20</td>
<td>This implementation dependency no longer applies, in UltraSPARC Architecture 2007</td>
<td>—</td>
</tr>
<tr>
<td>232-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>233-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>235-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
</tbody>
</table>
APPENDIX B • Implementation Dependencies

TABLE B-2  UltraSPARC Architecture Implementation Dependencies (2 of 5)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>236-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>239-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>240-U3-Cs10</td>
<td>Reserved.</td>
<td>—</td>
</tr>
<tr>
<td>243-U3</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
</tbody>
</table>
| 244-U3-Cs10 | Data Watchpoint Reliability
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. | —    |
| 248-U3 | Conditions for fp_exception_other with unfinished_FPop
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. | 45    |
| 249-U3-Cs10 | Data Watchpoint for Partial Store Instruction
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[rn2] 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. | 262   |
| 250-U3-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2007. | —    |
| 251 | Reserved.                                                                  | —    |
| 252-U3-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005. | —    |
| 253-U3-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005. | —    |
| 255-U3-Cs10 | LDDFA with ASI E016 or E116 and misaligned destination register number
If an LDDFA opcode is used with an ASI of E016 or E116 (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. | 185   |
| 256-U3 | LDDFA with ASI E016 or E116 and misaligned memory address
If an LDDFA opcode is used with an ASI of E016 or E116 (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. | 185   |
| 257-U3 | LDDFA with ASI C016–C516 or C816–CD16 and misaligned memory address
If an LDDFA opcode is used with an ASI of C016–C516 or C816–CD16 (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. | 185   |
| 259–299 | Reserved.                                                                  | —    |
TABLE B-2 UltraSPARC Architecture Implementation Dependencies (3 of 5)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>300-U4-Cs10</td>
<td>Attempted access to ASI registers with LDTWA</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>If an LDTWA instruction referencing a non-memory ASI is executed, it generates a DAE_invalid_asi exception.</td>
<td></td>
</tr>
<tr>
<td>301-U4-Cs10</td>
<td>Attempted access to ASI registers with STTWA</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>If an STTWA instruction referencing a non-memory ASI is executed, it generates a DAE_invalid_asi exception.</td>
<td></td>
</tr>
<tr>
<td>302-U4-Cs10</td>
<td>Scratchpad registers</td>
<td>334</td>
</tr>
<tr>
<td></td>
<td>An UltraSPARC Architecture processor includes eight privileged Scratchpad registers (64 bits each, read/write accessible).</td>
<td></td>
</tr>
<tr>
<td>303-U4-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>305-U4-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>306-U4-Cs10</td>
<td>Trap type generated upon attempted access to noncacheable page with LDTXA</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>When an LDTXA instruction attempts access from an address that is not mapped to cacheable memory space, a DAE_nc_page exception is generated.</td>
<td></td>
</tr>
<tr>
<td>307-U4-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>308-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>309-U4-Cs10</td>
<td>Reserved.</td>
<td>—</td>
</tr>
<tr>
<td>310-U4-Cs10</td>
<td>Large page sizes</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>311–319</td>
<td>Reserved.</td>
<td>—</td>
</tr>
<tr>
<td>327–399</td>
<td>Reserved</td>
<td>—</td>
</tr>
<tr>
<td>400-S10</td>
<td>Global Level register (GL) implementation</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>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 MAXPGL for that implementation. If any bits of GL are not implemented, they read as zero and writes to them are ignored.</td>
<td></td>
</tr>
<tr>
<td>401-S10</td>
<td>Maximum Global Level (MAXPGL)</td>
<td>68, 70</td>
</tr>
<tr>
<td></td>
<td>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). In a typical implementation MAXPGL = MAXPTL (see impl. dep. #101-V9-CS10). Architecturally, MAXPTL must be ≥ 2.</td>
<td></td>
</tr>
<tr>
<td>403-S10</td>
<td>Setting of “dirty” bits in FPRS</td>
<td>53, 53</td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>404-S10</td>
<td>Scratchpad registers 4 through 7</td>
<td>334</td>
</tr>
<tr>
<td></td>
<td>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, or (3) inaccessible (cause a DAE_invalid_asi exception).</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX B • Implementation Dependencies

TABLE B-2 UltraSPARC Architecture Implementation Dependencies (4 of 5)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>405-S10</td>
<td>Virtual address range</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>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)</td>
<td></td>
</tr>
</tbody>
</table>

| 409-S10 | FLUSH instruction and memory consistency                                  | 134  |
|        | 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. |      |

| 410-S10 | Block Load behavior                                                       | 179  |
|        | 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 the MMU ignores the side-effect bit (TTE.e) for LDBLOCKF accesses (in which case, LDBLOCKFs behave as if TTE.e = 0) | 307  |
|        | • Whether VA_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 | 179, 180 |

| 411-S10 | Block Store behavior                                                     | 251, 252 |
|        | The following aspects of the behavior of block store (STBLOCKF) instructions are implementation dependent: |      |
|        | • The memory ordering model that STBLOCKF follows (other than as constrained by the rules outlined on page 251). |      |
|        | • Whether VA_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. |      |
|        | • Whether STBLOCKFs to non-cacheable pages execute in strict program order or not. If not, a STBLOCKF to a non-cacheable page causes a DAE_nc_page 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). |      |
|        | • Whether the MMU ignores the side-effect bit (TTE.e) for STBLOCKF accesses (in which case, STBLOCKFs behave as if TTE.e = 0) | 307  |
|        | • Any other restrictions on the behavior of STBLOCKF, as described in implementation-specific documentation. |      |

| 412-S10 | MEMBAR behavior                                                          | 202  |
|        | An UltraSPARC Architecture implementation may define the operation of each MEMBAR variant in any manner that provides the required semantics. |      |
It is implementation dependent whether VA_watchpoint exceptions are recognized on accesses to all 16 bytes of a LTDXA instruction (the recommended behavior) or only on accesses to the first 8 bytes.

Reserved.

The size of context ID fields in MMU context registers is implementation-dependent and may range from 13 to 16 bits.

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 (zeros) the more-significant 32 bits of the values it places into PC and NPC.

If an accurate count cannot always be returned when STICK is read, any inaccuracy should be small, bounded, and documented. 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 high-order bits not implemented must read as 0.

Availability of the control_transfer_instruction exception feature is implementation dependent. If not implemented, trap type 07416 is unused, PSTATE.tct always reads as zero, and writes to PSTATE.tct are ignored.

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)

The number of real address (RA) bits supported is implementation dependent. A minimum of 40 bits and maximum of 56 bits can be provided for real addresses (RA). See implementation-specific documentation for details.

Unified vs. Split Instruction and Data MMUs

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.

Reserved for UltraSPARC Architecture 2007

Reserved for future use
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 409.
- Syntax Design on page 414.
- Synthetic Instructions on page 414.

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, rs0 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-%r7 (out registers; same as %r8-%r15)
%10-%i7 (local registers; same as %r16-%r23)
%0-%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:

---

\(^1\) In actual usage, the %sp, %fp, %gn, %on, %ln, and %in forms are preferred over %rn.
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 $%f62$)
- $%d0$, $%d2$, $%d4$, ..., $%d60$, $%d62$ ($%dn$, where $n \mod 2 = 0$, only)
- $%q0$, $%q4$, $%q8$, ..., $%q56$, $%q60$ ($%qn$, where $n \mod 4 = 0$, only)

See Floating-Point Registers on page 38 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)

i_or_x_cc. An i_or_x_cc specifies a set of integer condition codes, those based on either the 32-bit result of an operation (icc) or on the full 64-bit result (xcc). It may have either of the following values:

- $%icc$
- $%xcc$

fccn. An fccn specifies a set of floating-point condition codes. It can have any of the following values:

- $%fcc0$
- $%fcc1$
- $%fcc2$
- $%fcc3$

### 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)
%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_cmp† System Timer Compare (STICK_CMP) register
%tba Trap Base Address (TBA) register
%tick Cycle count (TICK) register
%tick_cmp Timer Compare (TICK_CMP) 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_cmp were, respectively, %sys_tick and %sys_tick_cmp, which are now deprecated. Over time, assemblers will support the new %stick and %stick_cmp names for these registers (which are consistent with %tick and %tick_cmp). 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)

<table>
<thead>
<tr>
<th>Value Name in Assembly Language</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>for PREFETCH instruction “fcn” field</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#n_reads</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>#one_read</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>#n_writes</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>#one_write</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>#page</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>#unified</td>
<td>17 (11_{16})</td>
<td></td>
</tr>
<tr>
<td>#n_reads_strong</td>
<td>20 (14_{16})</td>
<td></td>
</tr>
</tbody>
</table>
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_{rs}1` (equivalent to `reg_{rs}1 + %g0`)
- `reg_{rs}1 + simm13`
\[ \text{reg}_{rs1} - \text{simm13} \]
\[ \text{simm13} \quad \text{(equivalent to} \%g0 + \text{simm13)} \]
\[ \text{simm13} + \text{reg}_{rs1} \quad \text{(equivalent to} \text{reg}_{rs1} + \text{simm13)} \]

**address**
Can be any of the following:
\[ \text{reg}_{rs1} \quad \text{(equivalent to} \text{reg}_{rs1} + \%g0) \]
\[ \text{reg}_{rs1} + \text{simm13} \]
\[ \text{reg}_{rs1} - \text{simm13} \]
\[ \text{simm13} \quad \text{(equivalent to} \%g0 + \text{simm13)} \]
\[ \text{simm13} + \text{reg}_{rs1} \quad \text{(equivalent to} \text{reg}_{rs1} + \text{simm13)} \]
\[ \text{reg}_{rs1} + \text{reg}_{rs2} \]

**membar_mask**
Is the following:
\[ \text{const7} \]
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 202 for a complete list of mnemonics).

**prefetch_fcn (prefetch function)**
Can be any of the following:
\[ \text{0–31} \]
Predefined constants (the values of which fall in the 0-31 range) useful as \text{prefetch_fcn} values can be found in TABLE C-1 on page 411.

**regaddr (register-only address)**
Can be any of the following:
\[ \text{reg}_{rs1} \quad \text{(equivalent to} \text{reg}_{rs1} + \%g0) \]
\[ \text{reg}_{rs1} + \text{reg}_{rs2} \]

**reg_or_imm** *(register or immediate value)*
Can be either of:
\[ \text{reg}_{rs2} \]
\[ \text{simm13} \]

**reg_or_imm5** *(register or immediate value)*
Can be either of:
\[ \text{reg}_{rs2} \]
\[ \text{simm5} \]

**reg_or_imm10** *(register or immediate value)*
Can be either of:
\[ \text{reg}_{rs2} \]
\[ \text{simm10} \]

**reg_or_imm11** *(register or immediate value)*
Can be either of:
\[ \text{reg}_{rs2} \]
\[ \text{simm11} \]

**reg_or_shcnt** *(register or shift count value)*
Can be any of:
\[ \text{reg}_{rs2} \]
software_trap_number Can be any of the following:

- \( r_{\text{rs1}} \) (equivalent to \( r_{\text{rs1}} + %g0 \))
- \( r_{\text{rs1}} + r_{\text{rs2}} \)
- \( r_{\text{rs1}} + \text{simm8} \)
- \( r_{\text{rs1}} - \text{simm8} \)
- \( \text{simm8} \) (equivalent to \( %g0 + \text{simm8} \))
- \( \text{simm8} + r_{\text{rs1}} \) (equivalent to \( r_{\text{rs1}} + \text{simm8} \))

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 JMP, CALL, or SETHI) is specified directly, without square brackets.

The follow additional syntax constraints have been adopted for UltraSPARC Architecture:

- Instruction mnemonics should be limited to a maximum of 15 characters.

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>
<thead>
<tr>
<th>Synthetic Instruction</th>
<th>SPARC V9 Instruction(s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>( r_{\text{rs1}}, \text{reg_or_imm} ) subcc ( r_{\text{rs1}}, \text{reg_or_imm}, %g0 )</td>
<td>Compare.</td>
</tr>
<tr>
<td>jmp</td>
<td>address</td>
<td>jmp1 address, %g0</td>
</tr>
<tr>
<td>call</td>
<td>address</td>
<td>jmp1 address, %o7</td>
</tr>
</tbody>
</table>
### TABLE C-2: Mapping Synthetic to SPARC V9 Instructions (2 of 3)

<table>
<thead>
<tr>
<th>Synthetic Instruction</th>
<th>SPARC V9 Instruction(s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>iprefetch label</td>
<td>bn,a,pt \xcc,label</td>
<td>Originally envisioned as an encoding for an &quot;instruction prefetch&quot; operation, but functions as a NOP on all UltraSPARC Architecture implementations. (See PREFETCH function 17 on page 219 for an alternative method of prefetching instructions.)</td>
</tr>
<tr>
<td>tst</td>
<td>orcc %g0, %rg, %g0</td>
<td>Test.</td>
</tr>
<tr>
<td>ret</td>
<td>jmpl %i7+8, %g0</td>
<td>Return from subroutine.</td>
</tr>
<tr>
<td>retl</td>
<td>jmpl %o7+8, %g0</td>
<td>Return from leaf subroutine.</td>
</tr>
<tr>
<td>restore</td>
<td>restore %g0, %g0, %g0</td>
<td>Trivial RESTORE.</td>
</tr>
<tr>
<td>save</td>
<td>save %g0, %g0, %g0</td>
<td>Trivial SAVE. (Warning: trivial SAVE should only be used in kernel code!)</td>
</tr>
<tr>
<td>setuw value, %rg</td>
<td>sethi %hi(value), %rg</td>
<td>(When ((value &amp; 3FF16) == 0).)</td>
</tr>
<tr>
<td></td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%g0, value, %rg</td>
<td>(When 0 ≤ value ≤ 4095).</td>
</tr>
<tr>
<td></td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sethi %hi(value), %rg</td>
<td>(Otherwise)</td>
</tr>
<tr>
<td>setw value, %rg</td>
<td>sethi %hi(value), %rg</td>
<td>Warning: do not use setuw in the delay slot of a DCTI.</td>
</tr>
<tr>
<td></td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%g0, value, %rg</td>
<td>(When 4096 ≤ value ≤ 4095).</td>
</tr>
<tr>
<td></td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sethi %hi(value), %rg</td>
<td>(Otherwise, if (value &lt; 0) and ((value &amp; 3FF16) = = 0))</td>
</tr>
<tr>
<td>setx value, %rg, %rg</td>
<td>sethi %hh(value), %rg</td>
<td>Create 64-bit constant.</td>
</tr>
<tr>
<td></td>
<td>or %rg, %hh(value), %rg</td>
<td>(&quot;%reg&quot; is used as a temporary register.)</td>
</tr>
<tr>
<td></td>
<td>sllx %rg, 32, %rg</td>
<td>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.</td>
</tr>
<tr>
<td>signx %rg, %rg</td>
<td>sra %rg, %g0, %rg</td>
<td>Sign-extend 32-bit value to 64 bits.</td>
</tr>
</tbody>
</table>
### TABLE C-2: Mapping Synthetic to SPARC V9 Instructions (3 of 3)

<table>
<thead>
<tr>
<th>Synthetic Instruction</th>
<th>SPARC V9 Instruction(s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>not ®rd, ®rd</td>
<td>xnor ®rd, %g0, ®rd</td>
<td>One's complement.</td>
</tr>
<tr>
<td>not ®rd</td>
<td>xnor ®rd, %g0, ®rd</td>
<td>One's complement.</td>
</tr>
<tr>
<td>neg ®rd, ®rd</td>
<td>sub ®rd, %g0, ®rd</td>
<td>Two's complement.</td>
</tr>
<tr>
<td>neg ®rd</td>
<td>sub ®rd, %g0, ®rd</td>
<td>Two's complement.</td>
</tr>
<tr>
<td>cas [ ®rd1, ®rd2, ®rd ]</td>
<td>casa [ ®rd1, &amp;ASI_P, ®rd2, ®rd ]</td>
<td>Compare and swap.</td>
</tr>
<tr>
<td>casl [ ®rd1, ®rd2, ®rd ]</td>
<td>casla [ ®rd1, &amp;ASI_P_L, ®rd2, ®rd ]</td>
<td>Compare and swap, little-endian.</td>
</tr>
<tr>
<td>casx [ ®rd1, ®rd2, ®rd ]</td>
<td>casxa [ ®rd1, &amp;ASI_P, ®rd2, ®rd ]</td>
<td>Compare and swap extended.</td>
</tr>
<tr>
<td>casxl [ ®rd1, ®rd2, ®rd ]</td>
<td>casxla [ ®rd1, &amp;ASI_P_L, ®rd2, ®rd ]</td>
<td>Compare and swap extended, little-endian.</td>
</tr>
<tr>
<td>incr ®rd</td>
<td>add ®rd, 1, ®rd</td>
<td>Increment by 1.</td>
</tr>
<tr>
<td>incr const13, ®rd</td>
<td>add ®rd, const13, ®rd</td>
<td>Increment by const13.</td>
</tr>
<tr>
<td>inc ®rd</td>
<td>addc ®rd, 1, ®rd</td>
<td>Increment by 1; set &amp;icc &amp; &amp;xcc.</td>
</tr>
<tr>
<td>inc ®rd</td>
<td>addc ®rd, 1, ®rd</td>
<td>Increment by 1; set &amp;icc &amp; &amp;xcc.</td>
</tr>
<tr>
<td>dec ®rd</td>
<td>sub ®rd, 1, ®rd</td>
<td>Decrement by 1.</td>
</tr>
<tr>
<td>dec ®rd</td>
<td>sub ®rd, const13, ®rd</td>
<td>Decrement by const13.</td>
</tr>
<tr>
<td>dec ®rd</td>
<td>sub ®rd, const13, ®rd</td>
<td>Decrement by const13.</td>
</tr>
<tr>
<td>dec ®rd</td>
<td>sub ®rd, const13, ®rd</td>
<td>Decrement by const13.</td>
</tr>
<tr>
<td>bcle &amp;reg_or_imm, ®rd</td>
<td>and ®rd, &amp;reg_or_imm, %g0</td>
<td>Decr by const13; set &amp;icc &amp; &amp;xcc.</td>
</tr>
<tr>
<td>bcle &amp;reg_or_imm, ®rd</td>
<td>or ®rd, &amp;reg_or_imm, ®rd</td>
<td>Decr by const13; set &amp;icc &amp; &amp;xcc.</td>
</tr>
<tr>
<td>btst &amp;reg_or_imm, ®rd</td>
<td>andcc ®rd, &amp;reg_or_imm, %g0</td>
<td>Bit test.</td>
</tr>
<tr>
<td>btst &amp;reg_or_imm, ®rd</td>
<td>or ®rd, &amp;reg_or_imm, ®rd</td>
<td>Bit test.</td>
</tr>
<tr>
<td>btst &amp;reg_or_imm, ®rd</td>
<td>or ®rd, &amp;reg_or_imm, ®rd</td>
<td>Bit test.</td>
</tr>
<tr>
<td>clr ®rd</td>
<td>or ®rd, %g0, ®rd</td>
<td>Bit clear.</td>
</tr>
<tr>
<td>clr ®rd</td>
<td>or ®rd, %g0, ®rd</td>
<td>Bit clear.</td>
</tr>
<tr>
<td>clrb [ &amp;address ]</td>
<td>stb ®rd, %g0, [ &amp;address ]</td>
<td>Clear (zero) register.</td>
</tr>
<tr>
<td>clrb [ &amp;address ]</td>
<td>stb ®rd, %g0, [ &amp;address ]</td>
<td>Clear byte.</td>
</tr>
<tr>
<td>clrh [ &amp;address ]</td>
<td>sth ®rd, %g0, [ &amp;address ]</td>
<td>Clear half-word.</td>
</tr>
<tr>
<td>clrh [ &amp;address ]</td>
<td>sth ®rd, %g0, [ &amp;address ]</td>
<td>Clear word.</td>
</tr>
<tr>
<td>clrx [ &amp;address ]</td>
<td>stx ®rd, %g0, [ &amp;address ]</td>
<td>Clear extended word.</td>
</tr>
<tr>
<td>clruw ®rd, ®rd</td>
<td>srl ®rd, %g0, ®rd</td>
<td>Copy and clear upper word.</td>
</tr>
<tr>
<td>clruw ®rd, ®rd</td>
<td>srl ®rd, %g0, ®rd</td>
<td>Clear upper word.</td>
</tr>
<tr>
<td>mov &amp;reg_or_imm, ®rd</td>
<td>or ®rd, &amp;reg_or_imm, ®rd</td>
<td></td>
</tr>
<tr>
<td>mov &amp;reg_or_imm, ®rd</td>
<td>or ®rd, &amp;reg_or_imm, ®rd</td>
<td></td>
</tr>
<tr>
<td>mov &amp;reg_or_imm, %y</td>
<td>wr ®rd, %g0, &amp;reg_or_imm, %y</td>
<td></td>
</tr>
<tr>
<td>mov &amp;reg_or_imm, %asrn</td>
<td>wr ®rd, %g0, &amp;reg_or_imm, %asrn</td>
<td></td>
</tr>
</tbody>
</table>

416 UltraSPARC Architecture 2007 • Draft D0.9.3b, 20 Oct 2009
Index

A
a (annul) instruction field
branch instructions, 104, 107, 109, 122, 125
accesses
  cacheable, 306
  I/O, 307
  restricted ASI, 309
  with side effects, 307, 314
accrued exception (aexc) field of FSR register, 46, 348, 398
ADD instruction, 97
ADDC instruction, 97
ADDCc instruction, 97, 245
ADDCccc instruction, 97
address
  operand syntax, 413
  space identifier (ASI), 321
address mask (am) field of PSTATE register
    description, 66
address space, 5, 13
address space identifier (ASI), 5, 305
  accessing MMU registers, 380
  appended to memory address, 17, 71
  architecturally specified, 309
  changed in, 335
  changed in UA
    ASI_REAL, 334
    ASI_REAL_IO, 334
    ASI_REAL_IO_LITTLE, 334
    ASI_REAL_LITTLE, 334
    ASI_TWIX_N, 335
    ASI_TWIX_NL, 335
    ASI_TWIX_NUCLEUS_LITTLE, 335
  definition, 5
  encoding address space information, 73
  explicit, 76
  explicitly specified in instruction, 77
  implicit, See implicit ASIs
nontranslating, 8, 195, 268
nontranslating ASI, 322
  with prefetch instructions, 220
real ASI, 322
  restricted, 309, 321
    privileged, 310
  restriction indicator, 51
SPARC V9 address, 308
  translating ASI, 322
    unrestricted, 310, 321
address space identifier (ASI) register
  for load/store alternate instructions, 51
  address for explicit ASI, 76
  and LDDA instruction, 183, 194
  and LDTSTUBA instruction, 191
  load integer from alternate space instructions, 176
  with prefetch instructions, 220
  for register-immediate addressing, 309
  restoring saved state, 114, 233
  saving state, 341
  and STDAA instruction, 267
store floating-point into alternate space instructions, 255
  store integer to alternate space instructions, 248
  and SWAPA instruction, 272
  after trap, 21
  and TSTATE register, 63
  and write state register instructions, 286
addressing modes, 14
ADDX instruction (SPARC V8), 97
ADDCx instruction (SPARC V8), 97
alias
  floating-point registers, 38
aliased, 5
ALIGNADDRESS instruction, 98
ALIGNADDRESS_LITTLE instruction, 98
alignment
  data (load/store), 18, 73, 308
  doubleword, 18, 73, 308
  extended-word, 73
  halfword, 18, 73, 308
  instructions, 18, 73, 308
  integer registers, 185, 192, 195
  memory, 308, 361
  quadword, 18, 73, 308
  word, 18, 73, 308
ALLCLEAN instruction, 99
alternate space instructions, 19, 51
ancillary state registers (ASRs)
  access, 48
  assembly language syntax, 410
  I/O register access, 19
  possible registers included, 226, 286
  privileged, 20, 397
reading/writing implementation-dependent processor
registers, 20, 397
writing to, 286
AND instruction, 100
ANDcc instruction, 100
ANDN instruction, 100
ANDNcc instruction, 100
annul bit
in branch instructions, 109
in conditional branches, 123
annulled branches, 109
application program, 5, 48
architectural direction note, 4
architecture, meaning for SPARC V9, 13
arithmetic overflow, 51
ARRAY16 instruction, 101
ARRAY32 instruction, 101
ARRAY8 instruction, 101
ASI, invalid, and DAE_invalid_asi, 359
ASI register, 49
ASI, See address space identifier (ASI)
ASI_AIUP, 323, 329
ASI_AIUPL, 323, 330
ASI_AIUS, 323, 329
ASI_AIUS_L, 197
ASI_AIUSL, 323, 330
ASI_AS_IF_USER*, 67, 373
ASI_AS_IF_USER_NONFAULT_LITTLE, 310
ASI_AS_IF_USER_PRIMARY, 323, 329, 359, 372
ASI_AS_IF_USER_PRIMARY_LITTLE, 310, 323, 330, 359
ASI_AS_IF_USER_SECONDARY, 310, 323, 329, 359, 372
ASI_AS_IF_USER_SECONDARY_LITTLE, 310, 323, 330, 359
ASI_ASS_IF_USER_SECONDARY_NONFAULT_LITTLE, 310
ASI_BLK_AIUP, 323, 329
ASI_BLK_AIUPL, 323, 330
ASI_BLK_AIUS, 323, 329
ASI_BLK_COMMIT_P, 327
ASI_BLK_COMMIT_S, 327
ASI_BLK_P, 327
ASI_BLK_PL, 327
ASI_BLK_S, 327
ASI_BLK_SL, 327
ASI_BLOCK_AS_IF_USER_PRIMARY, 323, 329
ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE, 323, 330
ASI_BLOCK_AS_IF_USER_SECONDARY, 323, 329
ASI_BLOCK_COMMIT_PRIMARY, 327
ASI_BLOCK_COMMIT_SECONDARY, 327
ASI_BLOCK_PRIMARY, 327
ASI_BLOCK_PRIMARY_LITTLE, 327
ASI_BLOCK_SECONDARY, 327
ASI_BLOCK_SECONDARY_LITTLE, 327
ASI_F16_P, 326
ASI_F16_PL, 327
ASI_F16_PRIMARY, 326
ASI_F16_PRIMARY_LITTLE, 327
ASI_F16_S, 326
ASI_F16_SECONDARY, 326
ASI_F16_SECONDARY_LITTLE, 327
ASI_FL8_PRIMARY, 326
ASI_FL8_PRIMARY_LITTLE, 326
ASI_FL8_S, 326
ASI_FL8_SECONDARY, 326
ASI_FL8_SECONDARY_LITTLE, 327
ASI_F18_SL, 327
ASI_MMU_CONTEXTID, 324
ASI_N, 323
ASI_NL, 323
ASI_NUCLEUS, 76, 77, 323, 372
ASI_NUCLEUS_LITTLE, 77, 323
ASI_NUCLEUS_QUAD_LDD (deprecated), 335
ASI_NUCLEUS_QUAD_LDD_L (deprecated), 335
ASI_NUCLEUS_QUAD_LDD_LITTLE (deprecated), 335
ASI_P, 325
ASI_PHY_BYPASS_EC_WITH_EBIT_L, 334
ASI_PHYS_BYPASS_EC_WITH_EBIT, 334
ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE, 334
ASI_PHYS_USE_EC, 334
ASI_PHYS_USE_EC_L, 334
ASI_PHYS_USE_EC_LITTLE, 334
ASI_PN, 325
ASI_PNF, 325
ASI_PNFL, 325
ASI_PRIMARY, 76, 309, 310, 325
ASI_PRIMARY_L, 77, 309, 325, 378
ASI_PRIMARY_NO_FAULT, 307, 319, 325
ASI_PRIMARY_NO_FAULT_LITTLE, 307, 319, 325, 359
ASI_PRIMARY_NOFAULT_LITTLE, 310
ASI_PST16_P, 260, 326
ASI_PST16_PL, 260, 326
ASI_PST16_PRIMARY, 326
ASI_PST16_PRIMARY_LITTLE, 326
ASI_PST16_S, 260, 326
ASI_PST16_SECONDARY, 326
ASI_PST16_SECONDARY_LITTLE, 326
ASI_PST16_SL, 260
ASI_PST32_P, 260, 326
ASI_PST32_PL, 260, 326
ASI_PST32_PRIMARY, 326
ASI_PST32_PRIMARY_LITTLE, 326
ASI_PST32_SECONDARY, 326
ASI_PST32_SECONDARY_LITTLE, 326
ASI_PST32_SL, 260, 326
ASI_PST8_P, 326
ASI_PST8_PL, 326
ASI_PST8_PRIMARY, 326
ASI_PST8_PRIMARY_LITTLE, 326
ASI_PST8_S, 326
ASI_PST8_SECONDARY, 326
ASI_PST8_SECONDARY_LITTLE, 326
ASI_PST8_SL, 260, 326
ASI_QUAD_LDD_REAL (deprecated), 325
ASI_QUAD_LDD_REAL_LITTLE (deprecated), 325
Index

ASI_REAL, 323, 330, 334
ASI_REAL_IO, 323, 330, 334
ASI_REAL_IO_L, 323
ASI_REAL_LITTLE, 323, 330, 334
ASI_S, 325
ASI_SECONDARY, 325, 372
ASI_SECONDARY_LITTLE, 323
ASI_SECONDARY_NO_FAULT, 319, 325, 359
ASI_SECONDARY_NO_FAULT_LITTLE, 319, 326, 359
ASI_SECONDARY_NOFAULT, 310
ASI_SL, 325
ASI_SNF, 325
ASI_SNFL, 326
ASI_TWINX_AIUP, 197, 324, 331
ASI_TWINX_AIUP_L, 197, 331
ASI_TWINX_AIUS, 197, 331
ASI_TWINX_AIUS_L, 325, 331
ASI_TWINX_AS_IF_USER_PRIMARY, 324, 331
ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE, 325, 331
ASI_TWINX_AS_IF_USER_SECONDARY, 324, 331
ASI_TWINX_AS_IF_USER_SECONDARY_LITTLE, 325, 331
ASI_TWINX_N, 197, 325, 335
ASI_TWINX_NL, 197, 325, 331, 335
ASI_TWINX_NUCLEUS, 325, 331, 335
ASI_TWINX_NUCLEUS_L, 308
ASI_TWINX_NUCLEUS_LITTLE, 325, 331, 335
ASI_TWINX_P, 197, 327
ASI_TWINX_PL, 197, 327
ASI_TWINX_PRIMARY, 327, 332
ASI_TWINX_PRIMARY_LITTLE, 327, 332
ASI_TWINX_R, 325, 332
ASI_TWINX_REAL, 197, 325, 332
ASI_TWINX_REAL_L, 325, 332
ASI_TWINX_REAL_LITTLE, 325, 332
ASI_TWINX_S, 197, 327
ASI_TWINX_SECONDARY, 327, 332
ASI_TWINX_SECONDARY_LITTLE, 327, 332
ASI_TWINX_SL, 197, 327
ASR, 5
asr_reg, 410
atomic
  memory operations, 198, 316, 317
  store doubleword instruction, 265, 267
  store instructions, 247, 248
atomic load-store instructions
  compare and swap, 112
  load-store unsigned byte, 190, 272
  load-store unsigned byte to alternate space, 191
  simultaneously addressing doublewords, 271
  swap R register with alternate space memory, 272
  swap R register with memory, 112, 271
atomicity, 307, 403
B
  BA instruction, 104, 389
BCC instruction, 104, 389
bclrg synthetic instruction, 416
BCS instruction, 104, 389
BE instruction, 104, 389
Berkeley RISCs, 15
BG instruction, 104, 389
BGE instruction, 104, 389
BGU instruction, 104, 389
Bicc instructions, 104, 383
big-endian, 5
big-endian byte order, 18, 65, 74
binary compatibility, 15
BL instruction, 104, 389
BLD, 5
BLD, See LDBLOCKF instruction
BLE instruction, 104, 389
BLEU instruction, 104, 389
block load instructions, 38, 178, 333
block store instructions, 38, 250, 333
  with commit, 184, 251, 333
blocked byte formatting, 101
BMASK instruction, 106
BN instruction, 104, 389
BNE instruction, 104, 389
BNEG instruction, 104, 389
BP instructions, 389
BPA instruction, 107, 389
BPCC instruction, 107, 389
BPcc instructions, 51, 107, 390
BPCS instruction, 107, 389
BPE instruction, 107, 389
BPG instruction, 107, 389
BPGE instruction, 107, 389
BPGU instruction, 107, 389
BPL instruction, 107, 389
BLE instruction, 107, 389
BLEU instruction, 107, 389
BPN instruction, 107, 389
BPNEG instruction, 107, 389
BPOS instruction, 104, 389
BPPOS instruction, 107, 389
BPVC instruction, 107, 389
BPVS instruction, 107, 389
branch
  annullled, 109
  delayed, 71
  elimination, 83
  fcc-conditional, 122, 125
  icc-conditional, 104
instructions
  on floating-point condition codes, 122
  on floating-point condition codes with prediction, 124
  on integer condition codes with prediction (BPcc), 107
  on integer condition codes, See Bicc instructions
  when contents of integer register match condition, 109
  prediction bit, 109
  unconditional, 104, 108, 122, 125
  with prediction, 14
BRGEZ instruction, 109
BRGZ instruction, 109
BRLEZ instruction, 109
BRLZ instruction, 109
BRNZ instruction, 109
BRZ instruction, 109
bset synthetic instruction, 416
BSHUFFLE instruction, 106
BST, 5
BST, See STBLOCKF instruction
btog synthetic instruction, 416
btst synthetic instruction, 416
BVC instruction, 104, 389
BVS instruction, 104, 389
byte, 5
  addressing, 76
  data format, 23
  order, 18
  order, big-endian, 18
  order, little-endian, 18
byte order
  big-endian, 65
  implicit, 65
  in trap handlers, 347
  little-endian, 65

C
  cache
    coherency protocol, 306
    data, 312
    instruction, 312
    miss, 224
    nonconsistent instruction cache, 312
  caching, TSB, 377
  CALL instruction
    description, 111
    displacement, 20
    does not change CWP, 35
    and JMPL instruction, 174
    writing address into R[15], 37
  call synthetic instruction, 414
  CANRESTORE (restorable windows) register, 59
    and clean_window exception, 84
    and CLEANWIN register, 59, 61, 363
    counting windows, 60
    decremented by RESTORE instruction, 230
    decremented by SAVED instruction, 239
    detecting window underflow, 36
    if registered window was spilled, 230
    incremented by SAVE instruction, 237
    modified by NORMALW instruction, 213
    modified by OTHERW instruction, 215
    range of values, 58, 403
    RESTORE instruction, 84
    specification for RDPR instruction, 228
    specification for WRPR instruction, 288
    window underflow, 363
  CANSAVE (savable windows) register, 59
  decremented by SAVE instruction, 237
  detecting window overflow, 36
  FLUSHW instruction, 136
  if equals zero, 84
  incremented by RESTORE, 230
  incremented by SAVED instruction, 239
  range of values, 58, 403
  SAVE instruction, 364
  specification for RDPR instruction, 228
  specification for WRPR instruction, 288
  window overflow, 363
  CAS synthetic instruction, 317
  CASA instruction, 112
    32-bit compare-and-swap, 316
    alternate space addressing, 18
    and DAE_nc_page exception, 359
    atomic operation, 190
    hardware primitives for mutual exclusion of CASXA, 316
    in multiprocessor system, 191, 271, 272
    R register use, 73
    word access (memory), 73
  cash synthetic instructions, 416
  CASX synthetic instruction, 316, 317
  CASXA instruction, 112
    64-bit compare-and-swap, 316
    alternate space addressing, 18
    and DAE_nc_page exception, 359
    atomic operation, 191
    doubleword access (memory), 73
    hardware primitives for mutual exclusion of CASA, 316
    in multiprocessor system, 190, 191, 271, 272
    R register use, 73
    catastrophic error exception, 341
  cc0 instruction field
    branch instructions, 107, 125
    floating point compare instructions, 128
    move instructions, 205, 389
  cc1 instruction field
    branch instructions, 107, 125
    floating point compare instructions, 128
    move instructions, 205, 389
  cc2 instruction field
    move instructions, 205, 389
  CCR (condition codes register), 5
  CCR (condition codes) register, 50
    32-bit operation (icc) bit of condition field, 50, 51
    64-bit operation (xcc) bit of condition field, 50, 51
  ADD instructions, 97
  ASR for, 49
  carry (c) bit of condition fields, 51
    icc field, See CCR.icc field
  MULScc instruction, 209
  negative (n) bit of condition fields, 50
  overflow bit (v) in condition fields, 51
  restored by RETRY instruction, 114, 233
  saved after trap, 341
  saving after trap, 21
  TSTATE register, 63
  write instructions, 286
  xcc field, See CCR.xcc field
zero (z) bit of condition fields, 51

CCR.icc field
- add instructions, 97, 274
- bit setting for signed division, 241
- bit setting for signed/unsigned multiply, 246, 283
- bit setting for unsigned division, 282
- branch instructions, 104, 108, 206
- integer subtraction instructions, 270
- logical operation instructions, 100, 214, 290
- MULScc instruction, 209
- Tcc instruction, 277

CCR.xcc field
- add instructions, 97, 274
- bit setting for signed/unsigned divide, 241, 282
- bit setting for signed/unsigned multiply, 246, 283
- branch instructions, 108, 206
- logical operation instructions, 100, 214, 290
- subtract instructions, 270
- Tcc instruction, 277

Clean register window, 237, 358
- and window traps, 61, 362
- CLEANWIN register, 61
- definition, 363
- number is zero, 84
- trap handling, 364

Clean_window exception, 59, 84, 238, 358, 363, 400
CLEANWIN (clean windows) register, 59
- CANSAVE instruction, 84
- clean window counting, 59
- incremented by trap handler, 364
- range of values, 58, 403
- specification for RDPR instruction, 228
- specification for WRPR instruction, 288
- specifying number of available clean windows, 363
- value calculation, 61

clock cycle, counts for virtual processor, 52

clock tick registers, See TICK and STICK registers
clock-tick register (TICK), 361
cmpn synthetic instructions, 416
cmp synthetic instruction, 270, 414
code
- self-modifying, 317

Coherence, 5
- between processors, 403
- data cache, 312
- domain, 306
- memory, 307
- unit, memory, 308

Compare and swap instructions, 112
Comparison instruction, 79, 270

Compatibility note, 4
Completed (memory operation), 5
Compliance
- SPARC V9, 379
- compliant SPARC V9 implementation, 16

Cond instruction field
- branch instructions, 104, 107, 122, 125
- floating point move instructions, 140
- move instructions, 205

Condition codes
- adding, 274
- effect of compare-and-swap instructions, 113
- extended integer (xcc), 51
- floating-point, 122
- icc field, 50
- integer, 50
- results of integer operation (icc), 51
- subtracting, 270, 279
- trapping on, 277
- xcc field, 50

Condition codes register, See CCR register
Conditional branches, 104, 122, 125
Conditional move instructions, 21
Conforming SPARC V9 implementation, 16

Consistency
- between instruction and data spaces, 317
- processor, 312, 315
- processor self-consistency, 314
- sequential, 307, 313
- strong, 313

Const22 instruction field of ILLTRAP instruction, 172
Constants, generating, 242
Context, 5
- nucleus, 135
- selection for translation, 377

Context identifier, 308

Context register
determination of, 377
Nucleus, 381
Primary, 381
Secondary, 381

Control transfer
- pseudo-control-transfer via WRPR to PSTATE.am, 67

Control_transfer_instruction exception, 358

CALL/JMPL instructions, 111, 174
DONE/RETRY instructions, 115, 233, 277
RETURN, 235
and Tcc instruction, 278
with branch instructions, 105, 108, 109, 123, 125
control-transfer instructions, 20
control-transfer instructions (CTIs), 20, 114, 233

Conventions
- font, 2
- notational, 2

Conversion
- between floating-point formats instructions, 168
- floating-point to integer instructions, 167, 293
- integer to floating-point instructions, 132, 171
- planar to packed, 160

Copyback, 5

CPI, 6
CPU, pipeline draining, 58, 61

Cpu_monido exception, 358

Cross-call, 6

CTI, 6, 11

Current exception (cexc) field of FSR register, 46, 86, 398
Current window, 6
Current window pointer register, See CWP register
Current little endian (cle) field of PSTATE register, 65, 309
CWP (current window pointer) register
and instructions
  CALL and JMPL instructions, 35
  FLUSHW instruction, 136
  RDPR instruction, 228
  RESTORE instruction, 84, 230
  SAVE instruction, 84, 230, 237
  WRPR instruction, 288
and traps
  after spill trap, 364
  after spill/fill trap, 21
  on window trap, 363
  saved by hardware, 341
CWP (current window pointer) register, 59
  clean windows, 59
  definition, 6
  incremented/decremented, 35, 230, 237
  overlapping windows, 35
  range of values, 58, 403
  restored during RETRY, 114, 233
  specifying windows for use without cleaning, 363
  and TST A TE register, 63
cycle, 6

D
D superscript on instruction name, 87
  d16hi instruction field
    branch instructions, 109
  d16lo instruction field
    branch instructions, 109
DAE_invalid_ASI exception
  with load instructions and ASIs, 185, 331, 332, 333, 334
  with store instructions and ASIs, 185, 331, 332, 333, 334
DAE_invalid_ASI exception, 359
  accessing noncacheable page, 316
  SWAP/SWAPA, 273
  with compare-and-swap instructions, 113
  with load alternate instructions, 113, 177, 184, 191, 195, 248, 249, 256, 268, 272
  with load instructions, 185, 191, 196
  with load instructions and ASIs, 185
  with nonfaulting load, 319
  with store instructions, 191, 249, 257, 268
DAE_invalid_ASI exception (replacing SPARC V9 data access exception), 359
DAE_invalid_ASI exception
  with load instructions and ASIs, 332
DAE_nc_page exception, 359
  accessing noncacheable page, 316
  with compare-and-swap instructions, 113
  with load instructions, 179, 190, 191, 198
  with store instructions, 191
  with SWAP instructions, 273
DAE_nc_page exception (replacing SPARC V9 data access exception), 359
DAE_nfo_page exception, 359
  and TTE.nfo, 374
  with compare-and-swap instructions, 113
  with FLUSH instructions, 135
  with LDTXA instructions, 198
  with load alternate instructions, 177
  with load instructions, 175, 180, 182, 185, 187, 189, 190, 191, 193, 196, 200, 266
  with nonfaulting load, 319
  with store instructions, 191, 249, 252, 254, 257, 258, 262, 264, 268, 269
  with SWAP instruction, 271
  with SWAP instructions, 273
DAE_nfo_page exception (replacing SPARC V9 data access exception), 359
DAE_noncacheable_page exception
  with LDTXA instructions, 198
  with LDXFSR instructions, 187, 200
DAE_privilege_violation exception, 359
  and TTE.p, 376
  with load alternate instructions, 177
  with load instructions, 180, 185, 187, 191, 193, 196, 252, 266
  with store instructions, 191, 249, 254, 257, 258, 262, 264, 268, 269, 271
  with SWAP instructions, 113, 175, 273
DAE_privilege_violation exception (replacing SPARC V9 data access exception), 359
DAE_side_effect_page exception
  with nonfaulting loads, 307
DAE_side_effect_page exception, 359
  with load alternate instructions, 177
  with load instructions, 180, 185, 196
  with nonfaulting load, 319
DAE_side_effect_page exception (replacing SPARC V9 data access exception), 359
data
  access, 6
  cache coherence, 312
  conversion between SIMD formats, 29
  flow order constraints
    memory reference instructions, 311
    register reference instructions, 311
formats
  byte, 23
  doubleword, 23
  halfword, 23
  Int16 SIMD, 30
  Int32 SIMD, 30
  quadword, 23
  tagged word, 23
  Uint8 SIMD, 30
  word, 23
memory, 318
types
  floating-point, 23
  signed integer, 23
  unsigned integer, 23
  width, 23
watchpoint exception, 261
Data Cache Unit Control register, See DCUCR
data access exception (SPARC V9), 359
DCTI couple, 82
DCTI instructions, 6
behavior, 71
RETURN instruction effects, 235
de synthetic instructions, 416
de synthetic instructions, 416
deferred trap, 344
distinguishing from disrupting trap, 345
floating-point, 229
restartable
implementation dependency, 345
software actions, 345
delay instruction
and annul field of branch instruction, 122
annulling, 20
conditional branches, 125
DONE instruction, 114
executed after branch taken, 109
following delayed control transfer, 20
RETRY instruction, 233
RETURN instruction, 235
unconditional branches, 125
with conditional branch, 108
delayed branch, 71
delayed control transfer, 109
delayed CTI, See DCTI
denormalized number, 6
deprecated, 6
deprecated exceptions
tag overflow, 362
deprecated instructions
FBA, 122
FBE, 122
FBG, 122
FBGE, 122
FBL, 122
FBLE, 122
FBLG, 122
FBN, 122
FBNE, 122
FBO, 122
FBU, 122
FBUE, 122
FBUGE, 122
FBUL, 122
FBULE, 122
LDFSR, 186
LDTW, 192
LDTWA, 194
MULScc, 50, 209
RDY, 49, 50, 225
SDIV, 50, 240
SDIVcc, 50, 240
SMUL, 50, 246
SMULcc, 50, 246
STFSR, 258
STTWA, 265
STTWA, 267
SWAP, 271
SWAPA, 272
T ADDc cTV, 275
T SUBc cTV, 280
UDIV, 50, 281
UDIVcc, 50, 281
UMUL, 50, 283
UMULcc, 50, 283
WRY, 49, 50, 285
dev_mondo exception, 359
disp19 instruction field
branch instructions, 107, 125
disp22 instruction field
branch instructions, 104, 122
disp30 instruction field
word displacement (CALL), 111
disrupting trap, 345
divide instructions, 20, 211, 240, 281
division by zero exception, 79, 211, 360
division-by-zero bits of FSR.aexc/FSR.cexc fields, 48
DMMU
context register usage, 379
Secondary Context register, 381
DONE instruction, 114
effect on TNPC register, 62
effect on TSTATE register, 63
generating illegal instruction exception, 361
modifying CCR.xcc condition codes, 51
return from trap, 341
return from trap handler with different GL value, 70
target address, 20
doubleword, 6
addressing, 76
alignment, 18, 73, 308
data format, 23
definition, 6

E
EDGE16 instruction, 116
EDGE16L instruction, 116
EDGE16LN instruction, 118
EDGE16N instruction, 118
EDGE32 instruction, 116
EDGE32L instruction, 116
EDGE32LN instruction, 118
EDGE32N instruction, 118
EDGE8 instruction, 116
EDGE8L instruction, 116
EDGE8LN instruction, 118
EDGE8N instruction, 118
emulating multiple unsigned condition codes, 83
enable floating-point
See FPRS register, fef field
See PSTATE register, pef field
even parity, 6
target address, 20
even parity, 6
exception, 6
exceptions
See also individual exceptions
catastrophic error, 341
causing traps, 341
clean_window, 358, 400
control transfer instruction, 358
cpu_mondo, 358
Index 7
DAE_invalid_asi, 359
DAE_invalid_ASI (replacing SPARC V9
data_access_exception), 359
DAE_nc_page, 359
DAE_nc_page (replacing SPARC V9
data_access_exception), 359
DAE_nfo_page, 359
DAE_nfo_page (replacing SPARC V9
data_access_exception), 359
DAE_privilegeViolation, 359
DAE_privilegeViolation (replacing SPARC V9
data_access_exception), 359
DAE_side_effect_page, 359
DAE_side_effect_page (replacing SPARC V9
data_access_exception), 359
data_access_exception (SPARC V9), 359
definition, 341
dev_mondo, 359
division_by_zero, 360
fill_n_normal, 360
fill_n_other, 360
fp_disabled
and GSR, 54
fp_disabled, 360
fp_exception_ieee_754, 360
fp_exception_other, 360
htrap_instruction, 360
IAE_privilegeViolation, 360
illegal_instruction, 360
instruction_access_exception (SPARC V9), 361
instruction_VA_watchpoint, 361
interrupt_level_14
and SOFTINT.int_level, 55
and STICK_CMPR.stick_cmpr, 58
and tick_cmpr.tick_cmpr, 56
interrupt_level_14, 361
interrupt_level_15
and SOFTINT.int_level, 55
interrupt_level_n
and SOFTINT.register, 54
and SOFTINT.int_level, 55
interrupt_level_n, 346, 361
LDDF_mem_address_not_aligned, 361
LDOF_mem_address_not_aligned, 362
mem_address_not_aligned, 362
nonresumable_error, 361
pending, 21
privileged_action, 361
privileged_opcode
and access to register-window PR state registers, 58, 61,
69, 70
and access to SOFTINT, 54
and access to SOFTINT_CLR, 56
and access to SOFTINT_SET, 55
and access to STICK_CMPR, 57
and access to tick_cmpr, 56
privileged_opcode, 361
resumable_error, 361
spill_n_normal, 238, 362
spill_n_other, 238, 362
STDF_mem_address_not_aligned, 362
STQF_mem_address_not_aligned, 362
tag_overflow (deprecated), 362
trap_instruction, 362
unimplemented_LDTW, 362
unimplemented_STTW, 362
VA_watchpoint, 362
execute unit, 310
execute_state
trap processing, 356
explicit ASI, 6, 76, 322
extended word, 6
addressing, 76

F
F registers, 6, 17, 85, 291, 348
FABSd instruction, 119, 387, 388
FABSq instruction, 119, 119, 387, 388
FABs instruction, 119
FADD, 120
FADDd instruction, 120
FADDq instruction, 120, 120, 130
FADDS instruction, 120
FALIGNDATA instruction, 121
FAND instruction, 165
FANDNOT1 instruction, 165
FANDNOT1S instruction, 165
FANDNOT2 instruction, 165
FANDNOT2S instruction, 165
FANDS instruction, 165
FBA instruction, 122, 389
FBE instruction, 122, 389
FBfcc instructions, 42, 122, 360, 383, 389
FBG instruction, 122, 389
FBGE instruction, 122, 389
FBL instruction, 122, 389
FBLE instruction, 122, 389
FBLG instruction, 122, 389
FBN instruction, 122, 389
FBNNE instruction, 122, 389
FBO instruction, 122, 389
FBPA instruction, 124, 125, 389
FBPE instruction, 124, 389
FBPfcc instructions, 42, 124, 383, 389, 390
FBPG instruction, 124, 389
FBPGE instruction, 124, 389
FBPL instruction, 124, 389
FBPLE instruction, 124, 389
FBPLG instruction, 124, 389
FBPN instruction, 124, 389
FBPNE instruction, 124, 389
FBPO instruction, 124, 389
FBPU instruction, 124, 389
FBPUE instruction, 124, 389
FBPUG instruction, 124, 389
FBPUGE instruction, 124, 389
FBPUL instruction, 124, 389
FBPULE instruction, 124, 389
FBU instruction, 122, 389

Index 8
FBUE instruction, 122, 389
FBUG instruction, 122, 389
FBUGE instruction, 122, 389
FBUL instruction, 122, 389
FBULE instruction, 122, 389
fcc-conditional branches, 122, 125
fcc, 6
FCMP instructions, 390
FCMP* instructions, 42, 128
FCMPd instruction, 128, 388
FCMPE instructions, 390
FCMPE* instructions, 42, 128
FCMPEd instruction, 128, 388
FCMPGT instruction, 126
FCMPGT16 instruction, 126
FCMPGT32 instruction, 126
FCMPLE16 instruction, 126
FCMPLE32 instruction, 126
FCMPNE16 instruction, 126
FCMPNE32 instruction, 126
FCMPq instruction, 128, 129, 388
fcn instruction field
DONE instruction, 114
PREFETCH, 219
RETRY instruction, 233
FDIVd instruction, 130
FDIVq instruction, 130
FDIVs instructions, 130
FdMULq instruction, 151, 151
FdTOI instruction, 167, 293
FdTOq instruction, 168, 168
FdTOs instruction, 168
FdTOx instruction, 167, 388
fcr field of FPRS register, 53
and access to GSR, 54
and fp_disabled exception, 360
branch operations, 127, 129
byte permutation, 106
comparison operations, 127, 129
component distance, 216
data formatting operations, 131, 153, 160
data movement operations, 206
enabling FPU, 66
floating-point operations, 119, 120, 130, 132, 138, 139, 142, 145, 151, 152, 166, 167, 168, 170, 171, 181, 183, 186, 188, 199
integer arithmetic operations, 146, 159, 162
logical operations, 163, 164, 165
memory operations, 179
read operations, 227, 243, 252
special addressing operations, 98, 121, 253, 258, 261, 263, 269, 286
fcr, See FPRS register, fcr field
FEXPAND instruction, 131
FEXPAND operation, 131
fill handler, 230
fill register window, 360
overflow/underflow, 36
RESTORE instruction, 61, 230, 363
RESTORED instruction, 85, 232, 364
RETRY instruction, 364
selection of, 363
trap handling, 363, 364
trap vectors, 230
window state, 60
fill_n_normal exception, 231, 236, 360
fill_n_other exception, 231, 236, 360
FiTOd instruction, 132
FiTOq instruction, 132, 132, 171
FiTOs instruction, 132
fixed-point scaling, 147
floating point
absolute value instructions, 119
add instructions, 120
compare instructions, 42, 128, 128
condition code bits, 122
condition codes (fcc) fields of FSR register, 44, 122, 125, 128
data type, 23
deferred-trap queue (FQ), 229
divide instructions, 130
exception, 6
exception, encoding type, 44
FPRS register, 286
FSR condition codes, 42
move instructions, 139
multiply instructions, 151
multiply-add/subtract, 137
negate instructions, 152
operate (FPop) instructions, 6, 21, 44, 46, 85, 186
registers
destination F, 291
FPRS, See FPRS register
FSR, See FSR register
programming, 41
rounding direction, 43
square root instructions, 166
subtract instructions, 170
trap types, 7
IEEE_754_exception, 44, 45, 46, 48, 291, 292
invalid_fp_register, 170
unfinished_FPop, 44, 45, 48, 120, 130, 151, 166, 168, 170, 291
results after recovery, 44
unimplemented_FPop, 48, 145, 292
traps
deferred, 229
precise, 229
floating-point condition codes (fcc) fields of FSR register, 348
floating-point operate (FPop) instructions, 360
floating-point trap types
IEEE_754_exception, 348, 360
floating-point unit (FPU), 7, 17
FLUSH instruction, 134
memory ordering control, 202
FLUSH instruction
memory/instruction synchronization, 133
FLUSH instruction, 133, 318
data access, 6
immediacy of effect, 135
in multiprocessor system, 133
in self-modifying code, 134
latency, 403
flush instruction memory, See FLUSH instruction
flush register windows instruction, 136
FLUSHW instruction, 136, 362
effect, 21
management by window traps, 61, 362
spill exception, 85, 136, 364
FMA instructions
fused, 137
FMADDD instruction, 137
FMADDS instruction, 137
FMOVcc instructions
conditionally moving floating-point register contents, 51
conditions for copying floating-point register contents, 83
copying a register, 42
encoding of opf<84> bits, 388
encoding of opf_cc instruction field, 389
encoding of rcond instruction field, 389
floating-point moves, 140
FPop instruction, 85
used to avoid branches, 143, 206
FMOVccd instruction, 388
FMOVccq instruction, 388
FMOVd instruction, 139, 387, 388
FMOVdcc instructions, 140
FMOVdGEZ instruction, 144
FMOVdGZ instruction, 144
FMOVdicc instructions, 140
FMOVdLEZ instruction, 144
FMOVdLZ instruction, 144
FMOVdNZ instruction, 144
FMOVdZ instruction, 144
FMOVq instruction, 139, 387, 388
FMOVQcc instructions, 140, 142
FMOVqGEZ instruction, 144
FMOVqGZ instruction, 144
FMOVQicc instructions, 140, 142
FMOVqLEZ instruction, 144
FMOVqLZ instruction, 144
FMOVqNZ instruction, 144
FMOVqZ instruction, 144
FMOVr instructions, 85, 389
FMOVRq instructions, 145
FMOVRsGZ instruction, 144
FMOVRsLEZ instruction, 144
FMOVRsLZ instruction, 144
FMOVRsNZ instruction, 144
FMOVRsZ instruction, 144
FMOVs instruction, 139
FMOVScc instructions, 142
FMOVSfcc instructions, 140
FMOVsGEZ instruction, 144
FMOVSicc instructions, 140
FMOVSxcc instructions, 140
FMOVxcc instructions, 140, 142
FMSUBd instruction, 137
FMSUBs instruction, 137
FMUL8SUx16 instruction, 146, 148
FMUL8ULx16 instruction, 146, 148
FMUL8x16 instruction, 146, 147
FMUL8x16AL instruction, 146, 148
FMUL8x16AU instruction, 146, 147
FMULd instruction, 151
FMULD8SUx16 instruction, 146, 149
FMULD8ULx16 instruction, 146, 150
FMULq instruction, 151, 151
FMULs instruction, 151
FNAND instruction, 165
FNANDS instruction, 165
FNEG instructions, 152
FNEGd instruction, 152, 387, 388
FNEGq instruction, 152, 152, 387, 388
FNEGvs instruction, 152, 152, 387, 388
FNEGvs instruction, 152
FNADD instruction, 165
FNAND instruction, 165
FORNOT1 instruction, 164
FORNOT1S instruction, 164
FORNOT2 instruction, 164
FORNOT2S instruction, 164
FONE instruction, 163
FONES instruction, 163
FOR instruction, 165
formats, instruction, 72
FORNOT1 instruction, 165
FORNOT1S instruction, 165
FORNOT2 instruction, 165
FORNOT2S instruction, 165
FORS instruction, 165
fp_disabled exception, 360
absolute value instructions, 119, 120, 170
and gsr, 54
FPop instructions, 85
FPERS.fel disabled, 53
PSTATE.pes not set, 53, 406
with branch instructions, 123, 125
with compare instructions, 127
with conversion instructions, 132, 167, 168, 171
with floating-point arithmetic instructions, 130, 138, 151,
159, 162
with FMOV instructions, 139
with load instructions, 185, 188
with move instructions, 143, 145, 206
with negate instructions, 152
with store instructions, 253, 254, 257, 258, 261, 263, 269,
286
fp_exception exception, 46
fp_exception_ieee_754 "invalid" exception, 167
fp_exception_ieee_754 exception, 360
  and tem bit of FSR, 43
cause encoded in FSR.ftt, 44
FSR.aexc, 46
FSR.cexc, 47
FSR.ftt, 46
  generated by FCMP or FCMPE, 42
  and IEEE 754 overflow/underflow conditions, 46, 47
  trap handler, 292
  when FSR.ns = 1, 294, 398
  when FSR.tem = 0, 348
  when FSR.tem = 1, 348
  with floating-point arithmetic instructions, 120, 130, 138, 151, 170
fp_exception_other exception, 48, 360
  cause encoded in FSR.ftt, 44
FSUBq instruction, 170
  incorrect IEEE Std 754-1985 result, 86, 397
  supervisor handling, 292
  trap type of unfinished_FPop, 45
  when quad FPop unimplemented in hardware, 46
  with floating-point arithmetic instructions, 130, 138, 151
FPACK instruction, 54
FPACK instructions, 153–156
  FPACK16 instruction, 153, 154
  FPACK16 operation, 154
  FPACK32 instruction, 153, 155
  FPACK32 operation, 155
  FPACKFIX instruction, 153, 156
  FPACKFIX operation, 156
  FPADD16 instruction, 158
  FPADD16S instruction, 158
  FPADD32 instruction, 158
  FPADD32S instruction, 158
  FPMERGE instruction, 160
FPop, 7
FPop, See floating-point operate (FPop) instructions
FPRS register
  See also floating-point registers state (FPRS) register
FPRS register, 53
  ASR summary, 49
definition, 7
  fef field, 85, 347
  RDFPRS instruction, 226
FPRS register fields
  dl (dirty lower fp registers), 53
  du (dirty upper fp registers, 53
  fef, 53
  fef, See also fef field of FPRS register
FPSUB16 instruction, 161
FPSUB16S instruction, 161
FPSUB32 instruction, 161
FPSUB32S instruction, 161
FPU, 6, 7
FxTOd instruction, 168, 168
FxTOi instruction, 167, 167, 293
FxToS instruction, 168, 168
FxTOx instruction, 167, 167, 387, 388
freg, 410
FsMULd instruction, 151
FSQRTd instruction, 166
FSQRTq instruction, 166, 166
FSQRTs instruction, 166
FSR (floating-point state) register fields
  aexc (accrued exception), 44, 45, 46, 46, 291
  aexc (accrued exceptions), 138
  in user-mode trap handler, 292
  -- dza (division by zero) bit of aexc, 48
  -- nxa (rounding) bit of aexc, 48
cexc (current exception), 43, 44, 45, 46, 47, 291, 360
cexc (current exceptions), 138
  in user-mode trap handler, 292
  -- dzc (division by zero) bit of cexc, 48
  -- nxc (rounding) bit of cexc, 48
cfcc (condition codes), 42, 44, 45, 292, 410
fccn, 42
  ftt (floating-point trap type), 42, 44, 46, 86, 199, 258, 269, 360
  in user-mode trap handler, 292
  not modified by LDFSR/LDXFSR instructions, 42
  ns (nonstandard mode), 42, 186, 199
  qne (queue not empty), 42, 186, 199
  in user-mode trap handler, 292
  rd (rounding), 43
tem (trap enable mask), 43, 46, 47, 138, 293, 360
ver, 43
ver (version), 42, 199
FSR (floating-point state) register, 42
  after floating-point trap, 291
  compliance with IEEE Std 754-1985, 48
  LDFSR instruction, 186
  reading/writing, 42
  values in ftt field, 44
  writing to memory, 258, 269
FSRC1 instruction, 164
FSRC1S instruction, 164
FSRC2 instruction, 164
FSRC2S instruction, 164
FsTOd instruction, 168
FsTOi instruction, 167, 293
FsTOq instruction, 168, 168
FsTOx instruction, 167, 387, 388
FSUBd instruction, 170
FSUBq instruction, 170, 170
FSUBs instruction, 170
  functional choice, implementation-dependent, 396
  fused FMA instructions, 137
FXNOR instruction, 165
FXNORS instruction, 165
FXOR instruction, 165
FXORS instruction, 165
FxTOd instruction, 171, 388
FxTOq instruction, 171, 388
FxToS instruction, 171, 388
FZERO instruction, 163
FZEROS instruction, 163
**G**

- General status register, See GSR (general status) register
- Generating constants, 242
- GL register, 69
  - Access, 70
  - During trap processing, 356
  - Function, 69
  - Reading with RDPR instruction, 228, 288
  - Relationship to TL, 70
  - Restored during RETRY, 114, 233
  - SPARC V9 compatibility, 68
  - And TST A TE register, 63
  - Value restored from TSTATE[TL], 70
  - Writing to, 70
- Global level register, See GL register
- Global registers, 14, 17, 32, 33, 397
- Graphics status register, See GSR (general status) register
- GSR (general status) register fields
  - Align, 54
  - Im (interval mode) field, 54
  - IrmD (rounding), 54
  - Mask, 54
  - Scale, 54
- GSR (general status) register
  - ASR summary, 49

**H**

- Halfword, 7
  - Alignment, 18, 73, 308
  - Data format, 23
- Hardware dependency, 396
- Traps, 349
- Hardware trap stack, 21
- Htrap_instruction exception, 278, 360
- Hyperprivileged, 7

**I**

- I (integer) instruction field
  - Arithmetic instructions, 209, 211, 214, 240, 246, 281, 283
  - Floating point load instructions, 181, 183, 186, 199
  - Flush memory instruction, 133
  - Flush register instruction, 136
  - Jump-and-link instruction, 174
  - Load instructions, 175, 190, 191, 192, 194
  - Logical operation instructions, 100, 214, 290
  - Move instructions, 205, 207
  - POPC, 217
  - PREFETCH, 219
  - RETURN, 235
- I/O
  - Access, 307
  - Memory, 306
  - Memory-mapped, 306
- IAE_privilegeViolation exception, 360
  - And TTE.p, 376
- IAE_unauth_access exception, 376

**IEEE 754, 7**

- IEEE Std 754-1985, 7, 13, 43, 45, 47, 48, 86, 291, 397
- IEEE._754_exception floating-point trap type, 7, 44, 45, 46, 48, 291, 292, 348, 360
- IEEE-754 exception, 7
- IER register (SPARC V8), 286
- Illegal_instruction exception, 136, 360
  - Attempt to write in nonprivileged mode, 57
  - DONE/RETRY, 115, 234, 235
  - ILLTRAP, 172
  - Not implemented in hardware, 96
  - POPC, 218
  - PREFETCH, 224
  - RETURN, 236
  - With BP instruction, 109
  - With branch instructions, 108, 110
  - With CASA and CASXA instructions, 112, 214
  - With CASXA instruction, 113
  - With DONE instruction, 115
  - With FMOV instructions, 139
  - With FMOVcc instructions, 143
  - With FMOVR instructions, 145
  - With load instructions, 38, 180, 182, 192, 195, 200, 333
  - With move instructions, 206, 208
  - With read hyperprivileged register instructions, 228
  - With read instructions, 226, 228, 289, 399
  - With store instructions, 183, 254, 258, 265, 266, 267, 268, 269
  - With Tcc instructions, 278
  - With TPC register, 61
  - With TSTATE register, 63
  - With write instructions, 286, 289
  - Write to ASR 5, 52
  - Write to STICK register, 57
  - Write to TICK register, 52
- ILLTRAP instruction, 172, 360
- Imm_asi instruction field
  - Explicit ASI, providing, 76
  - Floating point load instructions, 183
  - Load instructions, 191, 192, 194
  - PREFETCH, 219
- Immediate CTI, 71
- I-MMU
  - And instruction prefetching, 307
- IMMU
  - Context register usage, 379
- IMPDEP1 instructions, 86
- IMPDEP2 instructions, 86
- IMPDEP2A instructions, 400
- IMPDEP2B instructions, 86
- Implementation, 7
- Implementation dependency, 395
- Implementation dependent, 7
- Implementation note, 3, 4
- Implementation-dependent functional choice, 396
- Implicit ASI, 7, 76, 322
- Implicit ASI memory access
  - LDFSR, 186
  - LDSTUB, 190
  - Load fp instructions, 181, 199
load integer doubleword instructions, 192
load integer instructions, 175
STD, 265
STFSR, 258
store floating-point instructions, 253, 269
store integer instructions, 247
SWAP, 271
implicit byte order, 65
in registers, 33, 35, 237
inccc synthetic instructions, 416
inexact accrued (nxa) bit of aexc field of FSR register, 292
inexact current (nxc) bit of cexc field of FSR register, 292
inexact mask (nxm) field of FSR.templ, 47
inexact quotient, 240, 281
infinity, 293
initiated, 7
input/output (I/O) locations
access by nonprivileged code, 397
behavior, 306
contents and addresses, 397
identifying, 402
order, 306
semantics, 403
value semantics, 306
instruction fields, 7
See also individual instruction fields
definition, 7
instruction group, 7
instruction MMU, See I-MMU
instruction prefetch buffer, invalidation, 134
instruction set architecture (ISA), 7, 7, 14
instruction_access_exception exception (SPARC V9), 361
instruction_VA_watchpoint exception, 361
instructions
32-bit wide, 13
alignment, 73
alignment, 18, 98, 308
arithmetic, integer
addition, 97, 274
division, 20, 211, 240, 281
multiplication, 20, 209, 211, 246, 283
subtraction, 270, 279
tagged, 20
array addressing, 101
atomic
CASA/CASXA, 112
load twin extended word from alternate space, 197
load-store, 73, 112, 190, 191, 271, 272
load-store unsigned byte, 190, 191
successful loads, 175, 176, 193, 195
successful stores, 247, 248
branch
branch if contents of integer register match condition, 109
branch on floating-point condition codes, 122, 124
branch on integer condition codes, 104, 107
cache, 312
causing illegal instruction, 172
compare and swap, 112
comparison, 79, 270
conditional move, 21
control-transfer (CTIs), 20, 114, 233
conversion
convert between floating-point formats, 168
convert floating-point to integer, 167
convert integer to floating-point, 132, 171
floating-point to integer, 293
count of number of bits, 217
dge handling, 116
detches, 73
floating point
compare, 42, 128
floating-point add, 120
floating-point divide, 130
floating-point load, 73, 181
floating-point load from alternate space, 183
floating-point load state register, 181, 199
floating-point move, 139, 140, 144
floating-point operate (FPop), 21, 186
floating-point square root, 166
floating-point store, 73, 253
floating-point store to alternate space, 255
floating-point subtract, 170
operate (FPop), 44, 46
short floating-point load, 188
short floating-point store, 263
status of floating-point load, 186
flush instruction memory, 133
flush register windows, 136
formats, 72
jump and link, 20, 174
loads
block load, 178
floating point, See instructions: floating point integer, 73
simultaneously addressing doublewords, 271
unsigned byte, 112, 190
unsigned byte to alternate space, 191
logical operations
64-bit/32-bit, 164, 165
AND, 100
logical 1-operand ops on F registers, 163
logical 2-operand ops on F registers, 164
logical 3-operand ops on F registers, 165
logical XOR, 290
OR, 214
memory, 318
moves
floating point, See instructions: floating point integer register, 204, 207
on condition, 14
ordering MEMBAR, 78
permuting bytes specified by GSR.mask, 106
pixel component distance, 216, 216
pixel formatting (PACK), 153
prefetch data, 219
read privileged register, 228
read state register, 20, 225
register window management, 21
reordering, 311
Index

14

reserved, 86
reserved fields, 96
RETRY
    and restartable deferred traps, 345
RETURN vs. RESTORE, 235
sequencing MEMBAR, 78
set high bits of low word, 242
set interval arithmetic mode, 243
setting GSR.field, 106
shift, 19
shift, 244
shift count, 244
SIMD, 10
simultaneous addressing of doublewords, 272
stores
    block store, 250
    floating point, See instructions: floating point
    integer, 73, 247
    integer (except doubleword), 247
    integer into alternate space, 248
    partial, 260
    unsigned byte, 112
    unsigned byte to alternate space, 191
    unsigned bytes, 190
swap R register, 271, 272
synthetic (for assembly language programmers), 414–416
tagged addition, 274
test-and-set, 316
timing, 96
trap on integer condition codes, 276
write privileged register, 288
write state register, 286
integer unit (IU)
    condition codes, 51
definition, 7
description, 16
interrupt
    enable (ie) field of PSTATE register, 346, 347
    level, 69
    request, 7, 21, 341
interrupt_level_14 exception, 55, 361
    and SOFTINT.int_level, 55
    and STICK_CMPR.tick_cmpr, 58
    and TICK_CMPR.tick_cmpr, 56
interrupt_level_15 exception
    and SOFTINT.int_level, 55
interrupt_level_n exception, 346, 361
    and SOFTINT.int_level, 54
    and SOFTINT.int_level, 55
inter-strand operation, 7
intra-strand operation, 7
invalid accrued (nva) bit of aexc field of FSR register, 48
invalid ASI
    and DAE_invalid.asi, 359
invalid current (nvc) bit of cexc field of FSR register, 48, 293
invalid mask (nvm) field of FSR.tem, 47, 293
invalid_exception exception, 167
invalid_fp_register floating-point trap type, 145, 152, 170
INVALW instruction, 173
iprefetch synthetic instruction, 415
ISA, 8
ISA, See instruction set architecture
issue unit, 310, 310
issued, 8
italic font, in assembly language syntax, 409
IU, 8
ixc synthetic instructions, 416

J
jmp synthetic instruction, 414
JMPLE instruction, 174
computing target address, 20
does not change CWP, 35
mem_address_not_aligned exception, 361
reexecuting trapped instruction, 235
jump and link, See JMPLE instruction

L
LD instruction (SPARC V8), 175
LDBLOCKF instruction, 178, 333
LDBLOCKF instruction, DAE_nj_page exception, 359
LDD instruction (SPARC V8 and V9), 192
LDDA instruction, 332
LDDA instruction (SPARC V8 and V9), 194
LDDF instruction, 73, 181, 361
LDDF_mem_address_not_aligned exception, 361
    address not doubleword aligned, 401
    address not quadword aligned, 402
    load instruction with partial store ASI and misaligned
    address, 185
    with load instructions, 181, 183, 333
    with store instructions, 255, 333
LDDF_mem_not_aligned exception, 41
LDDFA instruction, 183, 262
alignment, 73
ASIs for fp load operations, 333
behavior with block store with Commit ASIs, 184
behavior with partial store ASIs, 182–??, 185, 185–??, 199–??, 333–??
causing LDDF_mem_address_not_aligned exception, 73, 361
for block load operations, 333
used with ASIs, 333
LDF instruction, 41, 181
LDF instruction, 41, 183
LDFA instruction, 42, 52, 186, 361
LDQF instruction, 181, 362
LDQF_mem_address_not_aligned exception, 362
    address not quadword aligned, 402
    load instruction with partial store ASI and misaligned
    address, 183
LDQF/LDQFA instruction, 74
LDQF/LDQFA instruction, 74
with load instructions, 183
LDQFA instruction, 183
LDSB instruction, 175
LDSB instruction, 175
LDSBA instruction, 175
LDSH instruction, 175
LDSHA instruction, 176
LDSHORTF instruction, 188

Index 14
LDSTUB instruction, 73, 190, 191, 316, 317
and DAE_nc_page exception, 359
hardware primitives for mutual exclusion of
LDSTUB, 316
LDSTUBA instruction, 190, 191
alternate space addressing, 18
and DAE_nc_page exception, 359
hardware primitives for mutual exclusion of
LDSTUBA, 316
LDSW instruction, 175
LDSWA instruction, 176
LDTW instruction, 38, 73
LDTW instruction (deprecated), 192
LDTWA instruction, 38, 73
LDTWA instruction (deprecated), 194
LDTX instruction, 331
LDTX instruction, DAE_nc_page exception, 359
LDTXA instruction, 74, 76, 197, 331
access alignment, 73
access size, 73
LDUB instruction, 175
LDUBA instruction, 176
LDUH instruction, 175
LDUHA instruction, 176
LDUW instruction, 175
LDUWA instruction, 176
LDX instruction, 175
LDXA instruction, 176, 195, 314
LDFSR instruction, 42, 44, 186, 199, 239, 361
leaf procedure
modifying windowed registers, 84
little-endian byte order, 8, 18, 65
load
block, See block load instructions
floating-point from alternate space instructions, 183
floating-point instructions, 181, 186
floating-point state register instructions, 181, 199
from alternate space, 19, 51, 76
instructions, 8
instructions accessing memory, 73
nonfaulting, 310
short floating-point, See short floating-point load
instructions
LoadLoad MEMBAR relationship, 201
LoadLoad MEMBAR relationship, 317
LoadLoad predefined constant, 413
loads
nonfaulting, 319
load-store alignment, 18, 73, 308
load-store instructions
compare and swap, 112
definition, 8
load-store unsigned byte, 112, 190, 271, 272
load-store unsigned byte to alternate space, 191
memory access, 17
swap R register with alternate space memory, 272
swap R register with memory, 112, 271
LoadStore MEMBAR relationship, 201, 317
LoadStore predefined constant, 413
local registers, 33, 35, 230
logical XOR instructions, 290
Lookaside predefined constant, 413
LSTPARTIALF instruction, 333

M
MAXPGL, 17, 32, 33, 68, 70, 70, 406
MAXPTL
and MAXPGL, 70
instances of TNPC register, 62
instances of TPC register, 61
instances of TSTATE register, 63
instances of TT register, 64
may (keyword), 8
mem_address_not_aligned exception, 361
generated by virtual processor, 185
JMPL instruction, 174
LDTXA, 331, 332
load instruction with partial store ASI and misaligned
address, 185
RETURN, 235, 236
when recognized, 113
with CASA instruction, 112
with compare instructions, 113
with load instructions, 73–74, 175, 176, 181, 185, 186, 188,
192, 193, 195, 199, 269, 333
with store instructions, 73–74, 185, 247, 248, 249, 257, 258,
263, 266, 268, 333
with swap instructions (deprecated), 271, 273
MEMBAR
#Sync
semantics, 203
instruction
atomic operation ordering, 317
FLUSH instruction, 133, 318
functions, 201, 316–318
memory ordering, 202
memory synchronization, 78
side-effect accesses, 307
STBAR instruction, 202
mask encodings
#LoadLoad, 201, 317
#LoadStore, 201, 317
#Lookaside, 202, 318
#MemIssue, 202, 318
#StoreLoad, 201, 317
#StoreStore, 201, 317
#Sync, 202, 318
predefined constants
#LoadLoad, 413
#LoadStore, 413
#Lookaside, 413
#MemIssue, 413
#StoreLoad, 413
#StoreStore, 413
#Sync, 413
MEMBAR
#Lookaside, 314
#StoreLoad, 314
membar_mask, 413
memIssue predefined constant, 413
memory
access instructions, 17, 73
alignment, 308
atomic operations, 316
atomicity, 403
cached, 306
cohere, 307, 403
cohere, 307, 403
data, 318
instruction, 318
location, 305
models, 305
ordering unit, 308
real, 306
reference instructions, data flow order constraints, 311
synchronization, 202
virtual address, 305
virtual address 0, 319
Memory Management Unit
definition, 8
Memory Management Unit, See MMU
memory model
mode control, 313
partial store order (PSO), 313
relaxed memory order (RMO), 203, 313
sequential consistency, 313
strong, 313
total store order (TSO), 203, 313, 314
weak, 313
memory model (mm) field of PSTATE register, 66
memory order
pending transactions, 312
program order, 310
memory_model (mm) field of PSTATE register, 313
memory-mapped I/O, 306
metrics
for architectural performance, 338
for implementation performance, 338
See also performance monitoring hardware
MMU
accessing registers, 380
contexts, 372
definition, 8
page sizes, 369
SPARC V9 compliance, 379
mode
nonprivileged, 15
privileged, 16, 61, 310
motion estimation, 216
MOVA instruction, 204
MOVCC instruction, 204
MOVcc instructions, 204
conditionally moving integer register contents, 51
conditions for copying integer register contents, 31
copying a register, 42
encoding of cond field, 389
encoding of opf_cc instruction field, 389
used to avoid branches, 143, 206
MOVCS instruction, 204
move floating-point register if condition is true, 140
move floating-point register if contents of integer register satisfy condition, 144
MOVE instruction, 204
move integer register if condition is satisfied instructions, 204
move integer register if contents of integer register satisfies condition instructions, 207
move on condition instructions, 14
MOVFA instruction, 205
MOVFE instruction, 205
MOVFG instruction, 205
MOVFL instruction, 205
MOVFLE instruction, 205
MOVFLG instruction, 205
MOVFN instruction, 205
MOVFNE instruction, 205
MOVFO instruction, 205
MOVFU instruction, 205
MOVFUE instruction, 205
MOVFLUG instruction, 205
MOVFLUGE instruction, 205
MOVFUL instruction, 205
MOVFLUGH instruction, 205
MOVFLUGE instruction, 205
MOVFLUGH instruction, 205
MOVL instruction, 204
MOVL instruction, 204
MOVLE instruction, 204
MOVLEU instruction, 204
MOVNEG instruction, 204
MOVNEG instruction, 204
MOVPOS instruction, 204
MOVPOS instruction, 204
MOVr instructions, 83, 307, 389
MOVrRGEZ instruction, 207
MOVrRGZ instruction, 207
MOVrRLEZ instruction, 207
MOVrLZ instruction, 207
MOVrNZ instruction, 207
MOVrZ instruction, 207
MOVCC instruction, 204
MOVVS instruction, 204
multiple unsigned condition codes, emulating, 83
multiply instructions, 211, 246, 283
multiply-add instructions (fused), 137
multiply-subtract instructions (fused), 137
multiprocessor synchronization instructions, 112, 271, 272
multiprocessor system, 8, 133, 223, 271, 272, 312, 403
MULX instruction, 211
must (keyword), 8
N
N superscript on instruction name, 87
N_REG_WINDOWS, 9
integer unit registers, 17, 397
RESTORE instruction, 230
SAVE instruction, 237
value of, 32, 58
NaN (not-a-number)
conversion to integer, 293
converting floating-point to integer, 167
signalling, 42, 128, 168
neg synthetic instructions, 416
negative infinity, 293
negative multiply-add instructions (fused), 137
negative multiply-subtract instructions (fused), 137
nested traps, 14
next program counter register, See NPC register
NFO, 8
noncacheable accesses, 306
nonfaulting load, 8, 310
nonfaulting loads
behavior, 319
use by optimizer, 319
nonfaulting-only page
illegal access to, 359
nonfaulting-only page, illegal access to and TTE.nfo, 374
nonleaf routine, 174
nonprivileged mode, 5, 8, 15, 16, 44
software, 53
nonresumable_error exception, 361
nonstandard floating-point, See floating-point status register (FSR) NS field
nontranslating ASI, 8, 195, 268, 322
nonvirtual memory, 223
NOP instruction, 104, 122, 125, 212, 220, 277
normal traps, 349
NORMALW instruction, 213
not synthetic instructions, 416
note
architectural direction, 4
compatibility, 4
general, 3
implementation, 3
programming, 3
NPC (next program counter) register, 53
and PSTATE.tct, 65
control flow alteration, 11
definition, 8
DONE instruction, 114
instruction execution, 71
relation to TNPc register, 62
RETURN instruction, 233
saving after trap, 21
npt, 9
nucleus context, 135
Nucleus Context register, 381
nucleus software, 9
NUMA, 9
nvm (invalid mask) field of FSR.tem, 47, 293
NWIN, See N_REG_WINDOWS
nxm (inexact mask) field of FSR.tem, 47
O
octlet, 9
odd parity, 9
ofm (overflow mask) field of FSR.tem, 47
op3 instruction field
arithmetic instructions, 97, 108, 109, 112, 209, 211, 240, 246, 281, 283
floating point load instructions, 181, 183, 186, 199
flush instructions, 133, 136
jump-and-link instruction, 174
load instructions, 175, 190, 191, 192, 194
logical operation instructions, 100, 214, 290
PREFETCH, 219
RETURN, 235
opcode
definition, 9
opf instruction field
floating point arithmetic instructions, 120, 130, 151, 166
floating point compare instructions, 128
floating point conversion instructions, 167, 168, 171
floating point instructions, 119
floating point integer conversion, 132
floating point move instructions, 139
floating point negate instructions, 152
opf_cc instruction field
floating point move instructions, 140
move instructions, 389
opf_low instruction field, 140
optional, 9
OR instruction, 214
ORcc instruction, 214
ordering MEMBAR instructions, 78
ordering unit, memory, 308
ORN instruction, 214
ORNcc instruction, 214
OTHERW instruction, 215
OTHERWIN (other windows) register, 60
FLUSHW instruction, 136
keeping consistent state, 61
modified by OTHERW instruction, 215
partitioned, 61
range of values, 58, 403
rd designation for WRPR instruction, 288
rs1 designation for RDPR instruction, 228
SAVE instruction, 237
zeroed by INVALIDW instruction, 173
zeroed by NORMALW instruction, 213
OTHERWIN register trap vectors
fill/spill traps, 363
handling spill/fill traps, 363
selecting spill/fill vectors, 363
out register #7, 37
out registers, 33, 35, 237
overflow
bits
(v) in condition fields of CCR, 79
accrued (ofa) in aexc field of FSR register, 48
current (ofc) in cexc field of FSR register, 48
causing spill trap, 363
tagged add/subtract instructions, 79

Index 17
Index

overflow mask (ofm) field of FSR, 47

P

p (predict) instruction field of branch instructions, 107, 109, 125
P superscript on instruction name, 87
packed-to-planar conversion, 160
packing instructions, See FPACK instructions
page fault, 223
page table entry (PTE), See translation table entry (TTE)
parity, even, 6
parity, odd, 9
partial store instructions, 260, 333
partial store order (PSO) memory model, 313, 313
partitioned
additions, 158
subtracts, 161
PASI superscript on instruction name, 87
PASR superscript on instruction name, 87
PC (program counter) register, 9, 49, 52
after instruction execution, 71
and PSTATE.tct, 65
CALL instruction, 111
changed by NOP instruction, 212
copied by JMPL instruction, 174
saving after trap, 21
set by DONE instruction, 114
set by RETRY instruction, 233
Trap Program Counter register, 61
PDIST instruction, 216
pef field of PSTATE register
and access to GSR, 54
and fp_disabled exception, 360
and FPop instructions, 85
byte permutation, 106
comparison operations, 127, 129
component distance, 216
data formatting operations, 131, 153, 160
data movement operations, 206
enabling FPU, 53
floating-point operations, 119, 120, 130, 132, 138, 139, 142, 145, 151, 152, 166, 167, 168, 170, 171, 181, 183, 186, 188, 199
integer arithmetic operations, 146, 159, 162
logical operations, 163, 164, 165
memory operations, 179
read operations, 227, 243, 252
special addressing operations, 98, 121, 253, 258, 261, 263, 269, 286
trap control, 347
pef, See PSTATE, pef field
performance monitoring hardware
accuracy requirements, 338
classes of data reported, 338
counters and controls, 339
high-level requirements, 337
kinds of user needs, 337
See also instruction sampling
physical processor, 9
pil (processor interrupt level) register, 69
interrupt conditioning, 346
interrupt request level, 348
interrupt_level_n, 361
specification of register to read, 228
specification of register to write, 288
trap processing control, 347
pipeline, 9
pipeline draining of CPU, 58, 61
pixel instructions
compare, 126
component distance, 216, 216
formatting, 153
planar-to-packed conversion, 160
Pspt superscript on instruction name, 87
POPC instruction, 217
positive infinity, 293
precise floating-point traps, 229
precise trap, 344
conditions for, 344
software actions, 344
vs. disrupting trap, 345
predefined constants
LoadLoad, 413
lookaside, 413
MemIssue, 413
StoreLoad, 413
StoreStore, 413
Sync, 413
predict bit, 109
prefetch
for one read, 222
for one write, 223
for several reads, 222
for several writes, 223
page, 223
to nearest unified cache, 223
prefetch data instruction, 219
PREFETCH instruction, 73, 219, 400
prefetch_fcn, 413
PREFETCHA instruction, 219, 400
and invalid ASI or VA, 359
prefetchable, 9
Primary Context ID 0, 380
Primary Context ID 1, 380
Primary Context register, 381
priority of traps, 347, 356
privileged_action exception
read from Tick register when access disabled, 52
privilege violation, and DAE_privilege_violation exception, 359
privileged, 9
mode, 16, 61
registers, 61
software, 15, 36, 44, 66, 78, 136, 349, 400
privileged (priv) field of PSTATE register, 68, 113, 115, 176, 179, 183, 184, 191, 195, 248, 252, 256, 267, 272, 310, 361
privileged mode, 9
privileged_action exception, 361

Index 18
accessing restricted ASIs, 309
read from TICK register when access disabled, 52, 226
restricted ASI access attempt, 78, 321
TICK register access attempt, 51
with CASA instruction, 113
with compare instructions, 113
with load alternate instructions, 176, 179, 184, 191, 195, 248, 252, 256, 267, 272
with load instructions, 183
with RDasr instructions, 227
with read instructions, 227
with store instructions, 257
with swap instructions, 273
privileged Opcode exception, 361
DONE instruction, 115
RETRY instruction, 234
SAVED instruction, 239
with DONE instruction, 115, 228, 233, 289
with write instructions, 289
WRPR in nonprivileged mode, 52
processor, 9
execute unit, 310
issue unit, 310, 310
privilege-mode transition diagram, 343
reorder unit, 310
self-consistency, 311
processor cluster, See processor module
processor consistency, 312, 315
processor interrupt level register, See PIL register
processor self-consistency, 311, 314
processor state register, See PSTATE register
processor states
execute_state, 356
program counter register, See PC register
program counters, saving, 341
program order, 310, 311
programming note, 3
PSO, See partial store order (PSO) memory model
PSR register (SPARC V8), 286
PSTATE register
entering privileged execution mode, 341
restored by RETRY instruction, 114, 233
saved after trap, 341
saving after trap, 21
specification for RDPR instruction, 228
specification for WRPR instruction, 288
and TSTATE register, 63
PSTATE register fields
ag
unimplemented, 68
am
CALL instruction, 111
description, 66
masked/unmasked address, 114, 174, 233, 235
cle
and implicit ASIs, 76
and PSTATE.ile, 65
description, 65
ie
description, 68
enabling disrupting traps, 346
interrupt conditioning, 346
masking disrupting trap, 350
mm
description, 66
implementation dependencies, 66, 313, 402
reserved values, 66
pef
and FPRS.fef, 66
description, 66
See also pef field of PSTATE register
priv
access to register-window PR state registers, 61
accessing restricted ASIs, 309
description, 68
determining mode, 8, 9, 376
tct
branch instructions, 105, 108, 109, 123, 125
CALL instruction, 111
description, 65
DONE instruction, 115
JMPL instruction, 174
RETRY instruction, 233, 277
RETURN instruction, 235
tle
description, 65
PTE (page table entry), See translation table entry (TTE)
Q
quadword, 9
alignment, 18, 73, 308
data format, 23
quiet NaN (not-a-number), 42, 128
R
R register, 9
#15, 37
special-purpose, 37
alignment, 192, 195
rational quotient, 281
R-A-W, See read-after-write memory hazard
rcond instruction field
branch instructions, 109
encoding of, 389
move instructions, 207
rd (rounding), 10
rd instruction field
arithmetic instructions, 97, 108, 109, 112, 209, 211, 240, 246, 281, 283
floating point arithmetic, 120
floating point arithmetic instructions, 130, 151, 166
floating point conversion instructions, 167, 168, 171
floating point integer conversion, 132
floating point load instructions, 181, 183, 186, 199
floating point move instructions, 139, 140
floating point negate instructions, 152
floating-point instructions, 119
jump-and-link instruction, 174
load instructions, 175, 190, 191, 192, 194
logical operation instructions, 100, 214, 290
move instructions, 205, 207
POPC, 217
RDASI instruction, 49, 51, 225
RDasr instruction, 225
accessing I/O registers, 19
implementation dependencies, 226, 399
reading ASRs, 48
RDCCR instruction, 49, 50, 225, 225
RDFPRS instruction, 49, 53, 225
RDGSR instruction, 49, 54, 225
RDPC instruction, 49, 225
reading PC register, 52
RDPR instruction, 49, 228
accessing GL register, 70
accessing non-register-window PR state registers, 61
accessing register-window PR state registers, 58
and register-window PR state registers, 58
effect on TNPC register, 62
effect on TPC register, 62
effect on TSTATE register, 63
effect on TT register, 64
reading privileged registers, 61
reading PSTATE register, 65
reading the TICK register, 52
registers read, 228
RDSOFTINT instruction, 49, 54, 225
RDSTICK instruction, 49, 57, 225, 227
RDSTICK_CMPR instruction, 49, 225
RDTICK instruction, 49, 52, 225, 226
RDTICK_CMPR instruction, 49, 225
RDY instruction, 50
read ancillary state register (RDasr) instructions, 225
read state register instructions, 20
read-after-write memory hazard, 311
real address, 10
real ASI, 322
real memory, 306
reference MMU, 409
reg, 409
reg_or_imm, 413
reg_plus_imm, 412
regaddr, 413
register reference instructions, data flow order
constraints, 311
register window, 33, 34
register window management instructions, 21
register windows
clean, 59, 61, 84, 358, 362, 363, 364
fill, 36, 60, 61, 84, 85, 230, 232, 239, 360, 363, 364
management of, 15
overlapping, 35–37
spill, 36, 60, 61, 84, 85, 237, 239, 362, 363, 364
registers
See also individual register (common) names
accessing MMU registers, 380
address space identifier (ASI), 309
ASI (address space identifier), 51
chip-level multithreading, See CMT
clean windows (CLEANWIN), 59
clock-tick (TICK), 361
current window pointer (CWP), 59
F (floating point), 291, 348
floating-point, 17
programming, 41
floating-point registers state (FPRS), 53
floating-point state (FSR), 42
genral status (GSR), 54
global, 14, 17, 32, 33, 39, 397
global level (GL), 69
IER (SPARC V8), 286
in, 33, 35, 237
local, 33, 35
next program counter (NPC)
and PSTATE.tct, 65
next program counter (NPC), 53
other windows (OTHERWIN), 60
out, 33, 35, 237
out #7, 37
processor interrupt level (PIL)
and SOFTINT, 55
and STICK_CMPR, 58
and TICK_CMPR, 56
processor interrupt level (PIL), 69
program counter (PC)
and PSTATE.tct, 65
program counter (PC), 52
PSR (SPARC V8), 286
R register #15, 37
renaming mechanism, 311
restorable windows (CANRESTORE), 59, 59
savable windows (CANSAVE), 59
scratchpad
privileged, 334
SOFTINT, 49
SOFTINT_CLR pseudo-register, 49, 56
SOFTINT_SET pseudo-register, 49, 55
STICK, 57
TICK_CMPR
ASR summary, 49
int_dis field, 55, 58
stick_cmp field, 58
and system software trapping, 57
TBR (SPARC V8), 286
TICK, 52
TICK_CMPR
int_dis field, 55, 56
tick_cmp field, 56
TICK_CMPR, 49, 56
trap base address (TBA), 64
trap base address, See registers: TBA
trap level (TL), 68
trap level, See registers: TL
trap next program counter (TNPC), 62
trap next program counter, See registers: TNPC
trap program counter (TPC), 61
trap program counter, See registers: TPC
trap state (TSTATE), 63
trap state, See registers: TSTATE
trap type (TT), 64, 349

trap type, See registers: TT

VA_WATCHPOINT, 361, 362

visible to software in privileged mode, 61–70

WIM (SPARC V8), 286

window state (WSTATE), 60

window state, See registers: WSTATE

Y (32-bit multiply/divide), 50

relaxed memory order (RMO) memory model, 203, 313

renaming mechanism, register, 311

reorder unit, 310

reordering instruction, 311

reserved, 10

fields in instructions, 96

register field, 32

reset

reset trap, 345

restartable deferred trap, 344

restorable windows register, See CANRESTORE register

RESTORE instruction, 35, 230–231

actions, 84

and current window, 37

decremented CWP register, 35

followed by SAVE instruction, 36

managing register windows, 21

operation, 230

performance trade-off, 230, 237

and restorable windows (CANRESTORE) register, 59

restoring register window, 230

role in register state partitioning, 60, 61

restore synthetic instruction, 415

RESTORED instruction, 85, 232

creating inconsistent window state, 232

fill trap handler, 230

fill trap handler, 85, 364

register window management, 21

restricted, 10

restricted address space identifier, 78

restricted ASI, 309, 321

resumable_error exception, 361

ret/ret1 synthetic instructions, 415

RETRY instruction, 233

and restartable deferred traps, 345

effect on TNPC register, 62

effect on TPC register, 62

effect on TSTATE register, 63

generating illegal_instruction exception, 361

modifying CCR.xcc, 51

reexecuting trapped instruction, 364

restoring gl value in gl, 70

return from trap, 341

returning to instruction after trap, 347

target address, return from privileged traps, 20

RETURN instruction, 235–236

computing target address, 20

fill trap, 360

mem_address_not_aligned exception, 361

operation, 235

reexecuting trapped instruction, 235

RETURN vs. RESTORE instructions, 235

RMO, 10

RMO, See relaxed memory order (RMO) memory model

rounding

for floating-point results, 43

in signed division, 240

rounding direction (rd) field of FSR register, 120, 130, 151, 166, 167, 168, 170, 171

routine, nonleaf, 174

rs1 instruction field

arithmetic instructions, 97, 108, 109, 112, 209, 211, 240, 246, 281, 283

branch instructions, 109

floating point arithmetic instructions, 120, 130, 151

floating point compare instructions, 128

floating point load instructions, 181, 183, 186, 199

flush memory instruction, 133

jump-and-link instruction, 174

load instructions, 175, 190, 191, 192, 194

logical operation instructions, 100, 214, 290

move instructions, 207

PREFETCH, 219

RETURN, 235

rs2 instruction field

arithmetic instructions, 97, 108, 109, 112, 209, 211, 240, 246, 281, 283

floating point arithmetic instructions, 120, 130, 151, 166

floating point compare instructions, 128

floating point conversion instructions, 167, 168, 171

floating point instructions, 119

floating point integer conversion, 132

floating point load instructions, 181, 183, 186, 199

floating point move instructions, 139, 140

floating point negate instructions, 152

flush memory instruction, 133

jump-and-link instruction, 174

load instructions, 175, 192, 194

logical operation instructions, 100, 290

move instructions, 205, 207

POPC, 217

PREFETCH, 219

RTO, 10

RTS, 10

S

savable windows register, See CANSAVE register

SAVE instruction, 35, 237

actions, 84

after RESTORE instruction, 235

clean_window exception, 358, 363

and current window, 37

decremented CWP register, 35

effect on privileged state, 237

leaf procedure, 174

and local/out registers of register window, 36

managing register windows, 21

no clean window available, 59

number of usable windows, 59

operation, 237
performance trade-off, 237
role in register state partitioning, 60, 61
and savable windows (CANSAVE) register, 59
spill trap, 362, 363, 364
save synthetic instruction, 415
SAVED instruction, 84, 239
creating inconsistent window state, 239
register window management, 21
spill handler, 238, 239
spill trap handler, 84, 364
scaling of the coefficient, 147
scratchpad registers
privileged, 334
SDIV instruction, 50, 240
SDIVcc instruction, 50, 240
SDIVX instruction, 211
Secondary Context ID 0, 380
Secondary Context ID 1, 380
Secondary Context register, 381
self-consistency, processor, 311
self-modifying code, 133, 134, 317
sequencing MEMBAR instructions, 78
sequential consistency, 307, 313
sequential consistency memory model, 313
SETHI instruction, 79, 242
creating 32-bit constant in R register, 19
and NOP instruction, 212
with rd = 0, 242
setn synthetic instructions, 415
shall (keyword), 10
shared memory, 305
shift count encodings, 244
shift instructions, 19
shift instructions, 79, 244
short floating-point load and store instructions, 333
short floating-point load instructions, 188
short floating-point store instructions, 263
should (keyword), 10
SIAM instruction, 243
side effect
accesses, 307
definition, 10
I/O locations, 306
instruction prefetching, 307
real memory storage, 306
visible, 306
side-effect page, illegal access to, 359
signalling NaN (not-a-number), 42, 168
signed integer data type, 23
signx synthetic instructions, 415
SIMD, 10
instruction data formats, 29–30
simm10 instruction field
move instructions, 207
simm11 instruction field
move instructions, 205
simm13 instruction field
floating point
load instructions, 181, 199
simm13 instruction field
arithmetic instructions, 209, 211, 214, 240, 246, 281, 283
floating point load instructions, 183, 186
flush memory instruction, 133
jump-and-link instruction, 174
load instructions, 175, 190, 191, 192, 194
logical operation instructions, 100, 290
POPC, 217
PREFETCH, 219
RETURN, 235
single instruction/multiple data, See SIMD
SLL instruction, 244
SLLX instruction, 244
SMUL instruction, 50, 246
SMULcc instruction, 50, 246
SOFTINT register, 49, 54
clearing, 366
clearing of selected bits, 56
communication from nucleus code to kernel code, 366
scheduling interrupt vectors, 365, 366
setting, 366
SOFTINT register fields
int_level, 55
sm (stick_int), 55
tm (tick_int), 55
tm (tm), 56
SOFTINT_CLR pseudo-register, 49, 56
SOFTINT_SET pseudo-register, 49, 55, 55
software
nucleus, 9
software translation table, 371
software trap, 277, 349
software trap number (SWTN), 277
software, nonprivileged, 53
software_trap_number, 414
source operands, 158, 161
SPA
ASI_TWIN_DW_NUCLEUS, 335
SPARC V8 compatibility
LD, LDUW instructions, 175
operations to I/O locations, 307
read state register instructions, 226
STA instruction renamed, 249
STBAR instruction, 202
STD instruction, 265
STDA instruction, 268
tagged subtract instructions, 280
UNIMP instruction renamed, 172
window_overflow exception superseded, 360
write state register instructions, 286
SPARC V9
compliance, 9, 379
features, 13
SPARC V9 Application Binary Interface (ABI), 15
speculative load, 11
spill register window, 362
FLUSH instruction, 85
overflow/underflow, 36
RESTORE instruction, 84
SAVE instruction, 61, 84, 237, 363
SAVED instruction, 84, 239, 364
selection of, 363
trap handling, 364
trap vectors, 237, 364
window state, 60
spill_n_normal exception, 238, 362
and FLUSHW instruction, 136
spill_n_other exception, 238, 362
and FLUSHW instruction, 136
SRA instruction, 244
SRAX instruction, 244
SRL instruction, 244
SRLX instruction, 244
stack frame, 237
state registers (ASRs), 48–58
STB instruction, 247
STBA instruction, 248
STBAR instruction, 226, 286, 311, 317
STBLOCKF instruction, 250, 333
STDF instruction, 73, 253, 362
STDF_mem_address_not_aligned exception, 362
and store instructions, 254, 257
STDF/STDFA instruction, 73
STDFA instruction, 255
alignment, 73
ASIs for fp store operations, 333
causing DAE_invalid_ASI exception, 333
cauusing mem_address_not_aligned or illegal_instruction
exception, 333
cauusing STDF_mem_address_not_aligned exception, 73, 362
for block load operations, 333
for partial store operations, 333
used with ASIs, 333
STF instruction, 253
STFA instruction, 255
STFSR instruction, 42, 44, 361
STH instruction, 247
STHA instruction, 248
STICK register, 49, 52, 57
counter field, 57
npt field, 52, 57
RDSTICK instruction, 225
STICK_CMPR register, 49, 57
int_dis field, 55, 58
RDSTICK_CMPR instruction, 225
stick_cmp field, 58
store
block, See block store instructions
partial, See partial store instructions
short floating-point, See short floating-point store instructions
store buffer
merging, 307
store floating-point into alternate space instructions, 255
store instructions, 11, 73
StoreLoad MEMBAR relationship, 201, 317
StoreLoad predefined constant, 413
stores to alternate space, 19, 51, 76
StoreStore MEMBAR relationship, 201, 317
StoreStore predefined constant, 413
STPARTIALF instruction, 260
STPARTIALF instruction, DAE_nc_page exception, 359
STQF instruction, 74, 253, 362
STQF_mem_address_not_aligned exception, 362
STQF/STQFA instruction, 74
STQFA instruction, 74, 255
strand, 11
strong consistency memory model, 313
strong ordering, 313
Strong Sequential Order, 314
STSHORTF instruction, 263
STTW instruction, 38, 73
STTWA instruction (deprecated), 265
STTWA instruction, 38, 73
STTWA instruction (deprecated), 267
STW instruction, 247
STWA instruction, 248
STX instruction, 247
STXA instruction, 248
accessing nontranslating ASIs, 268
mem_address_not_aligned exception, 248
referencing internal ASIs, 314
STXFSR instruction, 42, 44, 269, 361
SUB instruction, 270, 270
SUBC instruction, 270, 270
SUBc instruction, 79, 270, 270
SUBCc instruction, 270, 270
subnormal number, 11
subtract instructions, 270
superscalar, 11
supervisor software
accessing special protected registers, 18
definition, 11
SWAP instruction, 18, 271
accessing doubleword simultaneously with other instructions, 272
and DAE_nc_page exception, 359
hardware primitive for mutual exclusion, 316
identification of R register to be exchanged, 73
in multiprocessor system, 190, 191
memory accessing, 271
ordering by MEMBAR, 317
swap R register
bit contents, 112
with alternate space memory instructions, 272
with memory instructions, 271
SWAPA instruction, 272
accessing doubleword simultaneously with other instructions, 272
alternate space addressing, 18
and DAE_nc_page exception, 359
hardware primitive for mutual exclusion, 316
in multiprocessor system, 190, 191
ordering by MEMBAR, 317
SWTN (software trap number), 277
Sync predefined constant, 413
synchronization, 203
synchronization, 11
synthetic instructions
mapping to SPARC V9 instructions, 414–416
for assembly language programmers, 414
mapping
bclrg, 416
bset, 416
btog, 416
btst, 416
call, 414
cast, 416
clnr, 416
cmp, 414
dec, 416
dec, 416
inc, 416
inc, 416
iprefetch, 415
jmp, 414
mov, 416
neg, 416
not, 416
restore, 415
ret/retl, 415
save, 415
setn, 415
signx, 415
tst, 415

vs. pseudo ops, 414
system clock-tick register (STICK), 57
system software
accessing memory space by server program, 309
ASIs allowing access to memory space, 310
FLUSH instruction, 135, 319
processing exceptions, 309
trap types from which software must recover, 44
System Tick Compare register, See STICK_CMPR register
System Tick register, See STICK register

T
TA instruction, 276, 389
TADDcc instruction, 79, 274
TADDcTTV instruction, 79, 362
tag overflow, 79
tag_overflow exception, 79, 274, 275, 279, 280
tag_overflow exception (deprecated), 362
tagged arithmetic, 79
tagged arithmetic instructions, 20
tagged word data format, 23
tagged words, 23
TBA (trap base address) register, 64, 342
establishing table address, 21, 341
initialization, 348
specification for RDPR instruction, 228
specification for WRPR instruction, 288
trap behavior, 11
TBR register (SPARC V8), 286
TCC instruction, 276
Tcc instructions, 276
at TL > 0, 349
causing trap, 341
causing trap to privileged trap handler, 349
CCR register bits, 51
generating htrap_instruction exception, 360
generating illegal_instruction exception, 360
generating trap_instruction exception, 362
opcode maps, 385, 389, 390
programming uses, 277
trap table space, 21
vector through trap table, 341
TCS instruction, 276, 389
TE instruction, 276, 389
termination deferred trap, 344
test-and-set instruction, 316
TG instruction, 276, 389
TGE instruction, 276, 389
TGU instruction, 276, 389
thread, 11
TICK register, 49
counter field, 52, 400, 408
inaccuracies between two readings of, 400, 408
specification for RDPR instruction, 228
TICK_CMPR register, 49, 56
int_dis field, 55, 56
tick_cmp field, 56
timer registers, See TICK register and STICK register
timing of instructions, 96
tininess (floating-point), 48
TL (trap level) register, 68, 342
affect on privilege level to which a trap is delivered, 348
and implicit ASIs, 76
displacement in trap table, 341
executing RESTORED instruction, 232
executing SAVED instruction, 239
indexing for WRPR instruction, 288
indexing privileged register after RDPR, 228
setting register value after WRPR, 288
specification for RDPR instruction, 228
specification for WRPR instruction, 288
and TBA register, 348
and TPC register, 61
and TSTATE register, 63
and TT register, 64
use in calculating privileged trap vector address, 348
and WSTATE register, 60
TL instruction, 276, 389
TLB
and 3-dimensional arrays, 103
miss
reloading TLB, 371, 376
specialized miss handler code, 380
TLE instruction, 276, 389
TLEU instruction, 276, 389
TN instruction, 276, 389
TNEG instruction, 276, 389
TNPC (trap-saved next program counter) register, 62
saving NPC, 344
specification for RDPR instruction, 228
specification for WRPR instruction, 288
TNPC (trap-saved next program counter) register, 11
total order, 312
total store order (TSO) memory model, 66, 203, 306, 313, 314
TPC (trap program counter) register, 11, 61
   address of trapping instruction, 229
   number of instances, 61
   specification for RDPR instructions, 228
   specification for WRPR instruction, 288
TPOS instruction, 276, 389
   translating ASI, 322
Translation Table Entry, See TTE
   trap
      See also exceptions and traps
      noncacheable accesses, 307
      when taken, 11
   trap enable mask (tem) field of FSR register, 347, 348, 398
   trap handler
      privileged mode, 349
      regular/nonfaulting loads, 8
      returning from, 114, 233
   user, 44, 293
   trap level register, See TL register
   trap next program counter register, See TNPC register
   Trap on Control Transfer
      and instructions
         Bicc, 105
         BFcc, 108
         BPr, 109
         CALL, 111
         DONE, 115, 233, 277
         FBfcx, 123, 125
         Jmpl, 174
   tct field of PSTATE register, 65
   trap on integer condition codes instructions, 276
   Trap Overflow Enable bit, 339
   trap program counter register, See TPC register
   trap state register, See TSTATE register
   trap type (TT) register, 349
   trap type register, See TT register
   trap_instruction (ISA) exception, 277, 278, 362
   trap_little_endian (tle) field of PSTATE register, 65
   traps, 11
      See also exceptions and individual trap names
      categories
         deferred, 343, 344, 345
         disrupting, 343, 345
         precise, 343, 344, 345
         priority, 347, 356
         reset, 344, 345
         restartable
            implementation dependency, 345
            restartable deferred, 344
            termination deferred, 344
      caused by undefined feature/behavior, 12
      causes, 21, 21
      definition, 21, 341
      hardware, 349
      hardware stack, 14
      level specification, 68
      model stipulations, 347
      nested, 14
   normal, 349
   processing, 356
   software, 277, 349
   stack, 356
   vector address, specifying, 64
   TSB, 11, 376
      cacheability, 377
      caching, 377
      indexing support, 376
      organization, 377
   TSO, 11
      TSO, See total store order (TSO) memory model
   tst synthetic instruction, 415
   TSTATE (trap state) register, 63
      DONE instruction, 114, 233
      registers saved after trap, 21
      restoring gl value, 70
      specification for RDPR instruction, 228
      specification for WRPR instruction, 288
   tstate, See trap state (TSTATE) register
   TSUBcc instruction, 79, 279
   TSUBccTV instruction, 79, 362
   TT (trap type) register, 64
      and privileged trap vector address, 348
      reserved values, 399
      specification for RDPR instruction, 228
      specification for WRPR instruction, 288
      and Tcc instructions, 277
      transferring trap control, 349
      window spill/fill exceptions, 60
   WRPR instruction, 288
   TTE, 11
      context ID field, 374
      cp (cacheability) field, 306
      cp field, 359, 375, 375
      cv field, 375, 375
      e field, 306, 319, 359, 375
      ie field, 375
      indexing support, 376
      nfo field, 319, 359, 374, 375
      p field, 359, 376
      size field, 376
      soft2 field, 374
      Sparc V8 equivalence, 373
      taddr field, 374
      va_tag field, 374
      w field, 376
   TVC instruction, 276, 389
   TVS instruction, 276, 389
   typewriter font, in assembly language syntax, 409
   U
   UDIV instruction, 50, 281
   UDIVcc instruction, 50, 281
   UDIVX instruction, 211
   uf (underflow mask) field of FSR,tem, 47
   UltraSPARC, previous ASIs
      ASI_NUCLEUS_QUAD_LDD (deprecated), 335
ASI_NUCLEUS_QUAD_LDD_L (deprecated), 335
ASI_NUCLEUS_QUAD_LDD_LITTLE (deprecated), 335
ASI_PHY_BYPASS_EC_WITH_EBIT_L, 334
ASI_PHY_BYPASS_EC_WITH_EBIT, 334
ASI_PHY_BYPASS_EC_WITH_EBIT_LITTLE, 334
ASI_PHYS_USE_EC, 334
ASI_PHYS_USE_EC_L, 334
ASI_PHYS_USE_EC_LITTLE, 334
UMUL instruction, 50
UMUL instruction (deprecated), 283
UMULcc instruction, 50
UMULcc instruction (deprecated), 283
unassigned, 11
unconditional branches, 104, 108, 122, 125
undefined, 12
underflow
bits of FSR register
  accrued (ufa) bit of aexc field, 48, 292
  current (ufc) bit of cexc, 48
  current (ufc) bit of ceicc field, 292
  mask (ufm) bit of tsr.tem, 48
  mask (ufm) bit of tem field, 293
detection, 36
occurrence, 363
underflow mask (ufm) field of FSR.tem, 47
unfinished_FFPop floating-point trap type, 45, 120, 130, 151, 166, 168, 170, 291
handling, 48
  in normal computation, 44
  results after recovery, 44
UNIMP instruction (SPARC V8), 172
unimplemented, 12
unimplemented_FPop floating-point trap type, 292
handling, 48
unimplemented_LDTW exception, 193, 362
unimplemented_STTW exception, 266, 362
uniprocessor system, 12
unrestricted, 12
unrestricted ASI, 321
unsigned integer data type, 23
user application program, 12
user trap handler, 44, 293

V
VA, 12
VA_watchpoint exception, 362
VA_WATCHPOINT register, 361, 362
value clipping, See FPACK instructions
value semantics of input/output (I/O) locations, 306
virtual
  address, 305
  address 0, 319
virtual address, 12
virtual core, 12
virtual memory, 223
VIS, 12
VIS instructions
  encoding, 391, 392
  implicitly referencing GSR register, 54
Visual Instruction Set, See VIS instructions

W
W-A-R, See write-after-read memory hazard
watchpoint comparator, 67
W-A-W, See write-after-write memory hazard
WM register (SPARC V8), 286
window fill exception, See also fill_n_normal exception
window fill trap handler, 21
window overflow, 36, 363
window spill exception, See also spill_n_normal exception
window spill trap handler, 21
window state register, See WSTATE register
window underflow, 363
window, clean, 237
window_fill exception, 60, 84
  RETURN, 235
window_spill exception, 60
word, 12
  alignment, 18, 73, 308
  data format, 23
WRASI instruction, 49, 51, 285
WRasr instruction, 285
  accessing I/O registers, 19
  attempt to write to ASR 5 (PC), 52
  cannot write to PC register, 52
  implementation dependencies, 399
  writing ASRs, 48
WRCCR instruction, 49, 50, 51, 285
WRFPRS instruction, 49, 53, 285
WRGSR instruction, 49, 54, 285
WRIER instruction (SPARC V8), 286
write ancillary state register (WRasr) instructions, 285
write ancillary state register instructions, See WRasr instruction
write privileged register instruction, 288
write-after-read memory hazard, 311
write-after-write memory hazard, 311
WRPR instruction
  accessing non-register-window PR state registers, 61
  accessing register-window PR state registers, 58
  and register-window PR state registers, 58
  effect on TNPC register, 62
  effect on TPC register, 62
  effect on TSTATE register, 63
  effect on TT register, 64
  writing the TICK register, 52
  writing to GL register, 70
  writing to PSTATE register, 65
  writing to TICK register, 52
WRPSR instruction (SPARC V8), 286
WRSOFTINT instruction, 49, 54, 285
WRSOFTINT_CLR instruction, 49, 54, 56, 285, 366
WRSOFTINT_SET instruction, 49, 54, 55, 285, 366
WRSTICK_CMPR instruction, 49, 285
WRTICK_CMP instruction, 49, 285
WRTRBR instruction (SPARC V8), 286
WRTCMP instruction, 49, 285
WRWIM instruction (SPARC V8), 286
WRY instruction, 49, 50, 285
WSTATE (window state) register
   description, 60
   and fill/spill exceptions, 363
   normal field, 363
   other field, 363
   overview, 58
   reading with RDPR instruction, 228
   spill exception, 136
   spill trap, 237
   writing with WRPR instruction, 288

X
XNOR instruction, 290
XNORcc instruction, 290
XOR instruction, 290
XORcc instruction, 290

Y
Y register, 49, 50
   after multiplication completed, 209
   content after divide operation, 240, 281
   divide operation, 240, 281
   multiplication, 209
   unsigned multiply results, 246, 283
   WRY instruction, 286
Y register (deprecated), 50

Z
zero virtual address, 319