UltraSPARC Architecture 2005

Draft D0.8.7, 27 Mar 2006

Privilege Levels: Privileged and Nonprivileged

Distribution: Public

Some portions of this specification are undergoing final review; please check monthly to see if an updated revision is available for download.
Contents

1 Preface .............................................................. i
  1.1 What’s New?.................................................... i
  1.2 Acknowledgements .............................................. ii

2 Document Overview .............................................. 1
  2.1 Navigating UltraSPARC Architecture 2005 ................... 1
  2.2 Fonts and Notational Conventions ............................ 2
    2.2.1 Implementation Dependencies .......................... 4
    2.2.2 Notation for Numbers .................................... 4
    2.2.3 Informational Notes ................................... 4
  2.3 Reporting Errors in this Specification ....................... 5

3 Definitions ...................................................... 7

4 Architecture Overview ........................................ 19
  4.1 The UltraSPARC Architecture 2005 .......................... 20
    4.1.1 Features ..................................................... 20
    4.1.2 Attributes .................................................. 21
      4.1.2.1 Design Goals ......................................... 21
      4.1.2.2 Register Windows .................................... 22
    4.1.3 System Components ....................................... 22
      4.1.3.1 Binary Compatibility ................................. 22
      4.1.3.2 UltraSPARC Architecture 2005 MMU .................. 22
      4.1.3.3 Privileged Software ................................ 23
    4.1.4 Architectural Definition ................................ 23
    4.1.5 UltraSPARC Architecture 2005 Compliance with SPARC V9
        Architecture ................................................. 23
    4.1.6 Implementation Compliance with UltraSPARC Architecture 2005
        23
  4.2 Processor Architecture ...................................... 24
    4.2.1 Integer Unit (IU) .......................................... 24
6.3 Floating-Point Registers ........................................... 52
  6.3.1 Floating-Point Register Number Encoding ................. 55
  6.3.2 Double and Quad Floating-Point Operands ................. 56

6.4 Floating-Point State Register (FSR) ........................... 58
  6.4.1 Floating-Point Condition Codes (fcc0, fcc1, fcc2, fcc3) . .... 58
  6.4.2 Rounding Direction (rd) ................................... 59
  6.4.3 Trap Enable Mask (tem) .................................. 59
  6.4.4 Nonstandard Floating-Point (ns) ........................... 60
  6.4.5 FPU Version (ver) ....................................... 60
  6.4.6 Floating-Point Trap Type (ftt) ............................ 60
  6.4.7 FQ Not Empty (qne) ...................................... 63
  6.4.8 Accrued Exceptions (aexc) ................................ 63
  6.4.9 Current Exception (cexc) .................................. 64
  6.4.10 Floating-Point Exception Fields ........................... 65
  6.4.11 FSR Conformance ........................................ 67

6.5 Ancillary State Registers ........................................ 67
  6.5.1 32-bit Multiply/Divide Register (Y) (ASR 0) ............. 69
  6.5.2 Integer Condition Codes Register (CCR) (ASR 2) .......... 69
    6.5.2.1 Condition Codes (CCR.xcc and CCR.icc) ............ 70
  6.5.3 Address Space Identifier (ASI) Register (ASR 3) .......... 71
  6.5.4 Tick (TICK) Register (ASR 4) ............................. 71
  6.5.5 Program Counters (PC, NPC) (ASR 5) ...................... 72
  6.5.6 Floating-Point Registers State (FPRS) Register (ASR 6) ... 73
  6.5.7 Performance Control Register (PCR) (ASR 16) .......... 74
  6.5.8 Performance Instrumentation Counter (PIC) Register (ASR 17) .... 75
  6.5.9 General Status Register (GSR) (ASR 19) ................... 76
  6.5.10 SOFTINTp Register (ASRs 20, 21, 22) ................. 77
    6.5.10.1 SOFTINT_SETp Pseudo-Register (ASR 20) ....... 78
    6.5.10.2 SOFTINT_CLRp Pseudo-Register (ASR 21) ....... 79
  6.5.11 Tick Compare (TICK_CMPRP) Register (ASR 23) .......... 79
  6.5.12 System Tick (STICK) Register (ASR 24) ............... 80
  6.5.13 System Tick Compare (STICK_CMPRP) Register (ASR 25) ... 81

6.6 Register-Window PR State Registers ............................ 81
  6.6.1 Current Window Pointer (CWPp) Register (PR 9) .......... 82
  6.6.2 Savable Windows (CANSAVEp) Register (PR 10) ........... 83
  6.6.3 Restorable Windows (CANRESTOREp) Register (PR 11) .... 83
  6.6.4 Clean Windows (CLEANWINp) Register (PR 12) .......... 83
  6.6.5 Other Windows (OTHERWINp) Register (PR 13) .......... 84
  6.6.6 Window State (WSTATEp) Register (PR 14) .............. 84
  6.6.7 Register Window Management .............................. 84
    6.6.7.1 Register Window State Definition .................... 85
    6.6.7.2 Register Window Traps ................................ 86

6.7 Non-Register-Window PR State Registers ........................ 86
  6.7.1 Trap Program Counter (TPCp) Register (PR 0) ............ 86
  6.7.2 Trap Next PC (TNPCp) Register (PR 1) .................... 87
  6.7.3 Trap State (TSTATEp) Register (PR 2) .................... 88
6.7.4 Trap Type (TT\textsuperscript{T}) Register (PR 3) 89
6.7.5 Trap Base Address (TBA\textsuperscript{T}) Register (PR 5) 89
6.7.6 Processor State (PSTATE\textsuperscript{T}) Register (PR 6) 90
6.7.7 Trap Level Register (TL\textsuperscript{T}) (PR 7) 94
6.7.8 Processor Interrupt Level (PIL\textsuperscript{T}) Register (PR 8) 95
6.7.9 Global Level Register (GL\textsuperscript{T}) (PR 16) 96

7 Instruction Set Overview 99
7.1 Instruction Execution 99
7.2 Instruction Formats 100
7.3 Instruction Categories 101
7.3.1 Memory Access Instructions 101
7.3.1.1 Memory Alignment Restrictions 102
7.3.1.2 Addressing Conventions 103
7.3.1.3 Address Space Identifiers (ASIs) 108
7.3.1.4 Separate Instruction Memory 109
7.3.2 Memory Synchronization Instructions 110
7.3.3 Integer Arithmetic and Logical Instructions 110
7.3.3.1 Setting Condition Codes 110
7.3.3.2 Shift Instructions 110
7.3.3.3 Set High 22 Bits of Low Word 110
7.3.3.4 Integer Multiply/Divide 111
7.3.3.5 Tagged Add/Subtract 111
7.3.4 Control-Transfer Instructions (CTIs) 111
7.3.4.1 Conditional Branches 113
7.3.4.2 Unconditional Branches 113
7.3.4.3 CALL and JMPL Instructions 114
7.3.4.4 RETURN Instruction 114
7.3.4.5 DONE and RETRY Instructions 114
7.3.4.6 Trap Instruction (Tcc) 114
7.3.4.7 DCTI Couples 115
7.3.5 Conditional Move Instructions 115
7.3.6 Register Window Management Instructions 116
7.3.6.1 SAVE Instruction 116
7.3.6.2 RESTORE Instruction 117
7.3.6.3 SAVED Instruction 118
7.3.6.4 RESTORED Instruction 118
7.3.6.5 Flush Windows Instruction 118
7.3.7 Ancillary State Register (ASR) Access 118
7.3.8 Privileged Register Access 119
7.3.9 Floating-Point Operate (FPop) Instructions 119
7.3.10 Implementation-Dependent Instructions 120
7.3.11 Reserved Opcodes and Instruction Fields 120

8 Instructions 123
8.30.1 FMUL8x16 Instruction 189
8.30.2 FMUL8x16AU Instruction ............................... 190
8.30.3 FMUL8x16AL Instruction ............................... 190
8.30.4 FMUL8SUx16 Instruction ............................... 191
8.30.5 FMUL8ULx16 Instruction ............................... 191
8.30.6 FMULD8SUx16 Instruction ............................... 192
8.30.7 FMULD8ULx16 Instruction ............................... 193
8.33.1 FPACK16 ............................................. 198
8.33.2 FPACK32 ............................................. 199
8.33.3 FPACKFIX ............................................. 201
8.46.1 IMPDEP1 Opcodes ....................................... 223
8.46.1.1 Opcode Formats ..................................... 224
8.46.2 IMDEP2B Opcodes ....................................... 224
8.61.1 Memory Synchronization .................................. 260
8.61.2 Synchronization of the Virtual Processor ................. 261
8.61.3 TSO Ordering Rules affecting Use of MEMBAR ............ 261
8.72.1 Exceptions ............................................. 280
8.72.2 Weak versus Strong Prefetches ............................ 281
8.72.3 Prefetch Variants ....................................... 281
8.72.3.1 Prefetch for Several Reads (fcn = 0, 20(1416)) ........ 282
8.72.3.2 Prefetch for One Read (fcn = 1, 21(1516)) ............ 282
8.72.3.3 Prefetch for Several Writes (and Possibly Reads) (fcn = 2, 22(1616)) 282
8.72.3.4 Prefetch for One Write (fcn = 3, 23(1716)) .......... 283
8.72.3.5 Prefetch Page (fcn = 4) ................................ 283
8.72.4 Implementation-Dependent Prefetch Variants (fcn = 16, 18, 19, and 24–31) 283
8.72.5 Additional Notes ....................................... 284
9 IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005 ........................................ 359
9.1 Traps Inhibiting Results ..................................... 359
9.2 NaN Operand and Result Definitions ......................... 360
9.2.1 Untrapped Result in Different Format from Operands ........ 360
9.2.2 Untrapped Result in Same Format as Operands .......... 361
9.3 Trapped Underflow Definition (ufm = 1) ....................... 362
9.4 Untrapped Underflow Definition (ufm = 0) .................... 362
9.5 Integer Overflow Definition .................................. 363
9.6 Floating-Point Nonstandard Mode ............................. 364
9 Memory ..................................................... 365
9.1 Memory Location Identification ............................... 366
9.2 Memory Accesses and Cacheability ........................... 366
9.2.1 Coherence Domains ....................................... 366
9.2.1.1 Cacheable Accesses .................................... 367
9.2.1.2 Noncacheable Accesses ................................. 367
9.2.1.3 Noncacheable Accesses with Side-Effect .................. 367
9.3 Memory Addressing and Alternate Address Spaces ........... 369

* Contents  v
9.3.1 Memory Addressing Types .......................... 369
9.3.2 Memory Address Spaces ............................. 370
9.3.3 Address Space Identifiers ........................... 370

9.4 SPARC V9 Memory Model .............................. 372
9.4.1 SPARC V9 Program Execution Model ................. 373
9.4.2 Virtual Processor/Memory Interface Model .......... 375

9.5 The UltraSPARC Architecture Memory Model — TSO ........ 376
9.5.1 Memory Model Selection ............................. 377
9.5.2 Programmer-Visible Properties of the UltraSPARC Architecture TSO Model 377
9.5.3 TSO Ordering Rules ................................. 378
9.5.4 Hardware Primitives for Mutual Exclusion .......... 379
9.5.4.1 Compare-and-Swap (CASA, CASXA) ............... 380
9.5.4.2 Swap (SWAP) .................................. 380
9.5.4.3 Load Store Unsigned Byte (LDSTUB) .......... 380
9.5.5 Memory Ordering and Synchronization ............... 381
9.5.5.1 Ordering MEMBAR Instructions .................. 381
9.5.5.2 Sequencing MEMBAR Instructions ................. 382
9.5.5.3 Synchronizing Instruction and Data Memory .... 383

9.6 Nonfaulting Load ...................................... 384
9.7 Store Coalescing ...................................... 385

10 Address Space Identifiers (ASIs) ........................ 387
10.1 Address Space Identifiers and Address Spaces ........... 387
10.2 ASI Values .............................................. 387
10.3 ASI Assignments ........................................ 388
10.3.1 Supported ASIs ...................................... 389
10.4 Special Memory Access ASIs ............................ 397
10.4.1 ASIs 10\_16, 11\_16, 16\_16, 17\_16 and 18\_16 (ASI\_\_*AS\_IF\_USER\_\_*) ........ 397
10.4.2 ASIs 18\_16, 19\_16, 1E\_16, and 1F\_16 (ASI\_\_*AS\_IF\_USER\_\_*\_LITTLE) ....... 398
10.4.3 ASI 14\_16 (ASI\_REAL) .............................. 399
10.4.4 ASI 15\_16 (ASI\_REAL\_IO) .......................... 399
10.4.5 ASI 1C\_16 (ASI\_REAL\_LITTLE) ...................... 400
10.4.6 ASI 1D\_16 (ASI\_REAL\_IO\_LITTLE) ................. 400
10.4.7 ASIs 22\_16, 23\_16, 27\_16, 2A\_16, 2B\_16, 2F\_16 (Privileged Load Integer Twin Extended Word) 400
10.4.8 ASIs 26\_16 and 2E\_16 (Privileged Load Integer Twin Extended Word, Real Addressing) 401
10.4.9 ASIs E2\_16, E3\_16, EA\_16, EB\_16 (Nonprivileged Load Integer Twin Extended Word) 402
10.4.10 Block Load and Store ASIs ......................... 403
10.4.11 Partial Store ASIs .................................. 404
10.4.12 Short Floating-Point Load and Store ASIs ........... 404
10.5 ASI-Accessible Registers ............................... 404
11 Performance Instrumentation ........................................... 407

12 Traps ........................................................................... 409
   12.1 Virtual Processor Privilege Modes ............................ 410
   12.2 Virtual Processor States and Traps .......................... 412
       12.2.0.1 Usage of Trap Levels ............................... 412
   12.3 Trap Categories .................................................... 412
       12.3.1 Precise Traps ................................................. 412
       12.3.2 Deferred Traps .............................................. 413
       12.3.3 Disrupting Traps ............................................ 415
           12.3.3.1 Disrupting versus Precise and Deferred Traps . 415
           12.3.3.2 Causes of Disrupting Traps ....................... 415
           12.3.3.3 Conditioning of Disrupting Traps ................ 415
           12.3.3.4 Trap Handler Actions for Disrupting Traps ...... 416
       12.3.4 Uses of the Trap Categories ............................. 417
   12.4 Trap Control .......................................................... 418
       12.4.1 PIL Control .................................................. 418
       12.4.2 FSR_tem Control ........................................... 418
   12.5 Trap-Table Entry Addresses .................................... 418
       12.5.1 Trap-Table Entry Address to Privileged Mode ..... 419
       12.5.2 Privileged Trap Table Organization ................... 420
       12.5.3 Trap Type (TT) .............................................. 420
           12.5.3.1 Trap Type for Spi ll/Fill Traps .................. 428
       12.5.4 Trap Priorities .............................................. 428
   12.6 Trap Processing ...................................................... 429
       12.6.1 Normal Trap Processing .................................. 429
   12.7 Exception and Interrupt Descriptions ....................... 431
       12.7.1 SPARC V9 Traps Not Used in UltraSPARC Architecture 2005 436
   12.8 Register Window Traps .......................................... 436
       12.8.1 Window Spill and Fill Traps ............................ 436
       12.8.2 clean_window Trap ....................................... 437
       12.8.3 Vectoring of Fill/Spill Traps .......................... 437
       12.8.4 CWP on Window Traps .................................... 438
       12.8.5 Window Trap Handlers .................................... 438

13 Interrupt Handling ..................................................... 441
   13.1 Interrupt Packets .................................................. 442
   13.2 Software Interrupt Register (SOFTINT) ..................... 442
       13.2.1 Setting the Software Interrupt Register ............. 442
       13.2.2 Clearing the Software Interrupt Register ........... 443
   13.3 Interrupt Queues .................................................. 443
       13.3.1 Interrupt Queue Registers .............................. 443
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.

1.1 What’s New?

For the first time, UltraSPARC Architecture 2005 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:

- the VISTM 1 and VIS 2 instruction sets and GSR register
- multiple levels of global registers, controlled by the GL register
- MMU architecture

Plus, now architectural features are tagged with Software Classes and Implementation Classes1. 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.

### 1.2 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 2005* 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

I hope you find the UltraSPARC Architecture 2005 specification more complete, accurate, and readable than its predecessors.

— David Weaver  
UltraSPARC Architecture coordinator and specification editor

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

This chapter discusses:
- Fonts and Notational Conventions on page 2.
- Reporting Errors in this Specification on page 5.

2.1 Navigating UltraSPARC Architecture 2005

If you are new to the SPARC architecture, read Chapter 4, Architecture Overview, study the definitions in Chapter 3, 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 2005, note that UltraSPARC Architecture 2005 conforms to the SPARC V9 Level 1 architecture (and most of Level 2), with numerous extensions — particularly with respect to VIS instructions. For additional details, see the following:
- Chapter 3, Definitions
- Chapter 5, Data Formats, for a description of the supported data formats
- Chapter 6, Registers, for a description of the register set
- Chapter 7, Instruction Set Overview, for a description of the new instructions
- Chapter 8, Instructions, for descriptions of instruction set extensions
- Chapter 9, IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005, for a description of the trap model
- Chapter 9, Memory
- Chapter 10, Address Space Identifiers (ASIs), for a complete list of supported ASIs
- Chapter 11, PerformanceInstrumentation
Chapter 12, Traps, for a description of the trap model
Chapter 13, Interrupt Handling, for information on how interrupts are handled
Chapter 14, Memory Management
Appendix A, Opcode Maps, to see the overall pictures of how the instruction opcodes are mapped
Appendix B, Implementation Dependencies, for descriptions of resolutions of all implementation dependencies
Appendix C, Assembly Language Syntax, to see extensions to the assembly language syntax; in particular, synthetic instructions are documented in this appendix
Appendix D, Formal Specification of the Memory Models

2.2 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 HELVETICA* font is used for register names; for example, FSR.
- *TYPEWRITER* (Courier) font is used for literal values, such as code (assembly language, C language, ASI names) and for state names. For example: %f0, ASI_PRIMARY, execute_state.

When a register field is shown along with its containing register name, they are separated by a period (’.’), for example, “FSR.cexc”.

- *UPPERCASE* words are acronyms or instruction names. Some common acronyms appear in the glossary in Chapter 3, 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 FCMP_E<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.
2.2.1 Implementation Dependencies

Implementors of UltraSPARC Architecture 2005 processors are allowed to resolve some aspects of the architecture in machine-dependent ways. Each possible implementation dependency is indicated by the notation "IMPL. DEP. #nn: Some descriptive text." In this specification, the number nn enumerates the dependencies in . References to implementation dependencies are indicated by the notation "(impl. dep. #nn)".

2.2.2 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).

2.2.3 Informational Notes

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

<table>
<thead>
<tr>
<th>Note</th>
<th>General notes contain incidental information relevant to the paragraph preceding the note.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Note</td>
<td>Programming notes contain incidental information about how software can use an architectural feature.</td>
</tr>
<tr>
<td>Implementation Note</td>
<td>An Implementation Note contains incidental information, describing how an UltraSPARC Architecture 2005 processor might implement an architectural feature.</td>
</tr>
<tr>
<td>V9 Compatibility Note</td>
<td>Note containing information about possible differences between UltraSPARC Architecture 2005 and SPARC V9 implementations. Such information is relevant to UltraSPARC Architecture 2005 implementations and might not apply to other SPARC V9 implementations.</td>
</tr>
<tr>
<td>Forward Compatibility Note</td>
<td>Note containing information about how the UltraSPARC Architecture is expected to evolve in the future. Such notes are not intended as a guarantee that the architecture will evolve as indicated, but as a guide to features that should not be depended upon to remain the same, by software intended to run on both current and future implementations.</td>
</tr>
</tbody>
</table>
2.3 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 email address: UA-editor@sun.com
Definitions

This chapter defines concepts and terminology common to all implementations of UltraSPARC Architecture 2005.

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

address space identifier (ASI)  An 8-bit value that identifies an address space. For each instruction or data access, an ASI is associated with the address. See also implicit ASI.

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

ASI  Address space identifier.

ASR  Ancillary State register.

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

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

**coherence**  A set of 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.

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

**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.

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

**DCTI**  Delayed control transfer instruction.

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

**deprecated**  The term applied to an architectural feature (such as an instruction or register) for which an UltraSPARC Architecture implementation provides support only for compatibility with previous versions of the architecture. Use of a deprecated feature must generate correct results but may compromise software performance.

Deprecated features should not be used in new UltraSPARC Architecture software and may not be supported in future versions of the architecture.
dispatch  To send a previously fetched instruction to one or more functional units for execution. Typically, the instruction is dispatched from a reservation station or other buffer of instructions waiting to be executed. (Other conventions for this term exist, but the this specification attempts to use dispatch consistently as defined here). See also issued.

doublet  Two bytes (16 bits) of data.

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.

floating-point exception  An exception that occurs during the execution of a floating-point operate (FPop) instruction. The exceptions are unfinished_FPop, unimplemented_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 (FPop) instructions  Instructions that perform floating-point calculations, as defined in Floating-Point Operate (FPop) Instructions on page 119. 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.flt field.

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

FPop  See floating-point operate (FPop) instructions.

FPRS  Floating-Point Register State register.
FGU  Floating-point and Graphics Unit (which, in most implementations, is a synonym for FPU).

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.


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 2005 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  (1) A memory transaction (load, store, or atomic load-store) is said to be “issued” when a virtual processor has sent the transaction to the memory subsystem and the completion of the request is out of the virtual processor’s control. Synonym for initiated.

(2) An instruction (or sequence of instructions) is said to be issued when released from the virtual processor’s instruction fetch unit. Typically, instructions are issued to a reservation station or other buffer of instructions waiting to be executed. (Other conventions for this term exist, but this specification attempts to use “issued” consistently as defined here.) See also dispatched.

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 physical addresses. The MMU is composed of the ASRs and ASI registers used to manage address translation. See also context and virtual address.

MMU  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:* shall.

next program counter (NPC)  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, nonprivileged mode;
(2) virtual processor state information that is accessible to software while the virtual processor is in either privileged mode or nonprivileged mode; for example, nonprivileged registers, nonprivileged ASRs, or, in general, nonprivileged state;
(3) an instruction that can be executed when the virtual processor is in either privileged mode or nonprivileged mode.

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

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.

PCR Performance Control register.

PIC Performance Instrumentation Counter.

PIL Processor Interrupt Level register.

pipeline Refers to an execution pipeline. It is a loose term for the basic collection of hardware needed to execute instructions. A pipeline may be used by one or more strands to execute instructions from one or more threads. Synonym for microcore. See also processor, strand, thread, and virtual processor.

POR Power-on reset.

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

privileged An adjective that describes: (1) the state of the processor when PSTATE.priv = 1, that is, privileged mode; (2) processor state that is only accessible to software while the 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 processor is in privileged mode.

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

processor The unit on which a shared interface is provided to control the configuration and execution of a collection of strands. A processor contains one or more physical cores, each of which contains one or more strands. On a more physical side, a processor is a physical module that plugs into a system. A processor is expected to appear logically as a single agent on the system interconnect fabric. Synonym for processor module. See also pipeline, strand, thread, and virtual processor.

processor core See virtual processor.


**processor module** Synonym for processor.

**program counter (PC)** 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** (1) Return Address Stack  
  (2) Reliability, Availability, and Serviceability
- **RAW** Read After Write (hazard)
- **rd** Rounding direction.
- **RDPR** Read Privileged Register instruction.

**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 2005 virtual processor when it encounters a nonzero value in a reserved instruction field is as defined in *Reserved Opcodes and Instruction Fields* on page 120.

* A reserved bit combination within an instruction field is defined in Chapter 8, *Instructions*. In all cases, an UltraSPARC Architecture 2005 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 46).

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

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

**retired** An instruction is said to be “retired” when one of (instruction) 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  Relaxed memory order.

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

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  Identifies the hardware state used to hold a software thread in order to execute it. Strand is specifically the software-visible architected state (program counter (PC), next program counter (NPC), general-purpose registers, floating-point...
registers, condition codes, status registers, ASRs, etc.) of a thread and any microarchitecture state required by hardware for its execution. See also pipeline, processor, thread, and virtual processor.

subnormal number Synonym for denormalized number.

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

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

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 physical address space.

taken A control-transfer instruction (CTI) is taken when the CTI alters the control flow by writing a value into NPC other than the default value NPC = 4. 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.

TEE Thread Execution Engine. Synonym for virtual processor and strand.

thread A software entity that can be run on hardware. A thread is scheduled, may or may not be actively running on hardware at any given time, and may migrate around the hardware of a system. See also pipeline, processor, strand, and virtual processor.

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 supervisor software through a table, the address of which is specified by the privileged Trap Base Address (TBA) register. See also exception.

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

TSO Total store order.

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

UA-2005 UltraSPARC Architecture 2005
unassigned  A value (for example, an ASI number), the semantics of which are not architecturally mandated and which may be determined independently by each implementation within any guidelines given.

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

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

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

unpredictable  *Synonym for undefined.*

uniprocessor system  A system containing a single virtual processor.

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

user application program  *Synonym for application program.*

VA  Virtual address.

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

virtual core, virtual processor, virtual processor core  *Synonyms: virtual processor.*

virtual processor  The term virtual processor, or virtual processor core, is used to identify each strand in a processor. A processor contains one or more physical cores, each of which contains one or more virtual processors (strands). Each virtual processor (strand) has its own interrupt ID. At any given time, an operating system can have a different thread scheduled on each virtual processor. See also pipeline, processor, strand, and thread.

VIS  VISTM Instruction Set.

Strand  *Abbreviation for Virtual Processor.*

word  A 4-byte datum. *Note:* The definition of this term is architecture dependent and may differ from that used in other processor architectures.
WRPR Write Privileged Register instruction.
The UltraSPARC Architecture supports 32- and 64-bit integer and 32- 64-, and 128-bit floating-point as its principal data types. The 32- and 64-bit floating-point types conform to IEEE Std 754-1985. The 128-bit floating-point type conforms to IEEE Std 1596.5-1992. The architecture defines general-purpose integer, floating-point, and special state/status register instructions, all encoded in 32-bit-wide instruction formats. The load/store instructions address a linear, 2^64-byte virtual address space.

The UltraSPARC Architecture 2005 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 2005 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 2005 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).
4.1 The UltraSPARC Architecture 2005

This section briefly describes features, attributes, and components of the UltraSPARC Architecture 2005 and, further, describes correct implementation of the architecture specification and SPARC V9-compliance levels.

4.1.1 Features

The UltraSPARC Architecture 2005, 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), 16 quad-precision (128-bit) registers, or a mixture thereof.
- **Fast trap handlers** — Traps are vectored through a table.
- **Multiprocessor synchronization instructions** — One instruction performs an atomic read-then-set-memory operation; another performs an atomic exchange-register-with-memory operation; another compares the contents of a register with a value in memory and exchanges memory with the contents of another register if the comparison was equal (compare and swap); two others synchronize the order of shared memory operations as observed by virtual processors.
- **Predicted branches** — The branch with prediction instructions allows the compiler or assembly language programmer to give the hardware a hint about whether a branch will be taken.
- **Branch elimination instructions** — Several instructions can be used to eliminate branches altogether (for example, Move on Condition). Eliminating branches increases performance in superscalar and superpipelined implementations.
Hardware trap stack — A hardware trap stack is provided to allow nested traps. It contains all of the machine state necessary to return to the previous trap level. The trap stack makes the handling of faults and error conditions simpler, faster, and safer.

In addition, UltraSPARC Architecture 2005 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 2005 provides multiple sets; typically, one set is used at each trap level.

- **Extended instruction set** — UltraSPARC Architecture 2005 provides many instruction set extensions, including the VIS instruction set for “vector” (SIMD) data operations.

- **More detailed, specific instruction descriptions** — UltraSPARC Architecture 2005 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 2005 provides a blueprint for the software view of the UltraSPARC MMU (TTEs and TSBs).

### 4.1.2 Attributes

UltraSPARC Architecture 2005 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 2005 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.

#### 4.1.2.1 Design Goals

The UltraSPARC Architecture 2005 architecture is designed to be a target for optimizing compilers and high-performance hardware implementations. This specification documents the UltraSPARC Architecture 2005 and provides a design spec against which an implementation can be verified, using appropriate verification software.
4.1.2.2 Register Windows

The UltraSPARC Architecture 2005 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 supervisor software, not user programs, manages the register windows. The supervisor can save a minimum number of registers (approximately 24) during a context switch, thereby optimizing context-switch latency.

4.1.3 System Components

The UltraSPARC Architecture 2005 allows for a spectrum of subarchitectures, such as cache system.

4.1.3.1 Binary Compatibility

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

Although different SPARC V9 systems can execute nonprivileged programs at different rates, they will generate the same results as long as they are run under the same memory model. See Chapter 9, Memory, for more information.

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

4.1.3.2 UltraSPARC Architecture 2005 MMU

Although the SPARC V9 architecture allows its implementations freedom in their MMU designs, UltraSPARC Architecture 2005 defines a common MMU architecture (see Chapter 14, Memory Management) with some specifics left to implementations (see processor implementation documents).
4.1.3.3 Privileged Software

UltraSPARC Architecture 2005 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.

4.1.4 Architectural Definition

The UltraSPARC Architecture 2005 is defined by the chapters and normative 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 normative appendixes.

UltraSPARC Architecture 2005 defines a set of implementations that conform to the SPARC V9 architecture, Level 1.

4.1.5 UltraSPARC Architecture 2005 Compliance with SPARC V9 Architecture

UltraSPARC Architecture 2005 fully complies with SPARC V9 Level 1 (nonprivileged). It partially complies with SPARC V9 Level 2 (privileged).

4.1.6 Implementation Compliance with UltraSPARC Architecture 2005

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 2005, SPARC V8, and SPARC V9 implementation dependencies. Documents for specific UltraSPARC Architecture 2005 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 2005 by being implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.
4.2 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.

4.2.1 Integer Unit (IU)

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

**IMPL. DEP. #2-V8:** An UltraSPARC Architecture implementation may contain from 72 to 640 general-purpose 64-bit R registers. This corresponds to a grouping of the registers into 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).

4.2.2 Floating-Point Unit (FPU)

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

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.

4.3 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

These classes are discussed in the following subsections.

4.3.1 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 doubleword (64-bit) accesses. Some versions of integer load instructions perform sign extension on 8-, 16-, and 32-bit values as they are loaded into a 64-bit destination register. Floating-point load and store instructions support word, doubleword, and quadword\(^1\) memory accesses.

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

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

\(^1\) No UltraSPARC Architecture processor currently implements the LDQF instruction in hardware; it generates an exception and is emulated in supervisor software.
4.3.1.1 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 102.

4.3.1.2 Addressing Conventions

The SPARC V9 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 SPARC V9 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. See Processor State (PSTATE) Register (PR 6) on page 90 for information about changing the implicit byte order to little-endian.

Addressing conventions are illustrated in FIGURE 7-2 on page 105 and FIGURE 7-3 on page 107.

4.3.1.3 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 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 \( 2^{64} - 1 \).

4.3.1.4 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 \( 00_{16} - 7F_{16} \) is restricted to privileged code, and access to alternate spaces \( 80_{16} - FF_{16} \) is unrestricted. Some of the ASIs are available for implementation-dependent uses. Supervisor software can use the implementation-dependent ASIs to access special protected registers, such as MMU, cache control, and virtual processor state registers, and other processor- or system-dependent values. See Address Space Identifiers (ASIs) on page 108 for more information.
Alternate space addressing is also provided for the atomic memory access instructions LDSTUBA, CASA, and CASXA.

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

### 4.3.1.5 Separate I and D 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 translation mechanism and cache and data references use another, although the same main memory is shared.

In such split-memory systems, the coherency mechanism may be split, so a write into data memory is not immediately reflected in instruction memory. For this reason, programs that modify their own code (self-modifying code) and that wish to be portable across all SPARC V9 processors must issue FLUSH instructions, or a system call with a similar effect, to bring the instruction and data caches into a consistent state.

An UltraSPARC Architecture virtual processor may or may not have coherent instruction and data caches. Even if it does, 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.

### 4.3.1.6 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.
4.3.1.7 Memory Synchronization

Two instructions are used for synchronization of memory operations: FLUSH and MEMBAR. Their operation is explained in Flush Instruction Memory on page 174 and Memory Barrier on page 258, respectively.

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

4.3.2 Arithmetic / Logical / Shift Instructions

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

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

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

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

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

**Note** 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 a \( R \) register and a 13-bit signed immediate value. The “branch on condition codes without prediction” instruction provides a displacement of \( \pm 8 \) Mbytes; the “branch on condition codes with prediction” instruction provides a displacement of \( \pm 1 \) Mbyte; the “branch on register contents” instruction provides a displacement of \( \pm 128 \) Kbytes; and the CALL instruction’s 30-bit word displacement allows a control transfer to any address within \( \pm 2 \) gigabytes (\( \pm 2^{31} \) bytes).

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

## 4.3.4 State Register Access

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

**IMPL. DEP. #8-V8-Cs20:** Ancillary state registers (ASRs) in the range 0–27 that are not defined in UltraSPARC Architecture 2005 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:** Whether each of the implementation-dependent read/write ancillary state register instructions (for ASRs 28–31) is privileged is implementation dependent.
4.3.5 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. FPop instructions compute a result that is a function of one or two source operands. The groups of instructions that are considered FPop are listed in Floating-Point Operate (FPop) Instructions on page 119.

4.3.6 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 upon the contents of an integer register. These instructions increase performance by reducing the number of branches.

4.3.7 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.

4.4 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.

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 34.
- Floating-Point Data Formats on page 38.
- SIMD Data Formats on page 41.

Names are assigned to individual subwords of the multiword data formats as described in these sections:
- Signed Integer Doubleword (64 bits) on page 35.
- Unsigned Integer Doubleword (64 bits) on page 37.
- Floating Point, Double Precision (64 bits) on page 39.
- Floating Point, Quad Precision (128 bits) on page 40.
5.1 Integer Data Formats

TABLE 5-1 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 $2^8 - 1$</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 5-2 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.

5.1.1 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

5.1.1.1 Signed Integer Byte, Halfword, and Word

FIGURE 5-1 illustrates the signed integer byte, halfword, and word data formats.

![Signed Integer Byte, Halfword, and Word Data Formats](image)

5.1.1.2 Signed Integer Doubleword (64 bits)

FIGURE 5-2 illustrates both components (SD-0 and SD-1) of the signed integer double data format.

![Signed Integer Double Data Format](image)
5.1.1.3 Signed Integer Extended-Word (64 bits)

Figure 5-3 illustrates the signed integer extended-word (SX) data format.

---

5.1.2 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

5.1.2.1 Unsigned Integer Byte, Halfword, and Word

Figure 5-4 illustrates the unsigned integer byte data format.
5.1.2.2  Unsigned Integer Doubleword (64 bits)

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

![Diagram of unsigned integer doubleword](image)

5.1.2.3  Unsigned Extended Integer (64 bits)

FIGURE 5-6 illustrates the unsigned extended integer (UX) data format.

![Diagram of unsigned extended integer](image)

5.1.3  Tagged Word (32 bits)

FIGURE 5-7 illustrates the tagged word data format.

![Diagram of tagged word](image)
5.2 Floating-Point Data Formats

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

5.2.1 Floating Point, Single Precision (32 bits)

FIGURE 5-8 illustrates the floating-point single-precision data format, and TABLE 5-3 describes the formats.

![Floating-Point Single-Precision Data Format](image)

<table>
<thead>
<tr>
<th>TABLE 5-3</th>
<th>Floating-Point Single-Precision Format Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>sign (1 bit)</td>
</tr>
<tr>
<td>e</td>
<td>biased exponent (8 bits)</td>
</tr>
<tr>
<td>f</td>
<td>fraction (23 bits)</td>
</tr>
<tr>
<td>u</td>
<td>undefined</td>
</tr>
</tbody>
</table>

- Normalized value (0 < e < 255): $(-1)^s \times 2^{e-127} \times 1.f$
- Subnormal value (e = 0): $(-1)^s \times 2^{-126} \times 0.f$
- Zero (e = 0, f = 0): $(-1)^s \times 0$
- Signalling NaN: $s = u; e = 255$ (max); $f = .0uu--uu$  
  (At least one bit of the fraction must be nonzero)
- Quiet NaN: $s = u; e = 255$ (max); $f = .1uu--uu$
- $-\infty$ (negative infinity): $s = 1; e = 255$ (max); $f = .000--00$
- $+\infty$ (positive infinity): $s = 0; e = 255$ (max); $f = .000--00$
5.2.2 Floating Point, Double Precision (64 bits)

FIGURE 5-9 illustrates both components (FD-0 and FD-1) of the floating-point double-precision data format, and TABLE 5-4 describes the formats.

### TABLE 5-4 Floating-Point Double-Precision Format Definition

- **s** = sign (1 bit)
- **e** = biased exponent (11 bits)
- **f** = fraction (52 bits)
- **u** = undefined

<table>
<thead>
<tr>
<th>Normalized value (0 &lt; e &lt; 2047):</th>
<th>((-1)^s \times 2^{e-1023} \times 1.f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subnormal value (e = 0):</td>
<td>((-1)^s \times 2^{-1022} \times 0.f)</td>
</tr>
<tr>
<td>Zero (e = 0, f = 0)</td>
<td>((-1)^s \times 0)</td>
</tr>
<tr>
<td>Signalling NaN</td>
<td>s = u; e = 2047 (max); f = .0uu--uu</td>
</tr>
<tr>
<td>(At least one bit of the fraction must be nonzero)</td>
<td></td>
</tr>
<tr>
<td>Quiet NaN</td>
<td>s = u; e = 2047 (max); f = .1uu--uu</td>
</tr>
<tr>
<td>(-\infty) (negative infinity)</td>
<td>s = 1; e = 2047 (max); f = .000--00</td>
</tr>
<tr>
<td>(+\infty) (positive infinity)</td>
<td>s = 0; e = 2047 (max); f = .000--00</td>
</tr>
</tbody>
</table>
5.2.3 Floating Point, Quad Precision (128 bits)

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

![Table 5-5: Floating-Point Quad-Precision Format Definition](image)

<table>
<thead>
<tr>
<th></th>
<th>exp{14:0}</th>
<th>fraction{111:96}</th>
</tr>
</thead>
<tbody>
<tr>
<td>FQ-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FQ-1</td>
<td></td>
<td>fraction{95:64}</td>
</tr>
<tr>
<td>FQ-2</td>
<td></td>
<td>fraction{63:32}</td>
</tr>
<tr>
<td>FQ-3</td>
<td></td>
<td>fraction{31:0}</td>
</tr>
</tbody>
</table>

FIGURE 5-10 Floating-Point Quad-Precision Data Format

### TABLE 5-5

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>sign (1 bit)</td>
</tr>
<tr>
<td>e</td>
<td>biased exponent (15 bits)</td>
</tr>
<tr>
<td>f</td>
<td>fraction (112 bits)</td>
</tr>
<tr>
<td>u</td>
<td>undefined</td>
</tr>
<tr>
<td>Normalized value (0 &lt; e &lt; 32767):</td>
<td>(-1)^s × 2^{-16383} × 1.f</td>
</tr>
<tr>
<td>Subnormal value (e = 0):</td>
<td>(-1)^s × 2^{-16382} × 0.f</td>
</tr>
<tr>
<td>Zero (e = 0, f = 0)</td>
<td>(-1)^s × 0</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>
</tr>
<tr>
<td>Quiet NaN</td>
<td>s = u; e = 32767 (max); f = .1uu--uu</td>
</tr>
<tr>
<td>-∞ (negative infinity)</td>
<td>s = 1; e = 32767 (max); f = .000--00</td>
</tr>
<tr>
<td>+∞ (positive infinity)</td>
<td>s = 0; e = 32767 (max); f = .000--00</td>
</tr>
</tbody>
</table>
5.2.4 Floating-Point Data Alignment in Memory and Registers

Table 5-6 describes the address and memory alignment for floating-point data.

**Table 5-6: 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</td>
</tr>
<tr>
<td>FD-1</td>
<td>fraction[31:0]</td>
<td>0 mod 4 †</td>
<td>1 mod 2</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</td>
</tr>
<tr>
<td>FQ-1</td>
<td>fraction[95:64]</td>
<td>0 mod 4 †</td>
<td>1 mod 4</td>
</tr>
<tr>
<td>FQ-2</td>
<td>fraction[63:32]</td>
<td>0 mod 4 †</td>
<td>2 mod 4</td>
</tr>
<tr>
<td>FQ-3</td>
<td>fraction[31:0]</td>
<td>0 mod 4 †</td>
<td>3 mod 4</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).

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

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

5.3 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.
5.3.1 **Uint8 SIMD Data Format**

The Uint8 SIMD data format consists of four unsigned 8-bit integers contained in a 32-bit word (see FIGURE 5-11).

![Uint8 SIMD Data Format](image)

5.3.2 **Int16 SIMD Data Formats**

The Int16 SIMD data format consists of four signed 16-bit integers contained in a 64-bit word (see FIGURE 5-12).

![Int16 SIMD Data Format](image)

5.3.3 **Int32 SIMD Data Format**

The Int32 SIMD data format consists of two signed 32-bit integers contained in a 64-bit word (see FIGURE 5-13).

![Int32 SIMD Data Format](image)
The integer SIMD data formats can be used to hold fixed-point data. The position of the binary point in a SIMD datum is implied by the programmer and does not influence the computations performed by instructions that operate on that SIMD data format.
CHAPTER 6

Registers

The following registers are described in this chapter:

- **General-Purpose R Registers** on page 46.
- **Floating-Point Registers** on page 52.
- **Floating-Point State Register (FSR)** on page 58.
- **Ancillary State Registers** on page 67. The following registers are included in this category:
  - 32-bit Multiply/Divide Register (Y) (ASR 0) on page 69.
  - Integer Condition Codes Register (CCR) (ASR 2) on page 69.
  - Address Space Identifier (ASI) Register (ASR 3) on page 71.
  - Tick (TICK) Register (ASR 4) on page 71.
  - Program Counters (PC, NPC) (ASR 5) on page 72.
  - Floating-Point Registers State (FPRS) Register (ASR 6) on page 73.
  - Performance Control Register (PCR) (ASR 16) on page 74.
  - Performance Instrumentation Counter (PIC) Register (ASR 17) on page 75.
  - General Status Register (GSR) (ASR 19) on page 76.
  - SOFTINTP Register (ASRs 20, 21, 22) on page 77.
  - SOFTINT_SETP Pseudo-Register (ASR 20) on page 78.
  - SOFTINT_CLR Pseudo-Register (ASR 21) on page 79.
  - Tick Compare (TICK_CMPRP) Register (ASR 23) on page 79.
  - System Tick (STICK) Register (ASR 24) on page 80.
  - System Tick Compare (STICK_CMPRP) Register (ASR 25) on page 81.
- **Register-Window PR State Registers** on page 81. The following registers are included in this subcategory:
  - Current Window Pointer (CWP) Register (PR 9) on page 82.
  - Savable Windows (CANSAVEP) Register (PR 10) on page 83.
  - Restorable Windows (CANRESTOREP) Register (PR 11) on page 83.
  - Clean Windows (CLEANWINP) Register (PR 12) on page 83.
  - Other Windows (OTHERWINP) Register (PR 13) on page 84.
  - Window State (WSTATEP) Register (PR 14) on page 84.
- **Non-Register-Window PR State Registers** on page 86. The following registers are included in this subcategory:
  - Trap Program Counter (TPCP) Register (PR 0) on page 86.
  - Trap Next PC (TNPC) Register (PR 1) on page 87.
6.1 Reserved Register Fields

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

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

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

6.2 General-Purpose R Registers

An UltraSPARC Architecture virtual processor contains an array of general-purpose 64-bit R registers. The array is partitioned into $\text{MAXPGL} + 1$ sets of eight global registers, plus $\text{N\_REG\_WINDOWS}$ groups of 16 registers each. The value of $\text{N\_REG\_WINDOWS}$ 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 6-1.

![General-Purpose Registers (as Visible at Any Given Time)](image_url)

**FIGURE 6-1** General-Purpose Registers (as Visible at Any Given Time)
6.2.1 Global R Registers (A1)

Registers $R[0]–R[7]$ refer to a set of eight registers called the global registers (labeled $g_0$ through $g_7$). At any time, one of $\text{MAXPGL} + 1$ 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 $\text{GL}$ register. See Global Level Register ($G_{L P}$) (PR 16) on page 96.

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

6.2.2 Windowed R Registers (A1)

A set of 24 R registers that is visible as $R[8]–R[31]$ at any given time is called a “register window”. The registers that become $R[8]–R[15]$ in a register window are called the out registers of the window. Note that the in registers of a register window become the out registers of an adjacent register window. See TABLE 6-1 and FIGURE 6-2.

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>
<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, $\text{N.REG WINDOWS}$, ranges from 3 to 32 (impl. dep. #2-V8). The maximum global register set index in the UltraSPARC Architecture, $\text{MAXPGL}$, ranges from 2 to 15. The number of implemented global register sets is $\text{MAXPGL} + 1$. The total number of R registers in a given UltraSPARC Architecture implementation is:

$((\text{N.REG WINDOWS} \times 16) + ((\text{MAXPGL} + 1) \times 8))$

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 \( a \), where \( 8 \leq a \leq 15 \), 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 \( a + 16 \) after the register window is advanced. Likewise, register address \( i \), where \( 24 \leq i \leq 31 \), refers to exactly the same
in register before the register window is restored by a RESTORE instruction (CWP is decremented by 1 \(\text{modulo } N\_\text{REG}\_\text{WINDOWS}\)) as does register address \(i-16\) after the window is restored. See FIGURE 6-2 on page 49 and FIGURE 6-3 on page 51.

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 \(N\_\text{REG}\_\text{WINDOWS}\), the highest-numbered implemented window overlaps with window 0. The \(\text{outs}\) of window \(N\_\text{REG}\_\text{WINDOWS} - 1\) are the \(\text{ins}\) of window 0. Implemented windows are numbered contiguously from 0 through \(N\_\text{REG}\_\text{WINDOWS} - 1\).

Because the windows overlap, the number of windows available to software is 1 less than the number of implemented windows; that is, \(N\_\text{REG}\_\text{WINDOWS} - 1\). When the register file is full, the \(\text{outs}\) of the newest window are the \(\text{ins}\) of the oldest window, which still contains valid data.

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

When a new register window is made visible through use of a SAVE instruction, the local and out registers are guaranteed to contain either zeroes or valid data from the current context. If software executes a RESTORE and later executes a SAVE, then the contents of the resulting window’s local and out registers are not guaranteed to be preserved between the RESTORE and the SAVE\(^1\). Those registers may even have been written with “dirty” data, that is, data created by software running in a different context. However, if the clean_window protocol is being used, system software must guarantee that registers in the current window after a SAVE always contains only zeroes or valid data from that context. See Clean Windows (CLEANWIN\(^6\)) Register (PR 12) on page 83, Savable Windows (CANSERVE\(^6\)) Register (PR 10) on page 83, and Restorable Windows (CANRESTORE\(^6\)) Register (PR 11) on page 83.

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

\(^1\) For example, any of those 16 registers might be altered due to the occurrence of a trap between the RESTORE and the SAVE, or might be altered during the RESTORE operation due to the way that register windows are implemented. After a RESTORE instruction executes, software must assume that the values of the affected 16 registers from before the RESTORE are unrecoverable.
Register Window Management Instructions on page 116 describes how the windowed integer registers are managed.

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

**FIGURE 6-3**  Windowed R Registers for \( N_{\text{REG\_WINDOWS}} = 8 \)
In FIGURE 6-3, \( \text{N\_REG\_WINDOWS} = 8 \). The eight global registers are not illustrated. \( \text{CWP} = 0 \), \( \text{CANSAVE} = 4 \), \( \text{OTHERWIN} = 1 \), and \( \text{CANRESTORE} = 1 \). If the procedure using window \( w_0 \) executes a RESTORE, then window \( w_7 \) becomes the current window. If the procedure using window \( w_0 \) executes a SAVE, then window \( w_1 \) becomes the current window.

6.2.3 Special \( R \) Registers

The use of two of the \( R \) registers is fixed, in whole or in part, by the architecture:

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

Register-Pair Operands. LDTW, LDTWA, STTW, and STTWA instructions access a pair of words (“twin words”) in adjacent \( R \) registers and require even-odd register alignment. The least significant bit of an \( R \) register number in these instructions is unused and must always be supplied as 0 by software.

When the \( R[0]–R[1] \) register pair is used as a destination in LDTW or LDTWA, only \( R[1] \) is modified. When the \( R[0]–R[1] \) register pair is used as a source in STTW or STTWA, 0 is read from \( R[0] \), so 0 is written to the 32-bit word at the lowest address, and the least significant 32 bits of \( R[1] \) are written to the 32-bit word at the highest address.

An attempt to execute an LDTW, LDTWA, STTW, or STTWA instruction that refers to a misaligned (odd) destination register number causes an illegal_instruction trap.

6.3 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 \( FQ[0], FQ[4], \ldots, FQ[60] \)
- Thirty-two 64-bit double-precision registers, referenced as \( FD[0], FD[2], \ldots, FD[62] \)
- Thirty-two 32-bit single-precision registers, referenced as \( FS[0], FS[1], \ldots, FS[31] \)
  (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 6-2. 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 6-2** 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>Register</td>
<td>Assembly Language</td>
<td>Bits</td>
</tr>
<tr>
<td>F_S[0]</td>
<td>%f0</td>
<td>63:32</td>
</tr>
<tr>
<td>F_S[1]</td>
<td>%f1</td>
<td>31:0</td>
</tr>
<tr>
<td>F_S[3]</td>
<td>%f3</td>
<td>31:0</td>
</tr>
<tr>
<td>F_S[5]</td>
<td>%f5</td>
<td>31:0</td>
</tr>
<tr>
<td>F_S[7]</td>
<td>%f7</td>
<td>31:0</td>
</tr>
<tr>
<td>F_S[9]</td>
<td>%f9</td>
<td>31:0</td>
</tr>
<tr>
<td>F_S[11]</td>
<td>%f11</td>
<td>31:0</td>
</tr>
<tr>
<td>F_S[13]</td>
<td>%f13</td>
<td>31:0</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[19]</td>
<td>%f19</td>
<td>31:0</td>
</tr>
<tr>
<td>F_S[21]</td>
<td>%f21</td>
<td>31:0</td>
</tr>
<tr>
<td>F_S[22]</td>
<td>%f22</td>
<td>63:32</td>
</tr>
<tr>
<td>F_S[23]</td>
<td>%f23</td>
<td>31:0</td>
</tr>
</tbody>
</table>
TABLE 6-2  Floating-Point Registers, with Aliasing (2 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>Register</td>
<td>Assembly Language</td>
<td>Bits</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>
<tr>
<td>$F_s[30]$</td>
<td>%f30</td>
<td>63:32</td>
</tr>
<tr>
<td>$F_s[31]$</td>
<td>%f31</td>
<td>31:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:32</td>
</tr>
</tbody>
</table>
6.3.1 Floating-Point Register Number Encoding

Register numbers for single, double, and quad registers are encoded differently in the 5-bit register number field of a floating-point instruction. If the bits in a register number field are labeled \( b_4 \) … \( b_0 \) (where \( b_4 \) is the most significant bit of the register number), the encoding of floating-point register numbers into 5-bit instruction fields is as given in TABLE 6-3.

<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 ) 0</td>
<td>( b_4 ) ( b_3 ) ( b_2 ) ( b_1 ) ( b_5 )</td>
</tr>
<tr>
<td>Quad</td>
<td>( b_5 ) ( b_4 ) ( b_3 ) ( b_2 ) 0 0</td>
<td>( b_4 ) ( b_3 ) ( b_2 ) 0 ( b_5 )</td>
</tr>
</tbody>
</table>
6.3.2 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.

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.
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 2005 implementation does not implement a particular quad floating-point arithmetic operation in hardware and an invalid quad register operand is specified, per FSR.ftt priorities in TABLE 6-7, the fp_exception_other exception occurs with FSR.ftt = 3 (unimplemented_FPop) instead of with FSR.ftt = 6 (invalid_fp_register).

UltraSPARC Architecture 2005 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 fp_exception_other exception with FSR.ftt = 3 (unimplemented_FPop).
6.4 Floating-Point State Register (FSR)

The Floating-Point State register (FSR) fields, illustrated in FIGURE 6-4, contain FPU mode and status information. The lower 32 bits of the FSR are read and written by the STFSR and LDFSR instructions; all 64 bits of the FSR are read and written by the STXFSR and LDXFSR instructions, respectively. FSR.ver, FSR.flt, and the reserved ("—") fields of FSR are not modified by LDFSR or LDXFSR.

![Figure 6-4: FSR Fields]

Bits 63–38, 29–28, 21–20, and 12 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 58 through 67 describe the remaining fields in the FSR.

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

The four sets of floating-point condition code fields are labeled 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 and written by STXFSR and LDXFSR instructions, respectively. The fcc0 field can also be read and written by STFSR and LDFSR, respectively. FBfcc and FBPfcc instructions base their control transfers on the content of these fields. The MOVcc and FMOVcc instructions can conditionally copy a register, based on the contents of these fields.
In Table 6-5, $f_{rs1}$ and $f_{rs2}$ correspond to the single, double, or quad values in the floating-point registers specified by a floating-point compare instruction’s rs1 and rs2 fields. The question mark (?) indicates an unordered relation, which is true if either $f_{rs1}$ or $f_{rs2}$ is a signalling NaN or a quiet NaN. If FCMP or FCMPE generates an fp_exception_ieee_754 exception, then fccn is unchanged.

**TABLE 6-4** Floating-Point Condition Codes (fccn) Fields of FSR

<table>
<thead>
<tr>
<th>Content of fccn</th>
<th>Indicated Relation</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>$F[rs1] = F[rs2]$ (unordered)</td>
</tr>
</tbody>
</table>

**TABLE 6-5** 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>$F[rs1] = F[rs2]$</td>
<td>0</td>
</tr>
<tr>
<td>$F[rs1] &lt; F[rs2]$</td>
<td>1</td>
</tr>
<tr>
<td>$F[rs1] &gt; F[rs2]$</td>
<td>2</td>
</tr>
<tr>
<td>$F[rs1] = F[rs2]$ (unordered)</td>
<td>3</td>
</tr>
</tbody>
</table>

**6.4.2 Rounding Direction (rd)**

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

**TABLE 6-6** 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.imd. See General Status Register (GSR) (ASR 19) on page 76 for further details.

**6.4.3 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 6-5 on page 66. If a floating-point instruction generates one or more exceptions and the
tem bit corresponding to any of the exceptions is 1, then this condition causes an 
fp_exception_ieee_754 trap. A tem bit value of 0 prevents the corresponding IEEE 
754 exception type from generating a trap.

6.4.4 Nonstandard Floating-Point (ns)

On an UltraSPARC Architecture 2005 processor, FSR.ns is a reserved bit; it always 
reads as 0 and writes to it are ignored. (impl. dep. #18-V8)

6.4.5 FPU Version (ver) (A1)

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

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

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

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

6.4.6 Floating-Point Trap Type (ftt) (A1)

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

The FSR.ftt field can be read by a STFSR or STXFSR instruction. The LDFSR and 
LDXFSR instructions do not affect FSR.ftt.
Privileged software that handles floating-point traps must execute an STFSR (or STXFSR) to determine the floating-point trap type. STFSR and STXFSR shall zero ft with the store completes without error. If the store generates an error and does not complete, ft remains unchanged.

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

FSR.ft 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.ft. The conditions leading to `fp_exception_other` and `fp_exception_ieee_754` exceptions, their relative priorities, and the corresponding FSR.ft values are listed in TABLE 6-7. Note that the FSR.ft 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 6-7 FSR Floating-Point Trap Type (ft) Field

<table>
<thead>
<tr>
<th>Condition Detected During Execution of an FPop</th>
<th>Relative Priority (1 = highest)</th>
<th>FSR.ft Set to Value</th>
<th>Exception Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>unimplemented_FPop</td>
<td>10</td>
<td>3</td>
<td><code>fp_exception_other</code></td>
</tr>
<tr>
<td>invalid_fp_register</td>
<td>20</td>
<td>6</td>
<td><code>fp_exception_other</code></td>
</tr>
<tr>
<td>unfinished_FPop</td>
<td>30</td>
<td>2</td>
<td><code>fp_exception_other</code></td>
</tr>
<tr>
<td>IEEE_754_exception</td>
<td>40</td>
<td>1</td>
<td><code>fp_exception_ieee_754</code></td>
</tr>
<tr>
<td>Reserved</td>
<td>—</td>
<td>4, 5, 7</td>
<td>—</td>
</tr>
<tr>
<td>(none detected)</td>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
</tbody>
</table>

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

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

1. The value of aex is unchanged.
2. When an `fp_exception_ieee_754` trap occurs, a bit corresponding to the trapping exception is set in cex. On other traps, the value of cex is unchanged.
3. The source and destination registers are unchanged.
4. The value of fcon 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 or unimplemented_FPop. In either case, `cexc` as seen by the trap handler reflects the exception causing the trap.

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

**fft = 1 (IEEE_754_exception).** The IEEE_754_exception floating-point trap type indicates the occurrence of a floating-point exception conforming to IEEE Std 754-1985. The IEEE 754 exception type (overflow, inexact, etc.) is set in the `cexc` field. The `aexc` and `fccn` fields and the destination F register are unchanged.

**fft = 2 (unfinished_FPop).** The unfinished_FPop floating-point trap type indicates that the virtual processor was unable to generate correct results or that exceptions as defined by IEEE Std 754-1985 have occurred. In cases where exceptions have occurred, the `cexc` field is unchanged.

**IMPL. DEP. #248-U3:** The conditions under which an `fp_exception_other` exception with floating-point trap type of unfinished_FPop can occur are implementation dependent. An implementation may cause `fp_exception_other` with `FSR.fft = unfinished_FPop` under a different (but specified) set of conditions.

**fft = 3 (unimplemented_FPop).** The unimplemented_FPop floating-point trap type indicates that the virtual processor decoded an FPop that it does not implement in hardware. In this case, the `cexc` field is unchanged.

For example, all quad-precision FPop variations in an UltraSPARC Architecture 2005 virtual processor cause an `fp_exception_other` exception, setting `FSR.fft = unimplemented_FPop`.

---

**Forward Compatibility Note:** The next revision of the UltraSPARC Architecture is expected to eliminate “unimplemented_FPop”, to simplify handling of unimplemented instructions. At that point, all conditions which currently cause cause `fp_exception_other` with `FSR.fft = 3` will cause an `illegal_instruction` exception, instead. `FSR.fft = 3` and the trap type associated with `fp_exception_other` will become reserved for other possible future uses.
ftt = 4 (Reserved).

**SPARC V9 Compatibility Note**

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

ftt = 5 (Reserved).

**SPARC V9 Compatibility Note**

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

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

**Implementation Note**

Per FSR.ftt priorities in TABLE 6-7, if an UltraSPARC Architecture 2005 processor does not implement a particular quad FPop in hardware, that FPop generates an *fp_exception_other* exception with FSR.ftt = 3 (unimplemented_FPop) instead of *fp_exception_other* with FSR.ftt = 6 (invalid_fp_register), regardless of the specified F registers.

6.4.7 FQ Not Empty (qne) (V2)

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

6.4.8 Accrued Exceptions (aexc) (A1)

Bits 9 through 5 accumulate IEEE_754 floating-point exceptions as long as floating-point exception traps are disabled through the tem field. See FIGURE 6-6 on page 66.
After an FPop completes with \( ftt = 0 \), the \( \text{tem} \) and \( \text{cexc} \) fields are logically \textbf{anded} together. If the result is nonzero, \( \text{aexc} \) is left unchanged and an \texttt{fp_exception_ieee_754} trap is generated; otherwise, the new \( \text{cexc} \) field is \texttt{ored} into the \( \text{aexc} \) field and no trap is generated. Thus, while (and only while) traps are masked, exceptions are accumulated in the \( \text{aexc} \) field.

\( \text{FSR.aexc} \) is written with the appropriate value when an LDFSR or LDXFSR instruction is executed.

### 6.4.9 Current Exception (\( \text{cexc} \)) (A1)

\( \text{FSR.cexc} \) (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 6-5 on page 66.

**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 \( \text{FSR.cexc} \) value before returning to a nonprivileged program.

The \( \text{cexc} \) bits are set as described in *Floating-Point Exception Fields* on page 65, by the execution of an FPop that either does not cause a trap or causes an \texttt{fp_exception_ieee_754} exception with \( \text{FSR.flt} = \text{IEEE_754_exception} \). An IEEE 754 exception that traps shall cause exactly one bit in \( \text{FSR.cexc} \) 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, \( \text{FSR.cexc} \) bits are set and trapping occurs as follows:

- If an IEEE 754 overflow condition occurs:
  - if \( \text{FSR.tem.ofm} = 0 \) and \( \text{tem.nxm} = 0 \), the \( \text{FSR.cexc.ofc} \) and \( \text{FSR.cexc.nxc} \) bits are both set to 1, the other three bits of \( \text{FSR.cexc} \) are set to 0, and an \texttt{fp_exception_ieee_754} trap does not occur.
  - if \( \text{FSR.tem.ofm} = 0 \) and \( \text{tem.nxm} = 1 \), the \( \text{FSR.cexc.nxc} \) bit is set to 1, the other four bits of \( \text{FSR.cexc} \) are set to 0, and an \texttt{fp_exception_ieee_754} trap does occur.
  - if \( \text{FSR.tem.ofm} = 1 \), the \( \text{FSR.cexc.ofc} \) bit is set to 1, the other four bits of \( \text{FSR.cexc} \) are set to 0, and an \texttt{fp_exception_ieee_754} trap does occur.

- If an IEEE 754 underflow condition occurs:
  - if \( \text{FSR.tem.ufm} = 0 \) and \( \text{FSR.tem.nxm} = 0 \), the \( \text{FSR.cexc.ufc} \) and \( \text{FSR.cexc.nxc} \) bits are both set to 1, the other three bits of \( \text{FSR.cexc} \) are set to 0, and an \texttt{fp_exception_ieee_754} trap does not occur.
- if $\text{FSR.tem.ufm} = 0$ and $\text{FSR.tem.nxm} = 1$, the $\text{FSR.cexc.nxc}$ bit is set to 1, the other four bits of $\text{FSR.cexc}$ are set to 0, and an $\text{fp_exception_ieee_754}$ trap does occur.

- if $\text{FSR.tem.ufm} = 1$, the $\text{FSR.cexc.ufc}$ bit is set to 1, the other four bits of $\text{FSR.cexc}$ are set to 0, and an $\text{fp_exception_ieee_754}$ trap does occur.

The above behavior is summarized in Table 6-8 (where “✔” indicates “exception was detected” and “x” indicates “don’t care”):

### Table 6-8  Setting of $\text{FSR.cexc}$ Bits

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exception(s)</strong></td>
<td><strong>Trap Enable Mask bits</strong></td>
</tr>
<tr>
<td><strong>Detected in F.p. operation</strong></td>
<td><strong>(in FSR.tem)</strong></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>-</td>
</tr>
<tr>
<td>✔</td>
<td>-</td>
</tr>
</tbody>
</table>

Notes:  
1 When the underflow trap is disabled ($\text{FSR.tem.ufm} = 0$), underflow is always accompanied by inexact.  
2 Overflow is always accompanied by inexact.

If the execution of an FPop causes a trap other than $\text{fp_exception_ieee_754}$, $\text{FSR.cexc}$ is left unchanged.

### 6.4.10 Floating-Point Exception Fields

The current and accrued exception fields and the trap enable mask assume the following definitions of the floating-point exception conditions (per IEEE Std 754-1985):
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 362 and Untrapped Underflow Definition (ufm = 0) on page 362 for additional details.

Division by zero (dzc, dza). \( X \div 0.0 \), 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.

6.4.11 FSR Conformance

An UltraSPARC Architecture implementation implements the \(\text{tem}, \text{cex},\) and \(\text{aex}\) 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 \text{fp\_exception\_other} (with FSR.ftt = unfinished.FPop or unimplemented.FPop) and \text{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.

6.5 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 6-9 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^D)</td>
<td>(Y) register (deprecated)</td>
<td>RDY(^D)</td>
<td>WRY(^D)</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>\text{Reserved}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CCR</td>
<td>Condition Codes register</td>
<td>RDCCR</td>
<td>WRCCCR</td>
</tr>
<tr>
<td>3</td>
<td>ASI</td>
<td>ASI register</td>
<td>RDASI</td>
<td>WRASI</td>
</tr>
</tbody>
</table>
## TABLE 6-9  ASR Register Summary  (Continued)

<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>4</td>
<td>TICK\textsuperscript{Pnpt}</td>
<td>TICK register</td>
<td>RDTICK\textsuperscript{Pnpt}, RDPR\textsuperscript{P} (TICK)</td>
<td>WRPR\textsuperscript{P} (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</td>
<td>—</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>15</td>
<td>—</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16–31</td>
<td>non-SPARC V9 ASRs</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>PCR\textsuperscript{P}</td>
<td>Performance Control registers (PCR)</td>
<td>RDPCR\textsuperscript{P}</td>
<td>WRPCR\textsuperscript{P}</td>
</tr>
<tr>
<td>17</td>
<td>PIC\textsuperscript{P}</td>
<td>Performance Instrumentation Counters (PIC)</td>
<td>RDPIC\textsuperscript{Pic}</td>
<td>WRPIC\textsuperscript{Pic}</td>
</tr>
<tr>
<td>18</td>
<td>—</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>19</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</td>
<td>SOFTINT_CLR\textsuperscript{P}</td>
<td>(pseudo-register, for &quot;Write 1s Clear&quot; to SOFTINT register, ASR 22)</td>
<td>—</td>
<td>WRSOFTINT_CLR\textsuperscript{P}</td>
</tr>
<tr>
<td>21</td>
<td>SOFTINT_SET\textsuperscript{P}</td>
<td>(pseudo-register, for &quot;Write 1s Set&quot; to SOFTINT register, ASR 22)</td>
<td>—</td>
<td>WRSOFTINT_SET\textsuperscript{P}</td>
</tr>
<tr>
<td>22</td>
<td>SOFTINT\textsuperscript{P}</td>
<td>per-virtual processor Soft Interrupt register</td>
<td>RDSOFTWARE\textsuperscript{P}</td>
<td>WRSOFTWARE\textsuperscript{P}</td>
</tr>
<tr>
<td>23</td>
<td>TICK_CMPR\textsuperscript{P}</td>
<td>Tick Compare register</td>
<td>RDTICK_CMPR\textsuperscript{P}</td>
<td>WRTICK_CMPR\textsuperscript{P}</td>
</tr>
<tr>
<td>24</td>
<td>STICK\textsuperscript{Pnpt}</td>
<td>System Tick register</td>
<td>RDSTICK\textsuperscript{Pnpt}</td>
<td>—</td>
</tr>
<tr>
<td>25</td>
<td>STICK_CMPR\textsuperscript{P}</td>
<td>System Tick Compare register</td>
<td>RDSTICK_CMPR\textsuperscript{P}</td>
<td>WRSSTICK_CMPR\textsuperscript{P}</td>
</tr>
<tr>
<td>26–31</td>
<td>—</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
6.5.1 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 page 351; for the multiply step instruction, see page 268; for division instructions, see page 348; for the read instruction, see page 286; and for the write instruction, see page 354.

The low-order 32 bits of the Y register, illustrated in FIGURE 6-8, contain the more significant word of the 64-bit product of an integer multiplication, as a result of either a 32-bit integer multiply (SMUL, SMULcc, UMUL, UMULcc) instruction or an integer multiply step (MULScc) instruction. The Y register also holds the more significant word of the 64-bit dividend for a 32-bit integer divide (SDIV, SDIVcc, UDIV, UDIVcc) instruction.

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.

6.5.2 Integer Condition Codes Register (CCR) (ASR 2)

The Condition Codes Register (CCR), shown in FIGURE 6-9, contains the integer condition codes. The CCR register may be explicitly read and written by the RDCCR and WRCCCR instructions, respectively.

![FIGURE 6-8 Y Register](image)

![FIGURE 6-9 Condition Codes Register](image)
6.5.2.1 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 6-10.

![FIGURE 6-10 Integer Condition Codes (CCR.icc and CCR.xcc)](image)

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 6-10).

<table>
<thead>
<tr>
<th>Unsigned Comparison of Operand Values</th>
<th>Setting of Carry bits in CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[rs1][31:0] ≥ R[rs2][31:0]</td>
<td>CCR.icc.c ← 0</td>
</tr>
<tr>
<td>R[rs1][31:0] &lt; R[rs2][31:0]</td>
<td>CCR.icc.c ← 1</td>
</tr>
<tr>
<td>R[rs1][63:0] ≥ R[rs2][63:0]</td>
<td>CCR.xcc.c ← 0</td>
</tr>
<tr>
<td>R[rs1][63:0] &lt; R[rs2][63:0]</td>
<td>CCR.xcc.c ← 1</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 (xccc).** 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.

### 6.5.3 Address Space Identifier (ASI) Register (ASR 3)

The Address Space Identifier register (FIGURE 6-11) 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 in the range 3016 to 7F16, a *privileged_action* exception is generated. See Chapter 10, *Address Space Identifiers (ASIs)* for details.

![FIGURE 6-11 Address Space Identifier Register](image)

### 6.5.4 Tick (TICK) Register (ASR 4)

FIGURE 6-12 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.

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 causes a privileged_action exception. Nonprivileged software cannot write the TICK register. An attempt by nonprivileged software to read the TICK register using the privileged RDPR 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.

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

### 6.5.5 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[T]L and/or TNPC[T]L and executes a RETRY instruction.

See Chapter 7, Instruction Set Overview, for details on how PC and NPC are used.

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

The Floating-Point Registers State (FPRS) register, shown in FIGURE 6-13, contains control information for the floating-point register file; this information is readable and writable by nonprivileged software.

![Floating-Point Registers State Register](FIGURE 6-13)

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.

| 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). | Programming Note |

---

CHAPTER 6 • Registers 73
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.

IMPL. DEP. #403-S10(a): An UltraSPARC Architecture 2005 virtual processor may set FPRS.du pessimistically; that is, it may be set whenever an FPop is issued, even though no destination F register is modified. The specific conditions under which a dirty bit is set pessimistically are implementation dependent.

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.

IMPL. DEP. #403-S10(b): An UltraSPARC Architecture 2005 virtual processor may set FPRS.dl pessimistically; that is, it may be set whenever an FPop is issued, even though no destination F register is modified. The specific conditions under which a dirty bit is set pessimistically are implementation dependent.

Implementation Note: If an instruction that normally writes to the F registers is executed and causes an fp_disabled exception, an UltraSPARC Architecture 2005 implementation still sets the “dirty” bit (FPRS.du or FPRS.dl) corresponding to the destination register to ‘1’.

Forward Compatibility Note: It is expected that in future revisions to the UltraSPARC Architecture, if an instruction that normally writes to the F registers is executed and causes an fp_disabled exception the “dirty” bit (FPRS.du or FPRS.dl) corresponding to the destination register will be left unchanged.

6.5.7 Performance Control Register (PCR^P) (ASR 16)

The PCR is used to control performance monitoring events collected in counter pairs, which are accessed via the Performance Instrumentation Counter (PIC) register (ASR 17) (see page 75). Unused PCR bits read as zero; they should be written only with zeroes or with values previously read from them.

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

When the virtual processor is operating in nonprivileged mode (PSTATE.priv = 0), an attempt to access PCR (using a RDPCR or WRPCR instruction) results in a privileged_opcode exception (impl. dep. #250-U3-Cs10).

The PCR is illustrated in FIGURE 6-14 and described in TABLE 6-11.
The values and semantics of bits 47:32, 26:17, and bit 3 of the PCR are implementation dependent.

**TABLE 6-11** PCR Bit Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:32</td>
<td>—</td>
<td>These bits are implementation dependent (impl. dep #207-U3).</td>
</tr>
<tr>
<td>26:17</td>
<td>—</td>
<td>These bits are implementation dependent (impl. dep. #207-U3).</td>
</tr>
<tr>
<td>16:11</td>
<td>su</td>
<td>Six-bit field selecting 1 of 64 event counts in the upper half (bits {63:32}) of the PIC.</td>
</tr>
<tr>
<td>9:4</td>
<td>sl</td>
<td>Six-bit field selecting 1 of 64 event counts in the lower half (bits {31:0}) of the PIC.</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>This bit is implementation dependent (impl. dep. #207-U3).</td>
</tr>
<tr>
<td>2</td>
<td>ut</td>
<td>User Trace Enable. If set to 1, events in nonprivileged (user) mode are counted.</td>
</tr>
<tr>
<td>1</td>
<td>st</td>
<td>System Trace Enable. If set to 1, events in privileged (system) mode are counted.</td>
</tr>
</tbody>
</table>

**Notes:**
- If both $\text{PCR}_{\text{ut}}$ and $\text{PCR}_{\text{st}}$ are set to 1, all selected events are counted.
- If both $\text{PCR}_{\text{ut}}$ and $\text{PCR}_{\text{st}}$ are zero, counting is disabled.
- $\text{PCR}_{\text{ut}}$ and $\text{PCR}_{\text{st}}$ are global fields which apply to all PIC pairs.

| 0     | priv  | Privileged. Controls access to the PIC register (via RDPIC or WRPIC instructions). If $\text{PCR}_{\text{priv}} = 0$, an attempt to access PIC will succeed regardless of the privilege state (PSTATE.$\text{priv}$). If $\text{PCR}_{\text{priv}} = 1$, access to PIC is restricted to privileged software; that is, an attempt to access PIC while PSTATE.$\text{priv}$ = 1 will succeed, but an attempt to access PIC while PSTATE.$\text{priv}$ = 0 will result in a privileged_action exception. |

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

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

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

PIC is normally a nonprivileged-access, read/write register. However, if the priv bit of the PCR (ASR 16) is set, attempted access by nonprivileged (user) code causes a privileged_action exception.
Multiple PICs may be implemented. Each is accessed through ASR 17, using an implementation-dependent PIC pair selection field in PCR (ASR 16) (impl. dep. #207-U3). Read/write access to the PIC will access the picu/picl counter pair selected by PCR.

The PIC is described below and illustrated in FIGURE 6-15.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>picu</td>
<td>32-bit counter representing the count of an event selected by the su field of the Performance Control Register (PCR) (ASR 16).</td>
</tr>
<tr>
<td>31:0</td>
<td>picl</td>
<td>32-bit counter representing the count of an event selected by the sl field of the Performance Control Register (PCR) (ASR 16).</td>
</tr>
</tbody>
</table>

**Counter Overflow.** On overflow, the effective counter wraps to 0, SOFTINT register bit 15 is set to 1, and an interrupt level 15 trap is generated if not masked by PSTATE.ie and P1L. The counter overflow trap is triggered on the transition from value FFFF FFFF 16 to value 0.

### 6.5.9 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 285) and written by the WRGSR instruction (see Write Ancillary State Register on page 353).

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 6-16 and described in TABLE 6-12.

---

\(^1\) This register was (inaccurately) referred to as the “Graphics Status Register” in early UltraSPARC implementations.

76 UltraSPARC Architecture 2005 • Draft D0.8.7, 27 Mar 2006
### 6.5.10 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 can be read with a RDSOFTINT instruction (see Read Ancillary State Register on page 285) and written with a WRSOFTINT, WRSOFTINT_SET, or WRSOFTINT_CLR instruction (see Write Ancillary State Register on page 353). An attempt to access to this register in nonprivileged mode causes a `privileged_opcode` exception.

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

The SOFTINT register is illustrated in FIGURE 6-17 and described in TABLE 6-13.

```
SOFTINT\(^P\) | RW | RW | RW
--- | --- | --- | ---
63:32 | mask | | |
31:28 | — | | |
27 | im | | |
26:25 | irnd | | |
24:8 | — | | |
7:3 | scale | 5-bit shift count in the range 0–31, used by the FPACK instructions for formatting.
2:0 | align | Least three significant bits of the address computed by the last-executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction.
```

**TABLE 6-12 GSR Bit Description**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>mask</td>
<td>This 32-bit field specifies the mask used by the BSHUFFLE instruction. The field contents are set by the BMASK instruction.</td>
</tr>
<tr>
<td>31:28</td>
<td>—</td>
<td>Reserved.</td>
</tr>
<tr>
<td>27</td>
<td>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</td>
<td>irnd</td>
<td>IEEE Std 754-1985 rounding direction to use in Interval Mode (GSR.im = 1), as follows:</td>
</tr>
<tr>
<td>24:8</td>
<td>—</td>
<td>Reserved.</td>
</tr>
<tr>
<td>7:3</td>
<td>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</td>
<td>align</td>
<td>Least three significant bits of the address computed by the last-executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction.</td>
</tr>
</tbody>
</table>

**FIGURE 6-17 SOFTINT Register (ASR 22)**
Setting any of SOFTINT.sm, SOFTINT.int_level{13} (SOFTINT{14}), or SOFTINT.tm to 1 causes a level-14 interrupt (interrupt_level_14). However, those three bits are independent; setting any one of them does not affect the other two.

See Software Interrupt Register (SOFTINT) on page 442 for additional information regarding the SOFTINT register.

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

A Write State register instruction to ASR 20 (WRSOFTINT_SET) atomically sets selected bits in the privileged SOFTINT Register (ASR 22) (see page 77). That is, bits 16:0 of the write data are or-ed into SOFTINT; any '1' bit in the write data causes the corresponding bit of SOFTINT to be set to 1. Bits 63:17 of the write data are ignored.

Access to ASR 20 is privileged and write-only. There is no instruction to read this pseudo-register. An attempt to write to ASR 20 in non-privileged mode, using the WRasr instruction, causes a privileged_opcode exception.

#### Programming Note
There is no actual "register" (machine state) corresponding to ASR 20; it is just a programming interface to conveniently set selected bits to '1' in the SOFTINT register, ASR 22.
FIGURE 6-18 illustrates the SOFTINT_SET pseudo-register.

![SOFTINT_SET](image)

**FIGURE 6-18 SOFTINT_SET Pseudo-Register (ASR 20)**

6.5.10.2 SOFTINT_CLR\(^P\) Pseudo-Register (ASR 21) (D2)

A Write State register instruction to ASR 21 (WRSOFTINT_CLR) atomically clears selected bits in the privileged SOFTINT register (ASR 22) (see page 77). That is, bits 16:0 of the write data are inverted and **and**ed into SOFTINT; any ‘1’ bit in the write data causes the corresponding bit of SOFTINT to be set to 0. Bits 63:17 of the write data are ignored.

Access to ASR 21 is privileged and write-only. There is no instruction to read this pseudo-register. An attempt to write to ASR 21 in non-privileged mode, using the WRasr instruction, causes a **privileged_opcode** exception.

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

FIGURE 6-19 illustrates the SOFTINT_CLR pseudo-register.

![SOFTINT_CLR](image)

**FIGURE 6-19 SOFTINT_CLR Pseudo-Register (ASR 21))**

6.5.11 Tick Compare (TICK_CMPR\(^P\)) Register (ASR 23) (D1)

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 431).

The TICK_CMPR register is illustrated in FIGURE 6-20 and described in TABLE 6-14.

![TICK_CMPR](image)

**FIGURE 6-20 TICK_CMPR Register**
6.5.12 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 6-21 and described below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>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</td>
<td>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>

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.
6.5.13 System Tick Compare (STICK_CMPRP) Register (ASR 25) \( \text{D2} \)

The privileged STICK_CMPR register allows system software to cause a trap when the STICK register reaches a specified value. Nonprivileged accesses to this register cause a privileged_opcode exception (see Exception and Interrupt Descriptions on page 431).

The System Tick Compare Register is illustrated in FIGURE 6-22 and described in TABLE 6-15.

![FIGURE 6-22 STICK_CMPR Register](image)

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

6.6 Register-Window PR State Registers

The state of the register windows is determined by the contents of a set of privileged registers. These state registers can be read/written by privileged software using the RDPR/WRPR instructions. An attempt by nonprivileged software to execute a RDPR or WRPR instruction causes a privileged_opcode exception. In addition, these registers are modified by instructions related to register windows and are used to generate traps that allow supervisor software to spill, fill, and clean register windows.
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 85.

Although the width of each of these five registers is architecturally 5 bits, the width is implementation dependent and shall be between \( \lceil \log_2(N_{\text{REG\_WINDOWS}}) \rceil \) 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 2005 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 116.

- **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 85. Hardware is not required to enforce this restriction; it is up to system software to keep the window state consistent.

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

### 6.6.1 Current Window Pointer (\( \text{CWP}^P \)) Register (PR 9)

The privileged CWP register, shown in FIGURE 6-23, is a counter that identifies the current window into the array of integer registers. See Register Window Management Instructions on page 116 and Chapter 12, Traps, for information on how hardware manipulates the CWP register.

![Current Window Pointer Register](image)
6.6.2 Savable Windows (CANSAVE\textsuperscript{P}) Register (PR 10)

The privileged CANSAVE register, shown in FIGURE 6-24, 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 6-24 CANSAVE Register, Figure 5-24, page 88](image)

6.6.3 Restorable Windows (CANRESTORE\textsuperscript{P}) Register (PR 11)

The privileged CANRESTORE register, shown in FIGURE 6-25, 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 6-25 CANRESTORE Register](image)

6.6.4 Clean Windows (CLEANWIN\textsuperscript{P}) Register (PR 12)

The privileged CLEANWIN register, shown in FIGURE 6-26, contains the number of windows that can be used by the SAVE instruction without causing a clean_window exception.

![FIGURE 6-26 CLEANWIN Register](image)

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.

6.6.5 Other Windows (OTHERWIN\textsuperscript{P}) Register (PR 13)

The privileged OTHERWIN register, shown in FIGURE 6-27, 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.

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

The privileged WSTATE register, shown in FIGURE 6-28, 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.

6.6.7 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 81.
Those registers are affected by the instructions described in Register Window
Management Instructions on page 116. Privileged software can read/write these state
registers directly by using RDPR/WRPR instructions.
6.6.7.1 Register Window State Definition

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

\[
\text{CANSAVE} + \text{CANRESTORE} + \text{OTHERWIN} = N_{\text{REG\_WINDOWS}} - 2
\]

FIGURE 6-3 on page 51 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 6-3, these are windows 0 and 5, respectively. They are always present and account for the “2” subtracted from \( N_{\text{REG\_WINDOWS}} \) 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 6-3) are counted in \( \text{CANSAVE} \).
- 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 6-3) are counted in \( \text{CANRESTORE} \).
- 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 6-3) are counted in \( \text{OTHERWIN} \).

In addition,

\[
\text{CLEANWIN} \geq \text{CANRESTORE}
\]

since \( \text{CLEANWIN} \) is the sum of \( \text{CANRESTORE} \) and the number of clean windows following \( \text{CWP} \).

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, \( \text{CANSAVE} \), \( \text{CANRESTORE} \), and \( \text{OTHERWIN} \) must never be greater than or equal to \( N_{\text{REG\_WINDOWS}} - 1 \).
6.6.7.2 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 436 for a detailed description of how fill, spill, and clean_window traps support register windowing.

6.7 Non-Register-Window PR State Registers

The registers described in this section are visible only to software running in privileged mode (that is, when PSTATE.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.

6.7.1 Trap Program Counter (TPC\textsuperscript{P}) Register (PR 0)

The privileged Trap Program Counter register (TPC; FIGURE 6-29) contains the program counter (PC) from the previous trap level. There are \textit{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.

\[
\begin{array}{c|c}
\text{TPC}_{1}^{P} & \text{pc\_high62 (PC\{63:2\} from trap while TL = 0)} \\
\text{TPC}_{2}^{P} & \text{pc\_high62 (PC\{63:2\} from trap while TL = 1)} \\
\text{TPC}_{3}^{P} & \text{pc\_high62 (PC\{63:2\} from trap while TL = 2)} \\
\vdots & \vdots \\
\text{TPC}_{\text{MAXPTL}}^{P} & \text{pc\_high62 (PC\{63:2\} from trap while TL = \text{MAXPTL} - 1)}
\end{array}
\]

\textbf{FIGURE 6-29} Trap Program Counter Register Stack

During normal operation, the value of TPC[\textit{n}], where \textit{n} is greater than the current trap level (\textit{n} > TL), is undefined.
TABLE 6-16 lists the events that cause TPC to be read or written.

**TABLE 6-16** 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)] ← PC</td>
</tr>
<tr>
<td>RETRY instruction</td>
<td>PC ← TPC[(n)]</td>
</tr>
<tr>
<td>RDPR (TPC)</td>
<td>R[rd] ← TPC[(n)]</td>
</tr>
<tr>
<td>WRPR (TPC)</td>
<td>TPC[(n)] ← \textit{value}</td>
</tr>
</tbody>
</table>

### 6.7.2 Trap Next PC (TNPC\(^P\)) Register (PR 1)

The privileged Trap Next Program Counter register (TNPC; **FIGURE 6-29**) is the next program counter (NPC) from the previous trap level. There are \textit{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 \textit{illegal_instruction} exception.

![FIGURE 6-30 Trap Next Program Counter Register Stack](image)

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

**TABLE 6-17** lists the events that cause TNPC to be read or written.

**TABLE 6-17** 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)] ← NPC</td>
</tr>
<tr>
<td>DONE instruction</td>
<td>PC ← TNPC[(n)]; NPC ← TNPC[(n)] + 4</td>
</tr>
<tr>
<td>RETRY instruction</td>
<td>NPC ← TNPC[(n)]</td>
</tr>
<tr>
<td>RDPR (TNPC)</td>
<td>R[rd] ← TNPC[(n)]</td>
</tr>
<tr>
<td>WRPR (TNPC)</td>
<td>TNPC[(n)] ← \textit{value}</td>
</tr>
</tbody>
</table>
6.7.3 Trap State (TSTATE\(^P\)) Register (PR 2)

The privileged Trap State register (TSTATE; FIGURE 6-31) 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 \(\text{MAXPTL}\) instances of the TSTATE register, but only one is accessible at a time. The current value in the TL register determines which instance of TSTATE is accessible. An attempt to read or write the TSTATE register when \(\text{TL} = 0\) causes an illegal_instruction exception.

**TABLE 6-19** lists the events that cause TSTATE to be read or written.

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

**TABLE 6-19** Events That Involve TSTATE, 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]) ← (registers)</td>
</tr>
<tr>
<td>DONE instruction</td>
<td>(registers) ← TSTATE([n])</td>
</tr>
<tr>
<td>RETRY instruction</td>
<td>(registers) ← TSTATE([n])</td>
</tr>
<tr>
<td>RDPR (TSTATE)</td>
<td>R[rd] ← TSTATE([n])</td>
</tr>
<tr>
<td>WRPR (TSTATE)</td>
<td>TSTATE([n]) ← value</td>
</tr>
</tbody>
</table>
6.7.4 Trap Type (TT\(^P\)) Register (PR 3)

The privileged Trap Type register (TT; see FIGURE 6-32) contains the trap type of the trap that caused entry to the current trap level. There are \( \text{MAXPTL} \) 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 \textit{illegal_instruction} exception.

\[ \begin{array}{|l|} \hline \text{TT}_1^P \quad \text{Trap type from trap while } TL = 0 \\ \text{TT}_2^P \quad \text{Trap type from trap while } TL = 1 \\ \vdots \quad \vdots \\ \text{TT}_{\text{MAXPTL}}^P \quad \text{Trap type from trap while } TL = \text{MAXPTL} - 1 \\ \hline \end{array} \]

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

TABLE 6-20 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] \leftarrow \text{trap type} )</td>
</tr>
<tr>
<td>RDPR (TT)</td>
<td>( R[rd] \leftarrow TT[n] )</td>
</tr>
<tr>
<td>WRPR (TT)</td>
<td>( TT[n] \leftarrow \text{value} )</td>
</tr>
</tbody>
</table>

6.7.5 Trap Base Address (TBA\(^P\)) Register (PR 5)

The privileged Trap Base Address register (TBA), shown in FIGURE 6-33, 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.

Details on how the full address for a trap vector is generated, using TBA and other state, are provided in \textit{Trap-Table Entry Address to Privileged Mode} on page 419.
6.7.6 Processor State (PSTATE\textsuperscript{P}) Register (PR 6)

The privileged Processor State register (PSTATE), shown in FIGURE 6-34, contains control fields for the current state of the virtual processor. There is only one instance of the PSTATE register per virtual processor.

![PSTATE Field](image)

**FIGURE 6-34 PSTATE Field**

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

The following subsections describe the fields of the PSTATE register.

**Current Little Endian (cle).** This bit affects the endianness of data accesses performed using an implicit ASI. When PSTATE\textsubscript{cle} = 1, all data accesses using an implicit ASI are performed in little-endian byte order. When PSTATE\textsubscript{cle} = 0, all data accesses using an implicit ASI are performed in big-endian byte order. Specific ASIs used are shown in TABLE 7-3 on page 108. 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\textsubscript{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\textsubscript{tle} bit is copied into PSTATE\textsubscript{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\textsubscript{tle} is set to 1, data accesses using an implicit ASI in the trap handler are little-endian.

The original state of PSTATE\textsubscript{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 6-21.

<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 2005 implementations. All software written to run correctly under RMO will run correctly under TSO on an UltraSPARC Architecture 2005 implementation.
**Enable FPU (pef).** When set to 1, the PSTATE.pef bit enables the floating-point unit. This allows privileged software to manage the FPU. For the FPU to be usable, both PSTATE.pef and FPRS.fef must be set to 1. Otherwise, any floating-point instruction that tries to reference the FPU causes an fp_disabled trap.

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

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

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

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

**Programming Note**

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

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

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

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

- Before any (virtual or real) data address is sent out of the virtual processor (notably, to the memory system, which includes MMU, internal caches, and external caches); this includes ASI accesses using ASI_AS_IF_USER* in privileged mode.

- 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 PC is stored to a general-purpose register by a CALL, JMPL, or RDPC instruction (closed impl.dep. #125-V9-Cs10)

- When the values of PC and NPC are written to TPC[TL] and TNPC[TL] (respectively) during a trap (closed impl.dep. #125-V9-Cs10)
Before any virtual address is sent to a watchpoint comparator

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

When a bypassing ASI (ASI_*REAL_*) is used in a load or store instruction (see ASI 1416, ASI_REAL, for an example).

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

- When a target address is written to NPC by a control transfer instruction
  **Forward Compatibility Note** This behavior is expected to change in the next revision of the architecture, such that implementations will explicitly mask out (not preserve) the more-significant 32 bits, in this case.

- When NPC is incremented to NPC + 4 during execution of an instruction that is not a taken control transfer
  **Forward Compatibility Note** This behavior is expected to change in the next revision of the architecture, such that implementations will explicitly mask out (not preserve) the more-significant 32 bits, in this case.

- When a WRPR instruction writes to TPC[TL] or TNPC[TL]
  **Programming Note** Since writes to PSTATE are nondelayed (see page 90), a change to PSTATE.am can affect the address of the next instruction executed. Specifically, if a WRPR to the PSTATE register changes the value of PSTATE.am from ‘0’ to ‘1’, and the more-significant 32 bits of NPC when the WRPR began execution were nonzero, then the next instruction that executes after the WRPR will not be from the address in NPC when the WRPR began execution but rather from that address truncated to a 32-bit address (NPC with its more-significant 32 bits set to zero).

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

If (1) TSTATE[TL] pstate.am = 1 and (2) a DONE or RETRY instruction is executed, 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).

---

1. which sets PSTATE.am to ‘1’, by restoring the value from TSTATE[TL] pstate.am to PSTATE.am
Privileged Mode (\texttt{priv}). When \texttt{PSTATE.priv} = 1, the virtual processor is operating in privileged mode.

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

\textbf{PSTATE\_interrupt\_enable (\texttt{ie})}. \texttt{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 \texttt{PSTATE.ie} = 1. At all other times, they are held pending. For more details, see \textit{Conditioning of Disrupting Traps} on page 415.

\textbf{SPARC V9 Compatibility Note}. Since the UltraSPARC Architecture provides a more general “alternate globals” facility (through use of the \texttt{GL} register) than does SPARC V9, an UltraSPARC Architecture processor does not implement the SPARC V9 \texttt{PSTATE.ag} bit.

6.7.7 Trap Level Register (\texttt{TLP}) (PR 7) \textit{D1}

The privileged Trap Level register (TL; \textit{FIGURE 6-35}) 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.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{trap_level_register.png}
\caption{Trap Level Register}
\end{figure}

The maximum valid value that the TL register may contain is \textit{MAXPTL}, which is always equal to the number of supported trap levels beyond level 0.

\textbf{IMPL. DEP. #101-V9-CS10}: The architectural parameter \textit{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 \textit{MAXPTL} = \textit{MAXPGL} (see impl. dep. #401-S10). Architecturally, \textit{MAXPTL} must be \geq 2.
In an UltraSPARC Architecture 2005 implementation, \( MAXPTL = 2 \). See Chapter 12, *Traps*, for more details regarding the TL register.

The effect of writing to TL with a WRPR instruction is summarized in **TABLE 6-22**.

**TABLE 6-22** 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 MAXPTL )</td>
<td>Nonprivileged: ( TL \leftarrow x )</td>
</tr>
<tr>
<td>( x &gt; MAXPTL )</td>
<td>Privileged: ( TL \leftarrow 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 \( MAXPTL \leq 3 \), 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 \( MAXPTL = 2 \), writing a value of \( 05_{16} \) to the TL register causes a value of \( 1_{16} \) to actually be stored in TL.

**Implementation Note** \( MAXPTL = 2 \) for all UltraSPARC Architecture 2005 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 \( TL > 0 \) for nonprivileged software†, an UltraSPARC Architecture virtual processor’s behavior when executing with \( TL > 0 \) in nonprivileged mode is undefined.

† by executing a WRPR to TSTATE followed by DONE instruction or RETRY instruction.

### 6.7.8 Processor Interrupt Level (PIL\(^P\)) Register (PR 8)

The privileged Processor Interrupt Level register (PIL; see **FIGURE 6-36**) specifies the interrupt level above which the virtual processor will accept an *interrupt_level_n* interrupt. Interrupt priorities are mapped so that interrupt level 2 has greater priority than interrupt level 1, and so on. See **TABLE 12-4** on page 422 for a list of exception and interrupt priorities.
6.7.9 Global Level Register (GL^P) (PR 16)

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

FIGURE 6-37 illustrates the Global Level register.

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 MAXPGL, where MAXPGL is one fewer than the number of global register sets available to the virtual processor.

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

In all UltraSPARC Architecture 2005 implementations, 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 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, TL = n does not imply that GL = n, and GL = n does not imply that TL = n. Furthermore, there may be a different total number of global levels (register sets) than there are trap levels; that is, $MAXPTL$ and $MAXPGL$ 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 $MAXPGL$ to the GL register causes $MAXPGL$ to be written to GL.

The effect of writing to GL with a WRPR instruction is summarized in TABLE 6-23.

<table>
<thead>
<tr>
<th>Value $x$ Written with WRPR</th>
<th>Privilege Level when WRPR is Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leq MAXPGL$</td>
<td>Privileged</td>
</tr>
<tr>
<td>$x &gt; MAXPGL$</td>
<td>GL $\leftarrow x$</td>
</tr>
<tr>
<td>privileged_opcode</td>
<td>GL $\leftarrow MAXPGL$ (no exception generated)</td>
</tr>
</tbody>
</table>

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 7

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 99.
- Instruction Formats on page 100.
- Instruction Categories on page 101.

7.1 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 to into the PC and the target address is copied into NPC. For an immediate CTI, at the end of execution, the target is copied to PC and target + 4 is copied to NPC. In the SPARC instruction set, many CTIs do not transfer control until after a delay of one instruction, hence the term “delayed CTI” (DCTI). Thus, the two program counters provide for a delayed-branch execution model.
For each instruction access and each normal data access, 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 108) can provide an arbitrary ASI with their data addresses or can use the ASI value currently contained in the ASI register.

7.2 Instruction Formats

Instructions are encoded in four major 32-bit formats and several minor formats, as shown in FIGURE 7-1. For detailed formats for specific instructions, see individual instruction descriptions in the Instructions chapter.

### op = 00: SETHI and Branches

<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>rcond</td>
</tr>
</tbody>
</table>

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|----|-----|------|-------|
| 0   |

### op = 01: CALL

<table>
<thead>
<tr>
<th>01</th>
<th>disp30</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### op = 10 or 11: 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>=i</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>=i</td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 7-1 Summary of Instruction Formats
7.3 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.

7.3.1 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. The LDFSR/LDXFSR and the STFSR/STXFSR are special load and store instructions that load or store the floating-point status instead of acting on 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.

7.3.1.1 Memory Alignment Restrictions

A halfword access must be aligned on a 2-byte boundary, a word access (including an instruction fetch) must be aligned on a 4-byte boundary, an extended-word (LDX, LDXA, STX, STXA) or integer twin word (LDTW, LDTWA, STTW, STTWA) access must be aligned on an 8-byte boundary, an integer twin-extended-word (LDTXA) access must be aligned on a 16-byte boundary, and a Block Load (LDBLOCKF) or Store (STBLOCKF) access must be aligned on a 64-byte boundary.

A floating-point doubleword access (LDDF, LDDFA, STDF, STDFA) should be aligned on an 8-byte boundary, but is only required to be aligned on a word (4-byte) boundary. A floating-point doubleword access to an address which 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 which 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).

Programming Note: For some instructions, by using simm13, any location in the lowest or highest 4 Kbytes of an address space can be accessed without using a register to hold part of the address.
An LDQF or LDQFA instruction accessing an address that is word aligned but not quadword aligned may cause an **LDQF_mem_address_not_aligned** exception (impl. dep. #111-V9-Cs10a).

**Implementation Note** | Although the architecture provides for the **LDQF_mem_address_not_aligned** exception, UltraSPARC Architecture 2005 implementations do not currently generate it.

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

**Implementation Note** | Although the architecture provides for the **STQF_mem_address_not_aligned** exception, UltraSPARC Architecture 2005 implementations do not currently generate it.

### 7.3.1.2 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 using selected ASIs. It is also possible to change the default byte order for implicit data accesses. See Processor State (**PSTATE**\(^3\)) Register (PR 6) on page 90 for more information.\(^1\)

**Big-endian Addressing Convention.** Within a multiple-byte integer, the byte with the smallest address is the most significant; a byte’s significance decreases as its address increases. The big-endian addressing conventions are described in TABLE 7-1 and illustrated in FIGURE 7-2.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>byte</strong></td>
<td>A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.</td>
</tr>
<tr>
<td><strong>halfword</strong></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>
</tbody>
</table>

---

\(^1\) Readers interested in more background information on big- vs. little-endian can also refer to Cohen, D., “On Holy Wars and a Plea for Peace,” *Computer* 14:10 (October 1981), pp. 48-54.
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.

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

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.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<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 extended word</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, 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.†Note that the LDTXA instruction, which is not an LDTWA operation but does share LDTWA’s opcode, is not deprecated.</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>Address</td>
<td>7</td>
</tr>
<tr>
<td>------------</td>
<td>---</td>
</tr>
<tr>
<td>Halfword</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Word</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Doubleword / Extended word</td>
<td>000</td>
</tr>
<tr>
<td>Address {2:0}</td>
<td>63</td>
</tr>
<tr>
<td>Address {2:0}</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Quadword</td>
<td>0000</td>
</tr>
<tr>
<td>Address {3:0}</td>
<td>127</td>
</tr>
<tr>
<td>Address {3:0}</td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td>95</td>
</tr>
<tr>
<td>Address {3:0}</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>63</td>
</tr>
<tr>
<td>Address {3:0}</td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

**FIGURE 7-2** 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 7-2 and illustrated in FIGURE 7-3.

<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.</td>
</tr>
<tr>
<td></td>
<td>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>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

### Halfword

<table>
<thead>
<tr>
<th>Address(0)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halfword</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

### Word

<table>
<thead>
<tr>
<th>Address(1:0)</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>7</td>
<td>0</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>16</td>
<td>31</td>
<td>24</td>
</tr>
</tbody>
</table>

### Doubleword / Extended word

<table>
<thead>
<tr>
<th>Address(2:0)</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doubleword</td>
<td>7</td>
<td>0</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>16</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td>Extended</td>
<td>39</td>
<td>32</td>
<td>47</td>
<td>40</td>
</tr>
<tr>
<td>Word</td>
<td>55</td>
<td>48</td>
<td>63</td>
<td>56</td>
</tr>
</tbody>
</table>

### Quadword

<table>
<thead>
<tr>
<th>Address(3:0)</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadword</td>
<td>7</td>
<td>0</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>16</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>Extended</td>
<td>39</td>
<td>32</td>
<td>47</td>
<td>40</td>
</tr>
<tr>
<td>Word</td>
<td>55</td>
<td>48</td>
<td>63</td>
<td>56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address(3:0)</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadword</td>
<td>71</td>
<td>64</td>
<td>79</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>80</td>
<td>95</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>Extended</td>
<td>103</td>
<td>96</td>
<td>111</td>
<td>104</td>
</tr>
<tr>
<td>Word</td>
<td>119</td>
<td>112</td>
<td>127</td>
<td>120</td>
</tr>
</tbody>
</table>

**FIGURE 7-3** Little-Endian Addressing Conventions
7.3.1.3 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 7-3.

**TABLE 7-3 ASIs Used for Data Accesses and Instruction Fetches**

<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>
<tr>
<td>Non-alternate-space</td>
<td>= 0</td>
<td>0</td>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>Load, Store, or Load-</td>
<td></td>
<td>1</td>
<td>ASI_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>Store</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>
<tr>
<td>Alternate-space Load,</td>
<td>any</td>
<td>any</td>
<td>ASI explicitly specified in the instruction (subject to privilege-level restrictions)</td>
</tr>
<tr>
<td>Store, or Load-Store</td>
<td></td>
<td></td>
<td></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.
See also Memory Addressing and Alternate Address Spaces on page 369.

ASIs $00_{16}$ through $7F_{16}$ 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 7-4.

### TABLE 7-4  Allowed Accesses to ASIs

<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>$00_{16}$–$7F_{16}$</td>
<td>Restricted</td>
<td>Nonprivileged (0)</td>
<td>privileged_action 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 2005 ASIs are implementation dependent. See TABLE 10-1 on page 389 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.

### 7.3.1.4 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.
7.3.2 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.

7.3.3 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.

7.3.3.1 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 486 for details.

7.3.3.2 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.

7.3.3.3 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.
7.3.3.4 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.

7.3.3.5 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 $\text{CCR}.\text{icc}.v$ bit; if 64-bit arithmetic overflow occurs, then they set the $\text{CCR}.\text{xcc}.v$ bit.

The trapping versions (TADDccTV, TSUBccTV) of these instructions are deprecated. See Tagged Add on page 339 and Tagged Subtract on page 345 for details.

7.3.4 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 (JMPL, RETURN)
- Return from trap (DONE, RETRY)
- Trap (Tcc)

A control-transfer instruction functions by changing the value of the next program counter (NPC) or by changing the value of both the program counter (PC) and the next program counter (NPC). When only the next program counter, NPC, is changed, the effect of the transfer of control is delayed by one instruction. Most control transfers are of the delayed variety. The instruction following a delayed control-transfer instruction is said to be in the delay slot of the control-transfer instruction.

Some control transfer instructions (branches) can optionally annul, that is, not execute, the instruction in the delay slot, depending upon whether the transfer is taken or not taken. Annulled instructions have no effect upon the program-visible state, nor can they cause a trap.

TABLE 7-5 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,
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.

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>
<tr>
<td>JMPI, 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>TPC[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 effective address, EA in TABLE 7-5, 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 two bits to create a byte displacement, and adding the result to the contents of the PC.

- **Register-indirect effective address** — A register-indirect effective address computes its target address as either \( R[rs1] + R[rs2] \) if \( i = 0 \), or \( R[rs1] + \text{sign}\_\text{ext}(\text{simm13}) \) if \( i = 1 \).

- **Trap vector effective address** — A trap vector effective address first computes the software trap number as the least significant 7 or 8 bits of \( R[rs1] + R[rs2] \) if \( i = 0 \), or as the least significant 7 or 8 bits of \( R[rs1] + \text{imm}\_\text{trap}\# \) if \( i = 1 \). Whether 7 or 8 bits is 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 429 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.

## 7.3.4.1 Conditional Branches

A conditional branch transfers control if the specified condition is **true**. If the annul bit is 0, the instruction in the delay slot is always executed. If the annul bit is 1, the instruction in the delay slot is executed only when the conditional branch is taken.

**Note** The annuling behavior of a taken conditional branch is different from that of an unconditional branch.

## 7.3.4.2 Unconditional Branches

An unconditional branch transfers control unconditionally if its specified condition is “always”; it never transfers control if its specified condition is “never.” If the annul bit is 0, then the instruction in the delay slot is always executed. If the annul bit is 1, then the instruction in the delay slot is **never** executed.

**Note** The annul behavior of an unconditional branch is different from that of a taken conditional branch.
7.3.4.3 CALL and JMPL Instructions

The CALL instruction writes the contents of the PC, which points to the CALL instruction itself, into R[15] (out register 7) and then causes a delayed transfer of control to a PC-relative effective address. The value written into R[15] is visible to the instruction in the delay slot.

The JMPL instruction writes the contents of the PC, which points to the JMPL instruction itself, into R[rd] and then causes a register-indirect delayed transfer of control to the address given by “R[rs1] + R[rs2]” or “R[rs1] + a signed immediate value.” The value written into R[rd] is visible to the instruction in the delay slot.

When PSTATE.am = 1, the value of the high-order 32 bits transmitted to R[15] by the CALL instruction or to R[rd] by the JMPL instruction is zero.

7.3.4.4 RETURN Instruction

The RETURN instruction is used to return from a trap handler executing in nonprivileged mode. RETURN combines the control-transfer characteristics of a JMPL instruction with R[0] specified as the destination register and the register-window semantics of a RESTORE instruction.

7.3.4.5 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.

7.3.4.6 Trap Instruction (Tcc)

The Tcc instruction initiates a trap if the condition specified by its cond field matches the current state of the condition code register specified in its cc field; otherwise, it executes as a NOP. If the trap is taken, it increments the TL register, computes a trap type that is stored in TT[TL], and transfers to a computed address in a trap table pointed to by a trap base address register.

A Tcc instruction can specify one of 256 software trap types (128 when in nonprivileged mode). When a Tcc is taken, 256 plus the 7 (in nonprivileged mode) or 8 (in privileged mode) least significant bits of the Tcc’s second source operand are
written to TT[TL]. The only visible difference between a software trap generated by a Tcc instruction and a hardware trap is the trap number in the TT register. See Chapter 12, Traps, for more information.

**Programming Note**

Tcc can be used to implement breakpointing, tracing, and calls to privileged or hyperprivileged software. Tcc can also be used for runtime checks, such as out-of-range array index checks or integer overflow checks.

7.3.4.7 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, as on future UltraSPARC Architecture implementations that construct may execute either slowly or differently than the programmer assumes it will.

**SPARC V8 and SPARC V9 Compatibility Note**

The SPARC V8 architecture left behavior undefined for a DCTI couple. The SPARC V9 architecture defined behavior in that case, but as of UltraSPARC Architecture 2005, use of DCTI couples is deprecated.

7.3.5 Conditional Move Instructions

This subsection describes two groups of instructions that copy or move the contents of any integer or floating-point register.

**MOVcc and FMOVcc Instructions.** The MOVcc and FMOVcc instructions copy the contents of any integer or floating-point register to a destination integer or floating-point register if a condition is satisfied. The condition to test is specified in the instruction and may be any of the conditions allowed in conditional delayed control-transfer instructions. This condition is tested against one of the six sets of condition codes (icc, xcc, fcc0, fcc1, fcc2, and fcc3), as specified by the instruction. For example:

```
fmovdg %fcc2, %f20, %f22
```

moves the contents of the double-precision floating-point register %f20 to register %f22 if floating-point condition code number 2 (fcc2) indicates a greater-than-relation (FSR.fcc2 = 2). If fcc2 does not indicate a greater-than relation (FSR.fcc2 ≠ 2), then the move is not performed.

The MOVcc and FMOVcc instructions can be used to eliminate some branches in programs. In most implementations, branches will be more expensive than the MOVcc or FMOVcc instructions. For example, the following C statement:

```
if (A > B) X = 1; else X = 0;
```

can be coded as
\begin{verbatim}
    cmp     %i0, %i2     ! (A > B)
or     %g0, 0, %i3     ! set X = 0
movg    %xcc, 1, %i3     ! overwrite X with 1 if A > B
\end{verbatim}
which eliminates the need for a branch.

**MOVr and FMOVr Instructions.** The MOVr and FMOVr instructions allow the contents of any integer or floating-point register to be moved to a destination integer or floating-point register if the contents of a register satisfy a specified condition. The conditions to test are enumerated in TABLE 7-6.

**TABLE 7-6** MOVr and FMOVr Test Conditions

<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,

\begin{verbatim}
    movrnz   %i2, %l4, %l6
\end{verbatim}

moves integer register %i2 to integer register %l4, %l6 if integer register %i2 contains a nonzero value.

MOVr and FMOVr can be used to eliminate some branches in programs or can emulate multiple unsigned condition codes by using an integer register to hold the result of a comparison.

### 7.3.6 Register Window Management Instructions

This subsection describes the instructions that manage register windows in the UltraSPARC Architecture. The privileged registers affected by these instructions are described in *Register-Window PR State Registers* on page 81.

#### 7.3.6.1 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 \textit{spill}\_\textit{n} \textit{<normal|other>} exceptions.
If $\text{CANSAVE} \neq 0$ but the number of clean windows is zero, that is, \((\text{CLEANWIN} - \text{CANRESTORE}) = 0\), then $\text{SAVE}$ causes a clean_window exception.

If $\text{SAVE}$ does not cause an exception, it performs an ADD operation, decrements $\text{CANSAVE}$, and increments $\text{CANRESTORE}$. The source registers for the ADD operation are from the old window (the one to which $\text{CWP}$ pointed before the $\text{SAVE}$), while the result is written into a register in the new window (the one to which the incremented $\text{CWP}$ points).

### 7.3.6.2 RESTORE Instruction

The $\text{RESTORE}$ instruction restores the previous register window by decrementing the $\text{CWP}$ register.

If $\text{CANRESTORE} = 0$, execution of a $\text{RESTORE}$ instruction causes a window fill exception, that is, one of the fill_n_\text{normal} or fill_n_\text{other} exceptions.

If $\text{RESTORE}$ does not cause an exception, it performs an ADD operation, decrements $\text{CANRESTORE}$, and increments $\text{CANSAVE}$. The source registers for the ADD are from the old window (the one to which $\text{CWP}$ pointed before the $\text{RESTORE}$), and the result is written into a register in the new window (the one to which the decremented $\text{CWP}$ points).

**Programming Note**

This note describes a common convention for use of register windows, $\text{SAVE}$, $\text{RESTORE}$, $\text{CALL}$, and JMPL instructions.

A procedure is invoked by executing a $\text{CALL}$ (or a JMPL) instruction. If the procedure requires a register window, it executes a $\text{SAVE}$ instruction in its prologue code. A routine that does not allocate a register window of its own (possibly a leaf procedure) should not modify any windowed registers except $\text{out}$ registers 0 through 6. This optimization, called “Leaf-Procedure Optimization”, is routinely performed by SPARC compilers.

A procedure that uses a register window returns by executing both a $\text{RESTORE}$ and a JMPL instruction. A procedure that has not allocated a register window returns by executing a JMPL only. The target address for the JMPL instruction is normally 8 plus the address saved by the calling instruction, that is, the instruction after the instruction in the delay slot of the calling instruction.

The $\text{SAVE}$ and $\text{RESTORE}$ instructions can be used to atomically establish a new memory stack pointer in an R register and switch to a new or previous register window.
7.3.6.3 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 300 for details.

7.3.6.4 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 292 for details.

7.3.6.5 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.

7.3.7 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 67.
7.3.8 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.

7.3.9 Floating-Point Operate (FPop) Instructions

Floating-point operate instructions (FPops) compute a result that is a function of one or two source operands and place the result in one or more destination F registers, with one exception: floating-point compare operations do not write to an F register but update one of the fccn fields of the FSR instead.

The term “FPop” refers to instructions in the FPop1, 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 115.

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 116.

If no floating-point unit is present or if PSTATE.pef = 0 or FPRS.fef = 0, then any instruction, including an FPop instruction, that attempts to access an FPU register generates an fp_disabled exception.

All FPop instructions clear the ftt field and set the cexc field unless they generate an exception. Floating-point compare instructions also write one of the 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 or FSR.ftt = unimplemented_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 62 to see which instructions can produce an fp_exception_other exception (with FSR.ftt = unfinished_FPop). See ftt = 3 (unimplemented_FPop) on page 62 to see which instructions can produce an fp_exception_other exception (with FSR.ftt = unimplemented_FPop).
7.3.10 Implementation-Dependent Instructions

The SPARC V9 architecture provided two instruction spaces that are entirely implementation dependent: IMPDEP1 and IMPDEP2.

In the UltraSPARC Architecture, the IMPDEP1 opcode space is used by VIS instructions.

In the UltraSPARC Architecture, IMPDEP2 is subdivided into IMPDEP2A and IMPDEP2B. IMPDEP2A remains implementation dependent. The IMPDEP2B opcode space is reserved for implementation of floating-point multiply-add/multiply-subtract instructions.

7.3.11 Reserved Opcodes and Instruction Fields

If a conforming UltraSPARC Architecture 2005 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.

- \( \{r=1\} \) If the instruction bit pattern does not encode an extension to the instruction set, but would decode as a valid instruction if nonzero bits in reserved instruction field(s) were ignored (read as 0):
  - The recommended behavior is to generate an `illegal_instruction` exception (or, for FPop, an `fp_exception_other` exception with FSR.flt = 3 (unimplemented_FPop)).
  - Alternatively, the implementation can ignore the nonzero reserved field bits and execute the instruction as if those bits had been zero.

- \( \{r=1\} \) If the instruction bit pattern does not encode an extension to the instruction set and would still not decode as a valid instruction if nonzero bits in reserved instruction field(s) were ignored, then the instruction bit pattern is invalid and causes an exception. Specifically, attempting to execute an FPop instruction (see Floating-Point Operate on page 30) causes an `fp_exception_other` exception (with FSR.flt = unimplemented_FPop); attempting to execute any other invalid instruction bit pattern causes an `illegal_instruction` exception.

<table>
<thead>
<tr>
<th>Forward Compatibility Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>To further enhance backward (and forward) binary compatibility, the next revision of the UltraSPARC Architecture is expected to require an <code>illegal_instruction</code> exception to be generated by any instruction bit pattern that encodes neither a known UltraSPARC Architecture instruction nor an implementation-specific extension instruction (including those with nonzero bits in reserved instruction fields).</td>
</tr>
</tbody>
</table>
See Appendix A, *Opcode Maps*, for an enumeration of the reserved instruction bit patterns (opcodes).

<table>
<thead>
<tr>
<th>Implementation Note</th>
<th>As described above, implementations are strongly encouraged, but not strictly required, to trap on nonzero values in reserved instruction fields.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Note</td>
<td>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” (“—”).</td>
</tr>
</tbody>
</table>
Instructions

UltraSPARC Architecture 2005 extends the standard SPARC V9 instruction set with additional classes of instructions:

- **Enhanced functionality:**
  - Instructions for alignment (**Align Address** on page 135)
  - Array handling (**Three-Dimensional Array Addressing** on page 138)
  - Byte-permutation instructions ()
  - Edge handling (**Edge Handling Instructions** on pages 156 and 158)
  - Logical operations on floating-point registers (**F Register Logical Operate** (1 operand) on page 211)
  - Partitioned arithmetic (**Fixed-point Partitioned Add** on page 203 and **Fixed-point Partitioned Subtract** on page 208)
  - Pixel manipulation (**FEXPAND** on page 172, **FPACK** on page 197, and **FPMERGE** on page 206)

- **Efficient memory access**
  - Partial store (**Store Partial Floating-Point** on page 325)
  - Short floating-point loads and stores (**Store Short Floating-Point** on page 328)
  - Block load and store (**Block Load** on page 232 and **Block Store** on page 312)

- **Efficient interval arithmetic:** **SIAM** (**Set Interval Arithmetic Mode** on page 304) and all instructions that reference **GSR.im**

**TABLE 8-2** provides a quick index of instructions, alphabetically by architectural instruction name.

**TABLE 8-3** 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 8-1.

**TABLE 8-1**  Instruction Superscripts

<table>
<thead>
<tr>
<th>Superscript</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Deprecated instruction</td>
</tr>
<tr>
<td>N</td>
<td>Nonportable instruction</td>
</tr>
<tr>
<td>P</td>
<td>Privileged instruction</td>
</tr>
<tr>
<td>PASI</td>
<td>Privileged action if bit 7 of the referenced ASI is 0</td>
</tr>
<tr>
<td>PASR</td>
<td>Privileged instruction if the referenced ASR register is privileged</td>
</tr>
<tr>
<td>P_{npt}</td>
<td>Privileged action if PSTATE.piv = 0 and (S)TICK.npt = 1</td>
</tr>
<tr>
<td>PPIC</td>
<td>Privileged action if PCR.piv = 1</td>
</tr>
<tr>
<td>Page</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>134</td>
<td>ADD (ADDcc)</td>
</tr>
<tr>
<td>134</td>
<td>ADDC (ADDCcc)</td>
</tr>
<tr>
<td>135</td>
<td>ALIGNADDRESS (_LITTLE)</td>
</tr>
<tr>
<td>136</td>
<td>ALLCLEAN</td>
</tr>
<tr>
<td>137</td>
<td>AND (ANDcc)</td>
</tr>
<tr>
<td>138</td>
<td>ARRAY(8,16,32)</td>
</tr>
<tr>
<td>142</td>
<td>Biccc</td>
</tr>
<tr>
<td>144</td>
<td>BMASK</td>
</tr>
<tr>
<td>145</td>
<td>BPcc</td>
</tr>
<tr>
<td>148</td>
<td>BPr</td>
</tr>
<tr>
<td>144</td>
<td>BSHUFFLE</td>
</tr>
<tr>
<td>150</td>
<td>CALL</td>
</tr>
<tr>
<td>151</td>
<td>CASAPASI</td>
</tr>
<tr>
<td>154</td>
<td>DONEP</td>
</tr>
<tr>
<td>156</td>
<td>EDGE(8,16,32)[L]cc</td>
</tr>
<tr>
<td>158</td>
<td>EDGE(8,16,32)[L]N</td>
</tr>
<tr>
<td>218</td>
<td>F(s,d,q)TO(s,d,q)</td>
</tr>
<tr>
<td>216</td>
<td>F(s,d,q)TOi</td>
</tr>
<tr>
<td>216</td>
<td>F(s,d,q)TOx</td>
</tr>
<tr>
<td>159</td>
<td>FABS(s,d,q)</td>
</tr>
<tr>
<td>160</td>
<td>FADD(s,d,q)</td>
</tr>
<tr>
<td>161</td>
<td>FALIGNDATA</td>
</tr>
<tr>
<td>214</td>
<td>FANDNOT(1,2)[s]</td>
</tr>
<tr>
<td>214</td>
<td>FAND[s]</td>
</tr>
<tr>
<td>162</td>
<td>FBfccD</td>
</tr>
<tr>
<td>164</td>
<td>FBPfcc</td>
</tr>
<tr>
<td>169</td>
<td>FCMP(s,d,q)</td>
</tr>
<tr>
<td>166</td>
<td>FCMP&lt;16,32&gt;</td>
</tr>
<tr>
<td>169</td>
<td>FCMP(s,d,q)</td>
</tr>
<tr>
<td>171</td>
<td>FDIV(s,d,q)</td>
</tr>
<tr>
<td>194</td>
<td>FdMULq</td>
</tr>
<tr>
<td>172</td>
<td>FEXPAND</td>
</tr>
<tr>
<td>173</td>
<td>FiTO(s,d,q)</td>
</tr>
<tr>
<td>174</td>
<td>FLUSH</td>
</tr>
<tr>
<td>177</td>
<td>FLUSHW</td>
</tr>
<tr>
<td>178</td>
<td>FMOV(s,d,q)</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>RDASI</td>
<td>285</td>
</tr>
<tr>
<td>RDassP ASI</td>
<td>285</td>
</tr>
<tr>
<td>RDCCR</td>
<td>285</td>
</tr>
<tr>
<td>RDFPRS</td>
<td>285</td>
</tr>
<tr>
<td>RDGSR</td>
<td>285</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>RDPC</td>
<td>285</td>
</tr>
<tr>
<td>RDPCR P</td>
<td>285</td>
</tr>
<tr>
<td>RDPRP P</td>
<td>288</td>
</tr>
<tr>
<td>RDSOFTINT P</td>
<td>285</td>
</tr>
<tr>
<td>RDSOCK_CMPRP</td>
<td>285</td>
</tr>
<tr>
<td>RDTICK_CMPRP P</td>
<td>285</td>
</tr>
<tr>
<td>RDTICK_CMPRP P</td>
<td>285</td>
</tr>
<tr>
<td>RESTORED P</td>
<td>292</td>
</tr>
<tr>
<td>RESTOREP P</td>
<td>290</td>
</tr>
<tr>
<td>RETRY P</td>
<td>294</td>
</tr>
<tr>
<td>RETURN</td>
<td>296</td>
</tr>
<tr>
<td>SAVED P</td>
<td>300</td>
</tr>
<tr>
<td>SAVE P</td>
<td>298</td>
</tr>
<tr>
<td>SDIVD (SDIVcc D)</td>
<td>348</td>
</tr>
<tr>
<td>SDIVX</td>
<td>270</td>
</tr>
<tr>
<td>SETHI</td>
<td>302</td>
</tr>
<tr>
<td>SHUTDOWN D P</td>
<td>303</td>
</tr>
<tr>
<td>SIAM</td>
<td>304</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SLL</td>
<td>305</td>
</tr>
<tr>
<td>SLLX</td>
<td>305</td>
</tr>
<tr>
<td>SMULD (SMULcc D)</td>
<td>351</td>
</tr>
<tr>
<td>SRA</td>
<td>305</td>
</tr>
<tr>
<td>SRAX</td>
<td>305</td>
</tr>
<tr>
<td>SRL</td>
<td>305</td>
</tr>
<tr>
<td>SRLX</td>
<td>305</td>
</tr>
<tr>
<td>STB</td>
<td>307</td>
</tr>
<tr>
<td>STBAP ASI</td>
<td>308</td>
</tr>
<tr>
<td>STBAR D</td>
<td>311</td>
</tr>
<tr>
<td>STBLOCK F</td>
<td>312</td>
</tr>
<tr>
<td>Instruction</td>
<td>Category and Function</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>MOVcc</td>
<td>Move integer register if condition is satisfied</td>
</tr>
<tr>
<td>MOVr</td>
<td>Move integer register on contents of integer register</td>
</tr>
<tr>
<td>FMOV(s,d,q)</td>
<td>Floating-point move</td>
</tr>
<tr>
<td>FMOV(s,d,q)cc</td>
<td>Move floating-point register if condition is satisfied</td>
</tr>
<tr>
<td>FMOV(s,d,q)R</td>
<td>Move f-p reg. if integer reg. contents satisfy condition</td>
</tr>
<tr>
<td>FSRC(1,2)[s]</td>
<td>Copy source</td>
</tr>
<tr>
<td>FiTO(s,d,q)</td>
<td>Convert 32-bit integer to floating-point</td>
</tr>
<tr>
<td>F(s,d,q)TOi</td>
<td>Convert floating point to integer</td>
</tr>
<tr>
<td>F(s,d,q)TOx</td>
<td>Convert floating point to 64-bit integer</td>
</tr>
<tr>
<td>F(s,d,q)TO(s,d,q)</td>
<td>Convert between floating-point formats</td>
</tr>
<tr>
<td>FxTO(s,d,q)</td>
<td>Convert 64-bit integer to floating-point</td>
</tr>
<tr>
<td>AND (ANDcc)</td>
<td>Logical <strong>and</strong> (and modify condition codes)</td>
</tr>
<tr>
<td>OR (ORcc)</td>
<td>Inclusive-<strong>or</strong> (and modify condition codes)</td>
</tr>
<tr>
<td>ORN (ORNcc)</td>
<td>Inclusive-<strong>or</strong> not (and modify condition codes)</td>
</tr>
<tr>
<td>XNOR (XNORcc)</td>
<td>Exclusive-<strong>nor</strong> (and modify condition codes)</td>
</tr>
<tr>
<td>XOR (XORcc)</td>
<td>Exclusive-<strong>or</strong> (and modify condition codes)</td>
</tr>
<tr>
<td>FAND[s]</td>
<td>Logical <strong>and</strong> operation</td>
</tr>
<tr>
<td>FANDNOT(1,2)[s]</td>
<td>Logical <strong>and</strong> operation with one inverted source</td>
</tr>
<tr>
<td>FNAND[s]</td>
<td>Logical <strong>nand</strong> operation</td>
</tr>
<tr>
<td>FNOR[s]</td>
<td>Logical <strong>nor</strong> operation</td>
</tr>
<tr>
<td>FNOT(1,2) (s)</td>
<td>Copy negated source</td>
</tr>
<tr>
<td>FONE[s]</td>
<td>One fill</td>
</tr>
<tr>
<td>FOR[s]</td>
<td>Logical <strong>or</strong> operation</td>
</tr>
<tr>
<td>FORNOT(1,2)[s]</td>
<td>Logical <strong>or</strong> operation with one inverted source</td>
</tr>
<tr>
<td>FXNOR[s]</td>
<td>Logical <strong>xor</strong> operation</td>
</tr>
<tr>
<td>FXOR[s]</td>
<td>Logical <strong>xor</strong> operation</td>
</tr>
<tr>
<td>FZERO[s]</td>
<td>Zero fill</td>
</tr>
<tr>
<td>SLL</td>
<td>Shift left logical</td>
</tr>
<tr>
<td>SLLX</td>
<td>Shift left logical, extended</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td>SRAX</td>
<td>Shift right arithmetic, extended</td>
</tr>
</tbody>
</table>
TABLE 8-3  Instruction Set - by Functional Category (2 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRL</td>
<td>Shift right logical</td>
<td>305</td>
<td>VIS 1</td>
</tr>
<tr>
<td>SRLX</td>
<td>Shift right logical, extended</td>
<td>305</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

**Special Addressing Operations**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGNADDRESS[_LITTLE]</td>
<td>Calculate address for misaligned data</td>
<td>135</td>
<td>VIS 1</td>
</tr>
<tr>
<td>ARRAY(8,16,32)</td>
<td>3-D array addressing instructions</td>
<td>138</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FALIGNDATA</td>
<td>Perform data alignment for misaligned data</td>
<td>161</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

**Control Transfers**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bicc</td>
<td>Branch on integer condition codes</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>BPcc</td>
<td>Branch on integer condition codes with prediction</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>BPr</td>
<td>Branch on contents of integer register with prediction</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>Call and link</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>DONEP</td>
<td>Return from trap</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>FBfccD</td>
<td>Branch on floating-point condition codes</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>FBPfcc</td>
<td>Branch on floating-point condition codes with prediction</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>ILLTRAP</td>
<td>Illegal instruction</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>JMLP</td>
<td>Jump and link</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>RETRYP</td>
<td>Return from trap and retry</td>
<td>294</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>Return</td>
<td>296</td>
<td></td>
</tr>
<tr>
<td>Tcc</td>
<td>Trap on integer condition codes</td>
<td>342</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>144</td>
<td>VIS 2</td>
</tr>
<tr>
<td>BSHUFFLE</td>
<td>Permute bytes as specified by GSR.mask</td>
<td>144</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>172</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPACK(16,32, FIX)</td>
<td>Pixel packing</td>
<td>197</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPMERGE</td>
<td>Pixel merge</td>
<td>206</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>232</td>
<td>VIS 1</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>Block stores</td>
<td>312</td>
<td>VIS 1</td>
</tr>
<tr>
<td>LDDF</td>
<td>Load double floating-point</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>LDDFA_PAS</td>
<td>Load double floating-point from alternate space</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>LDF</td>
<td>Load floating-point</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>LDFAS</td>
<td>Load floating-point from alternate space</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>LDQF</td>
<td>Load quad floating-point</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>LDQFA_PAS</td>
<td>Load quad floating-point from alternate space</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>LDHORDF</td>
<td>Short floating-point loads</td>
<td>245</td>
<td>VIS 1</td>
</tr>
<tr>
<td>STDF</td>
<td>Store double floating-point</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Category and Function</td>
<td>Page</td>
<td>Ext. to V9?</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>STDFAPASI</td>
<td>Store double floating-point into alternate space</td>
<td>319</td>
<td></td>
</tr>
<tr>
<td>STF</td>
<td>Store floating-point</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>STFAPASI</td>
<td>Store floating-point into alternate space</td>
<td>319</td>
<td></td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>Partial Store instructions</td>
<td>325</td>
<td>VIS 1</td>
</tr>
<tr>
<td>STQF</td>
<td>Store quad floating point</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>STQFAPASI</td>
<td>Store quad floating-point into alternate space</td>
<td>319</td>
<td></td>
</tr>
<tr>
<td>STSHORTF</td>
<td>Short floating-point stores</td>
<td>328</td>
<td>VIS 1</td>
</tr>
<tr>
<td>LDFSR</td>
<td>Load floating-point state register lower</td>
<td>243</td>
<td></td>
</tr>
<tr>
<td>LDXFSR</td>
<td>Load floating-point state register</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>MEMBAR</td>
<td>Memory barrier</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>PREFETCH</td>
<td>Prefetch data</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>PREFETCHAPASI</td>
<td>Prefetch data from alternate space</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>STFSRD</td>
<td>Store floating-point state register</td>
<td>323</td>
<td></td>
</tr>
<tr>
<td>STXFSR</td>
<td>Store extended floating-point state register</td>
<td>316</td>
<td></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>CASAPASI</td>
<td>Compare and swap word in alternate space</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>CASXAPASI</td>
<td>Compare and swap doubleword in alternate space</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>LDSTUB</td>
<td>Load-store unsigned byte</td>
<td>247</td>
<td></td>
</tr>
<tr>
<td>LDSTUBAPASI</td>
<td>Load-store unsigned byte from alternate space</td>
<td>248</td>
<td></td>
</tr>
<tr>
<td>SWAPD</td>
<td>Swap integer register with memory</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>SWAPAPASI</td>
<td>Swap integer register with memory in alternate space</td>
<td>337</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>LDSB</td>
<td>Load signed byte</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>LDSBAPASI</td>
<td>Load signed byte from alternate space</td>
<td>229</td>
<td></td>
</tr>
<tr>
<td>LDSH</td>
<td>Load signed halfword</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>LDSHAAPASI</td>
<td>Load signed halfword from alternate space</td>
<td>229</td>
<td></td>
</tr>
<tr>
<td>LDSW</td>
<td>Load signed word</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>LDWSWAPASI</td>
<td>Load signed word from alternate space</td>
<td>229</td>
<td></td>
</tr>
<tr>
<td>LDTXAN</td>
<td>Load integer twin extended word from alternate space</td>
<td>250</td>
<td>VIS 2+</td>
</tr>
<tr>
<td>LDTWD, PASI</td>
<td>Load integer twin word</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>LDTWAD, PASI</td>
<td>Load integer twin word from alternate space</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>LDUB</td>
<td>Load unsigned byte</td>
<td>247</td>
<td></td>
</tr>
<tr>
<td>LDUBAPASI</td>
<td>Load unsigned byte from alternate space</td>
<td>229</td>
<td></td>
</tr>
<tr>
<td>LDUH</td>
<td>Load unsigned halfword</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>LDUHAAPASI</td>
<td>Load unsigned halfword from alternate space</td>
<td>229</td>
<td></td>
</tr>
<tr>
<td>LDUW</td>
<td>Load unsigned word</td>
<td>227</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 8-3 Instruction Set - by Functional Category (4 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUWA&lt;sub&gt;P,ASI&lt;/sub&gt;</td>
<td>Load unsigned word from alternate space</td>
<td>229</td>
</tr>
<tr>
<td>LDX</td>
<td>Load extended</td>
<td>227</td>
</tr>
<tr>
<td>LDXA&lt;sub&gt;P,ASI&lt;/sub&gt;</td>
<td>Load extended from alternate space</td>
<td>229</td>
</tr>
<tr>
<td>STB</td>
<td>Store byte</td>
<td>307</td>
</tr>
<tr>
<td>STB&lt;sub&gt;P,ASI&lt;/sub&gt;</td>
<td>Store byte into alternate space</td>
<td>308</td>
</tr>
<tr>
<td>STBAR&lt;sub&gt;D&lt;/sub&gt;</td>
<td>Store barrier</td>
<td>311</td>
</tr>
<tr>
<td>STTW&lt;sub&gt;D&lt;/sub&gt;</td>
<td>Store twin word</td>
<td>330</td>
</tr>
<tr>
<td>STTWA&lt;sub&gt;D, P,ASI&lt;/sub&gt;</td>
<td>Store twin word into alternate space</td>
<td>332</td>
</tr>
<tr>
<td>STH</td>
<td>Store halfword</td>
<td>307</td>
</tr>
<tr>
<td>STH&lt;sub&gt;P,ASI&lt;/sub&gt;</td>
<td>Store halfword into alternate space</td>
<td>308</td>
</tr>
<tr>
<td>STW</td>
<td>Store word</td>
<td>307</td>
</tr>
<tr>
<td>STWA&lt;sub&gt;P,ASI&lt;/sub&gt;</td>
<td>Store word into alternate space</td>
<td>308</td>
</tr>
<tr>
<td>STX</td>
<td>Store extended</td>
<td>307</td>
</tr>
<tr>
<td>STX&lt;sub&gt;P,ASI&lt;/sub&gt;</td>
<td>Store extended into alternate space</td>
<td>308</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(s,d,q)</td>
<td>Floating-point absolute value</td>
<td>159</td>
</tr>
<tr>
<td>FADD(s,d,q)</td>
<td>Floating-point add</td>
<td>160</td>
</tr>
<tr>
<td>FDIV(s,d,q)</td>
<td>Floating-point divide</td>
<td>171</td>
</tr>
<tr>
<td>FdMULq</td>
<td>Floating-point multiply double to quad</td>
<td>194</td>
</tr>
<tr>
<td>FMUL(s,d,q)</td>
<td>Floating-point multiply</td>
<td>194</td>
</tr>
<tr>
<td>FNEG(s,d,q)</td>
<td>Floating-point negate</td>
<td>196</td>
</tr>
<tr>
<td>FsMULd</td>
<td>Floating-point multiply single to double</td>
<td>194</td>
</tr>
<tr>
<td>FSQRT(s,d,q)</td>
<td>Floating-point square root</td>
<td>215</td>
</tr>
<tr>
<td>FSUB(s,d,q)</td>
<td>Floating-point subtract</td>
<td>220</td>
</tr>
</tbody>
</table>

**Floating-Point Comparison Operations**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMP*&lt;sub&gt;≤&lt;/sub&gt;16,32&gt;</td>
<td>Compare four 16-bit signed values or two 32-bit signed values</td>
<td>166</td>
</tr>
<tr>
<td>FCMP(s,d,q)</td>
<td>Floating-point compare</td>
<td>169</td>
</tr>
<tr>
<td>FCMPE(s,d,q)</td>
<td>Floating-point compare (exception if unordered)</td>
<td>169</td>
</tr>
</tbody>
</table>

**Register-Window Control Operations**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLCLEAN</td>
<td>Mark all register window sets as “clean”</td>
<td>136</td>
</tr>
<tr>
<td>INVALW</td>
<td>Mark all register window sets as “invalid”</td>
<td>225</td>
</tr>
<tr>
<td>FLUSHW</td>
<td>Flush register windows</td>
<td>177</td>
</tr>
<tr>
<td>NORMALW</td>
<td>“Other” register windows become “normal” register windows</td>
<td>272</td>
</tr>
<tr>
<td>OTHERW</td>
<td>“Normal” register windows become “other” register windows</td>
<td>274</td>
</tr>
<tr>
<td>RESTORE&lt;sup&gt;p&lt;/sup&gt;</td>
<td>Restore caller’s window</td>
<td>290</td>
</tr>
<tr>
<td>RESTORED&lt;sup&gt;p&lt;/sup&gt;</td>
<td>Window has been restored</td>
<td>292</td>
</tr>
<tr>
<td>SAVE&lt;sup&gt;p&lt;/sup&gt;</td>
<td>Save caller’s window</td>
<td>298</td>
</tr>
</tbody>
</table>
### TABLE 8-3  Instruction Set - by Functional Category (5 of 6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Ext. to V9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVED&lt;sup&gt;P&lt;/sup&gt;</td>
<td>Window has been saved</td>
<td>300</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>174</td>
<td></td>
</tr>
<tr>
<td>IMPDEP2A</td>
<td>Implementation-dependent instructions</td>
<td>223</td>
<td></td>
</tr>
<tr>
<td>IMPDEP2B</td>
<td>Implementation-dependent instructions (reserved)</td>
<td>223</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>SHUTDOWN&lt;sup&gt;D,P&lt;/sup&gt;</td>
<td>Shut down the virtual processor</td>
<td>303</td>
<td>VIS 1</td>
</tr>
<tr>
<td><strong>Integer SIMD Operations on F Registers</strong></td>
<td></td>
<td></td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPADD&lt;16,32&gt;[S]</td>
<td>Fixed-point partitioned add</td>
<td>203</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPSUB&lt;16,32&gt;[S]</td>
<td>Fixed-point partitioned subtract</td>
<td>208</td>
<td>VIS 1</td>
</tr>
<tr>
<td><strong>Integer Arithmetic Operations on R Registers</strong></td>
<td></td>
<td></td>
<td>VIS 1</td>
</tr>
<tr>
<td>ADD (ADDcc)</td>
<td>Add (and modify condition codes)</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>ADC (ADDcc)</td>
<td>Add with carry (and modify condition codes)</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>MULScc&lt;sup&gt;D&lt;/sup&gt;</td>
<td>Multiply step (and modify condition codes)</td>
<td>268</td>
<td></td>
</tr>
<tr>
<td>MULX</td>
<td>Multiply 64-bit integers</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>SDIV&lt;sup&gt;D&lt;/sup&gt; (SDIVcc&lt;sup&gt;D&lt;/sup&gt;)</td>
<td>32-bit signed integer divide (and modify condition codes)</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>SDIVX</td>
<td>64-bit signed integer divide</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>SMUL&lt;sup&gt;D&lt;/sup&gt; (SMULcc&lt;sup&gt;D&lt;/sup&gt;)</td>
<td>Signed integer multiply (and modify condition codes)</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>SUB (SUBcc)</td>
<td>Subtract (and modify condition codes)</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>SUBC (SUBCcc)</td>
<td>Subtract with carry (and modify condition codes)</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>TADDcc</td>
<td>Tagged add and modify condition codes (trap on overflow)</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td>TADDccTV&lt;sup&gt;D&lt;/sup&gt;</td>
<td>Tagged add and modify condition codes (trap on overflow)</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td>TSUBcc</td>
<td>Tagged subtract and modify condition codes (trap on overflow)</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>TSUBccTV&lt;sup&gt;D&lt;/sup&gt;</td>
<td>Tagged subtract and modify condition codes (trap on overflow)</td>
<td>346</td>
<td></td>
</tr>
<tr>
<td>UDIV&lt;sup&gt;D&lt;/sup&gt; (UDIVcc&lt;sup&gt;D&lt;/sup&gt;)</td>
<td>Unsigned integer divide (and modify condition codes)</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>UDIVX</td>
<td>64-bit unsigned integer divide</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>UMUL&lt;sup&gt;D&lt;/sup&gt; (UMULcc&lt;sup&gt;D&lt;/sup&gt;)</td>
<td>Unsigned integer multiply (and modify condition codes)</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td><strong>Integer Arithmetic Operations on F Registers</strong></td>
<td></td>
<td></td>
<td>VIS 1</td>
</tr>
<tr>
<td>FMUL8x16</td>
<td>8x16 partitioned product</td>
<td>188</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FMUL8x16(AU,AL)</td>
<td>8x16 upper/lower α partitioned product</td>
<td>188</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FMUL8(SU,UL)x16</td>
<td>8x16 upper/lower partitioned product</td>
<td>188</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FMULD8(SU,UL)x16</td>
<td>8x16 upper/lower partitioned product</td>
<td>188</td>
<td>VIS 1</td>
</tr>
<tr>
<td><strong>Miscellaneous Operations on R Registers</strong></td>
<td></td>
<td></td>
<td>VIS 1</td>
</tr>
<tr>
<td>POPC</td>
<td>Population count</td>
<td>276</td>
<td></td>
</tr>
<tr>
<td>SETHI</td>
<td>Set high 22 bits of low word of integer register</td>
<td>302</td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous Operations on F Registers</strong></td>
<td></td>
<td></td>
<td>VIS 1</td>
</tr>
<tr>
<td>EDGE(8,16,32)[L]cc</td>
<td>Edge handling instructions (and modify condition codes)</td>
<td>156</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 8-3 Instruction Set - by Functional Category (6 of 6)

<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(8,16,32)(L)N</td>
<td>Edge handling instructions</td>
<td>158</td>
<td>VIS 2</td>
</tr>
<tr>
<td>PDIST</td>
<td>Pixel component distance</td>
<td>275</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>285</td>
<td></td>
</tr>
<tr>
<td>RDasrPASR</td>
<td>Read ancillary state register</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDCCR</td>
<td>Read Condition Codes register (CCR)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDFPRS</td>
<td>Read Floating-Point Registers State register (FPRS)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDGSR</td>
<td>Read General Status register (GSR)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDPC</td>
<td>Read Program Counter register (PC)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDPCR(^P)</td>
<td>Read Performance Control register (PCR)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDPIC(^P)</td>
<td>Read Performance Instrumentation Counters register (PIC)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDPR(^P)</td>
<td>Read privileged register</td>
<td>288</td>
<td></td>
</tr>
<tr>
<td>RDSTICK(^P)</td>
<td>Read System Tick register (STICK)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDSTICK_CMPRP</td>
<td>Read System Tick Compare register (STICK_CMPR)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDTICK(^P)</td>
<td>Read Tick register (TICK)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>RDTICK_CMPRP</td>
<td>Read Tick Compare register (TICK_CMPR)</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>SIAM</td>
<td>Set interval arithmetic mode</td>
<td>304</td>
<td>VIS 2</td>
</tr>
<tr>
<td>WRASI</td>
<td>Write ASI register</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRasrPASR</td>
<td>Write ancillary state register</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRCCR</td>
<td>Write Condition Codes register (CCR)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRFPRS</td>
<td>Write Floating-Point Registers State register (FPRS)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRGSR</td>
<td>Write General Status register (GSR)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRPCR(^P)</td>
<td>Write Performance Control register (PCR)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRPIC(^P)</td>
<td>Write Performance Instrumentation Counters register (PIC)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRPR(^P)</td>
<td>Write privileged register</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>WRSOFTINT(^P)</td>
<td>Write per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRSOFTINT_CLR(^P)</td>
<td>Clear bits of per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRSOFTINT_SET(^P)</td>
<td>Set bits of per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRTICK(^P)</td>
<td>Write Tick Compare register (TICK_CMPR)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRSTICK(^P)</td>
<td>Write System Tick register (STICK)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRSTICK_CMPRP</td>
<td>Write System Tick Compare register (STICK_CMPR)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>WRY(^D)</td>
<td>Write Y register</td>
<td>353</td>
<td></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)

(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 120.

   **Note** Instruction classes are subject to change, and are not yet defined in this document. The classes will be defined in a later draft of this document and in the meantime are subject to change.

(3) Description. This subsection describes the operation of the instruction, its features, restrictions, and exception-causing conditions.

(4) Exceptions. The exception that can occur as a consequence of attempting to execute the instruction(s). Exceptions due to an instruction_access_exception, and interrupts are not listed because they can occur on any instruction. An FPop that is not implemented in hardware generates an fp_exception_other exception with FSR.itt = unimplemented_FPop when executed. A non-FPop 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 428), 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.
### 8.1 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 reg&lt;rs1&gt;, reg_or_imm, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>ADDcc</td>
<td>01000</td>
<td>Add and modify cc’s</td>
<td>addcc reg&lt;rs1&gt;, reg_or_imm, reg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>ADDC</td>
<td>00100</td>
<td>Add with 32-bit Carry</td>
<td>adde reg&lt;rs1&gt;, reg_or_imm, reg&lt;rd&gt;</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 reg&lt;rs1&gt;, reg_or_imm, reg&lt;rd&gt;</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] + sign_ext (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] + icc,c” and if i = 1, they compute “R[rs1] + sign_ext (simm13) + icc,c”. In either case, the sum is written to R[rd].

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,c), not the 64-bit condition codes’ carry bit (CCR.xcc.c).

**SPARC V8 Compatibility Note**

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

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

**Exceptions**

*illegal_instruction*
ALIGNADDRESS

8.2 Align Address [VIS1]

<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>ALIGNADDRESS</td>
<td>00001</td>
<td>Calculate address for misaligned data access</td>
<td><code>alignaddr rs1, rs2, rd</code></td>
<td>A1</td>
</tr>
<tr>
<td>ALIGNADDRESS_LITTLE</td>
<td>00001</td>
<td>Calculate address for misaligned data access little-endian</td>
<td><code>alignaddrL rs1, rs2, rd</code></td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

ALIGNADDRESS adds two integer values, R[rs1] and R[rs2], and stores the result (with the least significant 3 bits forced to 0) in the integer register R[rd]. The least significant 3 bits of the result are stored in the GSR.align field.

ALIGNADDRESS_LITTLE is the same as ALIGNADDRESS except that the two’s complement of the least significant 3 bits of the result is stored in GSR.align.

**Note**

ALIGNADDRESS_LITTLE generates the opposite-endian byte ordering for a subsequent FALIGNDATA operation.

A byte-aligned 64-bit load can be performed as shown below.

```
alignaddr Address, Offset, Address !set GSR.align
ldd [Address], %d0
ldd [Address + 8], %d2
faligndata %d0, %d2, %d4 !use GSR.align to select bytes
```

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction causes an `fp_disabled` exception.

**Exceptions**

`fp_disabled`

**See Also**

Align Data on page 161
### ALLCLEAN

#### 8.3 Mark All Register Window Sets “Clean”

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLCLEANP</td>
<td>Mark all register window sets as “clean”</td>
<td>allclean</td>
<td>C1</td>
</tr>
</tbody>
</table>

```
|   10 | fcn = 0 0010 | 11 0001 | —      |
```

**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.

**Exceptions**
- illegal_instruction (not implemented in hardware in UltraSPARC Architecture 2005)
- privileged_opcode

**See Also**
- INVALW on page 225
- NORMALW on page 272
- OTHERW on page 274
- RESTORED on page 292
- SAVED on page 300
8.4 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:

- \( \text{icc} . v \), \( \text{icc} . c \), \( \text{xcc} . v \), and \( \text{xcc} . c \) are set to 0
- \( \text{icc} . n \) is copied from bit 31 of the result
- \( \text{xcc} . n \) is copied from bit 63 of the result
- \( \text{icc} . z \) is set to 1 if bits 31:0 of the result are zero (otherwise to 0)
- \( \text{xcc} . z \) 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.

---

### Description

> These instructions implement bitwise logical and operations. They compute “R[rs1] op R[rs2]” if \( i = 0 \), or “R[rs1] op sign_ext(simm13)” if \( i = 1 \), and write the result into R[rd].

### Exceptions

> illegal_instruction
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 8-4**. Illegal values produce undefined results in the destination register, $R[rd]$.

**TABLE 8-4  3D R[rs2] Array X and Y Dimensions**

<table>
<thead>
<tr>
<th>$R[rs2]$ Value ($n$)</th>
<th>Number of Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>3</td>
<td>512</td>
</tr>
<tr>
<td>4</td>
<td>1024</td>
</tr>
<tr>
<td>5</td>
<td>2048</td>
</tr>
</tbody>
</table>

**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.
ARRAY<8|16|32>

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 $N \times N \times M$ volume, where $N = 2^n \times 64$, $M = m \times 32$, $0 \leq n \leq 5$, $1 \leq m \leq 16$ should be composed of $64 \times 64 \times 32$ smaller volumes, which in turn should be composed of $4 \times 4 \times 2$ volumes. This data structure is optimal for 16-bit data. For 16-bit data, the $4 \times 4 \times 2$ volume has 64 bytes of data, which is ideal for reducing cache-line misses; the $64 \times 64 \times 32$ volume will have 256 Kbytes of data, which is good for improving the TLB hit rate. FIGURE 8-1 illustrates how the data has to be organized, where the origin (0,0,0) is assumed to be at the lower-left front corner and the $x$ coordinate varies faster than $y$ than $z$. That is, when traversing the volume from the origin to the upper right back, you go from left to right, front to back, bottom to top.

![FIGURE 8-1  Blocked-Byte Data Formatting Structure](image)

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 8-2.

![FIGURE 8-2 Three-Dimensional Array Fixed-Point Address Format](image)

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 8-3 for ARRAY8, FIGURE 8-4 for ARRAY16, and FIGURE 8-5 for ARRAY32.

![FIGURE 8-3 Three-Dimensional Array Blocked-Address Format (ARRAY8)](image)

![FIGURE 8-4 Three-Dimensional Array Blocked-Address Format (ARRAY16)](image)
FIGURE 8-5 Three Dimensional Array Blocked-Address Format (ARRAY32)

The bits above Z upper are set to 0. The number of zeroes in the least significant bits is determined by the element size. An element size of 8 bits has no zeroes, an element size of 16 bits has one zero, and an element size of 32 bits has two zeroes. Bits in X and Y above the size specified by R[rs2] are ignored.

<table>
<thead>
<tr>
<th>TABLE 8-5</th>
<th>ARRAY8 Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result [R[rd]]</td>
<td>Source [R[rs1]]</td>
</tr>
<tr>
<td>1:0</td>
<td>12:11</td>
</tr>
<tr>
<td>3:2</td>
<td>34:33</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
</tr>
<tr>
<td>8:5</td>
<td>16:13</td>
</tr>
<tr>
<td>12:9</td>
<td>38:35</td>
</tr>
<tr>
<td>16:13</td>
<td>59:56</td>
</tr>
<tr>
<td>17+n-1:17</td>
<td>17+n-1:17</td>
</tr>
<tr>
<td>17+2n-1:17+1</td>
<td>39+n-1:39</td>
</tr>
<tr>
<td>20+2n:17+2n</td>
<td>63:60</td>
</tr>
<tr>
<td>63:20+2n+1</td>
<td>n/a</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.

```
add    Addr, DeltaAddr, Addr
array8 Addr, %g0, bAddr
ldda   [bAddr] #ASI_FL8_PRIMARY, data
falign data, accum, accum
```

**Exceptions** None
8.6 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>1000</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>1001</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>1010</td>
<td>Branch on Greater</td>
<td>not (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>1011</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 or Equal, 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>bvcc{,a} label</td>
<td>A1</td>
</tr>
<tr>
<td>BVS</td>
<td>0111</td>
<td>Branch on Overflow Set</td>
<td>V</td>
<td>bvss{,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.
Bicc Instructions

BA (Branch Always) causes an unconditional PC-relative, delayed control transfer to the address “PC + (4 × sign_ext (disp22))”. If the annul (a) bit of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul bit is 0 (a = 0), the delay instruction is executed.

- **icc-conditional branches** — Conditional Bicc instructions (all except BA and BN) evaluate the 32-bit integer condition codes (icc), according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address “PC + (4 × sign_ext (disp22))”. If FALSE, the branch is not taken.

If a conditional branch is taken, the delay instruction is always executed regardless of the value of the annul field. If a conditional branch is not taken and the annul bit is 1 (a = 1), the delay instruction is annulled (not executed).

**Note** | The annul bit has a different effect on conditional branches than it does on unconditional branches.

Annullment, delay instructions, and delayed control transfers are described further in Chapter 7, *Instruction Set Overview*.

**Exceptions** | None
8.7 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[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.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a BMASK or BSHUFFLE instruction causes an fp_disabled exception.

Exceptions

fp_disabled
8.8 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) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPN 0000</td>
<td>Branch Never</td>
<td>0</td>
<td>bn, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPNE 1001</td>
<td>Branch on Not Equal</td>
<td>not Z</td>
<td>bne, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPE 0001</td>
<td>Branch on Equal</td>
<td>Z</td>
<td>be, a, (pt, pn) i_or_x_cc, label A1</td>
<td></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) i_or_x_cc, label A1</td>
<td></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) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BGPE 1011</td>
<td>Branch on Greater or Equal</td>
<td>not (N xor V)</td>
<td>bge, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPL 0011</td>
<td>Branch on Less</td>
<td>N xor V</td>
<td>bl, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPGU 1100</td>
<td>Branch on Greater Unsigned</td>
<td>not (C or Z)</td>
<td>bgu, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPLEU 0100</td>
<td>Branch on Less or Equal Unsigned</td>
<td>C or Z</td>
<td>bleu, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPCC 1101</td>
<td>Branch on Carry Clear</td>
<td>not C</td>
<td>bcc, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPCS 0101</td>
<td>Branch on Carry Set</td>
<td>C</td>
<td>bcs, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPPOS 1110</td>
<td>Branch on Positive</td>
<td>not N</td>
<td>bpos, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPNeg 0110</td>
<td>Branch on Negative</td>
<td>N</td>
<td>bneg, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPVC 1111</td>
<td>Branch on Overflow Clear</td>
<td>not V</td>
<td>bvc, a, (pt, pn) i_or_x_cc, label A1</td>
<td></td>
</tr>
<tr>
<td>BPVS 0111</td>
<td>Branch on Overflow Set</td>
<td>V</td>
<td>bvs, a, (pt, pn) i_or_x_cc, label A1</td>
<td></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 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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc0</th>
<th>Condition Code</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>
Unconditional branches and conditional branches are described below.

**Unconditional branches (BPA, BPN)** — A BPN (Branch Never with Prediction) instruction for this branch type \( \text{op2} = 1 \) may be used in the SPARC V9 architecture as an instruction prefetch; that is, the effective address \( \text{PC} + (4 \times \text{sign}_\text{ext}(\text{disp}19)) \) 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 \( \text{PC} + (4 \times \text{sign}_\text{ext}(\text{disp}19)) \). 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 \( \text{icc} \) or \( \text{xcc} \), as selected by \( \text{cc0} \) and \( \text{cc1} \), according to the \text{cond} field of the instruction, producing either a \text{TRUE} or \text{FALSE} result. If \text{TRUE}, the branch is taken; that is, the instruction causes a PC-relative, delayed control transfer to the address \( \text{PC} + (4 \times \text{sign}_\text{ext}(\text{disp}19)) \). If \text{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 7, *Instruction Set Overview*.

An attempt to execute a BPcc instruction with \( \text{cc0} = 1 \) (a reserved value) causes an \text{illegal_instruction} exception.

**Exceptions**

*illegal_instruction*
BPcc

See Also Branch on Integer Register with Prediction (BPr) on page 148
8.9 Branch on Integer Register with Prediction (BPr)

<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, pn]</td>
<td>regstr, label</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, pn]</td>
<td>regstr, label</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, pn]</td>
<td>regstr, label</td>
</tr>
<tr>
<td>BRGEZ</td>
<td>111</td>
<td>Branch on Register Greater Than Zero</td>
<td>R[rs1] ≥ 0</td>
<td>brgez [a], [pt, pn]</td>
<td>regstr, label</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00</th>
<th>a</th>
<th>0</th>
<th>rcond</th>
<th>011</th>
<th>d16hi</th>
<th>p</th>
<th>rs1</th>
<th>14</th>
<th>13</th>
<th>d16lo</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>25</td>
<td>24</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
</tbody>
</table>

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

**Programming Note**

To set the annul (a) bit for BPr instructions, append “,a” to the opcode mnemonic. For example, use “brz, a %i3, label.” In the preceding table, braces signify that the “,a” is optional. To set the branch prediction bit p, append either “,pt” for predict taken or “,pn” for predict not taken to the opcode mnemonic. If neither “,pt” nor “,pn” is specified, the assembler defaults to “,pt”.

**Description**

These instructions branch based on the contents of R[rs1]. They treat the register contents as a signed integer value.

A BPr instruction examines all 64 bits of R[rs1] according to the rcond field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken; that is, the instruction causes a PC-relative, delayed control transfer to the address “PC + (4 × sign_ext (d16hi :: d16lo))”. If FALSE, the branch is not taken.

If the branch is taken, the delay instruction is always executed, regardless of the value of the annul (a) bit. If the branch is not taken and the annul bit is 1 (a = 1), the delay instruction is annulled (not executed).
BPr

The predict bit \( (p) \) gives the hardware a hint about whether the branch is expected to be taken. If \( p = 1 \), the branch is expected to be taken; \( p = 0 \) indicates that the branch is expected not to be taken.

An attempt to execute a BPr instruction when instruction bit 28 = 1 or \( rcond \) is a reserved value (000_2 or 100_2) causes an \textit{illegal_instruction} exception.

Annullment, delay instructions, prediction, and delayed control transfers are described further in Chapter 7, \textit{Instruction Set Overview}.

<table>
<thead>
<tr>
<th>Implementation Note</th>
<th>If this instruction is implemented by tagging each register value with an N (negative) bit and Z (zero) bit, the table below can be used to determine if ( rcond ) is \textsc{true}:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branch</strong></td>
<td><strong>Test</strong></td>
</tr>
<tr>
<td>BRNZ</td>
<td>\text{not Z}</td>
</tr>
<tr>
<td>BRZ</td>
<td>Z</td>
</tr>
<tr>
<td>BRGEZ</td>
<td>\text{not N}</td>
</tr>
<tr>
<td>BRLZ</td>
<td>N</td>
</tr>
<tr>
<td>BRLEZ</td>
<td>N \text{ or Z}</td>
</tr>
<tr>
<td>BRGZ</td>
<td>\text{not (N or Z)}</td>
</tr>
</tbody>
</table>

\textit{Exceptions} \textit{illegal_instruction}

\textit{See Also} Branch on Integer Condition Codes with Prediction (BPcc) on page 145
8.10 Call and Link

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OP</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>01</td>
<td>Call and Link</td>
<td>call label</td>
<td>A1</td>
</tr>
</tbody>
</table>

**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 R[15] (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 R[15]. (closed impl. dep. #125-V9-Cs10)

**Exceptions**

None

**See Also**

JMPL on page 226
8.11 Compare and Swap

Description

Concurrent processes use these instructions for synchronization and memory updates. Uses of compare-and-swap include spin-lock operations, updates of shared counters, and updates of linked-list pointers. The last two can use wait-free (nonlocking) protocols.

The CASXA instruction compares the value in register R[rs2] with the doubleword in memory pointed to by the doubleword address in R[rs1]. If the values are equal, the value in R[rd] is swapped with the doubleword pointed to by the doubleword address in R[rs1]. If the values are not equal, the contents of the doubleword pointed to by R[rs1] replaces the value in R[rd], but the memory location remains unchanged.

The CASA instruction compares the low-order 32 bits of register R[rs2] with a word in memory pointed to by the word address in R[rs1]. If the values are equal, then the low-order 32 bits of register R[rd] are swapped with the contents of the memory word pointed to by the address in R[rs1] and the high-order 32 bits of register R[rd] are set to 0. If the values are not equal, the memory location remains unchanged, but the contents of the memory word pointed to by R[rs1] replace the low-order 32 bits of R[rd] and the high-order 32 bits of register R[rd] are set to 0.

A compare-and-swap instruction comprises three operations: a load, a compare, and a swap. The overall instruction is atomic; that is, no intervening interrupts or deferred traps are recognized by the virtual processor and no intervening update resulting from a compare-and-swap, swap, load, load-store unsigned byte, or store instruction to the doubleword containing the addressed location, or any portion of it, is performed by the memory system.

---

### CASA / CASXA

#### 8.11 Compare and Swap

<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>CASA[^A\text{Asi}]\</td>
<td>11 1100</td>
<td>Compare and Swap Word from Alternate Space</td>
<td>casa [r_{Srs1}] imm_asi, r_{Srs2}, r_{Srd}</td>
<td>A1</td>
</tr>
<tr>
<td>CASXA[^A\text{Asi}]\</td>
<td>11 1110</td>
<td>Compare and Swap Extended from Alternate Space</td>
<td>casxa [r_{Srs1}] imm_asi, r_{Srs2}, r_{Srd}</td>
<td>A1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>imm_asi</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 25 24</td>
<td>19 18 14 13 12 5 4 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Instruction**

Concurrent processes use these instructions for synchronization and memory updates. Uses of compare-and-swap include spin-lock operations, updates of shared counters, and updates of linked-list pointers. The last two can use wait-free (nonlocking) protocols.

The CASXA instruction compares the value in register R[rs2] with the doubleword in memory pointed to by the doubleword address in R[rs1]. If the values are equal, the value in R[rd] is swapped with the doubleword pointed to by the doubleword address in R[rs1]. If the values are not equal, the contents of the doubleword pointed to by R[rs1] replaces the value in R[rd], but the memory location remains unchanged.

The CASA instruction compares the low-order 32 bits of register R[rs2] with a word in memory pointed to by the word address in R[rs1]. If the values are equal, then the low-order 32 bits of register R[rd] are swapped with the contents of the memory word pointed to by the address in R[rs1] and the high-order 32 bits of register R[rd] are set to 0. If the values are not equal, the memory location remains unchanged, but the contents of the memory word pointed to by R[rs1] replace the low-order 32 bits of R[rd] and the high-order 32 bits of register R[rd] are set to 0.

A compare-and-swap instruction comprises three operations: a load, a compare, and a swap. The overall instruction is atomic; that is, no intervening interrupts or deferred traps are recognized by the virtual processor and no intervening update resulting from a compare-and-swap, swap, load, load-store unsigned byte, or store instruction to the doubleword containing the addressed location, or any portion of it, is performed by the memory system.
A compare-and-swap operation does not imply any memory barrier semantics. When compare-and-swap is used for synchronization, the same consideration should be given to memory barriers as if a load, store, or swap instruction were used.

A compare-and-swap operation behaves as if it performs a store, either of a new value from $R[rd]$ or of the previous value in memory. The addressed location must be writable, even if the values in memory and $R[rs2]$ are not equal.

If $i = 0$, the address space of the memory location is specified in the imm_asi field; if $i = 1$, the address space is specified in the ASI register.

An attempt to execute a CASXA or CASA instruction when $i = 1$ and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

A mem_address_not_aligned exception is generated if the address in $R[rs1]$ is not properly aligned.

In nonprivileged mode ($PSTATE.priv = 0$), 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 to 7F, 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 data_access_exception exception.

<table>
<thead>
<tr>
<th>ASI 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
data_access_exception
8.12 DONE

<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>DONEF</td>
<td>11 1110</td>
<td>Return from Trap (skip trapped instruction)</td>
<td>done</td>
<td>C1</td>
</tr>
</tbody>
</table>

**Description**

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

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.
DONE

In nonprivileged mode (PSTATE.priv = 0), an attempt to execute DONE causes a privileged_opcode exception.

**Implementation**

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).

**Exceptions**

illegal_instruction  
privileged_opcode

**See Also**

RETRY on page 294
8.13 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.

EDGE8Lcc, EDGE16Lcc, and EDGE32Lcc are little-endian versions of EDGE8cc, EDGE16cc, and EDGE32cc. They produce an edge mask that is bit-reversed from their big-endian counterparts but are otherwise identical. This makes the mask consistent with the mask produced by the Partial Store instruction (see Partial Store on page 298) on little-endian data.

A 2-bit (EDGE32cc), 4-bit (EDGE16cc), or 8-bit (EDGE8cc) pixel mask is stored in the least significant bits of \( R[rd] \). The mask is computed from left and right edge masks as follows:

1. The left edge mask is computed from the 3 least significant bits of \( R[rs1] \) and the right edge mask is computed from the 3 least significant bits of \( R[rs2] \), according to TABLE 8-6.
2. If a 32-bit address masking is disabled (PSTATE.am = 0, 64-bit addressing) and the upper 61 bits of \( R[rs1] \) are equal to the corresponding bits in \( R[rs2] \), \( R[rd] \) is set to the right edge mask anded with the left edge mask.

---

**Table: Edge Handling Instructions**

<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>EDGE8cc</td>
<td>0000 0000</td>
<td>Eight 8-bit edge boundary processing</td>
<td>edge8cc</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE8Lcc</td>
<td>0000 0010</td>
<td>Eight 8-bit edge boundary processing, little-endian</td>
<td>edge8lcc</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE16cc</td>
<td>0000 0100</td>
<td>Four 16-bit edge boundary processing</td>
<td>edge16cc</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE16Lcc</td>
<td>0000 0110</td>
<td>Four 16-bit edge boundary processing, little-endian</td>
<td>edge16lcc</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE32cc</td>
<td>0000 1000</td>
<td>Two 32-bit edge boundary processing</td>
<td>edge32cc</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE32Lcc</td>
<td>0000 1010</td>
<td>Two 32-bit edge boundary processing, little-endian</td>
<td>edge32lcc</td>
<td>C3</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>
<thead>
<tr>
<th>10</th>
<th>rd</th>
<th>110110</th>
<th>rs1</th>
<th>14</th>
<th>13</th>
<th>opf</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>25</td>
<td>24</td>
<td>19</td>
<td>18</td>
<td>14</td>
<td>13</td>
<td>5</td>
</tr>
</tbody>
</table>
EDGE<8|16|32>{L}cc

3. If 32-bit address masking is enabled (PSTATE.am = 1, 32-bit addressing) and bits 31:3 of R[rs1] match bits 31:3 of R[rs2], R[rd] is set to the right edge mask anded with the left edge mask.

4. Otherwise, R[rd] is set to the left edge mask.

The integer condition codes are set per the rules of the SUBcc instruction with the same operands (see Subtract on page 303).

TABLE 8-6 lists edge mask specifications.

<table>
<thead>
<tr>
<th>Edge Size</th>
<th>R[rsn] (2:0)</th>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>000</td>
<td>1111 1111</td>
<td>1000 0000</td>
</tr>
<tr>
<td>8</td>
<td>001</td>
<td>0111 1111</td>
<td>1100 0000</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>0011 1111</td>
<td>1110 0000</td>
</tr>
<tr>
<td>8</td>
<td>011</td>
<td>0001 1111</td>
<td>1111 0000</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 illegal_instruction

See Also EDGE(8,16,32){L}N on page 158
## 8.14 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>0 0000 0001</td>
<td>Eight 8-bit edge boundary processing, no CC</td>
<td>edge8n reg&lt;rs1&gt;, reg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE8LN</td>
<td>0 0000 0011</td>
<td>Eight 8-bit edge boundary processing, little-endian, no CC</td>
<td>edge8ln reg&lt;rs1&gt;, reg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE16N</td>
<td>0 0000 0101</td>
<td>Four 16-bit edge boundary processing, no CC</td>
<td>edge16n reg&lt;rs1&gt;, reg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE16LN</td>
<td>0 0000 0111</td>
<td>Four 16-bit edge boundary processing, little-endian, no CC</td>
<td>edge16ln reg&lt;rs1&gt;, reg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE32N</td>
<td>0 0000 1001</td>
<td>Two 32-bit edge boundary processing, no CC</td>
<td>edge32n reg&lt;rs1&gt;, reg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>C3</td>
</tr>
<tr>
<td>EDGE32LN</td>
<td>0 0000 1011</td>
<td>Two 32-bit edge boundary processing, little-endian, no CC</td>
<td>edge32ln reg&lt;rs1&gt;, reg&lt;rs2&gt;, reg&lt;rd&gt;</td>
<td>C3</td>
</tr>
</tbody>
</table>

### Description


See Edge Handling Instructions on page 156 for details.

### Exceptions

illegal_instruction

### See Also

EDGE-8,16,32>[L]cc on page 156
8.15 Floating-Point Absolute Value

**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 2005 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.ief = 0 or PSTATE.ipef = 0) or if no FPU is present, an attempt to execute an FABS instruction causes an fp_disabled exception.

**Exceptions**

- illegal_instruction
- fp_disabled
- fp_exception_other (FSR.ftt = unimplemented_FPop (FABSq))
8.16 Floating-Point Add

**Description**

The floating-point add instructions add the floating-point register(s) specified by the \( rs1 \) field and the floating-point register(s) specified by the \( rs2 \) field. The instructions then write the sum into the floating-point register(s) specified by the \( rd \) field.

Rounding is performed as specified by \( \text{FSR} \). \( rd \).

**Note**

UltraSPARC Architecture 2005 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FADDq instruction causes an \( \text{illegal_instruction} \) exception, allowing privileged software to emulate the instruction.

If the FPU is not enabled (\( \text{FPRS} . \text{fef} = 0 \) or \( \text{PSTATE} . \text{pef} = 0 \)) or if no FPU is present, an attempt to execute an FADD instruction causes an \( \text{fp_disabled} \) exception.

If the FPU is enabled, FADDq causes an \( \text{fp_exception_other} \) (with \( \text{FSR} . \text{ftt} = \text{unimplemented_FPop} \)), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.

**Note**

An \( \text{fp_exception_other} \) with \( \text{FSR} . \text{ftt} = \text{unfinished_FPop} \) can occur if the operation detects unusual, implementation-specific conditions.

For more details regarding floating-point exceptions, see Chapter 9, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005*.

**Exceptions**

- \( \text{illegal_instruction} \)
- \( \text{fp_disabled} \)
- \( \text{fp_exception_other} \) (\( \text{FSR} . \text{ftt} = \text{unimplemented_FPop} \) (FADDq))
- \( \text{fp_exception_other} \) (\( \text{FSR} . \text{ftt} = \text{unfinished_FPop} \))
- \( \text{fp_exception_ieee_754} \) (OF, UF, NX, NV)
8.17 Align Data [VIS 1]

**Description**
FALIGNDATA concatenates the two 64-bit floating-point registers specified by rs1 and rs2 to form a 128-bit (16-byte) intermediate value. The contents of the first source operand form the more-significant 8 bytes of the intermediate value, and the contents of the second source operand form the less significant 8 bytes of the intermediate value. Bytes in the intermediate value are numbered from most significant (byte 0) to least significant (byte 15). Eight bytes are extracted from the intermediate value and stored in the 64-bit floating-point destination register specified by rd. GSR.align specifies the number of the most significant byte to extract (and, therefore, the least significant byte extracted is numbered GSR.align+7).

GSR.align is normally set by a previous ALIGNADDRESS instruction.

If the FPU is not enabled (FPRS.ief = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FALIGNDATA instruction causes an *fp_disabled* exception.

**See Also**
Align Address on page 135

---

### Instruction

<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>FALIGNDATA</td>
<td>0 0100 1000</td>
<td>Perform data alignment for misaligned data</td>
<td>faligndata frs1, frs2, frd</td>
<td>A1</td>
</tr>
</tbody>
</table>

---

**FIGURE 8-6 FALIGNDATA**

A byte-aligned 64-bit load can be performed as shown below.

<table>
<thead>
<tr>
<th>alignaddr</th>
<th>Address, Offset, Address</th>
<th>!set GSR.align</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldd</td>
<td>Address, %d0</td>
<td></td>
</tr>
<tr>
<td>ldd</td>
<td>Address + 8, %d2</td>
<td></td>
</tr>
<tr>
<td>faligndata</td>
<td>%d0, %d2, %d4</td>
<td>!use GSR.align to select bytes</td>
</tr>
</tbody>
</table>

---

### Exceptions

*fp_disabled*
8.18 Branch on Floating-Point Condition Codes (FBfcc)

The FBfcc instructions are deprecated and should not be used in new software. The FBPfcc instructions should be used instead.

The FBfcc instructions are deprecated and should not be used in new software. The FBPfcc instructions should be used instead.

FBfcc (Deprecated)

## 8.18 Branch on Floating-Point Condition Codes (FBfcc)

The FBfcc instructions are deprecated and should not be used in new software. The FBPfcc instructions should be used instead.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>cond</th>
<th>Operation</th>
<th>fcc Test</th>
<th>Operation Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBA^D</td>
<td>1000</td>
<td>Branch Always</td>
<td>1</td>
<td>fba{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBN^D</td>
<td>0000</td>
<td>Branch Never</td>
<td>0</td>
<td>fbn{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBU^D</td>
<td>0111</td>
<td>Branch on Unordered</td>
<td>U</td>
<td>fbu{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBG^D</td>
<td>0110</td>
<td>Branch on Greater</td>
<td>G</td>
<td>fbg{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUG^D</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^D</td>
<td>0100</td>
<td>Branch on Less</td>
<td>L</td>
<td>fbl{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUL^D</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>FBLG^D</td>
<td>0010</td>
<td>Branch on Less or Greater</td>
<td>L or G</td>
<td>fblg{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FRNE^D</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^D</td>
<td>1001</td>
<td>Branch on Equal</td>
<td>E</td>
<td>fbe{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUOE^D</td>
<td>1010</td>
<td>Branch on Unordered or Equal</td>
<td>E or U</td>
<td>fbuo{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBGE^D</td>
<td>1011</td>
<td>Branch on Greater or Equal</td>
<td>E or G</td>
<td>fbge{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUGE^D</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>FBLE^D</td>
<td>1101</td>
<td>Branch on Less or Equal</td>
<td>E or L</td>
<td>fble{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBULE^D</td>
<td>1110</td>
<td>Branch on Unordered or Less or Equal</td>
<td>E or L or U</td>
<td>fbul{a} label</td>
<td>A1</td>
</tr>
<tr>
<td>FBO^D</td>
<td>1111</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: fbnz ‡ synonym: fbz

### Programming Note
To set the annul (a) bit for FBfcc instructions, append “,a” to the opcode mnemonic. For example, use “fbl,a label”. In the preceding table, braces around “,a” signify that “,a” is optional.

### Description
Unconditional and Fcc branches are described below:
FBfcc (Deprecated)

- **Unconditional branches** (FBA, FBN) — If its annul field is 0, an FBN (Branch Never) instruction acts like a NOP. If its annul field is 1, the following (delay) instruction is annulled (not executed) when the FBN is executed. In neither case does a transfer of control take place.

  FBA (Branch Always) causes a PC-relative, delayed control transfer to the address “PC + (4 × sign_ext (disp22))” regardless of the value of the floating-point condition code bits. If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul (a) bit is 0, the delay instruction is executed.

- **Fcc-conditional branches** — Conditional FBfcc instructions (except FBA and FBN) evaluate floating-point condition code zero (fcc0) according to the cond field of the instruction. Such evaluation produces either a **true** or **false** result. If **true**, the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address “PC + (4 × sign_ext (disp22))”. If **false**, the branch is not taken.

  If a conditional branch is taken, the delay instruction is always executed, regardless of the value of the annul (a) bit. If a conditional branch is not taken and the annul bit is 1 (a = 1), the delay instruction is annulled (not executed).

  **Note** | The annul bit has a different effect on conditional branches than it does on unconditional branches.

    Annulment, delay instructions, and delayed control transfers are described further in Chapter 6.

If the FPU is not enabled (FPRS.efd = 0 or PSTATE.ped = 0) or if no FPU is present, an attempt to execute an FBfcc instruction causes an **fp_disabled** exception.

**Exceptions** | **fp_disabled**
8.19 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>FBPA</td>
<td>1000</td>
<td>Branch Always</td>
<td>1</td>
<td>fba{a],[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPN</td>
<td>0000</td>
<td>Branch Never</td>
<td>0</td>
<td>fbn{a],[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPU</td>
<td>0111</td>
<td>Branch on Unordered</td>
<td>U</td>
<td>fbu{a],[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPG</td>
<td>0110</td>
<td>Branch on Greater</td>
<td>G</td>
<td>fbg{a],[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPGUG</td>
<td>0101</td>
<td>Branch on Unordered or Greater</td>
<td>G or U</td>
<td>fbugu{a],[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPL</td>
<td>0100</td>
<td>Branch on Less</td>
<td>L</td>
<td>fbl{a],[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPUL</td>
<td>0011</td>
<td>Branch on Unordered or Less</td>
<td>L or U</td>
<td>fbul{a],[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPLG</td>
<td>0010</td>
<td>Branch on Less or Greater</td>
<td>L or G</td>
<td>fblg{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPLNE</td>
<td>0001</td>
<td>Branch on Not Equal</td>
<td>L or G or U</td>
<td>fblne{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPE</td>
<td>1001</td>
<td>Branch on Equal</td>
<td>E</td>
<td>fbe{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPLUE</td>
<td>1010</td>
<td>Branch on Unordered or Equal</td>
<td>E or U</td>
<td>fble{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPGE</td>
<td>1011</td>
<td>Branch on Greater or Equal</td>
<td>E or G</td>
<td>fble{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPUGEL</td>
<td>1100</td>
<td>Branch on Unordered or Greater or Equal</td>
<td>E or G or U</td>
<td>fble{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPLE</td>
<td>1101</td>
<td>Branch on Less or Equal</td>
<td>E or L</td>
<td>fble{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPULE</td>
<td>1110</td>
<td>Branch on Unordered or Less or Equal</td>
<td>E or L or U</td>
<td>fble{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
<tr>
<td>FBPO</td>
<td>1111</td>
<td>Branch on Ordered</td>
<td>E or L or G</td>
<td>fbo{a}[p],{l},p{n}</td>
<td>%fcc, label</td>
</tr>
</tbody>
</table>

† synonym: fbnz  ‡ synonym: fbz

<table>
<thead>
<tr>
<th>00</th>
<th>a</th>
<th>cond</th>
<th>101</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>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc0</th>
<th>Condition Code</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>

164 UltraSPARC Architecture 2005 • Draft D0.8.7, 27 Mar 2006
FBPfcc

**Description**

Unconditional branches and Fcc-conditional branches are described below.

- **Unconditional branches (FBPA, FBPN)** — If its annul field is 0, an FBPN (Floating-Point Branch Never with Prediction) instruction acts like a NOP. If the Branch Never’s annul field is 0, the following (delay) instruction is executed; if the annul (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 x 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 x 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 7, 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.

**Exceptions**

- **fp_disabled**
8.20 SIMD Signed Compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>s1</th>
<th>s2</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMPLE16</td>
<td>00100000</td>
<td>Four 16-bit compare; set R[rd] if src1 ≤ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64 fcmple16 fregrs1tr, fregrs2, regrd</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPE16</td>
<td>00100010</td>
<td>Four 16-bit compare; set R[rd] if src1 ≥ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64 fcmpge16 fregrs1tr, fregrs2, regrd</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPE32</td>
<td>00100100</td>
<td>Two 32-bit compare; set R[rd] if src1 ≤ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64 fcmple32 fregrs1, fregrs2, regrd</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPEQ32</td>
<td>00100110</td>
<td>Two 32-bit compare; set R[rd] if src1 = src2</td>
<td>f64</td>
<td>f64</td>
<td>i64 fcmpeq32 fregrs1, fregrs2, regrd</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPT16</td>
<td>00101000</td>
<td>Four 16-bit compare; set R[rd] if src1 &gt; src2</td>
<td>f64</td>
<td>f64</td>
<td>i64 fcmpgt16 fregrs1tr, fregrs2, regrd</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPEQ16</td>
<td>00101010</td>
<td>Four 16-bit compare; set R[rd] if src1 = src2</td>
<td>f64</td>
<td>f64</td>
<td>i64 fcmpeq16 fregrs1, fregrs2, regrd</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPT32</td>
<td>00101100</td>
<td>Two 32-bit compare; set R[rd] if src1 &gt; src2</td>
<td>f64</td>
<td>f64</td>
<td>i64 fcmpgt32 fregrs1, fregrs2, regrd</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPEQ32</td>
<td>00101110</td>
<td>Two 32-bit compare; set R[rd] if src1 = src2</td>
<td>f64</td>
<td>f64</td>
<td>i64 fcmpeq32 fregrs1, fregrs2, regrd</td>
<td>C3</td>
</tr>
</tbody>
</table>

**Description**

Either four 16-bit signed values or two 32-bit signed values in F_D[rs1] and F_D[rs2] are compared. The 4-bit or 2-bit condition-code results are stored in the least significant bits of the integer register R[rd]. The least significant 16-bit or 32-bit compare result corresponds to bit zero of R[rd].

**Note**

Bits 63:4 of the destination register R[rd] are set to zero for 16-bit compares. Bits 63:2 of the destination register R[rd] are set to zero for 32-bit compares.

For FCMPGT[16,32], each bit in the result is set to 1 if the corresponding signed value in F_D[rs1] is greater than the signed value in F_D[rs2]. Less-than comparisons are made by swapping the operands.

For FCMPLE[16,32], each bit in the result is set to 1 if the corresponding signed value in F_D[rs1] is less than or equal to the signed value in F_D[rs2]. Greater-than-or-equal comparisons are made by swapping the operands.

For FCMPEQ[16,32], each bit in the result is set to 1 if the corresponding signed value in F_D[rs1] is equal to the signed value in F_D[rs2].
FCMP*<16|32> (SIMD)

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 8-7 and FIGURE 8-8 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.cef = 0) or if no FPU is present, an attempt to execute a SIMD signed compare instruction causes an \( fp\_disabled \) exception.
FCMP*<16|32> (SIMD)

Exception       fp_disabled

See Also        STPARTIALF on page 325
8.21 Floating-Point Compare

These instructions compare the floating-point register(s) specified by the rs1 field with the floating-point register(s) specified by the rs2 field, and set the selected floating-point condition code (fcc) as shown below.

### Description

These instructions compare the floating-point register(s) specified by the rs1 field with the floating-point register(s) specified by the rs2 field, and set the selected floating-point condition code (fcc) as shown below.

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

The “?” in the preceding table means that the comparison is unordered. The unordered condition occurs when one or both of the operands to the compare is a signalling or quiet NaN.
The “compare and cause exception if unordered” (FCMPEs, FCMPEd, and FCMPEq) instructions cause an invalid (NV) exception if either operand is a NaN.

FCMP causes an invalid (NV) exception if either operand is a signalling NaN.

An attempt to execute an FCMP instruction when instruction bits 29:27 are nonzero causes an illegal_instruction exception.

UltraSPARC Architecture 2005 processors do not implement in hardware the instructions that refer to quad-precision floating-point registers. An attempt to execute FCMPq or FCMPEq generates fp_exception_other (with FSR.ftt = unimplemented_FPop), 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.

For more details regarding floating-point exceptions, see Chapter 9, IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005.
8.22 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.

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.

If the FPU is enabled, FDIVq causes an fp_exception_other (with FSR.ftt = unimplemented_FPop), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.

**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 9, IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005.

**Exceptions**

- illegal_instruction
- fp_disabled
- fp_exception_other (FSR.ftt = unimplemented_FPop (FDIVq only))
- fp_exception_other (FSR.ftt = unfinished_FPop (FDIVs, FDIV))
- fp_exception_ieee_754 (OF, UF, DZ, NV, NX)
### FEXPAND

#### 8.23 FEXPAND [VIS 1]

<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>FEXPAND</td>
<td>0 0100 1101</td>
<td>Four 16-bit expands</td>
<td>—</td>
<td>f32</td>
<td>f64</td>
<td>fexp and</td>
<td>C3</td>
</tr>
</tbody>
</table>

**Description**

FEXPAND takes four 8-bit unsigned integers from $F_S[rs2]$, converts each integer to a 16-bit fixed-point value, and stores the four resulting 16-bit values in a 64-bit floating-point register $F_D[rd]$. FIGURE 7-10 illustrates the operation.

![FIGURE 8-9 FEXPAND Operation](image)

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.

In an UltraSPARC Architecture 2005 implementation, this instruction is not implemented in hardware, causes an *illegal_instruction* exception, and is emulated in software.

**Exceptions**

*illegal_instruction*

**See Also**

FPMERGE on page 206
FPACK on page 197
8.24 Convert 32-bit Integer to Floating Point

### Description

FiTOS, FiTOd, and FiTOq convert the 32-bit signed integer operand in floating-point register $F_{S}[rs2]$ into a floating-point number in the destination format. All write their result into the floating-point register(s) specified by $rd$.

The value of $FS.R_d$ determines how rounding is performed by FiTOS.

**Note** UltraSPARC Architecture 2005 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 ($FPR.S.fef = 0$ 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.

If the FPU is enabled, FiTOq causes an *fp_exception_other* (with $FSR.flt = unimplemented_FPop$), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.

For more details regarding floating-point exceptions, see Chapter 9, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005*.

### Exceptions

- *illegal_instruction*
- *fp_disabled*
- *fp_exception_other* (with $FSR.flt = unimplemented_FPop$ (FiTOq))
- *fp_exception_ieee_754* (NX (FiTOs only))
8.25 Flush Instruction Memory

**Description**

FLUSH ensures that the aligned doubleword specified by the effective address is consistent across any local caches and, in a multiprocessor system, will eventually (impl. dep. #122-V9) become consistent everywhere.

The SPARC V9 instruction set architecture does not guarantee consistency between instruction memory and data memory. When software writes (stores) to a memory location containing an instruction (self-modifying code¹), a potential memory consistency problem arises, which is addressed by the FLUSH instruction. Use of FLUSH ensures that instruction and data memory are synchronized after instruction memory has been modified.

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_FLSH 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_FLSH appear to execute after any loads, stores, and atomic load-stores to that address issued by P_FLSH 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 was a store operation (see Memory Barrier on page 258).

If i = 0, the effective address operand for the FLUSH instruction is “R[rs1] + R[rs2]”; if i = 1, it is “R[rs1] + sign_ext (simm13)”¹. The three least-significant three bits of the effective address are ignored; that is, the effective address always refers to an aligned doubleword.

---

† 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 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-Cs20:** The implementation of the FLUSH instruction is implementation dependent. If the implementation automatically maintains consistency between instruction and data memory,
  1. the FLUSH address is ignored and
  2. the FLUSH instruction cannot cause any data access exceptions, because its effective address operand is not translated or used by the MMU.

On the other hand, if the implementation does not maintain consistency between instruction and data memory, the FLUSH address is used to access the MMU and the FLUSH instruction can cause data access exceptions.

**Programming Note**

For portability across all SPARC V9 implementations, software must always supply the target effective address in FLUSH instructions.

- If the implementation contains instruction prefetch buffers:
  - the instruction prefetch buffer(s) are invalidated
  - instruction prefetching is suspended, but may resume starting with the instruction immediately following the FLUSH

**Programming Notes**

1. Typically, FLUSH is used in self-modifying code. The use of self-modifying code is discouraged.
2. If a program includes self-modifying code, to be portable it must issue a FLUSH instruction for each modified doubleword of instructions (or make a call to privileged software that has an equivalent effect) after storing into the instruction stream.
**FLUSH**

3. The order in which memory is modified can be controlled by means of FLUSH and MEMBAR instructions interspersed appropriately between stores and atomic load-stores. FLUSH is needed only between a store and a subsequent instruction fetch from the modified location. When multiple processes may concurrently modify live (that is, potentially executing) code, the programmer must ensure that the order of update maintains the program in a semantically correct form at all times.

4. The memory model guarantees in a uniprocessor that data loads observe the results of the most recent store, even if there is no intervening FLUSH.

5. FLUSH may be a time-consuming operation. (see the Implementation Note below)

6. In a multiprocessor system, the effects of a FLUSH operation will be globally visible before any subsequent store becomes globally visible.

7. FLUSH is designed to act on a doubleword. On some implementations, FLUSH may trap to system software. For these reasons, system software should provide a service routine, callable by nonprivileged software, for flushing arbitrarily-sized regions of memory. On some implementations, this routine would issue a series of FLUSH instructions; on others, it might issue a single trap to system software that would then flush the entire region.

8. FLUSH operates using the current (implicit) context. Therefore, a FLUSH executed in privileged mode will use the nucleus context and will not necessarily affect instruction cache lines containing data from a user (nonprivileged) context.

<table>
<thead>
<tr>
<th>Implementation Note</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>V9 Compatibility Note</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Exceptions**

illegal_instruction
8.26 Flush Register Windows

**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 CANSAVE = N_REG_WINDOWS - 2. Otherwise, there is more than one active window, so FLUSHW causes a spill exception. The trap vector for the spill exception is based on the contents of OTHERWIN and WSTATE. The spill trap handler is invoked with the CWP set to the window to be spilled (that is, (CWP + CANSAVE + 2) mod N_REG_WINDOWS). See Register Window Management Instructions on page 116.

**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.

An attempt to execute a FLUSHW instruction when instruction bits 29:25, 18:14, or 12:0 are nonzero causes an *illegal_instruction* exception.

**Exceptions**
*illegal_instruction*
*spill_n_normal*
*spill_n_other*
8.27 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 2005 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMOVq instruction causes an illegal_instruction exception, allowing privileged software to emulate the instruction.

An attempt to execute an FMOV instruction when instruction bits 18:14 are nonzero causes an illegal_instruction exception.

If the FPU is not enabled (FPRS.ief = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMOV instruction causes an fp_disabled exception.

If the FPU is enabled, an attempt to execute an FMOVq instruction causes an fp_exception_other (with FSR.ftt = unimplemented_FPop), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.

For more details regarding floating-point exceptions, see Chapter 9, IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005.

Exceptions

- illegal_instruction
- fp_disabled
- fp_exception_other (FSR.ftt = unimplemented_FPop (FMOVq only))
FMOV

See Also  F Register Logical Operate (2 operand) on page 212
## 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>FMOVSccc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on 32-bit integer condition codes</td>
<td>fmovscc %icc, reg, reg</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>fmovdccc %icc, reg, reg</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, reg, reg</td>
<td>C3</td>
</tr>
<tr>
<td>FMOVScxc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on 64-bit integer condition codes</td>
<td>fmovsxc %xcc, reg, reg</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>fmovdxc %xcc, reg, reg</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>fmovqxc %xcc, reg, reg</td>
<td>C3</td>
</tr>
<tr>
<td>FMOVScfc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on floating-point condition codes</td>
<td>fmovsfcc %fcc, reg, reg</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 %fcc, reg, reg</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 %fcc, reg, reg</td>
<td>C3</td>
</tr>
</tbody>
</table>
### 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 1u)</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>
FMOVcc

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><strong>fcc</strong> Test</th>
<th><strong>fcc</strong> 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><strong>opf_cc</strong></th>
<th>Instruction</th>
<th>Condition Code to be Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>100_2</td>
<td>FMOV(S,D,Q)icc</td>
<td><strong>icc</strong></td>
</tr>
<tr>
<td>110_2</td>
<td>FMOV(S,D,Q)xcc</td>
<td><strong>xcc</strong></td>
</tr>
<tr>
<td>000_2</td>
<td>FMOV(S,D,Q)fcc</td>
<td><strong>fcc0</strong></td>
</tr>
<tr>
<td>001_2</td>
<td></td>
<td><strong>fcc1</strong></td>
</tr>
<tr>
<td>010_2</td>
<td></td>
<td><strong>fcc2</strong></td>
</tr>
<tr>
<td>011_2</td>
<td></td>
<td><strong>fcc3</strong></td>
</tr>
<tr>
<td>101_2</td>
<td>(illegal_instruction exception)</td>
<td></td>
</tr>
<tr>
<td>111_2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**FMOVcc**

*Description*  The FMOVcc instructions copy the floating-point register(s) specified by rs2 to the floating-point register(s) specified by rd if the condition indicated by the cond field is satisfied by the selected floating-point condition code field in FSR. The condition code used is specified by the opf_cc field of the instruction. If the condition is **FALSE**, then the destination register(s) are not changed.

These instructions read, but do not modify, any condition codes.

These instructions clear (set to 0) both FSR.cexc and FSR.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 2005 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.

If the FPU is enabled, an attempt to execute an FMOVQicc, FMOVQxcc, or FMOVQfcc instruction causes an *fp_exception_other* (with FSR.flt = unimplemented_FPopp), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.
FMOVcc

Programming Note  Branches cause the performance of most implementations to degrade significantly. Frequently, the MOVcc and FMOVcc instructions can be used to avoid branches. For example, the following C language segment:

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

```c
! assume A is in %f0; B is in %f2; %xx points to ! constant area
ldd [%xx+C_1.03],%f4 ! X = 1.03
fcmpd %fcc3,%f0,%f2 ! A > B
fble,a %fcc3,label
! following 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

```c
ldd [%xx+C_1.03],%f4 ! X = 1.03
fsubd %f4,%f4,%f6 ! X' = 0.0
fcmpd %fcc3,%f0,%f2 ! A > B
fmovdle %fcc3,%f6,%f4 ! X = 0.0
```

This code also takes four instructions but requires no branches and may boost performance significantly. Use MOVcc and FMOVcc instead of branches wherever these instructions would improve performance.

Exceptions

- illegal_instruction
- fp_disabled
- fp_exception_other (FSR.ftt = unimplemented_FPop (opf_cc = 1012 or 1112))
- fp_exception_other (FSR.ftt = unimplemented_FPop (FMOVQ instructions only))
8.29 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>FMOVRsZ</td>
<td>001</td>
<td>0 0101</td>
<td>Reserved</td>
<td>—</td>
<td>—</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>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>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>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>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>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>
**FMOVR**

**Assembly Language Syntax**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmovr{s,d,q}</td>
<td>reg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt; (synonym: fmovr{s,d,q}e)</td>
</tr>
<tr>
<td>fmovr{s,d,q}le</td>
<td>reg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
</tr>
<tr>
<td>fmovr{s,d,q} nz</td>
<td>reg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
</tr>
<tr>
<td>fmovr{s,d,q}g</td>
<td>reg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
</tr>
<tr>
<td>fmovr{s,d,q}ge</td>
<td>reg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
</tr>
</tbody>
</table>

**Description**

If the contents of integer register R[rs1] satisfy the condition specified in the rcond field, these instructions copy the contents of the floating-point register(s) specified by the rs2 field to the floating-point register(s) specified by the rd field. If the contents of R[rs1] do not satisfy the condition, the floating-point register(s) specified by the rd field are not modified.

These instructions treat the integer register contents as a signed integer value; they do not modify any condition codes.

These instructions clear (set to 0) both FSR.cexc and FSR.ftt. They do not round, do not modify FSR.aexc, and do not treat floating-point NaN values differently from other floating-point values.

**Note** UltraSPARC Architecture 2005 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 = 0002 or 1002 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 an FMOVR instruction causes an fp_disabled exception.

If the FPU is enabled, an attempt to execute an FMOVRq instruction causes an fp_exception_other (with FSR.ftt = unimplemented_FPop), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.
FMOVR

**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**
- `fp_disabled`
- `fp_exception_other` (FSR.ftt = unimplemented_FPop (`rcond` = 000₂ or 100₂))
- `fp_exception_other` (FSR.ftt = unimplemented_FPop (FMOVRq))
8.30 Partitioned Multiply Instructions

**Description**

The following sections describe the versions of partitioned multiplies.

In an UltraSPARC Architecture 2005 implementation, these instructions are not implemented in hardware, cause an `illegal_instruction` exception, and are emulated in software.

**Exceptions**

`illegal_instruction`

---

**Programming Note**

When software emulates an 8-bit unsigned by 16-bit signed multiply, the unsigned value must be zero-extended and the 16-bit value sign-extended before the multiplication.
8.30.1 FMUL8x16 Instruction

FMUL8x16 multiplies each unsigned 8-bit value (for example, a pixel component) in the 32-bit floating-point register Fₛ[rₜ₁] by the corresponding (signed) 16-bit fixed-point integer in the 64-bit floating-point register Fₚ[rₜ₂]. 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 Fₚ[rₜ₃]. FIGURE 8-10 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 8-10 FMUL8x16 Operation](image-url)
FMUL (partitioned)

8.30.2 FMUL8x16AU Instruction

FMUL8x16AU is the same as FMUL8x16, except that one 16-bit fixed-point value is used as the multiplier for all four multiplies. This multiplier is the most significant (“upper”) 16 bits of the 32-bit register $F_S[rs2]$ (typically an α pixel component value). FIGURE 8-11 illustrates the operation.

8.30.3 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 8-12 illustrates the operation.
8.30.4 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 8-13 illustrates the operation.

![FIGURE 8-13 FMUL8SUx16 Operation](image)

8.30.5 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 8-14 illustrates the operation; CODE EXAMPLE 8-1 exemplifies the operation.
FMUL (partitioned)

**FIGURE 8-14** FMUL8ULx16 Operation

**CODE EXAMPLE 8-1** 16-bit × 16-bit 16-bit Multiply

```
fmul8sux16 %f0, %f1, %f2
fmul8ulx16 %f0, %f1, %f3
fpadd16 %f2, %f3, %f4
```

### 8.30.6 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 8-15** illustrates the operation.

**FIGURE 8-15** FMULD8SUx16 Operation
8.30.7  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 8-16 illustrates the operation; CODE EXAMPLE 8-2 exemplifies the operation.

**FIGURE 8-16  FMULD8ULx16 Operation**

**CODE EXAMPLE 8-2  16-bit x 16-bit 32-bit Multiply**

```
fmul8sux16  %f0, %f1, %f2
fmul8ulx16  %f0, %f1, %f3
fpadd32     %f2, %f3, %f4
```
8.31 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 2005 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.xef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute any FMUL instruction causes an fp_disabled exception.

If the FPU is enabled, an attempt to execute an FMULq or FdMULq instruction causes an fp_exception_other (with FSR.ftt = unimplemented_FPop), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.

For more details regarding floating-point exceptions, see Chapter 9, IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005.
**FMUL<s|d|q>**

**Exceptions**

*illegal_instruction*

*fp_disabled*

*fp_exception_other* (FSR.ftt = unimplemented_FPop (FMULq, FdMULq only))

*fp_exception_other* (FSR.ftt = unfinished_FPop)

*fp_exception_ieee_754* (any: NV; FMUL(s,d,q) only: OF, UF, NX)
8.32 Floating-Point Negate

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.

### Exceptions
- illegal_instruction
- fp_disabled
- fp_exception_other (FSR.ftt = unimplemented_FPop (FNEGq only))
8.33 FPACK

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.

In an UltraSPARC Architecture 2005 implementation, these instructions are not implemented in hardware, cause an illegal_instruction exception, and are emulated in software.

**Exception**

illegal_instruction

**See Also**

FEXPAND on page 172
FPMERGE on page 206
FPACK

8.33.1 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 8-17 illustrates the FPACK16 operation.

This operation is carried out as follows:

1. Left-shift the value from $F_D[rs2]$ by the number of bits specified in $GSR.scale$ while maintaining clipping information.

2. Truncate and clip to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 7 and 6 for each 16-bit word). Truncation converts the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative (that is, its most significant bit is set), 0 is returned as the clipped value. If the value is greater than 255, then 255 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.

3. Store the result in the corresponding byte in the 32-bit destination register, $F_S[rd]$.

For each 16-bit partition, the sequence of operations performed is shown in the following example pseudo-code:

```
tmp ← source_operand(15:0) << GSR.scale;
// Pick off the bits from bit position 15+GSR.scale to
```

Note: FPACK16 ignores the most significant bit of $GSR.scale$(GSR.scale[4]).
FPACK

// 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};

8.33.2 FPACK32

FPACK32 takes two 32-bit fixed values from the second source operand (64-bit floating-point register \( \text{F}_D[rs2] \)) 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 \( \text{F}_D[rs1] \), left-shifted by 8 bits. The 64-bit result is stored in \( \text{F}_D[rd] \). Thus, successive FPACK32 instructions can assemble two pixels by using three or four pairs of 32-bit fixed values. FIGURE 8-18 illustrates the FPACK32 operation.

This operation, illustrated in FIGURE 8-18, is carried out as follows:

1. Left-shift each 32-bit value in \( \text{F}_D[rs2] \) by the number of bits specified in GSR.scale, while maintaining clipping information.
FPACK

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 Fp[rs1] by 8 bits.

4. Merge the two clipped 8-bit unsigned values into the corresponding least significant byte positions in the left-shifted Fp[rs2] value.

5. Store the result in the 64-bit destination register Fp[rd].

For each 32-bit partition, the sequence of operations performed is shown in the following pseudo-code:

tmp ← source_operand2{31:0} << GSR.scale;
// Pick off the bits from bit position 31+GSR.scale to bit position 23 from the shifted result
trunc_signed_value ← tmp{(31+GSR.scale):23};
if (trunc_signed_value < 0)
    unsigned_8bit_value ← 0;
else if (trunc_signed_value > 255)
    unsigned_8bit_value ← 255;
else
    unsigned_8bit_value ← trunc_signed_value{30:23};
Final_32bit_Result ← (source_operand1{31:0} << 8) | (unsigned_8bit_value{7:0});
FPACK

8.33.3 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 8-19 illustrates the FPACKFIX operation.

This operation is carried out as follows:

1. Left-shift each 32-bit value from \( F_D[rs2] \) by the number of bits specified in \( GSR.scale \), while maintaining clipping information.

2. For each 32-bit value, truncate and clip to a 16-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 16 and 15 for each 32-bit word). Truncation is performed to convert the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is less than \(-32768\), then \(-32768\) is returned as the clipped value. If the value is greater than 32767, then 32767 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.

3. Store the result in the 32-bit destination register \( F_S[rd] \).

For each 32-bit partition, the sequence of operations performed is shown in the following pseudo-code:

```c
    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
```
FPACK

\[
\text{trunc\_signed\_value} \leftarrow \text{tmp}\{(31+\text{GSR.scale}):16}\};
\]
if (\text{trunc\_signed\_value} < -32768)
   \text{signed\_16bit\_result} \leftarrow -32768;
else if (\text{trunc\_signed\_value} > 32767)
   \text{signed\_16bit\_result} \leftarrow 32767;
else
   \text{signed\_16bit\_result} \leftarrow \text{trunc\_signed\_value}\{31:16\};
8.34 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.

**FIGURE 8-20** FPADD16 Operation
FIGURE 8-21  FPADD32 Operation

FIGURE 8-22  FPADD16S Operation

FIGURE 8-23  FPADD32S Operation
FPADD

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*
8.35 FPMERGE

**Description**

FPMERGE interleaves eight 8-bit unsigned values in \( F_S[^{\text{rs1}}] \) and \( F_S[^{\text{rs2}}] \) to produce a 64-bit value in the destination register \( F_D[^{\text{rd}}] \). This instruction converts from packed to planar representation when it is applied twice in succession; for example, \( \text{R1G1B1A1,R3G3B3A3} \rightarrow \text{R1R3G1G3A1A3} \rightarrow \text{R1R2R3R4G1G2G3G4} \).

FPMERGE also converts from planar to packed when it is applied twice in succession; for example, \( \text{R1R2R3R4,B1B2B3B4} \rightarrow \text{R1B1R2B2R3B3B4B4} \rightarrow \text{R1G1B1A1R2G2B2A2} \).

FIGURE 8-24 illustrates the operation.
In an UltraSPARC Architecture 2005 implementation, these instructions are not implemented in hardware, cause an *illegal_instruction* exception, and are emulated in software.

**Exceptions**

*illegal_instruction*

**See Also**

FPACK on page 197
FEXPAND on page 172
8.36 Fixed-point Partitioned Subtract

<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>FPSUB16</td>
<td>0 0  1 0  1  0  0</td>
<td>Four 16-bit subtracts</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpsub16 freg_rs1, freg_rs2, freg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>FPSUB16S</td>
<td>0 0  1 0  1  0  0  0</td>
<td>Two 16-bit subtracts</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>fpsub16s freg_rs1, freg_rs2, freg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>FPSUB32</td>
<td>0 0  1 0  1  1  0  0</td>
<td>Two 32-bit subtracts</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpsub32 freg_rs1, freg_rs2, freg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>FPSUB32S</td>
<td>0 0  1 0  1  1  0  0  0</td>
<td>One 32-bit subtract</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>fpsub32s freg_rs1, freg_rs2, freg_rd</td>
<td>A1</td>
</tr>
</tbody>
</table>

Description

FPSUB16 (FPSUB32) performs four 16-bit (two 32-bit) partitioned subtractions between the corresponding fixed-point values contained in the source operands (FD[rs1], FD[rs2]). The values in FD[rs2] are subtracted from those in FD[rs1], and the result is placed in the destination register, FD[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 8-25 FPSUB16 Operation
**FIGURE 8-26** FPSUB32 Operation

**FIGURE 8-27** FPSUB16S Operation

**FIGURE 8-28** FPSUB32S Operation
**FPSUB**

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

*Exceptions*  
*fp_disabled*
F Register 1-operand Logical Ops

8.37 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>FZERO</td>
<td>0110 0000</td>
<td>Zero fill</td>
<td>fzero freg rd</td>
<td>A1</td>
</tr>
<tr>
<td>FZEROs</td>
<td>0110 0001</td>
<td>Zero fill, 32-bit</td>
<td>fzeros freg rd</td>
<td>A1</td>
</tr>
<tr>
<td>FONE</td>
<td>0111 1110</td>
<td>One fill</td>
<td>fone freg rd</td>
<td>A1</td>
</tr>
<tr>
<td>FONEs</td>
<td>0111 1111</td>
<td>One fill, 32-bit</td>
<td>fones freg rd</td>
<td>A1</td>
</tr>
</tbody>
</table>

Description
FZERO and FONE fill the 64-bit destination register, F_D[rd], with all ‘0’ bits or all ‘1’ bits (respectively).

FZEROs and FONEs fill the 32-bit destination register, F_D[rd], with all ‘0’ bits or all ‘1’ bits (respectively).

An attempt to execute an FZERO or FONE instruction when instruction bits 18:14 or bits 4:0 are nonzero causes an illegal_instruction exception.

If the FPU is not enabled (FPRES.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FZERO[s] or FONE[s] instruction causes an fp_disabled exception.

Exceptions
illegal_instruction
fp_disabled

See Also
F Register 2-operand Logical Operations on page 212
F Register 3-operand Logical Operations on page 214
F Register 2-operand Logical Ops

8.38 F Register Logical Operate (2 operand) [VIS1]

<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>FSRC1</td>
<td>0 0111 0100</td>
<td>Copy F_D[r1] to F_D[r2]</td>
<td>fsrc1</td>
<td>freg[r1], freg[r2]</td>
</tr>
<tr>
<td>FSRC1s</td>
<td>0 0111 0101</td>
<td>Copy F_S[r1] to F_S[r2], 32-bit</td>
<td>fsrc1s</td>
<td>freg[r1], freg[r2]</td>
</tr>
<tr>
<td>FSRC2</td>
<td>0 0111 1000</td>
<td>Copy F_D[r2] to F_D[r1]</td>
<td>fsrc2</td>
<td>freg[r2], freg[r1]</td>
</tr>
<tr>
<td>FSRC2s</td>
<td>0 0111 1001</td>
<td>Copy F_S[r2] to F_S[r1], 32-bit</td>
<td>fsrc2s</td>
<td>freg[r2], freg[r1]</td>
</tr>
<tr>
<td>FNOT1</td>
<td>0 0110 1010</td>
<td>Negate (1’s complement) F_D[r1]</td>
<td>fnot1</td>
<td>freg[r1]</td>
</tr>
<tr>
<td>FNOT1s</td>
<td>0 0110 1011</td>
<td>Negate (1’s complement) F_S[r1], 32-bit</td>
<td>fnot1s</td>
<td>freg[r1]</td>
</tr>
<tr>
<td>FNOT2</td>
<td>0 0110 0110</td>
<td>Negate (1’s complement) F_D[r2]</td>
<td>fnot2</td>
<td>freg[r2]</td>
</tr>
<tr>
<td>FNOT2s</td>
<td>0 0110 0111</td>
<td>Negate (1’s complement) F_S[r2], 32-bit</td>
<td>fnot2s</td>
<td>freg[r2]</td>
</tr>
</tbody>
</table>

**Description**

The standard 64-bit versions of these instructions perform one of four 64-bit logical operations on the 64-bit floating-point register F_D[r1] (or F_D[r2]) and store the result in the 64-bit floating-point destination register F_D[r2].

The 32-bit (single-precision) versions of these instructions perform 32-bit logical operations on F_S[r1] (or F_S[r2]) and store the result in F_S[r2].

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.ief = 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 178). Programmers are encouraged to use FMOVs (FMOVd) instead of FSRC1s (FSRC1) whenever practical.

**Exceptions**

illegal_instruction
fp_disabled
F Register 2-operand Logical Ops

See Also

Floating-Point Move on page 178
F Register 1-operand Logical Operations on page 211
F Register 3-operand Logical Operations on page 214
## F Register 3-operands Logical Ops

### 8.39 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>FOR</td>
<td>0 0111 1100</td>
<td>Logical or</td>
<td>for fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FORs</td>
<td>0 0111 1101</td>
<td>Logical or, 32-bit</td>
<td>for fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FNORs</td>
<td>0 0110 0011</td>
<td>Logical nor</td>
<td>fnor fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FANDs</td>
<td>0 0111 0000</td>
<td>Logical and</td>
<td>fand fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FNANDs</td>
<td>0 0110 1110</td>
<td>Logical nand</td>
<td>fnands fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FXORs</td>
<td>0 0110 1111</td>
<td>Logical xor</td>
<td>fxors fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FXNORS</td>
<td>0 0111 0011</td>
<td>Logical xnor</td>
<td>fxnors fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FORNOT1</td>
<td>0 0111 1010</td>
<td>(not F[rₘₚ]) or F[rₘₕ]</td>
<td>fornot1 fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FORNOT1s</td>
<td>0 0111 1011</td>
<td>(not F[rₘₚ]) or F[rₘₕ], 32-bit</td>
<td>fornot1s fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FANDNOT1</td>
<td>0 0110 1010</td>
<td>(not F[rₘₚ]) and F[rₘₕ]</td>
<td>fandnot1 fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FANDNOT1s</td>
<td>0 0110 1011</td>
<td>(not F[rₘₚ]) and F[rₘₕ], 32-bit</td>
<td>fandnot1s fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FANDNOT2</td>
<td>0 0110 1000</td>
<td>(not F[rₘₚ]) and (not F[rₘₕ])</td>
<td>fandnot2 fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</td>
<td>A1</td>
</tr>
<tr>
<td>FANDNOT2s</td>
<td>0 0110 0100</td>
<td>(not F[rₘₚ]) and (not F[rₘₕ]), 32-bit</td>
<td>fandnot2s fregₘₚ, fregₘₕ, fregₘₖ, fregₘₗ</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 Fₕ[rₘₚ] and Fₕ[rₘₕ]. The result is stored in the 64-bit floating-point destination register Fₕ[rd].

The 32-bit (single-precision) versions of these instructions perform 32-bit logical operations between Fₕ[rₘₚ] and Fₕ[rₘₕ], storing the result in Fₕ[rd].

If the FPU is not enabled (FPRS.pef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute any 3-operands F Register Logical Operate instruction causes an fp_disabled exception.

### Exceptions

fp_disabled

### See Also

F Register 1-operands Logical Operations on page 211
F Register 2-operands Logical Operations on page 212
8.40 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 2005 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FSQRTq instruction causes an illegal_instruction exception, allowing privileged software to emulate the instruction.

An attempt to execute an FSQRT instruction when instruction bits 18:14 are nonzero causes an illegal_instruction exception.

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

If the FPU is enabled, an fp_exception_other (with FSR.ftt = unimplemented_FPpop) exception occurs, since the FSQRT instructions are not implemented in hardware in UltraSPARC Architecture 2005 implementations.

For more details regarding floating-point exceptions, see Chapter 9, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005*.

### Exceptions

- illegal_instruction
- fp_disabled
- fp_exception_other (FSR.ftt = unimplemented_FPpop (FSQRT is not implemented in hardware))
8.41 Convert Floating-Point to Integer

<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>FsTOx</td>
<td>0000 1000</td>
<td>Convert Single to 64-bit Integer</td>
<td>—</td>
<td>f32</td>
<td>f64</td>
<td>fstox frsra2, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FdTOx</td>
<td>0000 1000</td>
<td>Convert Double to 64-bit Integer</td>
<td>—</td>
<td>f64</td>
<td>f64</td>
<td>fdtrox frsra2, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FqTOx</td>
<td>0000 1000</td>
<td>Convert Quad to 64-bit Integer</td>
<td>—</td>
<td>f128</td>
<td>f64</td>
<td>fqtx frsra2, frsrd</td>
<td>C3</td>
</tr>
<tr>
<td>FsTOi</td>
<td>1101 0000</td>
<td>Convert Single to 32-bit Integer</td>
<td>—</td>
<td>f32</td>
<td>f32</td>
<td>fstoi frsra2, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FdTOi</td>
<td>1101 0000</td>
<td>Convert Double to 32-bit Integer</td>
<td>—</td>
<td>f64</td>
<td>f32</td>
<td>fdtoi frsra2, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FqTOi</td>
<td>1101 0000</td>
<td>Convert Quad to 32-bit Integer</td>
<td>—</td>
<td>f128</td>
<td>f32</td>
<td>fqtoi frsra2, frsrd</td>
<td>C3</td>
</tr>
</tbody>
</table>

### 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 2005 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.ief = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an F<s|d|q>TO<i|x> instruction causes an *fp_disabled* exception.

If the FPU is enabled, FqTOi and FqTOx cause *fp_exception_other* (with FSR.ftt = unimplemented_FPPop), since those instructions are not implemented in hardware in UltraSPARC Architecture 2005 implementations.

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 363.
F<s|d|q>TOi

For more details regarding floating-point exceptions, see Chapter 9, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005*.

Exceptions

- illegal_instruction
- fp_disabled
- fp_exception_other (FSR.itt = unimplemented_FPop (FqTOx, FqTOi only))
- fp_exception_ieee_754 (NV, NX)
8.42 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 2005 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 (\( FPERS.\text{fref} = 0 \) or \( PSTATE.\text{pdef} = 0 \)) or if no FPU is present, an attempt to execute an \( F<s|d|q>TO<s|d|q> \) instruction causes an \textit{fp_disabled} exception.

If the FPU is enabled, \( FsTOq \), \( FdTOq \), \( FqTOs \), and \( FqTOd \) cause \textit{fp_exception_other} with \( FSR.\text{ftt} = \text{unimplemented}_{\text{FPop}} \), since those instructions are not implemented in hardware in UltraSPARC Architecture 2005 implementations.

\( FqTOd \), \( FqTOs \), and \( FdTOs \) (the “narrowing” conversion instructions) can cause \textit{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 \textit{fp_exception_ieee_754} NV exception if the source operand is a signalling NaN.
Untrapped Result in Different Format from Operands on page 360 defines the rules for converting NaNs from one floating-point format to another.

**Note** | For FdTOs and FsTOd, an `fp_exception_other` with FSR.ftt = unfinished_FPop can occur if implementation-dependent conditions are detected during the conversion operation.

For more details regarding floating-point exceptions, see Chapter 9, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005*.

**Exceptions**
- `illegal_instruction`
- `fp_disabled`
- `fp_exception_other` (FSR.ftt = unimplemented_FPop (FsTOq, FqTOs, FdTOq, and FqTOd only))

- `fp_exception_other` (FSR.ftt = unfinished_FPop)
- `fp_exception_ieee_754` (NV)
- `fp_exception_ieee_754` (OF, UF, NX (FqTOd, FqTOs, and FdTOs))
8.43 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 2005 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.

If the FPU is enabled, FSUBq causes an `fp_exception_other` (with FSR.ftt = unimplemented_FPop), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.

*Note* An `fp_exception_other` with FSR.ftt = unfinished_FPop can occur if the operation detects unusual, implementation-specific conditions (for FSUBs or FSUBd).

For more details regarding floating-point exceptions, see Chapter 9, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005*.

**Exceptions**

- `illegal_instruction`
- `fp_disabled`
- `fp_exception_other` (FSR.ftt = unimplemented_FPop (FSUBq))
- `fp_exception_other` (FSR.ftt = unfinished_FPop)
- `fp_exception_ieee_754` (OF, UF, NX, NV)
8.44 Convert 64-bit Integer to Floating Point

FxTO(<s|d|q>)

FxTOs, FxTOd, and FxTOq convert the 64-bit signed integer operand in the floating-point register \( F_\text{D}[rs2] \) into a floating-point number in the destination format. All write their result into the floating-point register(s) specified by \( rd \).

The value of FSR.\( r_d \) determines how rounding is performed by FxTOs and FxTOd.

An attempt to execute an FxTO(s,d,q) instruction when instruction bits 18:14 are nonzero causes an illegal_instruction exception. An attempt to execute a FxTOq instruction causes an illegal_instruction exception, allowing privileged software to emulate the instruction.

If the FPU is not enabled (FPRS.\( \text{fef} = 0 \) or PSTATE.\( \text{pef} = 0 \)) or if no FPU is present, an attempt to execute an FxTO(s,d,q) instruction causes an fp_disabled exception.

If the FPU is enabled, FxTOq causes an fp_exception_other (with FSR.\( \text{ftt} = \text{unimplemented\_FPop} \)), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.

For more details regarding floating-point exceptions, see Chapter 9, IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005.

<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>11 0100</td>
<td>0 1000 0100</td>
<td>Convert 64-bit Integer to Single</td>
<td>—</td>
<td>i64</td>
<td>f32</td>
<td>fxtos freg_{rs2}, freg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>FxTOd</td>
<td>11 0100</td>
<td>0 1000 1000</td>
<td>Convert 64-bit Integer to Double</td>
<td>—</td>
<td>i64</td>
<td>f64</td>
<td>fxtod freg_{rs2}, freg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>FxTOq</td>
<td>11 0100</td>
<td>0 1000 1100</td>
<td>Convert 64-bit Integer to Quad</td>
<td>—</td>
<td>i64</td>
<td>f128</td>
<td>fxtoq freg_{rs2}, freg_{rd}</td>
<td>C3</td>
</tr>
</tbody>
</table>

---

**Description**

FxTOs, FxTOd, and FxTOq convert the 64-bit signed integer operand in the floating-point register \( F_\text{D}[rs2] \) into a floating-point number in the destination format. All write their result into the floating-point register(s) specified by \( rd \).

The value of FSR.\( r_d \) determines how rounding is performed by FxTOs and FxTOd.

**Note**

UltraSPARC Architecture 2005 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.\( \text{fef} = 0 \) or PSTATE.\( \text{pef} = 0 \)) or if no FPU is present, an attempt to execute an FxTO(s,d,q) instruction causes an fp_disabled exception.

If the FPU is enabled, FxTOq causes an fp_exception_other (with FSR.\( \text{ftt} = \text{unimplemented\_FPop} \)), since that instruction is not implemented in hardware in UltraSPARC Architecture 2005 implementations.

For more details regarding floating-point exceptions, see Chapter 9, IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005.

**Exceptions**

- illegal_instruction
- fp-disabled
- fp_exception_other (FSR.\( \text{ftt} = \text{unimplemented\_FPop} \) (FxTOq only))
- fp_exception_ieee_754 (NX (FxTOs and FxTOd only))
8.45 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 Note**

Except for its name, this instruction is identical to the SPARC V8 UNIMP instruction.

An attempt to execute an ILLTRAP instruction when reserved instruction bits 29:25 are nonzero (also) causes an *illegal_instruction* exception. However, software should not rely on this behavior, because a future version of the architecture may use nonzero values of bits 29:25 to encode other functions.

**Exceptions**

*illegal_instruction*
8.46 Implementation-Dependent Instructions

**Description**

IMPL. DEP. #106-V9: The IMPDEP2A opcode space is completely implementation dependent. Implementation-dependent aspects of IMPDEP2A instructions include their operation, the interpretation of bits 29–25, 18–7, and 4–0 in their encodings, and which (if any) exceptions they may cause.

IMPDEP2B opcodes are reserved; see **IMDEP2B Opcodes** on page 224.

See “Implementation-Dependent and Reserved Opcodes” in the "Extending the UltraSPARC Architecture" section of the separate document UltraSPARC Architecture Application Notes, for information about extending the instruction set by means of implementation-dependent instructions.

**Compatibility Note**

IMPDEP2A and IMPDEP2B are subsets of the SPARC V9 IMPDEP2 opcode space. The IMPDEP1 opcode space from SPARC V9 is occupied by various VIS instructions in the UltraSPARC Architecture, so it should not be used for implementation-dependent instructions.

**Exceptions**

implementation-dependent (IMPDEP2A, IMPDEP2B)

8.46.1 IMPDEP1 Opcodes

All operands of instructions using IMPDEP1 opcodes are in floating-point registers, unless otherwise specified. Pixel values are stored in single-precision floating point registers and fixed values are stored in double-precision floating point registers, unless otherwise specified.

**Note**

All instructions, regardless of whether they use floating-point registers or integer registers, leave FSR.cexc and FSR.aexc unchanged.
IMPDEP

8.46.1.1 Opcode Formats

Most of the VIS instruction set maps to the opcode space reserved for the Implementation-Dependent Instruction 1 (op3 = IMPDEP1 = 36_16) instructions.

8.46.2 IMDEP2B Opcodes

No instructions are currently encoded in the IMDEP2B opcode space; it is a reserved opcode space.
8.47 Mark Register Window Sets as “Invalid”

INVALW

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALW$^2$</td>
<td>Mark all register window sets as “invalid”</td>
<td>invalw</td>
<td>C1</td>
</tr>
</tbody>
</table>

Description

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

\[
\begin{align*}
\text{CANSAVE} & \leftarrow (N_{\text{REG WINDOWS}} - 2) \\
\text{CANRESTORE} & \leftarrow 0 \\
\text{OTHERWIN} & \leftarrow 0
\end{align*}
\]

In an UltraSPARC Architecture 2005 implementation, these instructions are not implemented in hardware, cause an illegal_instruction exception, and are emulated in software.

Exceptions

illegal_instruction (not implemented in hardware in UltraSPARC Architecture 2005)

See Also

ALLCLEAN on page 136
NORMALW on page 272
OTHERW on page 274
RESTORED on page 292
SAVED on page 300
8.48 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 field = 0, or “R[rs1] + sign_ext(simm13)” if i = 1.

The JMPL instruction copies the PC, which contains the address of the JMPL instruction, into register R[rd].

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

If either of the low-order two bits of the jump address is nonzero, a mem_address_not_aligned exception occurs.

**Programming Notes**

A JMPL instruction with rd = 15 functions as a register-indirect call using the standard link register.

A 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.

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]. (closed impl. dep. #125-V9-Cs10)

**Exceptions**

illegal_instruction
mem_address_not_aligned

**See Also**

CALL on page 150
Bicc on page 142
BPCC on page 148
8.49 Load Integer

### Description
The load integer instructions copy a byte, a halfword, a word, or an extended word from memory. All copy the fetched value into R[rd]. A fetched byte, halfword, or word is right-justified in the destination register R[rd]; it is either sign-extended or zero-filled on the left, depending on whether the opcode specifies a signed or unsigned operation, respectively.

Load integer instructions access memory using the implicit ASI (see page 104). The effective address is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + sign_ext(simm13)” if i = 1.

A successful load (notably, load extended) instruction operates atomically.

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

If the effective address is not halfword-aligned, an attempt to execute an LDUH or LDSH causes a mem_address_not_aligned exception. If the effective address is not word-aligned, an attempt to execute an LDUW or LDSW instruction causes a mem_address_not_aligned exception. If the effective address is not doubleword-aligned, an attempt to execute an LDX instruction causes a mem_address_not_aligned exception.

### V8 Compatibility Note
The SPARC V8 LD instruction was renamed LDUW in the SPARC V9 architecture. The LDSW instruction was new in the SPARC V9 architecture.

A load integer twin word (LDTW) instruction exists, but is deprecated; see Load Integer Twin Word on page 253 for details.
LD

Exceptions

illegal_instruction
mem_address_not_aligned (all except LDSB, LDUB)
VA_watchpoint
data_access_exception
8.50 Load Integer from 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>LDSBA_ASI</td>
<td>01 001</td>
<td>Load Signed Byte from Alternate Space</td>
<td>ldsba [regaddr] immasi, rsrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDSHA_ASI</td>
<td>01 1010</td>
<td>Load Signed Halfword from Alternate Space</td>
<td>ldsha [regaddr] immasi, rsrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDSWA_ASI</td>
<td>01 1000</td>
<td>Load Signed Word from Alternate Space</td>
<td>ldswa [regaddr] immasi, rsrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUBA_ASI</td>
<td>01 0001</td>
<td>Load Unsigned Byte from Alternate Space</td>
<td>lduba [regaddr] immasi, rsrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUHA_ASI</td>
<td>01 0010</td>
<td>Load Unsigned Halfword from Alternate Space</td>
<td>lduha [regaddr] immasi, rsrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUWA_ASI</td>
<td>01 0000</td>
<td>Load Unsigned Word from Alternate Space</td>
<td>lduwa† [regaddr] immasi, rsrd</td>
<td>A1</td>
</tr>
<tr>
<td>LDXA_ASI</td>
<td>01 1011</td>
<td>Load Extended Word from Alternate Space</td>
<td>ldxa [regaddr] immasi, rsrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonym: lda

Description

The load integer from alternate space instructions copy a byte, a halfword, a word, or an extended word from memory. All copy the fetched value into R[rd]. A fetched byte, halfword, or word is right-justified in the destination register R[rd]; it is either sign-extended or zero-filled on the left, depending on whether the opcode specifies a signed or unsigned operation, respectively.

The load integer from alternate space instructions contain the address space identifier (ASI) to be used for the load in the immasi field if i = 0, or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + sign_ext(simm13)” if i = 1.

A successful load (notably, load extended) instruction operates atomically.

A load integer twin word from alternate space (LDTWA) instruction exists, but is deprecated; see Load Integer Twin Word from Alternate Space on page 255 for details.

An attempt to execute a load integer from alternate space instruction when i = 0 and instruction bits 12:5 are nonzero causes an illegal_instruction exception.
LDA

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 3016 to 7F16, these instructions cause a `privileged_action` exception.

LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA can be used with any of the following ASIs, subject to the privilege mode rules described for the `privileged_action` exception above. Use of any other ASI with these instructions causes a `data_access_exception` 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 `data_access_exception` exception.

<table>
<thead>
<tr>
<th>ASIs invalid for LDXA (cause <code>data_access_exception</code> exception)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2416 (aliased to 2716, ASI_LDTX_N)</td>
</tr>
<tr>
<td>2216 (ASI_LDTX_AIUP)</td>
</tr>
<tr>
<td>2316 (ASI_LDTX_AIUS)</td>
</tr>
<tr>
<td>2616 (ASI_LDTX_REAL)</td>
</tr>
<tr>
<td>2716 (ASI_LDTX_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>
</tbody>
</table>

230 UltraSPARC Architecture 2005 • Draft D0.8.7, 27 Mar 2006
**LDA**

**Exceptions**

- mem_address_not_aligned (all except LDSBA and LDUBA)
- privileged_action
- VA_watchpoint
- data_access_exception

**See Also**

- LD on page 227
- STA on page 308
# LDBLOCKF

## 8.51 Block Load [VIS]

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>LDBLOCKF</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>B2</td>
</tr>
<tr>
<td>LDBLOCKF</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>B2</td>
</tr>
<tr>
<td>LDBLOCKF</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>B2</td>
</tr>
<tr>
<td>LDBLOCKF</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>B2</td>
</tr>
<tr>
<td>LDBLOCKF</td>
<td>F016</td>
<td>64-byte block load from primary address space</td>
<td>ldda [regaddr] #ASI_BLK_P, freg</td>
<td>B2</td>
</tr>
<tr>
<td>LDBLOCKF</td>
<td>F116</td>
<td>64-byte block load from secondary address space</td>
<td>ldda [regaddr] #ASI_BLK_S, freg</td>
<td>B2</td>
</tr>
<tr>
<td>LDBLOCKF</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>B2</td>
</tr>
<tr>
<td>LDBLOCKF</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>B2</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.
LDBLOCKF

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.

**Programmer's Note** LDBLOCKF is intended to be a processor-specific instruction (see the warning at the top of page 232). 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 LDBLOCKFs to non-cacheable locations are (a) strictly ordered, (b) not strictly ordered and cause an illegal_instruction exception, or (c) not strictly ordered and silently execute without causing an exception (option (c) is strongly discouraged)
- 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
LDBLOCKF

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.pef = 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 (ASI's 16, 17, 1E, 16, and 1F), LDBLOCKF causes a privileged_action exception.

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

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

Exceptions
illegal_instruction
fp_disabled
mem_address_not_aligned
privileged_action
VA_watchpoint (impl. dep. #410-S10)
data_access_exception
LDBLOCKF

See Also

STBLOCKF on page 312
LDF / LDDF / LDQF / LDXFSR

8.52 Load Floating-Point

<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], fregrd</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], fregrd</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], fregrd</td>
<td>C3</td>
</tr>
<tr>
<td>LDXFSR</td>
<td>10 0001</td>
<td>1</td>
<td>Load Floating-Point State Register</td>
<td>ldx [address], %fsr</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td>10 0001</td>
<td>2–31</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

‡ Encoded floating-point register value, as described on page 51.

<table>
<thead>
<tr>
<th>11 rd op3 rs1</th>
<th>i=0 — rs2</th>
<th>11 rd op3 rs1</th>
<th>i=1 simm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 25 24 19 18 14 13 12 5 4 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

The load single floating-point instruction (LDF) copies a word from memory into 32-bit floating-point destination register $F_S[rd]$. The load doubleword floating-point instruction (LDDF) copies a word-aligned doubleword from memory into a 64-bit floating-point destination register, $F_D[rd]$. The unit of atomicity for LDDF is 4 bytes (one word).

The load quad floating-point instruction (LDQF) copies a word-aligned quadword from memory into a 128-bit floating-point destination register, $F_Q[rd]$. The unit of atomicity for LDQF is 4 bytes (one word).

The load floating-point state register instruction (LDXFSR) waits for all FPop instructions that have not finished execution to complete and then loads a doubleword from memory into the FSR. LDXFSR does not alter the ver, fit, qne, or reserved fields of FSR (see page 58).

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.

These load floating-point instructions access memory using the implicit ASI (see page 104).

If $i = 0$, the effective address for these instructions is $R[rs1] + R[rs2]$ and if $i = 0$, the effective address is $R[rs1] + \text{sign_ext}(\text{simm13})$. 
LDF / LDDF / LDQF / LDXFSR

Exceptions. An attempt to execute an LDF, LDDF, LDQF, or LDXFSR instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an \textit{illegal\_instruction} exception. An attempt to execute an instruction encoded as \( \text{op} = 2, \text{op3} = 2116, \) and \( \text{rd} > 1 \) causes an \textit{illegal\_instruction} exception.

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

If the effective address is not word-aligned, an attempt to execute an LDF instruction causes a \textit{mem\_address\_not\_aligned} exception. If the effective address is not doubleword-aligned, an attempt to execute an LDXFSR instruction causes a \textit{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 \textit{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 \textit{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)).

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

Since UltraSPARC Architecture 2005 processors do not implement in hardware instructions (including LDQF) that refer to quadprecision floating-point registers, the \textit{LDQF\_mem\_address\_not\_aligned} and \textit{fp\_exception\_other} (with \( \text{FSR} \text{.ftt} = \text{invalid\_fp\_register} \)) 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.

Destination Register(s) when Exception Occurs. If \textit{aload} floating-point instruction generates an exception that causes a \textit{precise} trap, the destination floating-point register(s) remain unchanged.
LDF / LDDF / LDQF / LDXFSR

**IMPL. DEP. #44-V8-Cs10(a):** 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.

**Implementation Note:** LDXFSR shares an opcode with the LDFS instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the \( \text{op} = 11_2, \text{op3} = 100001_2 \) opcode with an invalid rd value \( (\text{rd} > 1) \) causes an *illegal_instruction* exception.

**Exceptions:**
- *illegal_instruction*
- *fp_disabled*
- *LDDF_mem_address_not_aligned*
- *mem_address_not_aligned*
- *fp_exception_other* (FSR.flt = invalid_fp_register (LDQF only))
- *VA_watchpoint*
- *data_access_exception*

**See Also**
- *Load Floating-Point from Alternate Space on page 239*
- *Load Floating-Point State Register on page 243*
- *Store Floating-Point on page 316*
### 8.53 Load Floating-Point from Alternate Space

<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>LDFA&lt;sup&gt;ASI&lt;/sup&gt;</td>
<td>11 0000</td>
<td>0–31</td>
<td>Load Floating-Point Register from Alternate Space</td>
<td>lda [regaddr] imm_asi, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>LDDFA&lt;sup&gt;ASI&lt;/sup&gt;</td>
<td>11 0011</td>
<td>†</td>
<td>Load Double Floating-Point Register from Alternate Space</td>
<td>ldda [regaddr] imm_asi, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>LDQFA&lt;sup&gt;ASI&lt;/sup&gt;</td>
<td>11 0010</td>
<td>†</td>
<td>Load Quad Floating-Point Register from Alternate Space</td>
<td>ldqa [regaddr] imm_asi, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>C3</td>
</tr>
</tbody>
</table>

† Encoded floating-point register value, as described in Floating-Point Register Number Encoding on page 51.

**Description**

The load single floating-point from alternate space instruction (LDFA) copies a word from memory into 32-bit floating-point destination register \( F_S[r_d] \).

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, \( F_D[r_d] \). 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, \( F_Q[r_d] \). 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 “\( R[r_{s1}] + R[r_{s2}] \)”. If \( i = 1 \), the ASI to be used is contained in the ASI register and the effective address for the instruction is “\( R[r_{s1}] + \text{sign_ext}(\text{simm13}) \)”.

**Exceptions.** If the FPU is not enabled (FPRS.<sub>eef</sub> = 0 or PSTATE.<sub>pe</sub> = 0) or if no FPU is present, an attempt to execute an LDFA, LDDFA, or LDQFA instruction causes an \textit{fp_disabled} exception.

LDFA causes a \textit{mem_address_not_aligned} exception if the effective memory address is not word-aligned.

**V9 Compatibility** | LDFA, LDDFA, and LDQFA cause a \textit{privileged_action} exception if PSTATE.<sub>priv</sub> = 0 and bit 7 of the ASI is 0.
LDFA / LDDFA / LDQFA

LDFA requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, LDFA causes an `LDDF_mem_address_not_aligned` exception. In this case, trap handler software must emulate the LDFA instruction and return (impl. dep. #109-V9-Cs10(b)).

LDQFA requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, LDQFA causes an `LDQF_mem_address_not_aligned` exception. In this case, trap handler software must emulate the LDQFA instruction and return (impl. dep. #111-V9-Cs10(b)).

An attempt to execute an LDQFA instruction when \( \text{rd}[1] \neq 0 \) causes an `fp_exception_other` (with \( \text{FSR.ftt} = \text{invalid_fp_register} \)) exception.

**Implementation Note**

Since UltraSPARC Architecture 2005 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 \( \text{FSR.ftt} = \text{invalid_fp_register} \)) 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 (\( \text{PSTATE.priv} = 0 \)), if bit 7 of the ASI is 0, this instruction causes a `privileged_action` exception. In privileged mode (\( \text{PSTATE.priv} = 1 \)), if the ASI is in the range \( 30_{16} \) to \( 7F_{16} \), this instruction causes a `privileged_action` exception.

LDFA and LDQFA can be used with any of the following ASIs, subject to the privilege mode rules described for the `privileged_action` exception above. Use of any other ASI with these instructions causes a `data_access_exception` 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>
LDFA / LDDFA / LDQFA

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 data_access_exception 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>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>

Behavior with Partial Store ASIs. ASIs C016–C516 and C816–CD16 are only defined for use in Partial Store operations (see page 325). 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 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 an exception. It is implementation dependent whether the generated exception is a data_access_exception, mem_address_not_aligned, or LDDF_mem_address_not_aligned exception.

2. If the memory address is correctly aligned, the virtual processor generates a data_access_exception.

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
mem_address_not_aligned
LDFA / LDDFA / LDQFA

\textit{fp\_exception\_other} (FSR.flt = invalid\_fp\_register (LDQFA only))
\textit{privileged\_action}
\textit{VA\_watchpoint}

See Also

Load Floating-Point on page 236
Block Load on page 232
Store Short Floating-Point on page 328
Store Floating-Point into Alternate Space on page 319
LDFSR - Deprecated

8.54 Load Floating-Point State Register

The LDFSR instruction is deprecated and should not be used in new software. The LDXFSR 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>LDFSR</td>
<td>10 0001</td>
<td>0</td>
<td>Load Floating-Point State Register Lower</td>
<td>ld [address], %fsr</td>
<td>C2</td>
</tr>
</tbody>
</table>

Description

The load floating-point state register lower instruction (LDFSR) waits for all FPop instructions that have not finished execution to complete and then loads a word from memory into the less significant 32 bits of the FSR. The upper 32 bits of FSR are unaffected by LDFSR. LDFSR does not alter the ver, ftt, qne, or reserved fields of FSR (see page 58).

Programming Note

For future compatibility, software should only issue an LDFSR instruction with a zero value (or a value previously read from the same field) in any reserved field of FSR.

LDFSR accesses memory using the implicit ASI (see page 108).

An attempt to execute an LDFSR instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

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

LDFSR causes a mem_address_not_aligned exception if the effective memory address is not word-aligned.

V8 Compatibility Note

The SPARC V9 architecture supports two different instructions to load the FSR: the 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.w
LDFSR - Deprecated

**Implementation Note**

LDFSR shares an opcode with the LDXFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction \( \text{rd} \) field. An attempt to execute the \( \text{op} = 11_2, \text{op3} = 100001_2 \) opcode with an invalid \( \text{rd} \) value (\( \text{rd} > 1 \)) causes an *illegal_instruction* exception.

**Exceptions**

- *illegal_instruction*
- *fp_disabled*
- *mem_address_not_aligned*
- *VA_watchpoint*
# LDSHORTF

## 8.55 Short Floating-Point Load

<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</td>
<td>D016</td>
<td>8-bit load from primary address space</td>
<td>ldda [regaddr] #ASI_FL8_P, freg</td>
<td>C3</td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>D116</td>
<td>8-bit load from secondary address space</td>
<td>ldda [regaddr] #ASI_FL8_S, freg</td>
<td>C3</td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>D816</td>
<td>8-bit load from primary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL8_PL, freg</td>
<td>C3</td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>D916</td>
<td>8-bit load from secondary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL8_SL, freg</td>
<td>C3</td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>D216</td>
<td>16-bit load from primary address space</td>
<td>ldda [regaddr] #ASI_FL16_P, freg</td>
<td>C3</td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>D316</td>
<td>16-bit load from secondary address space</td>
<td>ldda [regaddr] #ASI_FL16_S, freg</td>
<td>C3</td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>D816</td>
<td>16-bit load from primary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL16_PL, freg</td>
<td>C3</td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>D916</td>
<td>16-bit load from secondary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL16_SL, freg</td>
<td>C3</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.ef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an LDSHORTF instruction causes an **fp_disabled** exception.

An 8-bit load places the loaded value in the least significant byte of F_D[rd] and zeroes in the most-significant three bytes of F_D[rd]. An 8-bit LDSHORTF can be performed from an arbitrary byte address.

A 16-bit load places the loaded value in the least significant halfword of F_D[rd] and zeroes in the more-significant halfword of F_D[rd]. A 16-bit LDSHORTF from an address that is not halfword-aligned (an odd address) causes a **mem_address_not_aligned** exception.
LDshortcut

Little-endian ASIs transfer data in little-endian format from memory; otherwise, memory is assumed to be in big-endian byte order.

**Programming Note**

LDshortcut is typically used with the FALIGNDATA instruction (see *Align Address* on page 135) to assemble or store 64 bits from noncontiguous components.

**Implementation Note**

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

In an UltraSPARC Architecture 2005 implementation, these instructions are not implemented in hardware, cause a *data_access_exception* exception, and are emulated in software.

**Exceptions**

*VA_watchpoint*

*data_access_exception*
8.56 Load-Store Unsigned Byte

**Description**

The load-store unsigned byte instruction copies a byte from memory into R[rd], then rewrites the addressed byte in memory to all 1’s. The fetched byte is right-justified in the destination register R[rd] and zero-filled on the left.

The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing LDSTUB, LDSTUBA, CASA, CASXA, SWAP, or SWAPA instructions addressing all or parts of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

LDSTUB accesses memory using the implicit ASI (see page 104). The effective address for this instruction is “R[rs1] + R[rs2]” if \( i = 0 \), or “R[rs1] + sign_ext(simm13)” if \( i = 1 \).

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

An attempt to execute an LDSTUB instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an `illegal_instruction` exception.

**Exceptions**

- `illegal_instruction`
- `VA_watchpoint`
- `data_access_exception`
### 8.57 Load-Store Unsigned Byte to 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>LDSTUBAPASI</td>
<td>01 1101</td>
<td>Load-Store Unsigned Byte into Alternate Space</td>
<td>ldstuba [reg_addr] imm_asi, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>rd</td>
<td>op3</td>
<td>rs1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>rs1=0</td>
<td>imm_asi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>rs1=1</td>
<td>simm13</td>
</tr>
</tbody>
</table>

#### 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 data_access_exception exception.

#### ASIs valid for LDSTUBA

- ASI_NUCLEUS
- ASI_NUCLEUS_LITTLE
- ASI_AS_IF_USER_PRIMARY
- ASI_AS_IF_USER_PRIMARY_LITTLE
- ASI_AS_IF_USER_SECONDARY
- ASI_AS_IF_USER_SECONDARY_LITTLE
- ASI_REAL
- ASI_REAL_LITTLE
- ASI_PRIMARY
- ASI_PRIMARY_LITTLE
- ASI_SECONDARY
- ASI_SECONDARY_LITTLE

---

248 UltraSPARC Architecture 2005 • Draft D0.8.7, 27 Mar 2006
LDSTUBA

Exceptions

privileged_action
VA_watchpoint
data_access_exception
8.58 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_LDTX_AIUP, regrd</td>
<td>N1</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_LDTX_AIUS, regrd</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td>2616</td>
<td>Load Integer Twin Extended Word, real address</td>
<td>ldtxa [regaddr] #ASI_LDTX_REAL, regrd</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td>2716</td>
<td>Load Integer Twin Extended Word, nucleus context</td>
<td>ldtxa [regaddr] #ASI_LDTX_N, regrd</td>
<td>N1</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_LDTX_AIUP_L, regrd</td>
<td>N1</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_LDTX_AIUS_L, regrd</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td>2E16</td>
<td>Load Integer Twin Extended Word, real address, little endian</td>
<td>ldtxa [regaddr] #ASI_LDTX_REAL_L, regrd</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td>2F16</td>
<td>Load Integer Twin Extended Word, nucleus context, little-endian</td>
<td>ldtxa [regaddr] #ASI_LDTX_NL, regrd</td>
<td>N1</td>
</tr>
<tr>
<td>LDTXAN</td>
<td>E216</td>
<td>Load Integer Twin Extended Word, Primary address space</td>
<td>ldtxa [regaddr] #ASI_LDTX_P, regrd</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td>E316</td>
<td>Load Integer Twin Extended Word, Secondary address space</td>
<td>ldtxa [regaddr] #ASI_LDTX_S, regrd</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td>EA16</td>
<td>Load Integer Twin Extended Word, Primary address space, little endian</td>
<td>ldtxa [regaddr] #ASI_LDTX_PL, regrd</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td>EB16</td>
<td>Load Integer Twin Extended Word, Secondary address space, little-endian</td>
<td>ldtxa [regaddr] #ASI_LDTX_SL, regrd</td>
<td>N1</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.
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 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.

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 \( \text{rd}[0]=1 \) causes an \text{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 \text{mem_address_not_aligned} exception.

**IMPL. DEP. #413-S10:** It is implementation dependent whether \text{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 \text{data_access_exception} 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 \text{mem_address_not_aligned} trap is taken if the access is not aligned on a 128-byte boundary.
LDTXA

**Implementation**

LDTXA shares an opcode with the “i = 0” variant of the (deprecated) LDTWA instruction. See *Load Integer Twin Word from Alternate Space* on page 255.

**Exceptions**

illegal_instruction
mem_address_not_aligned
privileged_action
VA_watchpoint (impl. dep. #413-S10)
data_access_exception
8.59 Load Integer Twin Word

The LDTW instruction is deprecated and should not be used in new software. It is provided only for compatibility with previous versions of the architecture. The LDX instruction should be used instead.

**Description**

The load integer twin word instruction (LDTW) copies two words (with doubleword alignment) from memory into a pair of R registers. The word at the effective memory address is copied into the least significant 32 bits of the even-numbered R register. The word at the effective memory address + 4 is copied into the least significant 32 bits of the following odd-numbered R register. The most significant 32 bits of both the even-numbered and odd-numbered R registers are zero-filled.

With respect to little endian memory, an LDTW instruction behaves as if it comprises two 32-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

**IMPL. DEP. #107-V9a**: It is implementation dependent whether LDTW is implemented in hardware. If not, an attempt to execute an LDTW instruction will cause an `unimplemented_LDTW` exception.

**Programming Note**: LDTW is provided for compatibility with existing SPARC V8 software. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties.
LDTW (Deprecated)

SPARC V9 Compatibility Note

LDTW was (inaccurately) named LDD in the SPARC V8 and SPARC V9 specifications. It does not load a doubleword; it loads two words (into two registers), and has been renamed accordingly.

The least significant bit of the rd field in an LDTW instruction is unused and should always be set to 0 by software. An attempt to execute an LDTW instruction that refers to a misaligned (odd-numbered) destination register causes an illegal_instruction exception.

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

If the effective address is not doubleword-aligned, an attempt to execute an LDTW instruction causes a mem_address_not_aligned exception.

A successful LDTW instruction operates atomically.

Exceptions

unimplemented_LDTW
illegal_instruction
mem_address_not_aligned
VA_watchpoint
data_access_exception

See Also

LDW/LDX on page 227
STTW on page 330
**LDTWA (Deprecated)**

8.60 Load Integer Twin Word from Alternate Space

The LDTWA instruction is deprecated and should not be used in new software. The LDXA instruction should be used instead.

### Description

The load integer twin word from alternate space instruction (LDTWA) copies two words (with doubleword alignment) from memory into a pair of R registers. The word at the effective memory address is copied into the least significant 32 bits of the even-numbered R register. The word at the effective memory address + 4 is copied into the least significant 32 bits of the following odd-numbered R register. The most significant 32 bits of both the even-numbered and odd-numbered R registers are zero-filled.

### Note

Execution of an LDTWA instruction with rd = 0 modifies only R[1].

If i = 0, the LDTWA instruction contains the address space identifier (ASI) to be used for the load in its imm_asi field and the effective address for the instruction is “R[rs1] + R[rs2]”. If i = 1, the ASI to be used is contained in the ASI register and the effective address for the instruction is “R[rs1] + sign_ext (simm13)”. With respect to little endian memory, an LDTWA instruction behaves as if it is composed of two 32-bit loads, each of which is byte-swapped independently before being written into its respective destination register.
LDTWA (Deprecated)

**IMPL. DEP. #107-V9b:** It is implementation dependent whether LDTWA is implemented in hardware. If not, an attempt to execute an LDTWA instruction will cause an *unimplemented LDTW* exception so that it can be emulated.

| Programming Note | LDTWA is provided for compatibility with existing SPARC V8 software. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties. If LDTWA is emulated in software, an LDXA instruction should be used for the memory access in the emulation code in order to preserve atomicity. |
| SPARC V9 Compatibility Note | LDTWA was (inaccurately) named LDDA in the SPARC V8 and SPARC V9 specifications. |

The least significant bit of the \(rd\) field in an LDTWA instruction is unused and should always be set to 0 by software. An attempt to execute an LDTWA instruction that references 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 \(30_{16}\) to \(7F_{16}\), these instructions cause a *privileged_action* exception.

LDTWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *data_access_exception* exception (impl. dep. #300-U4-Cs10).

<table>
<thead>
<tr>
<th>ASIs valid for LDTWA</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>22_{16}‡ (ASI_LDTX_AIUP)</td>
</tr>
<tr>
<td>23_{16}‡ (ASI_LDTX_AIUUS)</td>
</tr>
<tr>
<td>24_{16}‡ (aliased to 27_{16}, ASI_LDTX_N)</td>
</tr>
<tr>
<td>26_{16}‡ (ASI_LDTX_REAL)</td>
</tr>
<tr>
<td>27_{16}‡ (ASI_LDTX_N)</td>
</tr>
</tbody>
</table>

---

*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.

---

256 UltraSPARC Architecture 2005 • Draft D0.8.7, 27 Mar 2006
LDTWA (Deprecated)

ASIs valid for LDTWA

<table>
<thead>
<tr>
<th>ASI_PRIMARY</th>
<th>ASI_PRIMARY_LITTLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_SECONDARY</td>
<td>ASI_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PRIMARY_NO_FAULT</td>
<td>ASI_PRIMARY_NO_FAULT_LITTLE</td>
</tr>
<tr>
<td>ASI_SECONDARY_NO_FAULT</td>
<td>ASI_SECONDARY_NO_FAULT_LITTLE</td>
</tr>
</tbody>
</table>

E2i† (ASI_LDTX_P)  EA1† (ASI_LDTX.PL)
E3i‡ (ASI_LDTX_S)  EB1‡ (ASI_LDTX_SL)

† If this ASI is used with the opcode for LDTWA and i = 0, the LDTXA instruction is executed instead of LDTWA. For behavior of LDTXA, see Load Integer Twin Extended Word from Alternate Space on page 250.

If this ASI is used with the opcode for LDTWA and i = 1, behavior is undefined.

Programming Note

Nontranslating ASIs (see page 387) 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 data_access_exception 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 250.

Exceptions

unimplemented_LDTW illegal_instruction
mem_address_not_aligned
privileged_action
VA_watchpoint
data_access_exception

See Also

LDWA/LDXA on page 229
STTWA on page 332
8.61 Memory Barrier

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 8-7 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.
The `cmask` field is encoded in bits 6 through 4 of the instruction. Bits in the `cmask` field, described in TABLE 8-8, specify additional constraints on the order of memory references and the processing of instructions. If `cmask` is zero, then MEMBAR enforces the partial ordering specified by the `mmask` field; if `cmask` is nonzero, then completion and partial order constraints are applied.

### TABLE 8-7 MEMBAR `mmask` Encodings

<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Assembly Language Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmask[3]</td>
<td>#StoreStore</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. Equivalent to the deprecated STBAR instruction.</td>
</tr>
<tr>
<td>mmask[2]</td>
<td>#LoadStore</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>mmask[1]</td>
<td>#StoreLoad</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>mmask[0]</td>
<td>#LoadLoad</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 `cmask` field is encoded in bits 6 through 4 of the instruction. Bits in the `cmask` field, described in TABLE 8-8, specify additional constraints on the order of memory references and the processing of instructions. If `cmask` is zero, then MEMBAR enforces the partial ordering specified by the `mmask` field; if `cmask` is nonzero, then completion and partial order constraints are applied.

### TABLE 8-8 MEMBAR `cmask` Encodings

<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Function</th>
<th>Assembly Language Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmask[2]</td>
<td>Synchronization barrier</td>
<td>#Sync</td>
<td>All operations (including nonmemory reference operations) appearing prior to the MEMBAR must have been performed and the effects of any exceptions be visible before any instruction after the MEMBAR may be initiated.</td>
</tr>
<tr>
<td>cmask[1]</td>
<td>Memory issue barrier</td>
<td>#MemIssue</td>
<td>All memory reference operations appearing prior to the MEMBAR must have been performed before any memory operation after the MEMBAR may be initiated.</td>
</tr>
<tr>
<td>cmask[0]</td>
<td>Lookaside barrier</td>
<td>#Lookaside</td>
<td>A store appearing prior to the MEMBAR must complete before any load following the MEMBAR referencing the same address can be initiated.</td>
</tr>
</tbody>
</table>

A MEMBAR instruction with both `mmask` = 0 and `cmask` = 0 is functionally a NOP.

For information on the use of MEMBAR, see Memory Ordering and Synchronization on page 381 and Programming with the Memory Models contained in the separate volume UltraSPARC Architecture Application Notes. For additional information about the memory models themselves, see Chapter 9, Memory.
MEMBAR

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

**V9 Compatibility Note**

MEMBAR with mmask = 8\_16 and cmask = 0\_16 (MEMBAR #StoreStore) is identical in function to the SPARC V8 STBAR instruction, which is deprecated.

An attempt to execute a MEMBAR instruction when instruction bits 12:7 are nonzero causes an *illegal_instruction* exception.

**Implementation Note**

MEMBAR shares an opcode with RDaSr and STBAR\^D; it is distinguished by rs1 = 15, rd = 0, i = 1, and bit 12 = 0.

### 8.61.1 Memory Synchronization

The UltraSPARC Architecture provides some level of software control over memory synchronization, through use of the MEMBAR and FLUSH instructions for explicit control of memory ordering in program execution.

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

**Implementation Note**

For an UltraSPARC Architecture virtual processor that only provides TSO memory ordering semantics, three of the ordering MEMBARs would normally be implemented as NOPs. TABLE 8-9 shows an acceptable implementation of MEMBAR for a TSO-only UltraSPARC Architecture implementation.

<table>
<thead>
<tr>
<th>MEMBAR variant</th>
<th>Preferred Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>#StoreStore, STBAR</td>
<td>NOP</td>
</tr>
<tr>
<td>#LoadStore</td>
<td>NOP</td>
</tr>
<tr>
<td>#StoreLoad</td>
<td>#Sync</td>
</tr>
<tr>
<td>#LoadLoad</td>
<td>NOP</td>
</tr>
<tr>
<td>#Sync</td>
<td>#Sync</td>
</tr>
<tr>
<td>#MemIssue</td>
<td>#Sync</td>
</tr>
<tr>
<td>#Lookaside</td>
<td>#Sync</td>
</tr>
</tbody>
</table>

If an UltraSPARC Architecture implementation provides a less restrictive memory model than TSO (for example, RMO), the implementation of the MEMBAR variants may be different. See implementation-specific documentation for details.
8.61.2 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.

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

8.61.3 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 378.

**Exceptions** | illegal_instruction
8.62 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 1000</td>
<td>Move Always</td>
<td>1</td>
<td>mova i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVN</td>
<td>10 1100 0000</td>
<td>Move Never</td>
<td>0</td>
<td>movn i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVNE</td>
<td>10 1100 1001</td>
<td>Move if Not Equal</td>
<td>not Z</td>
<td>movne i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVE</td>
<td>10 1100 0001</td>
<td>Move if Equal</td>
<td>Z</td>
<td>move i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVG</td>
<td>10 1100 1010</td>
<td>Move if Greater</td>
<td>not (Z or N xor V)</td>
<td>movg i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVLE</td>
<td>10 1100 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, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVGE</td>
<td>10 1100 1011</td>
<td>Move if Greater or Equal</td>
<td>not (N xor V)</td>
<td>movge i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVL</td>
<td>10 1100 0011</td>
<td>Move if Less</td>
<td>N xor V</td>
<td>movl i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVGU</td>
<td>10 1100 1100</td>
<td>Move if Greater, Unsigned</td>
<td>not (C or Z)</td>
<td>movgu i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVLEU</td>
<td>10 1100 0100</td>
<td>Move if Less or Equal, Unsigned</td>
<td>(C or Z)</td>
<td>movleu i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVCC</td>
<td>10 1100 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, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVCS</td>
<td>10 1100 0101</td>
<td>Move if Carry Set (Less than, Unsigned)</td>
<td>C</td>
<td>movcs i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVPOS</td>
<td>10 1100 1110</td>
<td>Move if Positive</td>
<td>not N</td>
<td>movpos i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVNEG</td>
<td>10 1100 0110</td>
<td>Move if Negative</td>
<td>N</td>
<td>movneg i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVVC</td>
<td>10 1100 1111</td>
<td>Move if Overflow Clear</td>
<td>not V</td>
<td>movvc i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>MOVVS</td>
<td>10 1100 0111</td>
<td>Move if Overflow Set</td>
<td>V</td>
<td>movvs i_or_x_cc, reg_or_imm11, reg Rd</td>
<td>A1</td>
<td></td>
</tr>
</tbody>
</table>

† synonym: movnz ‡ synonym: movz ◊ synonym: movgeu ∇ synonym: movleu ∨ synonym: movlu

Programming Note | In assembly language, to select the appropriate condition code, include %icc or %xcc before the reg_or_imm11 field.
For Floating-Point Condition Codes

<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>G or U</td>
<td>movug %fcc, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFG</td>
<td>10</td>
<td>1100</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</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>1000</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</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</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</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</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</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</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</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>1111</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.
**MOVcc**

<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>

**Description**

These instructions test to see if cond is True for the selected condition codes. If so, they copy the value in R[rs2] if i field = 0, or “sign_ext(simm11)” if i = 1 into R[rd]. The condition code used is specified by the cc2, cc1, and cc0 fields of the instruction. If the condition is False, then R[rd] is not changed.

These instructions copy an integer register to another integer register if the condition is True. The condition code that is used to determine whether the move will occur can be either integer condition code (icc or xcc) or any floating-point condition code (fcc0, fcc1, fcc2, or fcc3).

These instructions do not modify any condition codes.

**Programming Note**

Branches cause the performance of many implementations to degrade significantly. Frequently, the MOVcc and FMOVcc instructions can be used to avoid branches. For example, the C language if-then-else statement

if (A > B) then X = 1; else X = 0;

can be coded as

```asm
cmp %i0,%i2
bg,a %xcc,label
or %g0,1,%i3! X = 1
or %g0,0,%i3! X = 0
```

label:...

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

```asm
cmp %i0,%i2
or %g0,1,%i3! assume X = 1
movle %xcc,0,%i3! overwrite with X = 0
```

This approach takes only three instructions and no branches and may boost performance significantly. Use MOVcc and FMOVcc instead of branches wherever these instructions would increase performance.

An attempt to execute a MOVcc instruction when either instruction bits 10:5 are nonzero or (cc2 : cc1 : cc0) = 1012 or 1112 causes an illegal_instruction exception.
MOVcc

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

\begin{itemize}
  \item \texttt{Exceptions}
  \begin{itemize}
  \item \texttt{illegal\_instruction}
  \item \texttt{fp\_disabled}
  \end{itemize}
\end{itemize}
### 8.63 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>
<th>Class</th>
</tr>
</thead>
</table>
| MOVrZ        | 10 1111 | 000   | Move if Register Zero   | $R[rs1] = 0$ | movrz
testcond, reg_or_imm10, regrd | A1    |
| MOVrLEZ      | 10 1111 | 010   | Move if Register Less Than or Equal to Zero | $R[rs1] \leq 0$ | movrelz
testcond, reg_or_imm10, regrd | A1    |
| MOVrLZ       | 10 1111 | 011   | Move if Register Less Than Zero | $R[rs1] < 0$ | movrlz
testcond, reg_or_imm10, regrd | A1    |
| MOVrnZ       | 10 1111 | 100   | Move if Register Not Zero | $R[rs1] \neq 0$ | movrnz
testcond, reg_or_imm10, regrd | A1    |
| MOVrgZ       | 10 1111 | 110   | Move if Register Greater Than Zero | $R[rs1] > 0$ | movrgz
testcond, reg_or_imm10, regrd | A1    |
| MOVrGEZ      | 10 1111 | 111   | Move if Register Greater Than or Equal to Zero | $R[rs1] \geq 0$ | movrgez
testcond, reg_or_imm10, regrd | A1    |

<table>
<thead>
<tr>
<th></th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i=0</th>
<th>rcond</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>25</td>
<td>24</td>
<td>19</td>
<td>18</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>

† synonym: movre  ‡ synonym: movrne

**Description**

If the contents of integer register $R[rs1]$ satisfy the condition specified in the $rcond$ field, these instructions copy their second operand (if $i = 0$, $R[rs2]$; if $i = 1$, $\text{sign_ext}(\text{simm10})$) 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.
MOVr

**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>MOVRNZ</td>
<td><em>not Z</em></td>
</tr>
<tr>
<td>MOVRZ</td>
<td><em>Z</em></td>
</tr>
<tr>
<td>MOVGEZ</td>
<td><em>not N</em></td>
</tr>
<tr>
<td>MOVNZ</td>
<td><em>N</em></td>
</tr>
<tr>
<td>MOVNZ</td>
<td><em>N or Z</em></td>
</tr>
<tr>
<td>MOVNZ</td>
<td><em>N nor Z</em></td>
</tr>
</tbody>
</table>

An attempt to execute a MOVr instruction when either instruction bits 9:5 are nonzero or _rcond_ = 000₂ or 100₂ causes an _illegal_instruction_ exception.

**Exceptions** _illegal_instruction_
MULScc - Deprecated

8.64 Multiply Step

The MULScc instruction is 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>MULSccD</td>
<td>10 0100</td>
<td>Multiply Step and modify cc’s</td>
<td>mulscc rs1, reg_or_imm, reg</td>
<td>Y3</td>
</tr>
</tbody>
</table>

Description

MULScc treats the less-significant 32 bits of R[rs1] and the less-significant 32 bits of the Y register as a single 64-bit, right-shiftable doubleword register. The least significant bit of R[rs1] is treated as if it were adjacent to bit 31 of the Y register. The MULScc instruction performs an addition operation, based on the least significant bit of Y.

Multiplication assumes that the Y register initially contains the multiplier, R[rs1] contains the most significant bits of the product, and R[rs2] contains the multiplicand. Upon completion of the multiplication, the Y register contains the least significant bits of the product.

Note | In a standard MULScc instruction, rs1 = rd.

MULScc operates as follows:
1. If i = 0, the multiplicand is R[rs2]; if i = 1, the multiplicand is sign_ext(simm13).
2. A 32-bit value is computed by shifting the value from R[rs1] right by one bit with “CCR.icc.n xor CCR.icc.v” replacing bit 31 of R[rs1]. (This is the proper sign for the previous partial product.)
3. If the least significant bit of Y = 1, the shifted value from step (2) and the multiplicand are added. If the least significant bit of the Y = 0, then 0 is added to the shifted value from step (2).
MULScc - Deprecated

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][63:32]</td>
<td>undefined</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</td>
<td>undefined</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 *illegal_instruction* exception.

*Exceptions*  
*illegal_instruction*
8.65 Multiply and Divide (64-bit)

Description

MULX computes \( R[rs1] \times R[rs2] \) if \( i = 0 \) or \( R[rs1] \times \text{sign_ext}(\text{simm}13) \) 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 \( R[rs1] \div \text{sign_ext}(\text{simm}13) \) 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 \textit{illegal_instruction} exception.

Exceptions

\textit{illegal_instruction}

\textit{division_by_zero}
8.66 No Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op2</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>100</td>
<td>No Operation</td>
<td>nop</td>
<td>A1</td>
</tr>
</tbody>
</table>

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 only one guaranteed to be implemented efficiently across all implementations.

Exceptions

None
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.

In an UltraSPARC Architecture 2005 implementation, this instruction is not implemented in hardware, causes an illegal_instruction exception, and is emulated in software.

Exceptions
illegal_instruction (not implemented in hardware in UltraSPARC Architecture 2005)

See Also
ALLCLEAN on page 136
INVALW on page 225
OTHERW on page 274
RESTORED on page 292
SAVED on page 300
8.68  OR Logical Operation

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].

OR and ORcc 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
OTHERW

8.69 OTHERW

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTHERWP</td>
<td>“Normal” register windows become “other” otherw</td>
<td>otherw</td>
<td>C1</td>
</tr>
</tbody>
</table>

| 10 fcn = 0 0011 | 11 0001 | — |
| 31 30 29 25 24 19 18 0 |

**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.

In an UltraSPARC Architecture 2005 implementation, this instruction is not implemented in hardware, causes an illegal_instruction exception, and is emulated in software.

**Exceptions**

illegal_instruction (not implemented in hardware in UltraSPARC Architecture 2005)

**See Also**

ALLCLEAN on page 136
INVALW on page 225
NORMALW on page 272
RESTORED on page 292
SAVED on page 300
PDIST

8.70  Pixel Component Distance [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>0 0011 1110</td>
<td>Distance between eight 8-bit components, with accumulation</td>
<td>pdist rs1, rs2, rd</td>
<td>C3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
<th>rd</th>
<th>110110</th>
<th>rs1</th>
<th>opf</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>25</td>
<td>24</td>
<td>19</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**  
Typically, PDIST uses \( F_D[rd] \) as both a source and a destination register. PDIST uses \( F_D[rd] \) as both a source and a destination register. Typically, PDIST is used for motion estimation in video compression algorithms.

**Exceptions**  
illegal_instruction

In an UltraSPARC Architecture 2005 implementation, this instruction is not implemented in hardware, causes an illegal_instruction exception, and is emulated in software.
8.71 Population Count

**Description**

POPC counts the number of one bits in $R[rs2]$ if $i = 0$, or the number of one bits in $\text{sign\_ext}(\text{simm13})$ 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(zz) /* finds first 1 bit, counting from the LSB */
   unsigned zz;
{
    return popc ( zz ^ (~ (~zz))); /* for nonzero zz */
}
```

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

```
neg %IN, %M_IN  ! -zz(2’s complement)
 xor %IN, %M_IN, %TEMP  ! ^ -zz (exclusive nor)
popc %TEMP, %RESULT  ! result = popc(zz ^ -zz)
movz %IN, %g0,%RESULT  ! %RESULT should be 0 for %IN=0
```

where $IN$, $M\_IN$, $TEMP$, and $RESULT$ are integer registers.

**Example computation:**

```
IN = ...00101000 !1st ‘1’ bit from right is
-IN = ...11011000 ! bit 3 (4th bit)
~IN = ...00100111
IN ^ ~IN = ...00001111
popc(IN ^ ~IN) = 4
```

### 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>POPC</td>
<td>10 1110</td>
<td>Population Count</td>
<td>popc reg_or_imm, regrd</td>
<td>D3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
<th>rd</th>
<th>op3</th>
<th>0 0000</th>
<th>i=0</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 0000</td>
<td>0</td>
<td>---</td>
<td>0 0000</td>
</tr>
<tr>
<td>31 30 29 25 24 19 18 14 13 12 5 4 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In an UltraSPARC Architecture 2005 implementation, this instruction is not implemented in hardware, causes an illegal_instruction exception, and is emulated in software.

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

#### Assembly Language Syntax

<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>PREFETCH</td>
<td>10 1101</td>
<td>Prefetch Data</td>
<td>prefetch [address], prefetch_fcn</td>
<td>A1</td>
</tr>
<tr>
<td>PREFETCHAASI</td>
<td>11 1101</td>
<td>Prefetch Data from Alternate Space</td>
<td>prefetcha [regaddr] immasi, prefetch_fcn</td>
<td>A1</td>
</tr>
</tbody>
</table>

#### TABLE 8-10 Prefetch Variants, by Function Code

<table>
<thead>
<tr>
<th>fcn</th>
<th>Prefetch Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Weak) Prefetch for several reads</td>
</tr>
<tr>
<td>1</td>
<td>(Weak) Prefetch for one read</td>
</tr>
<tr>
<td>2</td>
<td>(Weak) Prefetch for several writes and possibly reads</td>
</tr>
<tr>
<td>3</td>
<td>(Weak) Prefetch for one write</td>
</tr>
<tr>
<td>4</td>
<td>Prefetch page</td>
</tr>
<tr>
<td>5–15</td>
<td>Reserved (illegal_instruction)</td>
</tr>
<tr>
<td>16</td>
<td>Implementation dependent (NOP if not implemented)</td>
</tr>
<tr>
<td>17</td>
<td>Implementation dependent (NOP if not implemented)</td>
</tr>
<tr>
<td>18–19</td>
<td>Implementation dependent (NOP if not implemented)</td>
</tr>
<tr>
<td>20</td>
<td>Strong Prefetch for several reads</td>
</tr>
<tr>
<td>21</td>
<td>Strong Prefetch for one read</td>
</tr>
<tr>
<td>22</td>
<td>Strong Prefetch for several writes and possibly reads</td>
</tr>
<tr>
<td>23</td>
<td>Strong Prefetch for one write</td>
</tr>
<tr>
<td>24–31</td>
<td>Implementation dependent (NOP if not implemented)</td>
</tr>
</tbody>
</table>
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.

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 \( (\text{ASI}_{\text{PRIMARY}}[_{\text{LITTLE}}]) \).

PREFETCHA instructions access an alternate address space. If \( i = 0 \), the address space identifier (ASI) to be used for the instruction is in the \( \text{imm}_\text{asi} \) 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 PREFETCH[A] instruction is non-blocking; subsequent instructions can continue to execute while the prefetch is in progress.

When executed in nonprivileged or privileged mode, PREFETCH[A] has the same observable effect as a NOP. A prefetch instruction will not cause a trap if applied to an illegal or nonexistent memory address. (impl. dep. #103-V9-Ms10(e))

\textbf{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.

Software may prefetch 64 bytes beginning at an arbitrary address by issuing the instructions

\begin{verbatim}
  prefetch [address], prefetch_fcn
  prefetch [address + 63], prefetch_fcn
\end{verbatim}

Variants of the prefetch instruction can be used to prepare the memory system for different types of accesses.

\textbf{IMPL. DEP. #103-V9-Ms10(b):} An implementation may implement none, some, or all of the defined PREFETCH[A] variants. It is implementation-dependent whether each variant is (1) not implemented and executes as a NOP, (2) is implemented and supports the full semantics for that variant, or (3) is implemented and only supports the simple common-case prefetching semantics for that variant.
8.72.1 Exceptions

Prefetch instructions PREFETCH and PREFETCHA generate exceptions under the conditions detailed in TABLE 8-11. Only the implementation-dependent prefetch variants (see TABLE 8-10) may generate an exception under conditions not listed in this table; the predefined variants only generate the exceptions listed here.

TABLE 8-11 Behavior of PREFETCH[A] Instructions Under Exceptional Conditions

<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}-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}^{16}-7F_{16} ), while in privileged mode (privileged_action condition)</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-3 (weak)</td>
<td>PREFETCH[A]</td>
<td>condition detected for MMU miss</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-4</td>
<td>PREFETCH[A]</td>
<td>variant unimplemented</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-4</td>
<td>PREFETCHA</td>
<td>reference to an invalid ASI (ASI not listed in following table)</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-4, 17, 20-23</td>
<td>PREFETCH[A]</td>
<td>condition detected for ( ((\text{TTE.cp} = 0) ) or ( ((\text{fcn} = 0) ) and ( \text{TTE.cv} = 0)) ), or ( (\text{TTE.e} = 1) )</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>4, 20-23 (strong)</td>
<td>PREFETCHA</td>
<td>prefetching the requested data would be a very time-consuming operation</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>5–15 (05_{16}^{16}–0F_{16})</td>
<td>PREFETCH[A]</td>
<td>(always)</td>
<td>illegal_instruction</td>
</tr>
<tr>
<td>16-31 (18_{16}^{16}-1F_{16})</td>
<td>PREFETCH[A]</td>
<td>variant unimplemented</td>
<td>executes as NOP</td>
</tr>
</tbody>
</table>

ASIs valid for PREFETCHA (all others are invalid)

- ASI_NUCLEUS
- ASI_INDEX
- ASI_INDEX_PRIMARY
- ASI_INDEX_SECONDARY
- ASI_INDEX_PRIMARY_NO_FAULT
- ASI_INDEX_SECONDARY_NO_FAULT
- ASI_INDEX_REAL

- ASI_INDEX_NUCLEUS
- ASI_INDEX_INDEX
- ASI_INDEX_INDEX_PRIMARY
- ASI_INDEX_INDEX_SECONDARY
- ASI_INDEX_INDEX_PRIMARY_NO_FAULT
- ASI_INDEX_INDEX_SECONDARY_NO_FAULT
- ASI_INDEX_INDEX_REAL

280 UltraSPARC Architecture 2005 • Draft D0.8.7, 27 Mar 2006
8.72.2 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 requests 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 NOP while a Strong prefetch may pay the cost of performing the time-consuming operation so it can finish initiating the requested data movement. Behavioral differences among loads and prefetches are compared in TABLE 8-12.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upon detection of privileged_action, data_access_exception or VA_watchpoint exception…</td>
<td>Traps 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 NOP‡</td>
</tr>
<tr>
<td>If page table entry has nfo = 1 for a non-NoFault access…</td>
<td>Traps 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 NOP‡</td>
</tr>
<tr>
<td>Instruction blocks until cache line filled?</td>
<td>Yes No</td>
</tr>
</tbody>
</table>

8.72.3 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 2005.
PREFETCH

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).

8.72.3.1 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. |

8.72.3.2 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.

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. |

8.72.3.3 Prefetch for Several Writes (and Possibly Reads) (fcn = 2, 22(1616))

The intent of this variant is to cause movement of data in preparation for multiple writes.
PREFETCH

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.

8.72.3.4 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.

8.72.3.5 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.

8.72.4 Implementation-Dependent Prefetch Variants (fcn = 16, 18, 19, and 24–31)

**IMPL. DEP. #103-V9-Ms10(c):** Whether and how PREFETCH fcn 16, 18, 19 and 24-31 are implemented are implementation dependent. If a variant is not implemented, it must execute as a NOP.
8.72.5 Additional Notes

**Programming Note**
Prefetch instructions do have some “cost to execute”. As long as the cost of executing a prefetch instruction is well less than the cost of a cache miss, use of prefetching provides a net gain in performance.

It does not appear that prefetching causes a significant number of useless fetches from memory, though it may increase the rate of useful fetches (and hence the bandwidth), because it more efficiently overlaps computing with fetching.

**Programming Note**
A compiler that generates PREFETCH instructions should generate each of the variants where its use is most appropriate. That will help portable software be reasonably efficient across a range of hardware configurations.

**Implementation Note**
Any effects of a data prefetch operation in privileged 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.

**Implementation Note**
A prefetch from a nonprefetchable location has no effect. It is up to memory management hardware to determine how locations are identified as not prefetchable.

*Exceptions* illegal_instruction
8.73 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&lt;sup&gt;D&lt;/sup&gt;</td>
<td>0</td>
<td>Read Y register (deprecated)</td>
<td>rd %y, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>C2</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, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>RDASI</td>
<td>3</td>
<td>Read ASI register</td>
<td>rd %asi, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>RDTICK&lt;sup&gt;P&lt;/sup&gt;&lt;sub&gt;opt&lt;/sub&gt;</td>
<td>4</td>
<td>Read TICK register</td>
<td>rd %tick, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>RDPC</td>
<td>5</td>
<td>Read Program Counter (PC)</td>
<td>rd %pc, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B2</td>
</tr>
<tr>
<td>RDFPRS</td>
<td>6</td>
<td>Read Floating-Point Registers Status (FPRS) register</td>
<td>rd %fprs, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>7–14</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See text</td>
<td>15</td>
<td>STBAR, MEMBAR or Reserved; see text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDPCR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>16</td>
<td>Read Performance Control registers (PCR)</td>
<td>rd %pcr, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>RDPI&lt;sup&gt;P&lt;/sup&gt;&lt;sub&gt;MC&lt;/sub&gt;</td>
<td>17</td>
<td>Read Performance Instrumentation Counters register (PIC)</td>
<td>rd %pic, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>18</td>
<td>Reserved (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDGSR</td>
<td>19</td>
<td>Read General Status register (GSR)</td>
<td>rd %gsr, rs&lt;sub&gt;rd&lt;/sub&gt;</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>RD&lt;sup&gt;P&lt;/sup&gt;&lt;sub&gt;SOFTINT&lt;/sub&gt;</td>
<td>22</td>
<td>Read per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>rd %softint, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N2</td>
</tr>
<tr>
<td>RDTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>23</td>
<td>Read Tick Compare register (TICK_CMPR)</td>
<td>rd %tick_cmp, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N2</td>
</tr>
<tr>
<td>RDSTICK&lt;sup&gt;P&lt;/sup&gt;&lt;sub&gt;opt&lt;/sub&gt;</td>
<td>24</td>
<td>Read System Tick Register (STICK)</td>
<td>rd %sys_tick, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N2</td>
</tr>
<tr>
<td>RDSTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>25</td>
<td>Read System Tick Compare register (STICK_CMPR)</td>
<td>rd %sys_tick_cmp, rs&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>N2</td>
</tr>
<tr>
<td>—</td>
<td>26–27</td>
<td>Reserved (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>28–31</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
<th>rd</th>
<th>10 1000</th>
<th>rs1</th>
<th>i=0</th>
<th>—</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>25</td>
<td>24</td>
<td>19</td>
</tr>
</tbody>
</table>

CHAPTER 8 • Instructions 285
**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)

RDIPRS waits for all pending FPop 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, 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 Note**

See the section “Read/Write Ancillary State Registers (ASRs)” in *Extending the UltraSPARC Architecture*, contained in the separate volume *UltraSPARC Architecture Application Notes*, for a discussion of extending the SPARC V9 instruction set using read/write ASR instructions.

**Note**

Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers.

**SPARC V8 Compatibility Note**

The SPARC V8 RDPSR, RDWIM, and RDTBR instructions do not exist in the UltraSPARC Architecture, since the PSR, WIM, and TBR registers do not exist.

See *Ancillary State Registers* on page 67 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 $rd \neq 0$ (reserved for future versions of the architecture)
- $rs1 = 1, 7-14, 18, 20-21, or 26-27$ (reserved for future versions of the architecture)
- instruction bits 13:0 are nonzero

An attempt to execute a RDPCR (impl. dep. #250-U3-Cs10), RDSTICK_CMPR, RDSTICK, or RDSTICK_CMPR instruction in nonprivileged mode ($PSTATE.priv = 0$) causes a privileged_opcode exception (impl. dep. #250-U3-Cs10).

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

In nonprivileged mode ($PSTATE.priv = 0$), the following cause a privileged_action exception:

- execution of RDTICK when $TICK.npt = 1$
- execution of RDSTICK when $STICK.npt = 1$
- execution of RDPIC when nonprivileged access to PIC is disabled ($PCR.priv = 1$)

Implementation Note | RDasr shares an opcode with MEMBAR and STBAR, it is distinguished by $rs1 = 15$ or $rd = 0$ or ($i = 0$, and bit 12 = 0).
8.74 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>rs1</td>
<td>rdpr %tpc, rsrd</td>
<td>C2</td>
</tr>
<tr>
<td>TPC</td>
<td>0</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %tnpc, rsrd</td>
<td></td>
</tr>
<tr>
<td>TNPC</td>
<td>1</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %tstate, rsrd</td>
<td></td>
</tr>
<tr>
<td>TSTATE</td>
<td>2</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %tt, rsrd</td>
<td></td>
</tr>
<tr>
<td>TT</td>
<td>3</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %tba, rsrd</td>
<td></td>
</tr>
<tr>
<td>TICK</td>
<td>4</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %tick, rsrd</td>
<td></td>
</tr>
<tr>
<td>TBA</td>
<td>5</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %tstate, rsrd</td>
<td></td>
</tr>
<tr>
<td>PSTATE</td>
<td>6</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %tstate, rsrd</td>
<td></td>
</tr>
<tr>
<td>TL</td>
<td>7</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %tl, rsrd</td>
<td></td>
</tr>
<tr>
<td>PIL</td>
<td>8</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %pl, rsrd</td>
<td></td>
</tr>
<tr>
<td>CWP</td>
<td>9</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %cwp, rsrd</td>
<td></td>
</tr>
<tr>
<td>CANSAVE</td>
<td>10</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %cansave, rsrd</td>
<td></td>
</tr>
<tr>
<td>CANRESTORE</td>
<td>11</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %canrestore, rsrd</td>
<td></td>
</tr>
<tr>
<td>CLEANWIN</td>
<td>12</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %cleanwin, rsrd</td>
<td></td>
</tr>
<tr>
<td>OTHERWIN</td>
<td>13</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %otherwin, rsrd</td>
<td></td>
</tr>
<tr>
<td>WSTATE</td>
<td>14</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %wstate, rsrd</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>15</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %gl, rsrd</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>16</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %gl, rsrd</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>17–31</td>
<td>rdpr</td>
<td>rs1</td>
<td>rdpr %gl, rsrd</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.
**RDPR**

**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 285  
WRPR on page 356
8.75 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, $N\_REG\_WINDOWS$.

Programming Notes

Typically, if a RESTORE instruction traps, the fill trap handler returns to the trapped instruction to reexecute it. So, although the ADD operation is not performed the first time (when the instruction traps), it is performed the second time the instruction executes. The same applies to changing the CWP.

There is a performance trade-off to consider between using SAVE/RESTORE and saving and restoring selected registers explicitly.

Description (Effect on Privileged State)

If a RESTORE instruction does not trap, it decrements the CWP (mod $N\_REG\_WINDOWS$) to restore the register window that was in use prior to the last SAVE instruction executed by the current process. It also updates the state of the register windows by decrementing CANRESTORE and incrementing CANSAVE.
RESTORE

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 428. The fill trap handler is invoked with CWP set to point to the window to be filled, that is, old CWP = 1.

**Programming Note**
The vectoring of fill traps can be controlled by setting the value of the OTHERWIN and WSTATE registers appropriately. For details, see the section “Splitting the Register Windows” in Software Considerations, contained in the separate volume UltraSPARC Architecture Application Notes.

The fill handler normally will end with a RESTORED instruction followed by a RETRY instruction.

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

**Exceptions**
- `illegal_instruction`
- `fill_n_normal` (n = 0–7)
- `fill_n_other` (n = 0–7)

**See Also**
SAVE on page 298
8.76 RESTORED

Description

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

RESTORED increments CANRESTORE.

If CLEANWIN < (N_REG_WINDOWS−1), then RESTORED increments CLEANWIN.

If OTHERWIN = 0, RESTORED decrements CANSAVE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

If CANSAVE = 0 or CANRESTORE ≥ (N_REG_WINDOWS − 2) just prior to execution of a RESTORED instruction, the subsequent behavior of the processor is undefined. In neither of these cases can RESTORED generate a register window state that is both valid (see Register Window State Definition on page 85) and consistent with the state prior to the RESTORED.

An attempt to execute a RESTORED instruction when instruction bits 18:0 are nonzero causes an illegal_instruction exception.

An attempt to execute a RESTORED instruction in nonprivileged mode (PSTATE.priv = 0) causes a privileged_opcode exception.

---

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.

---

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESTORED_P</td>
<td>Window has been restored</td>
<td>restored</td>
<td>C1</td>
</tr>
</tbody>
</table>

---

Instruction  Operation  Assembly Language Syntax  Class

| 10 | fcn = 0 0001 | 11 0001 | — |

---

31 19 24 18 0 25 30 29
Exceptions

illegal_instruction
privileged_opcode

See Also

ALLCLEAN on page 136
INVALW on page 225
NORMALW on page 272
OTHERW on page 274
SAVED on page 300
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).

If the saved TPC[TL] and TNPC[TL] were not altered by trap handler software, RETRY causes execution to resume at the instruction that originally caused the trap ("retrying" it).

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

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 the following condition is true causes an illegal_instruction exception:

- TL = 0 and the virtual processor is in privileged mode (PSTATE.priv = 1)
RETRY

An attempt to execute a RETRY instruction in nonprivileged mode (PSTATE.priv = 0) causes a privileged_opcode exception.

**Implementation**

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).

**Exceptions**

illegal_instruction
privileged_opcode

**See Also**

DONE on page 154
8.78 \textbf{RETURN}

The \textbf{RETURN} instruction causes a delayed transfer of control to the target address and has the window semantics of a \textbf{RESTORE} instruction; that is, it restores the register window prior to the last \textbf{SAVE} instruction. The target address is \textit{“R[rs1] + R[rs2]”} if \( i = 0 \), or \textit{“R[rs1] + sign_ext (simm13)”} if \( i = 1 \). Registers \( R[rs1] \) and \( R[rs2] \) come from the \textit{old} window.

Like other DCTIs, all effects of \textbf{RETURN} (including modification of CWP) are visible prior to execution of the delay slot instruction.

\textbf{Description}  
To reexecute the trapped instruction when returning from a user trap handler, use the \textbf{RETURN} instruction in the delay slot of a \textbf{JMPN} instruction, for example:

\begin{verbatim}
jmpl%l6,%g0 ! Trapped PC supplied to user trap handler
return%l7 ! Trapped NPC supplied to user trap handler
\end{verbatim}

\textbf{Programming Note}  
A routine that uses a register window may be structured either as:

\begin{verbatim}
save %sp, -framesize, %sp
... ret ! Same as jmpl %i7 + 8, %g0
restore ! Something useful like “restore
: %o2,%l2,%o0”
or as:
save %sp, -framesize, %sp
... return %i7 + 8
nop ! Could do some useful work in the
: caller’s window, e.g., “or %o1, %o2,%o0”
\end{verbatim}

An attempt to execute a \textbf{RETURN} instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an \textit{illegal_instruction} exception.

A \textbf{RETURN} instruction may cause a \textit{window_fill} exception as part of its \textbf{RESTORE} semantics.

When \texttt{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.
RETURN

A RETURN instruction causes a `mem_address_not_aligned` exception if either of the two least-significant bits of the target address is nonzero.

Exceptions

- `illegal_instruction`
- `fill_n_normal (n = 0–7)`
- `fill_n_other (n = 0–7)`
- `mem_address_not_aligned`
SAVE

8.79 SAVE

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.
SAVE

If the new register window is occupied (that is, CANSAVE = 0), a spill trap is generated. The trap vector for the spill trap is based on the value of OTHERWIN and WSTATE. The spill trap handler is invoked with the CWP set to point to the window to be spilled (that is, old CWP + 2).

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

If CANSAVE ≠ 0, the SAVE instruction checks whether the new window needs to be cleaned. It causes a clean_window trap if the number of unused clean windows is zero, that is, (CLEANWIN – CANRESTORE) = 0. The clean_window trap handler is invoked with the CWP set to point to the window to be cleaned (that is, old CWP + 1).

Exceptions

illegal_instruction
spill_n_normal (n = 0–7)
spill_n_other (n = 0–7)
clean_window

See Also

RESTORE on page 290

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.
8.80 SAVED

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVEDP</td>
<td>Window has been saved</td>
<td>saved</td>
<td>C1</td>
</tr>
</tbody>
</table>

Description

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

SAVED increments CANSAVE. If OTHERWIN = 0, SAVED decrements CANRESTORE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

Programming Notes

Trap handler software for register window spills uses the SAVED instruction to indicate that a window has been spilled successfully. For details, see the section “Example Code for Spill Handler” in Software Considerations, contained in the separate volume UltraSPARC Architecture Application Notes.

Normal privileged software would probably not execute a SAVED instruction from trap level zero (TL = 0). However, it is not illegal to do so and doing so does not cause a trap.

Executing a SAVED instruction outside of a window spill trap handler is likely to create an inconsistent window state. Hardware will not signal an exception, however, since maintaining a consistent window state is the responsibility of privileged software.

If CANSAVE ≥ (N_REG_WINDOWS – 2) or CANRESTORE = 0 just prior to execution of a SAVED instruction, the subsequent behavior of the processor is undefined. In neither of these cases can SAVED generate a register window state that is both valid (see Register Window State Definition on page 85) and consistent with the state prior to the SAVED.

An attempt to execute a SAVED instruction when instruction bits 18:0 are nonzero causes an illegal_instruction exception.

An attempt to execute a SAVED instruction in nonprivileged mode (PSTATE.priv = 0) causes a privileged_opcode exception.

Exceptions

illegal_instruction

privileged_opcode
SAVED

See Also

ALLCLEAN on page 136
INVALW on page 225
NORMALW on page 272
OTHERW on page 274
RESTORED on page 292
SETHI

8.81 SETHI

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op2</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETHI</td>
<td>100</td>
<td>Set High 22 Bits of Low Word</td>
<td>sethi const22, rd$rd  sethi %hi(value), rd$rd</td>
<td>A1</td>
</tr>
</tbody>
</table>

Description

SETHI zeroes the least significant 10 bits and the most significant 32 bits of \( R[r_d] \) and replaces bits 31 through 10 of \( R[r_d] \) with the value from its \( \text{imm22} \) field.

SETHI does not affect the condition codes.

Some SETHI instructions with \( r_d = 0 \) have special uses:

- \( r_d = 0 \) and \( \text{imm22} = 0 \): defined to be a NOP instruction (described in No Operation)
- \( r_d = 0 \) and \( \text{imm22} \neq 0 \) 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
8.82 SHUTDOWN [VIS 1]

The SHUTDOWN instruction is deprecated and should not be used in new software.

<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>SHUTDOWND,P</td>
<td>0 1000 0000</td>
<td>Enter low-power mode</td>
<td>shutdown</td>
<td>D3</td>
</tr>
</tbody>
</table>

### Description

SHUTDOWN is a deprecated, privileged instruction that was used in early UltraSPARC implementations to bring the virtual processor or its containing system into a low-power state in an orderly manner. It had no effect on software-visible virtual processor state.

On an UltraSPARC Architecture implementation operating in privileged mode, SHUTDOWN behaves like a NOP (impl. dep. #206-U3-Cs10).

In an UltraSPARC Architecture 2005 implementation, this instruction is not implemented in hardware, causes an `illegal_instruction` exception, and its effect is emulated in software.

### Exceptions

`illegal_instruction` (instruction not implemented in hardware)
## 8.83 Set Interval Arithmetic Mode [VIS2]

<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>SIAM</td>
<td>01000001</td>
<td>Set the interval arithmetic mode fields in the GSR</td>
<td><code>siam siam_mode</code></td>
<td>B1</td>
</tr>
</tbody>
</table>

### Description

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

- \( \text{GSR.im} \leftarrow \text{mode}[2] \)
- \( \text{GSR.irnd} \leftarrow \text{mode}[1:0] \)

#### Note

When \( \text{GSR.im} \) 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 \( \text{GSR.im} = 1 \), 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
8.84 Shift

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 8-13 shows the shift count encodings for all values of \( i \) and \( x \).

### TABLE 8-13 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>

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

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

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

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

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

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

These instructions do not modify the condition codes.

Programming Notes

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

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

An attempt to execute a SLL, SRL, or SRA instruction when instruction bits 11:5 are nonzero causes an illegal_instruction exception.

An attempt to execute a SLLX, SRLX, or SRAX instruction when either of the following conditions exist causes an illegal_instruction exception:

- $i = 0$ or $x = 0$ and instruction bits 11:5 are nonzero
- $x = 1$ and instruction bits 11:6 are nonzero

Exceptions

illegal_instruction
8.85 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&lt;sup&gt;†&lt;/sup&gt; &lt;sub&gt;reg&lt;/sub&gt;&lt;sub&gt;rd&lt;/sub&gt; [address]</td>
<td>A1</td>
</tr>
<tr>
<td>STH</td>
<td>00 0110</td>
<td>Store Halfword</td>
<td>sth&lt;sup&gt;‡&lt;/sup&gt; &lt;sub&gt;reg&lt;/sub&gt;&lt;sub&gt;rd&lt;/sub&gt; [address]</td>
<td>A1</td>
</tr>
<tr>
<td>STW</td>
<td>00 0100</td>
<td>Store Word</td>
<td>stw&lt;sup&gt;◊&lt;/sup&gt; &lt;sub&gt;reg&lt;/sub&gt;&lt;sub&gt;rd&lt;/sub&gt; [address]</td>
<td>A1</td>
</tr>
<tr>
<td>STX</td>
<td>00 1110</td>
<td>Store Extended Word</td>
<td>stx &lt;sub&gt;reg&lt;/sub&gt;&lt;sub&gt;rd&lt;/sub&gt; [address]</td>
<td>A1</td>
</tr>
</tbody>
</table>

<sup>†</sup> synonyms: stub, stab  <sup>‡</sup> synonyms: stuh, stah  <sup>◊</sup> 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<sub>rd</sub> into memory.

These instructions access memory using the implicit ASI (see page 104). The effective address for these instructions is “R<sub>rs1</sub> + R<sub>rs2</sub>” if <i>i</i> = 0, or “R<sub>rs1</sub> + sign_ext (simm13)” if <i>i</i> = 1.

A successful store (notably, STX) integer instruction operates atomically.

An attempt to execute a store integer instruction when <i>i</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 330
### 8.86 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>STBA\textsuperscript{P,ASI}</td>
<td>01 0101</td>
<td>Store Byte into Alternate Space</td>
<td>\texttt{stba}^{†} \texttt{regrd}, \texttt{regaddr} \texttt{imm Asi}</td>
<td>A1</td>
</tr>
<tr>
<td>STHA\textsuperscript{P,ASI}</td>
<td>01 0110</td>
<td>Store Halfword into Alternate Space</td>
<td>\texttt{stha}^{‡} \texttt{regrd}, \texttt{regaddr} \texttt{imm Asi}</td>
<td>A1</td>
</tr>
<tr>
<td>STWA\textsuperscript{P,ASI}</td>
<td>01 0100</td>
<td>Store Word into Alternate Space</td>
<td>\texttt{stwa}^{◊} \texttt{regrd}, \texttt{regaddr} \texttt{imm Asi}</td>
<td>A1</td>
</tr>
<tr>
<td>STXAPASI</td>
<td>01 1110</td>
<td>Store Extended Word into Alternate Space</td>
<td>\texttt{stxa} \texttt{regrd}, \texttt{regaddr} \texttt{imm Asi}</td>
<td>A1</td>
</tr>
</tbody>
</table>

\begin{itemize}
  \item \texttt{† synonyms: stuba, stsba}
  \item \texttt{‡ synonyms: stuha, stsha}
  \item \texttt{◊ synonyms: sta, stuwa, stswa}
\end{itemize}

### 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 \texttt{R[rd]} into memory.

Store integer to alternate space instructions contain the address space identifier (ASI) to be used for the store in the \texttt{imm Asi} field if \texttt{i = 0}, or in the ASI register if \texttt{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 \texttt{“R[rs1] + R[rs2]”} if \texttt{i = 0}, or \texttt{“R[rs1] + sign_ext (simm13)”} if \texttt{i = 1}.

A successful store (notably, STXA) instruction operates atomically.

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

STHA causes a \texttt{mem_address not aligned} exception if the effective address is not halfword-aligned. STWA causes a \texttt{mem_address not aligned} exception if the effective address is not word-aligned. STXA causes a \texttt{mem_address not aligned} exception if the effective address is not doubleword-aligned.
STBA / STHA / STWA / STXA

STBA, STHA, and STWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *data_access_exception* exception.

<table>
<thead>
<tr>
<th>ASIs valid for STBA, STHA, and STWA</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>

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 *data_access_exception* exception.

<table>
<thead>
<tr>
<th>ASIs invalid for STXA (cause <em>data_access_exception</em> exception)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 16 (aliased to 27 16, ASI_LDTX_N)</td>
</tr>
<tr>
<td>2C 16 (aliased to 2F 16, ASI_LDTX_NL)</td>
</tr>
<tr>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>24 16 (deprecated ASI_QUAD_LDD)</td>
</tr>
<tr>
<td>2C 16 (deprecated ASI_QUAD_LDD_L)</td>
</tr>
<tr>
<td>ASI_PST8_PRIMARY</td>
</tr>
<tr>
<td>ASI_PST8_SECONDARY</td>
</tr>
<tr>
<td>ASI_PST8_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PRIMARY_NO_FAULT</td>
</tr>
<tr>
<td>ASI_SECONDARY_NO_FAULT</td>
</tr>
<tr>
<td>ASI_PST16_PRIMARY</td>
</tr>
<tr>
<td>ASI_PST16_SECONDARY</td>
</tr>
<tr>
<td>ASI_PST16_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PST32_PRIMARY</td>
</tr>
<tr>
<td>ASI_PST32_SECONDARY</td>
</tr>
<tr>
<td>ASI_PST32_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_FL8_PRIMARY</td>
</tr>
<tr>
<td>ASI_FL8_SECONDARY</td>
</tr>
<tr>
<td>ASI_FL8_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_FL16_PRIMARY</td>
</tr>
<tr>
<td>ASI_FL16_SECONDARY</td>
</tr>
<tr>
<td>ASI_FL16_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_BLOCK_COMMIT_PRIMARY</td>
</tr>
<tr>
<td>ASI_BLOCK_COMMIT_SECONDARY</td>
</tr>
<tr>
<td>ASI_BLOCK_PRIMARY</td>
</tr>
<tr>
<td>ASI_BLOCK_SECONDARY</td>
</tr>
<tr>
<td>ASI_BLOCK_SECONDARY_LITTLE</td>
</tr>
</tbody>
</table>

**V8 Compatibility** The SPARC V8 STA instruction was renamed STWA in the SPARC V9 architecture.

**Note**
STBA / STHA / STWA / STXA

Exceptions

- mem_address_not_aligned (all except STBA)
- privileged_action
- VA_watchpoint

See Also

- LDA on page 229
- STTWA on page 332
8.87 Store Barrier

The STBAR instruction is deprecated. Use the MEMBAR instruction 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>STBAR</td>
<td>10 1000</td>
<td>Store Barrier</td>
<td>stbar</td>
<td>Y2</td>
</tr>
</tbody>
</table>

Description
The store barrier instruction (STBAR) forces all store and atomic load-store operations issued by a virtual processor prior to the STBAR to complete their effects on memory before any store or atomic load-store operations issued by that virtual processor subsequent to the STBAR are executed by memory.

V8 Compatibility
STBAR is identical in function to a MEMBAR instruction with mmask = 8_{16}. STBAR is retained for compatibility with existing SPARC V8 software.

For correctness, it is sufficient for a virtual processor to stop issuing new store and atomic load-store operations when an STBAR is encountered and to resume after all stores have completed and are observed in memory by all virtual processors. More efficient implementations may take advantage of the fact that the virtual processor is allowed to issue store and load-store operations after the STBAR, as long as those operations are guaranteed not to become visible before all the earlier stores and atomic load-stores have become visible to all virtual processors.

An attempt to execute a STBAR instruction when instruction bits 12:0 are nonzero causes an illegal_instruction exception.

Implementation Note
STBAR shares an opcode with MEMBAR, and RDasr; it is distinguished by rs1 = 15, rd = 0, i = 0, and bit 12 = 0.

Exceptions
illegal_instruction
8.88 Block Store [VIS1]

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, [regaddr] #ASI_BLK_AIUP</td>
<td>A2</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>1716</td>
<td>64-byte block store to secondary address space, user privilege</td>
<td>stda freg, [regaddr] #ASI_BLK_AIUS</td>
<td>A2</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, [regaddr] #ASI_BLK_AIUPL</td>
<td>A2</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, [regaddr] #ASI_BLK_AIUSL</td>
<td>A2</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>F016</td>
<td>64-byte block store to primary address space</td>
<td>stda freg, [regaddr] #ASI_BLK_P</td>
<td>A2</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>F116</td>
<td>64-byte block store to secondary address space</td>
<td>stda freg, [regaddr] #ASI_BLK_S</td>
<td>A2</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>F816</td>
<td>64-byte block store to primary address space, little-endian</td>
<td>stda freg, [regaddr] #ASI_BLK_PL</td>
<td>A2</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>F916</td>
<td>64-byte block store to secondary address space, little-endian</td>
<td>stda freg, [regaddr] #ASI_BLK_SL</td>
<td>A2</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...
STBLOCKF

format; otherwise, the access is assumed to be big-endian. Byte swapping is performed separately for each of the eight double-precision registers accessed by the instruction.

**Programming Note** The block store instruction, STBLOCKF, and its companion, LDBLOCKF, were originally defined to provide a fast mechanism for block-copy operations.

STBLOCKF stores data from the eight double-precision floating-point registers specified by rd to a 64-byte-aligned memory area. The lowest-addressed eight bytes in memory are stored from the lowest-numbered double-precision rd.

While a STBLOCKF operation is in progress, any of the following values may be observed in a destination doubleword memory locations: (1) the old data value, (2) zero, or (3) the new data value. When the operation is complete, only the new data values will be seen.

**Compatibility Note** Software written for older UltraSPARC implementations that reads data being written by STBLOCKF instructions may or may not allow for case (2) above. Such software should be checked to verify that either it always waits for STBLOCKF to complete before reading the values written, or that it will operate correctly if an intermediate value of zero (not the “old” or “new” data values) is observed while the STBLOCKF operation is in progress.

A Block Store only guarantees atomicity for each 64-bit (8-byte) portion of the 64 bytes that it stores.

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.
STBLOCKF

Programming Note

STBLOCKF is intended to be a processor-specific instruction (see the warning at the top of page 312). 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.
- 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 STBLOCKF forces the data to be written to memory and invalidates copies in all caches present.
- Any other restrictions on the behavior of STBLOCKF, as described in implementation-specific documentation.

Exceptions. An illegal_instruction exception occurs if the source floating-point registers are not aligned on an eight-register boundary.

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

If the least significant 6 bits of the memory address are not all zero, a mem_address_not_aligned exception occurs.

In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0 (ASIs 1616, 1716, 1E16, and 1F16), 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
mem_address_not_aligned
privileged_action
VA_watchpoint (impl. dep. #411-S10)
STBLOCKF

See Also    LDBLOCKF on page 232
8.89 Store Floating-Point

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>rd</th>
<th>Operation</th>
<th>Assembly Language</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STF</td>
<td>10 0100</td>
<td>0–31</td>
<td>Store Floating-Point register</td>
<td>\textit{st} \texttt{freg} {address}</td>
<td>A1</td>
</tr>
<tr>
<td>STDF</td>
<td>10 0111</td>
<td>†</td>
<td>Store Double Floating-Point register</td>
<td>\texttt{std} \texttt{freg} {address}</td>
<td>A1</td>
</tr>
<tr>
<td>STQF</td>
<td>10 0110</td>
<td>†</td>
<td>Store Quad Floating-Point register</td>
<td>\texttt{stq} \texttt{freg} {address}</td>
<td>C3</td>
</tr>
<tr>
<td>STXFSR</td>
<td>10 0101</td>
<td>1</td>
<td>Store Floating-Point State register</td>
<td>\texttt{stx} %\texttt{fsr}, {address}</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>10 0101</td>
<td>2–31</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† Encoded floating-point register value, as described on page 51.

**Description**

The store single floating-point instruction (STF) copies the contents of the 32-bit floating-point register $F_{S}[rd]$ into memory.

The store double floating-point instruction (STDF) copies the contents of 64-bit floating-point register $F_{D}[rd]$ into a word-aligned doubleword in memory. The unit of atomicity for STDF is 4 bytes (one word).

The store quad floating-point instruction (STQF) copies the contents of 128-bit floating-point register $F_{Q}[rd]$ into a word-aligned quadword in memory. The unit of atomicity for STQF is 4 bytes (one word).

The store floating-point state register instruction (STXFSR) waits for any currently executing $F$Pop instructions to complete, and then it writes all 64 bits of the FSR into memory.

STXFSR zeroes FSR.$\texttt{ftt}$ after writing the FSR to memory.

**Implementation Note**

FSR.$\texttt{ftt}$ should not be zeroed by STXFSR until it is known that the store will not cause a precise trap.

These instruction access memory using the implicit ASI (see page 104). The effective address for these instructions is “$R[rs1] + R[rs2]$” if $i = 0$, or “$R[rs1] + \text{sign\_ext}(\text{simm13})$” if $i = 1$.

**Exceptions.** An attempt to execute a STF, STDF, or STXFSR instruction when $i = 0$ and instruction bits 12:5 are nonzero causes an \texttt{illegal\_instruction} exception.
STF / STDF / STQF / STXFSR

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STF, STDF, or STXFSR instruction causes an fp_disabled exception.

STF causes a mem_address_not_aligned exception if the effective memory address is not word-aligned. STXFSR causes a mem_address_not_aligned exception if the address is not doubleword-aligned.

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)).

An attempt to execute an STQF instruction when rd{1} ≠ 0 causes an fp_exception_other (FSR.ftt = invalid_fp_register) exception.

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.

Implementation Note
Since UltraSPARC Architecture 2005 processors do not implement in hardware instructions (including STQF) that refer to quad-precision floating-point registers, the STQF_mem_address_not_aligned and fp_exception_other (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an illegal_instruction exception and subsequent trap.

Exceptions
illegal_instruction
fp_disabled
STDF_mem_address_not_aligned
STQF_mem_address_not_aligned (not used in UltraSPARC Architecture 2005)
mem_address_not_aligned
fp_exception_other (FSR.ftt = invalid_fp_register (STQF only))
VA_watchpoint
STF / STDF / STQF / STXFSR

See Also
- Load Floating-Point on page 236
- Block Store on page 312
- Store Floating-Point into Alternate Space on page 319
- Store (Lower) Floating-Point Status Register on page 323
- Store Short Floating-Point on page 328
- Store Partial Floating-Point on page 325
8.90 Store Floating-Point into Alternate Space

<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>STFA&lt;sup&gt;P&lt;/sup&gt;&lt;sub&gt;ASI&lt;/sub&gt;</td>
<td>11 0100</td>
<td>0–31</td>
<td>Store Floating-Point Register to Alternate Space</td>
<td>sta &lt;sub&gt;Fr&lt;sub&gt;rd&lt;/sub&gt;&lt;/sub&gt; [reg&lt;sub&gt;addr&lt;/sub&gt;] imm&lt;sub&gt;asi&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>STDFAPASI</td>
<td>11 0111</td>
<td>Store Double Floating-Point Register to Alternate Space</td>
<td>stda &lt;sub&gt;Fr&lt;sub&gt;rd&lt;/sub&gt;&lt;/sub&gt; [reg&lt;sub&gt;addr&lt;/sub&gt;] imm&lt;sub&gt;asi&lt;/sub&gt;</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>STQFAPASI</td>
<td>11 0110</td>
<td>Store Quad Floating-Point Register to Alternate Space</td>
<td>stqa &lt;sub&gt;Fr&lt;sub&gt;rd&lt;/sub&gt;&lt;/sub&gt; [reg&lt;sub&gt;addr&lt;/sub&gt;] imm&lt;sub&gt;asi&lt;/sub&gt;</td>
<td>C3</td>
<td></td>
</tr>
</tbody>
</table>

† Encoded floating-point register value, as described on page 51.

Description

The store single floating-point into alternate space instruction (STFA) copies the contents of the 32-bit floating-point register Fr<sub>rd</sub> into memory.

The store double floating-point into alternate space instruction (STDFAPASI) copies the contents of 64-bit floating-point register Fr<sub>rd</sub> into a word-aligned doubleword in memory. The unit of atomicity for STDFAPASI is 4 bytes (one word).

The store quad floating-point into alternate space instruction (STQFAPASI) copies the contents of 128-bit floating-point register Fr<sub>rd</sub> into a word-aligned quadword in memory. The unit of atomicity for STQFAPASI is 4 bytes (one word).

Store floating-point into alternate space instructions contain the address space identifier (ASI) to be used for the load in the imm<sub>asi</sub> 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<sub>ext</sub> (simm13)” if i = 1.

Some compilers issued sequences of single-precision stores for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned stores is expected to be fast, compilers should issue sets of single-precision stores only when they can determine that double- or quadword operands are not properly aligned.

Exceptions. STFA causes a <code>mem_address_not_aligned</code> exception if the effective memory address is not word-aligned.
STFA / STDFA / STQFA

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 a \texttt{STDF\_mem\_address\_not\_aligned} exception. In this case, trap handler software must emulate the STDFA instruction and return (impl. dep. #110-V9-Cs10(b)).

STQFA requires only word alignment in memory. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an STQFA instruction may cause an \texttt{STQF\_mem\_address\_not\_aligned} exception. In this case, the trap handler software must emulate the STQFA instruction and return (impl. dep. #112-V9-Cs10(b)).

\textbf{Implementation Note} | STDFA shares an opcode with the STBLOCKF, STPARTIALF, and STSHORTF instructions; it is distinguished by the ASI used.

An attempt to execute an STQFA instruction when \(rd[1] \neq 0\) causes an \texttt{fp\_exception\_other} (\(FSR.\texttt{ftt} = \text{invalid}\_\text{fp}\_\text{register}\)) exception.

\textbf{Implementation Note} | Since UltraSPARC Architecture 2005 processors do not implement in hardware instructions (including STQFA) that refer to quad-precision floating-point registers, the \texttt{STQF\_mem\_address\_not\_aligned} and \texttt{fp\_exception\_other} (with \(FSR.\texttt{ftt} = \text{invalid}\_\text{fp}\_\text{register}\)) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an \texttt{illegal\_instruction} exception and subsequent trap.

In nonprivileged mode (\texttt{PSTATE.priv} = 0), if bit 7 of the ASI is 0, this instruction causes a \texttt{privileged\_action} exception. In privileged mode (\texttt{PSTATE.priv} = 1), if the ASI is in the range \(30\text{16} \text{ to } 7\text{F16}\), this instruction causes a \texttt{privileged\_action} exception.

STFA and STQFA can be used with any of the following ASIs, subject to the privilege mode rules described for the \texttt{privileged\_action} exception above. Use of any other ASI with these instructions causes a \texttt{data\_access\_exception} exception.

\begin{tabular}{ll}
\textbf{ASI\_NUCLEUS} & ASI\_NUCLEUS\_LITTLE \\
\textbf{ASI\_AS\_IF\_USER\_PRIMARY} & ASI\_AS\_IF\_USER\_PRIMARY\_LITTLE \\
\textbf{ASI\_AS\_IF\_USER\_SECONDARY} & ASI\_AS\_IF\_USER\_SECONDARY\_LITTLE \\
\textbf{ASI\_REAL} & ASI\_REAL\_LITTLE \\
\textbf{ASI\_REAL\_IO} & ASI\_REAL\_IO\_LITTLE \\
\textbf{ASI\_PRIMARY} & ASI\_PRIMARY\_LITTLE \\
\textbf{ASI\_SECONDARY} & ASI\_SECONDARY\_LITTLE \\
\end{tabular}

320 UltraSPARC Architecture 2005 • Draft D0.8.7, 27 Mar 2006
STFA / STDFA / STQFA

STDFA can be used with any of the following ASIs, subject to the privilege mode rules described for the `privileged_action` exception above. Use of any other ASI with the STDFA instruction causes a `data_access_exception` exception.

### ASIs valid for STDFA

<table>
<thead>
<tr>
<th>ASI / ASI LITTLE</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_AS_IF_USER_PRIMARY †</td>
</tr>
<tr>
<td>ASI_BLOCK_AS_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_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>
<tr>
<td>ASI_PST32_SECONDARY *</td>
</tr>
</tbody>
</table>

† 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 312.

‡ 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 328.

* 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 325.

### Exceptions

- `illegal_instruction`
- `fp_disabled`
- `STDF_mem_address_not_aligned`
- `STQF_mem_address_not_aligned` (STQFA only) (not used in UA-2005)
- `mem_address_not_aligned`
- `fp_exception_other` (FSR.ftt = invalid_fp_register (STQFA only))
See Also

Load Floating-Point from Alternate Space on page 239
Block Store on page 312
Store Floating-Point on page 316
Store Short Floating-Point on page 328
Store Partial Floating-Point on page 325
8.91 Store (Lower) Floating-Point Status Register

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 $fsrc, [address]</td>
<td>D2</td>
</tr>
</tbody>
</table>

<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>11</td>
<td>25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></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 the FSR into memory. STFSR zeroes FSR.ftt after writing the FSR to memory.

**V9 Compatibility Note**

FSR.ftt should not be zeroed until it is known that the store will not cause a precise trap.

STFSR accesses memory using the implicit ASI (see page 104). The effective address for this instruction is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + sign_ext (simm13)” if i = 1.

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

If the floating-point unit is not enabled (FPRSfef = 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_not_aligned exception if the effective memory address is not word-aligned.

**V9 Compatibility Note**

Although STFSR is deprecated, UltraSPARC Architecture implementations continue to support it for compatibility with existing SPARC V8 software. The STFSR instruction is defined to store only 32 bits of the FSR into memory, while STXFSR allows SPARC V9 software to store all 64 bits of the FSR.
STFSR (Deprecated)

Exceptions

illegal_instruction
fp_disabled
mem_address_not_aligned
VA_watchpoint

See Also

Store Floating-Point on page 316
8.92 Store Partial Floating-Point

<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>STPARTIALF</td>
<td>C016</td>
<td>Eight 8-bit conditional stores to primary address space</td>
<td>stda freg, r^rs2^, [reg^rs1^] #ASI_PST8_P</td>
<td>C3</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C116</td>
<td>Eight 8-bit conditional stores to secondary address space</td>
<td>stda freg, r^rs2^, [reg^rs1^] #ASI_PST8_S</td>
<td>C3</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, r^rs2^, [reg^rs1^] #ASI_PST8_PL</td>
<td>C3</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, r^rs2^, [reg^rs1^] #ASI_PST8_SL</td>
<td>C3</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C216</td>
<td>Four 16-bit conditional stores to primary address space</td>
<td>stda freg, r^rs2^, [reg^rs1^] #ASI_PST16_P</td>
<td>C3</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C316</td>
<td>Four 16-bit conditional stores to secondary address space</td>
<td>stda freg, r^rs2^, [reg^rs1^] #ASI_PST16_S</td>
<td>C3</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, r^rs2^, [reg^rs1^] #ASI_PST16_PL</td>
<td>C3</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, r^rs2^, [reg^rs1^] #ASI_PST16_SL</td>
<td>C3</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C416</td>
<td>Two 32-bit conditional stores to primary address space</td>
<td>stda freg, r^rs2^, [reg^rs1^] #ASI_PST32_P</td>
<td>C3</td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>C516</td>
<td>Two 32-bit conditional stores to secondary address space</td>
<td>stda freg, r^rs2^, [reg^rs1^] #ASI_PST32_S</td>
<td>C3</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, r^rs2^, [reg^rs1^] #ASI_PST32_PL</td>
<td>C3</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, r^rs2^, [reg^rs1^] #ASI_PST32_SL</td>
<td>C3</td>
</tr>
</tbody>
</table>

† The original assembly language syntax for a Partial Store instruction ("stda freg, [reg^rs1^] r^rs2^, imm_asi") has been deprecated because of inconsistency with the rest of the SPARC assembly language. Over time, assemblers will support the new syntax for this instruction. In the meantime, some existing assemblers may only recognize the original syntax.

Description

The partial store instructions are selected by one of the partial store ASIs with the STDFA instruction.
**STPARTIALF**

Two 32-bit, four 16-bit, or eight 8-bit values from the 64-bit floating-point register \( F_D [\text{rd}] \) are conditionally stored at the address specified by \( R[\text{rs1}] \), using the mask specified in \( R[\text{rs2}] \). STPARTIALF has the effect of merging selected data from its source register, \( F_D [\text{rd}] \), into the existing data at the corresponding destination locations.

The mask value in \( R[\text{rs2}] \) has the same format as the result specified by the pixel compare instructions (see **SIMD Signed Compare** on page 166). The most significant bit of the mask (not of the entire register) corresponds to the most significant part of \( F_D [\text{rd}] \). 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.

![Mask Format for Partial Store](image)

**FIGURE 8-29** Mask Format for Partial Store

In an UltraSPARC Architecture 2005 implementation, these instructions are not implemented in hardware, cause a **data_access_exception** exception, and are emulated in software.

**Exceptions.** An attempt to execute a STPARTIALF instruction when \( i = 1 \) causes an **illegal_instruction** exception.
STPARTIALF

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STPARTIALF instruction causes an fp_disabled exception.

STPARTIALF causes a mem_address_not_aligned exception if the effective memory address is not word-aligned.

STPARTIALF requires only word alignment in memory for eight byte stores. If the effective address is word-aligned but not doubleword-aligned, it generates an STDF_mem_address_not_aligned exception. In this case, the trap handler software shall emulate the STDFA instruction and return.

IMPL. DEP. #249-U3-Cs10: For an STPARTIAL instruction, the following aspects of data watchpoints are implementation dependent: (a) whether data watchpoint logic examines the byte store mask in R[rs2] or it conservatively behaves as if every Partial Store always stores all 8 bytes, and (b) whether data watchpoint logic examines individual bits in the Virtual (Physical) Data Watchpoint Mask in the LSU Control register DCUCR to determine which bytes are being watched or (when the Watchpoint Mask is nonzero) it conservatively behaves as if all 8 bytes are being watched.

ASIs C016–C516 and C816–CD16 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 241.

Implementation Note STPARTIALF shares an opcode with the STBLOCKF, STDFA, and STSHORTF instructions; it is distinguished by the ASI used.

Exceptions

illegal_instruction
fp_disabled
data_access_exception (not implemented in hardware in UA-2005)
8.93 Store Short Floating-Point

<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>STSHORTF</td>
<td>D0₁₆</td>
<td>8-bit store to primary address space</td>
<td>stda $regrdr$ [$regaddr$] #ASI_FL₈_P</td>
<td>C3</td>
</tr>
<tr>
<td>STSHORTF</td>
<td>D1₁₆</td>
<td>8-bit store to secondary address space</td>
<td>stda $regrdr$ [$regaddr$] #ASI_FL₈_S</td>
<td>C3</td>
</tr>
<tr>
<td>STSHORTF</td>
<td>D8₁₆</td>
<td>8-bit store to primary address space, little-endian</td>
<td>stda $regrdr$ [$regaddr$] #ASI_FL₈_PL</td>
<td>C3</td>
</tr>
<tr>
<td>STSHORTF</td>
<td>D9₁₆</td>
<td>8-bit store to secondary address space, little-endian</td>
<td>stda $regrdr$ [$regaddr$] #ASI_FL₈_SL</td>
<td>C3</td>
</tr>
<tr>
<td>STSHORTF</td>
<td>D2₁₆</td>
<td>16-bit store to primary address space</td>
<td>stda $regrdr$ [$regaddr$] #ASI_FL₁₆_P</td>
<td>C3</td>
</tr>
<tr>
<td>STSHORTF</td>
<td>D3₁₆</td>
<td>16-bit store to secondary address space</td>
<td>stda $regrdr$ [$regaddr$] #ASI_FL₁₆_S</td>
<td>C3</td>
</tr>
<tr>
<td>STSHORTF</td>
<td>DA₁₆</td>
<td>16-bit store to primary address space, little-endian</td>
<td>stda $regrdr$ [$regaddr$] #ASI_FL₁₆_PL</td>
<td>C3</td>
</tr>
<tr>
<td>STSHORTF</td>
<td>DB₁₆</td>
<td>16-bit store to secondary address space, little-endian</td>
<td>stda $regrdr$ [$regaddr$] #ASI_FL₁₆_SL</td>
<td>C3</td>
</tr>
</tbody>
</table>

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

**Implementation Note**

STSHORTF shares an opcode with the STBLOCKF, STDFA, and STPARTIALF instructions; it is distinguished by the ASI used.

In an UltraSPARC Architecture 2005 implementation, these instructions are not implemented in hardware, cause an *data_access_exception* exception, and are emulated in software.
STSHORTF

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 D0\text{16}, D1_{16}, D8_{16}, or D9_{16}) can be performed to an arbitrary memory address (no alignment requirement).

A 16-bit STSHORTF (using ASI D2_{16}, D3_{16}, DA_{16}, or DB_{16}) to an address that is not halfword-aligned (an odd address) causes a mem_address_not_aligned exception.

Exceptions

<table>
<thead>
<tr>
<th>VA_watchpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_access_exception</td>
</tr>
</tbody>
</table>
8.94 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>STTWU</td>
<td>00 0111</td>
<td>Store Integer Twin Word</td>
<td>sttw reg_r [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 104). The effective address for this instruction is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + sign_ext (simm13)” if i = 1.

A successful store twin word instruction operates atomically.

IMPL. DEP. #108-V9a: It is implementation dependent whether STTW is implemented in hardware. If not, an attempt to execute it will cause an unimplemented_STTW exception. (STTW is implemented in hardware in all UltraSPARC Architecture 2005 implementations.)

An attempt to execute an STTW instruction when either of the following conditions exist causes an illegal_instruction exception:

- destination register number rd is an odd number (is misaligned)
- i = 0 and instruction bits 12:5 are nonzero
STTW (Deprecated)

STTW causes a mem_address_not_aligned exception if the effective address is not doubleword-aligned.

With respect to little-endian memory, an STTW instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.

Programming Notes

STTW is provided for compatibility with SPARC V8. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties. Therefore, software should avoid using STTW.

If STTW is emulated in software, STX instruction should be used for the memory access in the emulation code to preserve atomicity.

Exceptions

unimplemented_STTW
illegal_instruction
mem_address_not_aligned
VA_watchpoint

See Also

STW/STX on page 307
STTWA on page 332
8.95 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.

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 “\(R[rs1] + \text{sign ext } (\text{simm13})\)” if i = 1.

A successful store twin word instruction operates atomically.

With respect to little-endian memory, an STTWA instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.
STTWA (Deprecated)

IMPL. DEP. #108-V9b: It is implementation dependent whether STTWA is implemented in hardware. If not, an attempt to execute it will cause an unimplemented_STTW exception. (STTWA is implemented in hardware in all UltraSPARC Architecture 2005 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 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 data_access_exception 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 387) 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 data_access_exception 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
STTW (Deprecated)

privileged_action
VA_watchpoint

See Also
STWA/STXA on page 308
STTW on page 330
8.96 Subtract

These instructions compute "R[rs1] – R[rs2]" if \(i = 0\), or "R[rs1] – \text{sign\_ext}(\text{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] – \text{sign\_ext}(\text{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.

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
8.97 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.

<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>SWAPD</td>
<td>00 1111</td>
<td>Swap Register with Memory</td>
<td>swap [address], rs[^rd]</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 104). The effective address for these instructions is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + sign_ext(simm13)” if i = 1.

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

If the effective address is not word-aligned, an attempt to execute a SWAP instruction causes a mem_address_not_aligned exception.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

**Exceptions**

- illegal_instruction
- mem_address_not_aligned
- VA_watchpoint
SWAPA (Deprecated)

8.98 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.

### 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 \( \text{imm}_\text{asi} \) 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 this instruction is \( \text{``R[rs1] + R[rs2]''} \) if \( i = 0 \), or \( \text{``R[rs1] + sign_ext(simm13)''} \) if \( i = 1 \).

This instruction causes a \texttt{mem\_address\_not\_aligned} exception if the effective address is not word-aligned. It causes a \texttt{privileged\_action} exception if \texttt{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 \texttt{mem\_address\_not\_aligned} exception.
SWAPA ( Deprecated )

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 data_access_exception exception.

### Exceptions
- mem_address_not_aligned
- privileged_action
- VA_watchpoint
- data_access_exception

### ASIs valid for SWAPA

<table>
<thead>
<tr>
<th>ASI</th>
<th>ASI_valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_NUCLEUS</td>
<td>ASI_NUCLEUS_LITTLE</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_PRIMARY</td>
<td>ASI_AS_IF_USER_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_AS_IF_USER_SECONDARY</td>
<td>ASI_AS_IF_USER_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_PRIMARY</td>
<td>ASI_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>ASI_SECONDARY</td>
<td>ASI_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>ASI_REAL</td>
<td>ASI_REAL_LITTLE</td>
</tr>
</tbody>
</table>
8.99 Tagged Add

**TADDcc**

This instruction computes a sum that is “R[rs1] + R[rs2]” if \( i = 0 \), or “R[rs1] + sign_ext(simm13)” if \( i = 1 \).

TADDcc modifies the integer condition codes (icc and xcc).

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the addition generates 32-bit arithmetic overflow (that is, both operands have the same value in bit 31 and bit 31 of the sum is different).

If a TADDcc causes a tag overflow, the 32-bit overflow bit (CCR.icc.v) is set to 1; if TADDcc does not cause a tag overflow, CCR.icc.v is set to 0.

In either case, the remaining integer condition codes (both the other CCR.icc bits and all the CCR.xcc bits) are also updated as they would be for a normal ADD instruction. In particular, the setting of the CCR.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). CCR.xcc.v is set based on the 64-bit arithmetic overflow condition, like a normal 64-bit add.

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

**Exceptions**

illegal_instruction

**See Also**

TADDccTVD on page 340
TSUBcc on page 345
TADDccTV (Deprecated)

8.100 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 \( R[rs1] + \text{sign_ext}(\text{simm13}) \) if \( i = 1 \).

TADDccTV modifies the integer condition codes if it does not trap.

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

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the addition generates 32-bit arithmetic overflow (that is, both operands have the same value in bit 31 and bit 31 of the sum is different).

If TADDccTV causes a tag overflow, a tag_overflow exception is generated and \( R[rd] \) and the integer condition codes remain unchanged. If a TADDccTV does not cause a tag overflow, the sum is written into \( R[rd] \) and the integer condition codes are updated. CCR.icc.v is set to 0 to indicate no 32-bit overflow.

In either case, the remaining integer condition codes (both the other CCR.icc bits and all the CCR.xcc bits) are also updated as they would be for a normal ADD instruction. In particular, the setting of the CCR.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). CCR.xcc.v is set only on the basis of the normal 64-bit arithmetic overflow condition, like a normal 64-bit add.
TADDccTV (Deprecated)

**SPARC V8 Compatibility**

**Note**

TADDccTV traps based on the 32-bit overflow condition, just as in the SPARC V8 architecture. Although the tagged add instructions set the 64-bit condition codes $CCR.xcc$, there is no form of the instruction that traps on the 64-bit overflow condition.

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>illegal_instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tag_overflow</td>
</tr>
</tbody>
</table>

**See Also**

TADDcc on page 339
TSUBccTV on page 346
### 8.101 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 1010</td>
<td>1000</td>
<td>Trap Always</td>
<td>ta (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TN</td>
<td>11 1010</td>
<td>0000</td>
<td>Trap Never</td>
<td>0 (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TNE</td>
<td>11 1010</td>
<td>1001</td>
<td>Trap on Not Equal</td>
<td>(not\ Z) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TE</td>
<td>11 1010</td>
<td>0001</td>
<td>Trap on Equal</td>
<td>Z (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TG</td>
<td>11 1010</td>
<td>1010</td>
<td>Trap on Greater</td>
<td>(not\ (Z\ or\ (N\ xor\ V))) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TLE</td>
<td>11 1010</td>
<td>0010</td>
<td>Trap on Less or Equal</td>
<td>(Z\ or\ (N\ xor\ V)) (t_{le}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TGE</td>
<td>11 1010</td>
<td>1011</td>
<td>Trap on Greater or Equal</td>
<td>(not\ (N\ xor\ V)) (t_{ge}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TL</td>
<td>11 1010</td>
<td>0011</td>
<td>Trap on Less</td>
<td>(N\ xor\ V) (t_{l}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TGU</td>
<td>11 1010</td>
<td>1100</td>
<td>Trap on Greater, Unsigned</td>
<td>(not\ (C\ or\ Z)) (t_{gu}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TLEU</td>
<td>11 1010</td>
<td>0100</td>
<td>Trap on Less or Equal, Unsigned</td>
<td>(C\ or\ Z) (t_{leu}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TCC</td>
<td>11 1010</td>
<td>1101</td>
<td>Trap on Carry Clear</td>
<td>(not\ (C)) (t_{cc}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TCS</td>
<td>11 1010</td>
<td>0101</td>
<td>Trap on Carry Set</td>
<td>(C) (t_{cs}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TPOS</td>
<td>11 1010</td>
<td>1110</td>
<td>Trap on Positive or zero</td>
<td>(not\ (N)) (t_{pos}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TNEG</td>
<td>11 1010</td>
<td>0110</td>
<td>Trap on Negative</td>
<td>(N) (t_{neg}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>VC</td>
<td>11 1010</td>
<td>1111</td>
<td>Trap on Overflow Clear</td>
<td>(not\ (V)) (t_{vc}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
<tr>
<td>TVS</td>
<td>11 1010</td>
<td>0111</td>
<td>Trap on Overflow Set</td>
<td>(V) (t_{vs}) (i_{or_x_cc},\ software_trap_number)</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonym: \(t_{nz}\) ‡ synonym: \(t_{z}\) ◊ synonym: \(t_{geu}\) ∇ synonym: \(t_{lu}\)

---

```
<table>
<thead>
<tr>
<th>op3</th>
<th>cond</th>
<th>rs1</th>
<th>i=0</th>
<th>cc</th>
<th>cc0</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 1010</td>
<td>1000</td>
<td>1</td>
<td>ta</td>
<td>(i_{or_x_cc},\ software_trap_number)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>0000</td>
<td>0</td>
<td>tn</td>
<td>(i_{or_x_cc},\ software_trap_number)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>1001</td>
<td>(not\ Z)</td>
<td>tne</td>
<td>(i_{or_x_cc},\ software_trap_number)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>0001</td>
<td>Z</td>
<td>te</td>
<td>(i_{or_x_cc},\ software_trap_number)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>1010</td>
<td>(not\ (Z\ or\ (N\ xor\ V)))</td>
<td>tg</td>
<td>(i_{or_x_cc},\ software_trap_number)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>0010</td>
<td>(Z\ or\ (N\ xor\ V)) (t_{le})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>1011</td>
<td>(not\ (N\ xor\ V)) (t_{ge})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>0011</td>
<td>(N\ xor\ V) (t_{l})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>1100</td>
<td>(not\ (C\ or\ Z)) (t_{gu})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>0100</td>
<td>(C\ or\ Z) (t_{leu})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>1101</td>
<td>(not\ (C)) (t_{cc})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>0101</td>
<td>(C) (t_{cs})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>1110</td>
<td>(not\ (N)) (t_{pos})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>0110</td>
<td>(N) (t_{neg})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>1111</td>
<td>(not\ (V)) (t_{vc})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1010</td>
<td>0111</td>
<td>(V) (t_{vs})</td>
<td>i_{or_x_cc}, software_trap_number</td>
<td>A1</td>
<td></td>
<td></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 and should be supplied as zeroes by software.

Generally, values of \( 0 \leq \text{SWTN} \leq 127 \) are used to trap to privileged-mode software and values of \( 128 \leq \text{SWTN} \leq 255 \) 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 \leq \text{SWTN} \leq 127 )</th>
<th>( 128 \leq \text{SWTN} \leq 255 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonprivileged (PSTATE.priv = 0)</td>
<td>trap_instruction exception</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>(to privileged mode)</td>
<td>(not possible)</td>
</tr>
<tr>
<td></td>
<td>(256 \leq TT \leq 383)</td>
<td></td>
</tr>
<tr>
<td>Privileged (PSTATE.priv = 1)</td>
<td>trap_instruction exception</td>
<td>htrap_instruction exception</td>
</tr>
<tr>
<td></td>
<td>(to privileged mode)</td>
<td>(to hyperprivileged mode)</td>
</tr>
<tr>
<td></td>
<td>(256 \leq TT \leq 383)</td>
<td>(384 \leq TT \leq 511)</td>
</tr>
</tbody>
</table>
Tcc

**Programming Note**  Tcc can be used to implement breakpointing, tracing, and calls to privileged and hyperprivileged software. It can also be used for runtime checks, such as for out-of-range array indexes and integer overflow.

**Exceptions.** An attempt to execute a Tcc instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- Instruction bit 29 is nonzero
- \( i = 0 \) and instruction bits 12:5 are nonzero
- \( i = 1 \) and instruction bits 10:8 are nonzero
- \( \text{cc}0 = 1 \)

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 429.

**Exceptions**

- *illegal_instruction*
- *trap_instruction* \((0 \leq \text{SWTN} \leq 127)\)
- *htrap_instruction* \((128 \leq \text{SWTN} \leq 255)\)
8.102 Tagged Subtract

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 339
TSUBccTVD on page 346
8.103 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] – sign_ext (simm13)” if i = 1.

TSUBccTV modifies the integer condition codes (icc and xcc) if it does not trap.

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the subtraction generates 32-bit arithmetic overflow; that is, the operands have different values in bit 31 (the 32-bit sign bit) and the sign of the 32-bit difference in bit 31 differs from bit 31 of R[rs1].

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

If TSUBccTV causes a tag overflow, then a tag_overflow exception is generated and R[rd] and the integer condition codes remain unchanged. If a TSUBcctV does not cause a tag overflow condition, the difference is written into R[rd] and the integer condition codes are updated. CCR.icc.v is set to 0 to indicate no 32-bit overflow.

In either case, the remaining integer condition codes (both the other CCR.icc bits and all the CCR.xcc bits) are also updated as they would be for a normal subtract instruction. In particular, the setting of the CCR.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). CCR.xcc.v is set only on the basis of the normal 64-bit arithmetic overflow condition, like a normal 64-bit subtract.
TSUBccTV (Deprecated)

<table>
<thead>
<tr>
<th>SPARC V8 Compatibility Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSUBccTV traps based on the 32-bit overflow condition, just as in the SPARC V8 architecture. Although the tagged add instructions set the 64-bit condition codes CCR.xcc, there is no form of the instruction that traps on the 64-bit overflow condition.</td>
</tr>
</tbody>
</table>

Exceptions

illegal_instruction
tag_overflow

See Also

TADDccTV^D on page 340
TSUBcc on page 345
8.104  Divide (64-bit ÷ 32-bit)

The UDIV, UDIVcc, SDIV, and SDIVcc instructions are deprecated and should not be used in new software. The UDIVX and SDIVX instructions 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>UDIVD</td>
<td>00 1110</td>
<td>Unsigned Integer Divide</td>
<td>udiv reg_{rs1}, reg_or_imm, reg_{rd}</td>
<td>C2</td>
</tr>
<tr>
<td>SDIVD</td>
<td>00 1111</td>
<td>Signed Integer Divide</td>
<td>sdiv reg_{rs1}, reg_or_imm, reg_{rd}</td>
<td>C2</td>
</tr>
<tr>
<td>UDIVccD</td>
<td>01 1110</td>
<td>Unsigned Integer Divide and modify cc’s</td>
<td>udivcc reg_{rs1}, reg_or_imm, reg_{rd}</td>
<td>C2</td>
</tr>
<tr>
<td>SDIVccD</td>
<td>01 1111</td>
<td>Signed Integer Divide and modify cc’s</td>
<td>sdivcc reg_{rs1}, reg_or_imm, reg_{rd}</td>
<td>C2</td>
</tr>
</tbody>
</table>

**Description**

The divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If \( i = 0 \), they compute \((Y :: R[rs1][31:0]) ÷ R[rs2][31:0]\)”. Otherwise (that is, if \( i = 1 \)), the divide instructions compute \((Y :: R[rs1][31:0]) ÷ (\text{sign}_\text{ext}(\text{simm13})[31:0])\)”. 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 \((Y :: R[rs1][31:0])\) and an unsigned integer word divisor \(R[rs2][31:0]\) or \((\text{sign}_\text{ext}(\text{simm13})[31:0])\) and computes an unsigned integer word quotient \((R[rd])\). 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).
UDIV / UDIVcc / SDIV / SDIVcc (Deprecated)

The result of an unsigned divide instruction can overflow the less significant 32 bits of the destination register R[rd] under certain conditions. When overflow occurs, the largest appropriate unsigned integer is returned as the quotient in R[rd]. The condition under which overflow occurs and the value returned in R[rd] under this condition are specified in TABLE 8-14.

TABLE 8-14 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 (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 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>UDIVcc</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 8-14)</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>

Signed Divide
Signed divide (SDIV, SDIVcc) assumes a signed integer doubleword dividend (Y :: lower 32 bits of R[rs1]) and a signed integer word divisor (lower 32 bits of R[rs2] or lower 32 bits of sign_ext(simm13)) and computes a signed integer word quotient (R[rd]).

Signed division rounds an inexact quotient toward zero. For example, −7 ÷ 4 equals the rational quotient of −1.75, which rounds to −1 (not −2) when rounding toward zero.

The result of a signed divide can overflow the low-order 32 bits of the destination register R[rd] under certain conditions. When overflow occurs, the largest appropriate signed integer is returned as the quotient in R[rd]. The conditions under which overflow occurs and the value returned in R[rd] under those conditions are specified in TABLE 8-15.
When no overflow occurs, the 32-bit result is sign-extended to 64 bits and written into register \( R[rd] \).

SDIV does not affect the condition code bits. SDIVcc writes the integer condition code bits as shown in the following table. Note that negative (N) and zero (Z) are set according to the value of \( R[rd] \) after it has been set to reflect overflow, if any.

<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>

An attempt to execute a UDIV, UDIVcc, 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
UMUL / UMULcc / SMUL / SMULcc ( Deprecated )

8.105 Multiply (32-bit)

The UMUL, UMULcc, SMUL, and SMULcc 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>00 1010</td>
<td>Unsigned Integer Multiply</td>
<td>umul reg1, reg_or_imm, regd</td>
<td>C2</td>
</tr>
<tr>
<td>SMULD</td>
<td>00 1011</td>
<td>Signed Integer Multiply</td>
<td>smul reg1, reg_or_imm, regd</td>
<td>C2</td>
</tr>
<tr>
<td>UMULccD</td>
<td>01 1010</td>
<td>Unsigned Integer Multiply and modify cc's</td>
<td>umulcc reg1, reg_or_imm, regd</td>
<td>C2</td>
</tr>
<tr>
<td>SMULccD</td>
<td>01 1011</td>
<td>Signed Integer Multiply and modify cc's</td>
<td>smulcc reg1, reg_or_imm, regd</td>
<td>C2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i=0</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29</td>
<td>25 24</td>
<td>19 18</td>
<td>14 13 12</td>
<td>5 4</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Description

The multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute “R[rs1][31:0] × R[rs2][31:0]” if i = 0, or “R[rs1][31:0] × sign_ext(simm13)[31:0]” if i = 1. They write the 32 most significant bits of the product into the Y register and all 64 bits of the product into R[rd].

Unsigned multiply instructions (UMUL, UMULcc) operate on unsigned integer word operands and compute an unsigned integer doubleword product. Signed multiply instructions (SMUL, SMULcc) operate on signed integer word operands and compute a signed integer doubleword product.

UMUL and SMUL do not affect the condition code bits. UMULcc and SMULcc write the integer condition code bits, icc and xcc, as shown below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>UMULcc / 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>
An attempt to execute a UMUL, UMULcc, SMUL, or SMULcc instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an `illegal_instruction` exception.

**Exceptions**

`illegal_instruction`

**Programming Notes**

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

32-bit overflow after SMUL/SMULcc is indicated by \( Y \neq (R[rd] >> 31) \), where “\( >> \)” indicates 32-bit arithmetic right-shift.
## 8.106  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>WRY&lt;sup&gt;D&lt;/sup&gt;</td>
<td>0</td>
<td>Write Y register (deprecated)</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %y</td>
<td>C1</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 reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %ccr</td>
<td>A1</td>
</tr>
<tr>
<td>WRASI</td>
<td>3</td>
<td>Write ASI register</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, 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 reg&lt;sub&gt;rs1&lt;/sub&gt;, 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>WRPCR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>16</td>
<td>Write Performance Control register (PCR)</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %pcr</td>
<td>A1</td>
</tr>
<tr>
<td>WRPIC&lt;sup&gt;P&lt;/sup&gt;&lt;sub&gt;PIC&lt;/sub&gt;</td>
<td>17</td>
<td>Write Performance Instrumentation Counters (PIC)</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %pic</td>
<td>A1</td>
</tr>
<tr>
<td>—</td>
<td>18</td>
<td>Reserved (impl. dep. #8-V8-Cs20, #9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRGSR</td>
<td>19</td>
<td>Write General Status register (GSR)</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %gsr</td>
<td>A1</td>
</tr>
<tr>
<td>WRSOFTINT_SET&lt;sup&gt;P&lt;/sup&gt;</td>
<td>20</td>
<td>Set bits of per-virtual processor Soft Interrupt register</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %softint_set</td>
<td>N1</td>
</tr>
<tr>
<td>WRSOFTINT_CLR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>21</td>
<td>Clear bits of per-virtual processor Soft Interrupt register</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %softint_clr</td>
<td>N1</td>
</tr>
<tr>
<td>WRSOFTINT&lt;sup&gt;P&lt;/sup&gt;</td>
<td>22</td>
<td>Write per-virtual processor Soft Interrupt register</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %softint</td>
<td>N1</td>
</tr>
<tr>
<td>WRTICK_CMPRP&lt;sup&gt;P&lt;/sup&gt;</td>
<td>23</td>
<td>Write Tick Compare register</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %tick_cmpr</td>
<td>N1</td>
</tr>
<tr>
<td>—</td>
<td>24</td>
<td>used at higher privilege level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRSTICK_CMPRP&lt;sup&gt;P&lt;/sup&gt;</td>
<td>25</td>
<td>Write System Tick Compare register</td>
<td>wr reg&lt;sub&gt;rs1&lt;/sub&gt;, reg_or_imm, %sys_tick_cmpr</td>
<td>N1</td>
</tr>
<tr>
<td>—</td>
<td>26–27</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>28–31</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**WRasr**

<table>
<thead>
<tr>
<th>10</th>
<th>rd</th>
<th>op3 = 11 0000</th>
<th>rs1</th>
<th>i=0</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>31 30 29</th>
<th>25 24</th>
<th>19 18</th>
<th>14 13 12 5 4</th>
</tr>
</thead>
</table>

**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 \(R[rs1] \text{ xor sign_ext (simm13)}\).

**Note**

The operation is exclusive-or.

The WRasr instruction with \(rs1 = 0\) is a (deprecated) WRY instruction (which should not be used in new software). WRY is not a delayed-write instruction; the instruction immediately following a WRY observes the new value of the Y register.

WRCCCR, WRFPRS, and WRASI are not delayed-write instructions. The instruction immediately following a WRCCCR, WRFPRS, or WRASI observes the new value of the CCR, FPRS, or ASI register.

WRFPRS waits for any pending floating-point operations to complete before writing the FPRS register.

**IMPL. DEP. #48-V8-Cs20:** WRasr instructions with rd in the range 26–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRasr instruction with rd in the range 26–31, the following are implementation dependent:

- the interpretation of bits 18:0 in the instruction
- the operation(s) performed (for example, xor) to generate the value written to the ASR
- whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20), and
- whether an attempt to execute the instruction causes an illegal_instruction exception.

**Note**

See the section “Read/Write Ancillary State Registers (ASRs)” in *Extending the UltraSPARC Architecture*, contained in the separate volume *UltraSPARC Architecture Application Notes*, for a discussion of extending the SPARC V9 instruction set by means of read/write ASR instructions.
WRasr

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 67 for more detailed information regarding ASR registers.

**Exceptions.** An attempt to execute a WRasr instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- $i = 0$ and instruction bits 12:5 are nonzero
- $rd = 1, 4, 5, 7–14, 18,$ or 26–31
- $rd = 15$ and ($(rs1 ≠ 0)$ or ($i = 0$))

An attempt to execute a WRPCR (impl. dep. #250-U3-Cs10), WRSOFTINT_SET, WRSOFTINT_CLR, WRTICK_CMPR, or WRSTICK_CMPR instruction in nonprivileged mode ($PSTATE.priv = 0$) causes a *privileged_opcode* exception.

If the floating-point unit is not enabled ($FPRS.fef = 0$ or $PSTATE.pef = 0$) or if the FPU is not present, then an attempt to execute a WRGSR instruction causes an *fp_disabled* exception.

An attempt to execute a WRPIC instruction in nonprivileged mode ($PSTATE.priv = 0$) when $PCR.priv = 1$ causes a *privileged_action* exception.

**Exceptions**

- *illegal_instruction*
- *privileged_opcode*
- *fp_disabled*
- *privileged_action*

**See Also**

RDasr on page 285
WRPR on page 356
8.107 Write Privileged Register

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, TCK, 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 6-22 on page 95.

$MAXPGL$ 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 6-23 on page 97.

**Exceptions.** An attempt to execute a WRPR instruction in nonprivileged mode ($PSTATE_{priv} = 0$) causes a *privileged_opcode* exception.

An attempt to execute a WRPR instruction when any of the following conditions exist causes an *illegal_instruction* exception:
- $i = 0$ and instruction bits 12:5 are nonzero
- $rd = 4$
- $rd = 15$, or 17-31 (reserved for future versions of the architecture)
- $0 \leq rd \leq 3$ (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, *illegal_instruction* exception due to $0 \leq rd \leq 3$ and TL = 0 does not occur; the *privileged_opcode* exception occurs instead.

**See Also** RDPR on page 288
WRasr on page 353
## XOR / XNOR

### 8.108 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 0011</td>
<td>Exclusive or</td>
<td>xor rs1, reg_or_imm, reg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>XORcc</td>
<td>01 0011</td>
<td>Exclusive or and modify cc's</td>
<td>xorcc rs1, reg_or_imm, reg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>XNOR</td>
<td>00 0111</td>
<td>Exclusive nor</td>
<td>xnor rs1, reg_or_imm, reg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>XNORcc</td>
<td>01 0111</td>
<td>Exclusive nor and modify cc's</td>
<td>xnorcc rs1, reg_or_imm, reg_rd</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)

#### Programming Note
XNOR (and XNORcc) is identical to the xor_not (and set condition codes) xor_not_cc logical operation, respectively.

An attempt to execute an XOR, XORcc, XNOR, or XNORcc instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

#### Exceptions
illegal_instruction
IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2005

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 359.
- NaN Operand and Result Definitions on page 360.
- Trapped Underflow Definition (ufm = 1) on page 362.
- Untrapped Underflow Definition (ufm = 0) on page 362.
- Integer Overflow Definition on page 363.
- Floating-Point Nonstandard Mode on page 364.

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.

9.1 Traps Inhibiting Results

As described in Floating-Point State Register (FSR) on page 58 and elsewhere, when a floating-point trap occurs, the following conditions are true:

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

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

### Programming Note
A user-mode trap handler invoked for an IEEE_754_exception, whether as a direct result of a hardware fp_exception_ieee_754 trap or as an indirect result of privileged software handling of an fp_exception_other trap with FSR.ftt = unfinished_FPop or FSR.ftt = unimplemented_FPop, can rely on the following behavior:
- The address of the instruction that caused the exception will be available.
- The destination floating-point register(s) are unchanged from their state prior to that instruction’s execution.
- The floating-point condition codes (fcc0, fcc1, fcc2, and fcc3) are unchanged.
- The FSR.aexc field is unchanged.
- The FSR.cexc field contains exactly one bit set to 1, corresponding to the exception that caused the trap.
- The FSR.ftt, FSR.qne, and reserved fields of FSR are zero.

### 9.2 NaN Operand and Result Definitions

An untrapped floating-point result can be in a format that is either the same as, or different from, the format of the source operands. These two cases are described separately below.

#### 9.2.1 Untrapped Result in Different Format from Operands
- F<sdq>TO<sdq> or F<sd>MUL<dq> with a quiet NaN operand — No exception caused; result is a quiet NaN. The operand is transformed as follows:
**NaN transformation**: The most significant bits of the operand fraction are copied to the most significant bits of the result fraction. In conversion to a narrower format, excess low-order bits of the operand fraction are discarded (which is not considered a "rounding" operation). In conversion to a wider format, excess low-order bits of the result fraction are set to 0. The quiet bit (the most significant bit of the result fraction) is always set to 1, so the NaN transformation always produces a quiet NaN. The sign bit is copied from the operand to the result without modification.

- **F<sdq>TO<sdq> or F<sdq>MUL<dq> with a signalling NaN operand** — Invalid exception; result is the signalling NaN operand processed by the NaN transformation above to produce a quiet NaN.
- **FCMPE<sdq> with any NaN operand** — Invalid exception; the selected floating-point condition code is set to unordered.
- **FCMP<sdq> with any signalling NaN operand** — Invalid exception; the selected floating-point condition code is set to unordered.
- **FCMP<sdq> with any quiet NaN operand but no signalling NaN operand** — No exception; the selected floating-point condition code is set to unordered.

### 9.2.2 Untrapped Result in Same Format as Operands

- **No NaN operand** — For an invalid operation such as sqrt(–1.0) or 0.0 ÷ 0.0, the result is the quiet NaN with sign = zero, exponent = all 1’s, and fraction = all ones. The sign is zero to distinguish such results from storage initialized to all ones.
- **One operand, a quiet NaN** — No exception; result is the quiet NaN operand.
- **One operand, a signalling NaN** — Invalid exception; result is the signalling NaN with its quiet bit (most significant bit of fraction field) set to 1.
- **Two operands, both quiet NaNs** — No exception; result is the rs2 (second source) operand.
- **Two operands, both signalling NaNs** — Invalid exception; result is the rs2 operand with the quiet bit set to 1.
- **Two operands, only one is a signalling NaN** — Invalid exception; result is the signalling NaN operand with the quiet bit set to 1.
- **Two operands, neither is a signalling NaN, only one is a quiet NaN** — No exception; result is the quiet NaN operand.

In TABLE 9-1, NaNn means that the NaN is in rsn, Q means quiet, S signalling.
QSNaN means a quiet NaN produced by the NaN transformation on a signalling NaN from rsn; the invalid exception is always indicated. The QNaN results in the table never generate an exception, but IEEE 754 specifies several cases of invalid exceptions, and QNaN results from operands that are both numbers.

<table>
<thead>
<tr>
<th>rs1 operand</th>
<th>Number</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>IEEE 754</td>
<td>QNaN2</td>
<td>QNaN2</td>
</tr>
<tr>
<td>Number</td>
<td>IEEE 754</td>
<td>QNaN2</td>
<td>QNaN2</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN1</td>
<td>QNaN2</td>
<td>QNaN2</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN1</td>
<td>QNaN1</td>
<td>QNaN2</td>
</tr>
</tbody>
</table>

TABLE 9-1  Untrapped Floating-Point Results

9.3  Trapped Underflow Definition (ufm = 1)

An UltraSPARC Architecture virtual processor detects tininess before rounding occurs. (impl. dep. #55-V8-Cs10)

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 SPARC V9 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.

9.4  Untrapped Underflow Definition (ufm = 0)

On an implementation that detects tininess before rounding, untrapped underflow occurs when the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format and the correctly rounded result in the destination format is inexact.
TABLE 9-2 summarizes what happens on an implementation that detects tininess before rounding, 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.

### 9.5 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 should be set and \textit{fp_exception_IEEE_754} should be raised. If the floating-point invalid trap is disabled (FSR.tem.nvm = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is \( 2^{31} - 1 \); 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 should be set and \textit{fp_exception_IEEE_754} should be raised. If the floating-point invalid trap is disabled (FSR.tem.nvm = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is \( 2^{63} - 1 \); if the sign bit of the operand is 1, the result is \( -2^{63} \).
9.6 Floating-Point Nonstandard Mode

Please refer to Nonstandard Floating-Point (ns) on page 60 for information.
CHAPTER 9

Memory

The UltraSPARC Architecture memory models define the semantics of memory operations. The instruction set semantics require that loads and stores behave as if they are performed in the order in which they appear in the dynamic control flow of the program. The actual order in which they are processed by the memory may be different. The purpose of the memory models is to specify what constraints, if any, are placed on the order of memory operations.

The memory models apply both to uniprocessor and to shared memory multiprocessors. Formal memory models are necessary for precise definitions of the interactions between multiple virtual processors and input/output devices in a shared memory configuration. Programming shared memory multiprocessors requires a detailed understanding of the operative memory model and the ability to specify memory operations at a low level in order to build programs that can safely and reliably coordinate their activities. For additional information on the use of the models in programming real systems, see Programming with the Memory Models, contained in the separate volume UltraSPARC Architecture Application Notes.

This chapter contains a great deal of theoretical information so that the discussion of the UltraSPARC Architecture TSO memory model has sufficient background.

This chapter describes memory models in these sections:

- Memory Location Identification on page 366.
- Memory Accesses and Cacheability on page 366.
- Memory Addressing and Alternate Address Spaces on page 369.
- SPARC V9 Memory Model on page 372.
- The UltraSPARC Architecture Memory Model — TSO on page 376.
- Nonfaulting Load on page 384.
- Store Coalescing on page 385.
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 by the page translation (TTE.cp), while in other implementations, it is physical address bit specific.
Although SPARC V9 does not specify memory ordering between cacheable and noncacheable accesses, the UltraSPARC Architecture maintains TSO ordering between memory references regardless of their cacheability.

The UltraSPARC Architecture obeys the Sun-5 Ordering rules as documented in the “Sun-4u/Sun-5 Ordering with TSO” specification.

9.2.1.1 Cacheable Accesses
Accesses within the coherence domain are called cacheable accesses. They have these properties:
- Data reside in real memory locations.
- Accesses observe supported cache coherency protocol(s).
- The cache line size is $2^n$ bytes (where $n \geq 4$), and can be different for each cache.

9.2.1.2 Noncacheable Accesses
Noncacheable accesses are outside of the coherence domain. They have the following properties:
- Data might not reside in real memory locations. Accesses may result in programmer-visible side effects. An example is memory-mapped I/O control registers.
- Accesses do not observe supported cache coherency protocol(s).
- The smallest unit in each transaction is a single byte.

The UltraSPARC Architecture MMU optionally includes an attribute bit in each page translation, $TTE.e$, which when set signifies that this page has side effects.

Noncacheable accesses without side effects ($TTE.e = 0$) are processor consistent and obey TSO memory ordering. In particular, processor consistency ensures that a noncacheable load that references the same location as a previous noncacheable store will load the data of 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 385).
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 382.

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 \( ASI_{\text{PRIMARY\_NO\_FAULT}} \) or \( ASI_{\text{SECONDARY\_NO\_FAULT}} \)) with the \( TTE.e \) bit = 1 cause a 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 102, 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.

1. Two exceptions to this are the special ASI_TWIN_DW_NUCLEUS[_L] and ASI_LD_TWINX_REAL[_L] which provide hardware support for an atomic quad load to be used for TTE loads from TSBs.
Real addresses (RA). A real address is provided to privileged software to describe the underlying physical memory allocated to it. Translation storage buffers (TSBs) maintained by privileged software are used to translate privileged or nonprivileged mode virtual addresses into real addresses. MMU bypass addresses in privileged mode are also real addresses.

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).

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 389.

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 ≤ 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>
<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>0016 – 2F16</td>
<td>Restricted-to-privileged</td>
<td>privileged_action exception</td>
<td>Valid Access</td>
</tr>
<tr>
<td>8016 – FF16</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.

**FIGURE 9-1** Processor Model: Uniprocessor System

The Issue Unit reads instructions over the instruction path from memory and issues them in *program order to the Reorder Unit*. Program order is precisely the order determined by the control flow of the program and the instruction semantics, under the assumption that each instruction is performed independently and sequentially.

Issued instructions are collected and potentially reordered in the Reorder Unit, and then dispatched to the Execute Unit. Instruction reordering allows an implementation to perform some operations in parallel and to better allocate resources. The reordering of instructions is constrained to ensure that the results of program execution are the same as they would be if the instructions were performed in program order. This property is called *processor self-consistency*.

Processor self-consistency requires that the result of execution, in the absence of any shared memory interaction with another virtual processor, be identical to the result that would be observed if the instructions were performed in program order. In the model in FIGURE 9-1, instructions are issued in program order and placed in the reorder buffer. The virtual processor is allowed to reorder instructions, provided it does not violate any of the data-flow constraints for registers or for memory.

The data-flow order constraints for register reference instructions are these:

1. An instruction that reads from or writes to a register cannot be performed until all earlier instructions that write to that register have been performed (read-after-write hazard; write-after-write hazard).
2. An instruction cannot be performed that writes to a register until all earlier instructions that read that register have been performed (write-after-read hazard).

**V9 Compatibility Note** An implementation can avoid blocking instruction execution in case 2 and the write-after-write hazard in case 1 by using a renaming mechanism that provides the old value of the register to earlier instructions and the new value to later uses.

The data-flow order constraints for memory-reference instructions are those for register reference instructions, plus the following additional constraints:

1. A memory-reference instruction that uses (loads or stores) the value at a location cannot be performed until all earlier memory-reference instructions that set (store to) that location have been performed (read-after-write hazard, write-after-write hazard).

2. A memory-reference instruction that writes (stores to) a location cannot be performed until all previous instructions that read (load from) that location have been performed (write-after-read hazard).

Memory-barrier instruction (MEMBAR) and the TSO memory model also constrain the issue of memory-reference instructions. See Memory Ordering and Synchronization on page 381 and The UltraSPARC Architecture Memory Model — TSO on page 376 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.

Data caches are maintained by hardware to be consistent (coherent). Instruction caches need not be kept consistent with data caches and therefore require explicit program action to ensure consistency when a program modifies an executing instruction stream. See Synchronizing Instruction and Data Memory on page 383 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.

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

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.
may complete in any order. Once initiated, all memory operations are performed atomically: loads from one location all see the same value, and the result of stores is visible to all potential requestors at the same instant.

The order of memory operations observed at a single location is a total order that preserves the partial orderings of each virtual processor’s transactions to this address. There may be many legal total orders for a given program’s execution.

9.5 The UltraSPARC Architecture Memory Model — TSO

The UltraSPARC Architecture is a model that specifies the behavior observable by software on UltraSPARC Architecture systems. Therefore, access to memory can be implemented in any manner, as long as the behavior observed by software conforms to that of the models described here.

The SPARC V9 architecture defines three different memory models: Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory Order (RMO).

All SPARC V9 processors must provide Total Store Order (or a more strongly ordered model, for example, Sequential Consistency) to ensure compatibility for SPARC V8 application software.

All UltraSPARC Architecture virtual processors implement TSO ordering. The PSO and RMO models from SPARC V9 are not described in this UltraSPARC Architecture specification. UltraSPARC Architecture 2005 processors do not implement the PSO memory model directly, but all software written to run under PSO will execute correctly on an UltraSPARC Architecture 2005 processor (using the TSO model).

Whether memory models represented by PSTATE.mm = 102 or 112 are supported in an UltraSPARC Architecture processor is implementation dependent (impl. dep. #113-V9-Ms10). 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 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 having this model available.

9.5.1 Memory Model Selection

The active memory model is specified by the 2-bit value in PSTATE.mm. The value 00₂ represents the TSO memory model; increasing values of PSTATE.mm indicate increasingly weaker (less strongly ordered) memory models.

Writing a new value into PSTATE.mm causes subsequent memory reference instructions to be performed with the order constraints of the specified memory model.

IMPL. DEP. #119-Ms10: The effect of an attempt to write an unsupported memory model designation into PSTATE.mm is implementation dependent; however, it should never result in a value of PSTATE.mm value greater than the one that was written. In the case of an UltraSPARC Architecture implementation that only supports the TSO memory model, PSTATE.mm always reads as zero and attempts to write to it are ignored.

9.5.2 Programmer-Visible Properties of the UltraSPARC Architecture TSO Model

Total Store Order must be provided for compatibility with existing SPARC V8 programs. Programs that execute correctly in either RMO or PSO will execute correctly in the TSO model.

The rules for TSO, in addition to those required for self-consistency (see page 373), 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 376) is desired.
- Accesses that have side effects are all strongly ordered with respect to each other.
- A MEMBAR #Lookaside is not needed between a store and a subsequent load to the same noncacheable address.
- Load (LDXA) and store (STXA) instructions that reference certain internal ASIs perform both an intra-virtual processor synchronization (i.e. an implicit MEMBAR #Sync operation before the load or store is executed) and an inter-virtual processor synchronization (that is, all active virtual processors are brought to a point where synchronization is possible, the load or store is executed, and all virtual processors then resume instruction fetch and execution). The model-specific PRM should indicate which ASIs require intra-virtual processor synchronization, inter-virtual processor synchronization, or both.

9.5.3 TSO Ordering Rules

TABLE 9-2 summarizes the cases where a MEMBAR must be inserted between two memory operations on an UltraSPARC Architecture virtual processor running in TSO mode, to ensure that the operations appear to complete in a particular order. Memory operation ordering is not to be confused with processor consistency or deterministic operation; MEMBARs are required for deterministic operation of certain ASI register updates.

Programming Note

To ensure software portability across systems, the MEMBAR rules in this section should be followed (which may be stronger than the rules in SPARC V9).

TABLE 9-2 is to be read as follows: Reading from row to column, the first memory operation in program order in a row is followed by the memory operation found in the column. Symbols used as table entries:
# — No intervening operation is required.
M — an intervening MEMBAR #StoreLoad or MEMBAR #Sync or MEMBAR #MemIssue is required
S — an intervening MEMBAR #Sync or MEMBAR #MemIssue is required
nc — Noncacheable
e — Side effect
ne — No side effect

TABLE 9-2 Summary of UltraSPARC Architecture Ordering Rules (TSO Memory Model)

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

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 data_access_exception 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 data_access_exception 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 SPARCV9 instruction set architecture does not guarantee consistency between instruction and data spaces. A problem arises when instruction space is dynamically modified by a program writing to memory locations containing instructions (Self-Modifying Code). Examples are Lisp, debuggers, and dynamic linking. The FLUSH instruction synchronizes instruction and data memory after instruction space has been modified.

9.5.5.1 Ordering MEMBAR Instructions

Ordering MEMBAR instructions induce an ordering in the instruction stream of a single virtual processor. Sets of loads and stores that appear before the MEMBAR in program order are ordered with respect to sets of loads and stores that follow the MEMBAR in program order. Atomic operations (LDSTUB(A), SWAP(A), CASA, and CASXA) are ordered by MEMBAR as if they were both a load and a store, since they share the semantics of both. An STBAR instruction, with semantics that are a subset of MEMBAR, is provided for SPARC V8 compatibility. MEMBAR and STBAR operate on all pending memory operations in the reorder buffer, independently of their address or ASI, ordering them with respect to all future memory operations. This ordering applies only to memory-reference instructions issued by the virtual processor issuing the MEMBAR. Memory-reference instructions issued by other virtual processors are unaffected.

The ordering relationships are bit-encoded as shown in TABLE 9-3. For example, MEMBAR 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

1Sequencing 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.
(#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)&gt; 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)&gt; Load</td>
<td>#LoadLoad</td>
<td>nop</td>
<td>01(_{16})</td>
<td>0</td>
</tr>
<tr>
<td>Store (&lt;(m)&gt; Load</td>
<td>#StoreLoad</td>
<td>#StoreLoad</td>
<td>02(_{16})</td>
<td>1</td>
</tr>
<tr>
<td>Load (&lt;(m)&gt; Store</td>
<td>#LoadStore</td>
<td>nop</td>
<td>04(_{16})</td>
<td>2</td>
</tr>
<tr>
<td>Store (&lt;(m)&gt; Store</td>
<td>#StoreStore</td>
<td>nop</td>
<td>08(_{16})</td>
<td>3</td>
</tr>
</tbody>
</table>

Implementation Note
An UltraSPARC Architecture 2005 implementation that only implements the TSO memory model may implement MEMBAR #LoadLoad, MEMBAR #LoadStore, and MEMBAR #StoreStore as nops and MEMBAR #Storeload as a MEMBAR #Sync.

9.5.5.2 Sequencing MEMBAR Instructions

A sequencing MEMBAR exerts explicit control over the completion of operations. The three sequencing MEMBAR options each have a different degree of control and a different application.

- **Lookaside Barrier** — Ensures that loads following this MEMBAR are from memory and not from a lookaside into a write buffer. Lookaside Barrier requires that pending stores issued prior to the MEMBAR be completed before any load from that address following the MEMBAR may be issued. A Lookaside Barrier MEMBAR may be needed to provide lock fairness and to support some plausible I/O location semantics. See the example in “Control and Status Registers” in Programming with the Memory Models, contained in the separate volume UltraSPARC Architecture Application Notes.

- **Memory Issue Barrier** — Ensures that all memory operations appearing in program order before the sequencing MEMBAR complete before any new memory operation may be initiated. See the example in “I/O Registers with Side Effects” in Programming with the Memory Models, contained in the separate volume UltraSPARC Architecture Application Notes.
Synchronization Barrier  Ensures that all instructions (memory reference and others) preceding the MEMBAR complete and that the effects of any fault or error have become visible before any instruction following the MEMBAR in program order is initiated. A Synchronization Barrier MEMBAR fully synchronizes the virtual processor that issues it.

TABLE 9-4 shows the encoding of these functions in the MEMBAR instruction.

<table>
<thead>
<tr>
<th>Sequencing 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 2005 implementations, MEMBAR #Lookaside and MEMBAR #MemIssue are typically implemented as a MEMBAR #Sync.

For more details, see the MEMBAR instruction on page 258 of Chapter 8, 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 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.

**Programming Note** Because FLUSH is designed to act on a doubleword and because, on some implementations, FLUSH may trap to system software, it is recommended that system software provide a user-callable service routine for flushing arbitrarily sized regions of memory. On some implementations, this routine would issue a series of FLUSH instructions; on others, it might issue a single trap to system software that would then flush the entire region.

On an UltraSPARC Architecture virtual processor:

- A FLUSH instruction causes a synchronization with the virtual processor, which flushes the instruction pipeline in the virtual processor on which the FLUSH instruction is executed.

- Coherency between instruction and data memories may or may not be maintained by hardware. If it is, an UltraSPARC Architecture implementation may ignore the address in the operands of a FLUSH instruction.

**Programming Note** UltraSPARC Architecture virtual processors are not required to maintain coherency between instruction and data caches in hardware. Therefore, portable software must do the following:

1. must always assume that store instructions (except Block Store with Commit) do not coherently update instruction cache(s);
2. must, in every FLUSH instruction, supply the address of the instruction or instructions that were modified.

For more details, see the FLUSH instruction on page 174 of Chapter 8, *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 `data_access_exception` 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 `data_access_exception` 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 `data_access_exception` 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 016 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.
CHAPTER 10

Address Space Identifiers (ASIs)

This appendix describes address space identifiers (ASIs) in the following sections:
- Address Space Identifiers and Address Spaces on page 387.
- ASI Values on page 387.
- ASI Assignments on page 388.
- Special Memory Access ASIs on page 397.

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 00₁₆–7F₁₆ are restricted; they may only be accessed by software running in a mode with sufficient privilege for the particular ASI. ASIs in the range 00₁₆–2F₁₆ may only be accessed by software running in privileged or hyperprivileged mode and ASIs in the range 30₁₆–7F₁₆ may only be accessed by software running in hyperprivileged mode.

An attempt by nonprivileged software to access a restricted (privileged or hyperprivileged) ASI (00₁₆–7F₁₆) causes a *privileged_action* trap.

An attempt by privileged software to access a hyperprivileged ASI (30₁₆–7F₁₆) 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 **Normal** or **Translating** ASI is translated by the MMU.
- A **Nontranslating** ASI is not translated by the MMU; instead the address is passed through unchanged. Nontranslating ASIs are typically used for accessing internal registers.
- A **Bypass** ASI, like a nontranslating ASI, is not translated by the MMU and the address is passed through unchanged. However, unlike a nontranslating ASI, an access using a bypass ASI can cause exception(s) only visible in hyperprivileged mode. Bypass 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 2005 processors.

An ASI marked with an open bullet (❍) is defined by the UltraSPARC Architecture 2005 but is not necessarily implemented in all UltraSPARC Architecture 2005 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.

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 (●)</th>
<th>opt’l (❍)</th>
<th>ASI Name (and Abbreviation)</th>
<th>Access Type(s)</th>
<th>Virtual Address (VA)</th>
<th>T/ Non-T/ Bypass</th>
<th>Shared per strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0016–0316</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>_2,12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>0416</td>
<td>● ASI_NUCLEUS (ASI_N)</td>
<td>RW^2,4 (decoded)</td>
<td>T</td>
<td>—</td>
<td>Implicit address space, nucleus context, TL &gt; 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0516–0B16</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>_2,12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>0C16</td>
<td>● ASI_NUCLEUS_LITTLE (ASI_NL)</td>
<td>RW^2,4 (decoded)</td>
<td>T</td>
<td>—</td>
<td>Implicit address space, nucleus context, TL &gt; 0, little-endian</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CHAPTER 10 • Address Space Identifiers (ASIs) 389
<table>
<thead>
<tr>
<th>ASI</th>
<th>Access Type(s)</th>
<th>Virtual Address (VA)</th>
<th>T/Non-T</th>
<th>Shared/Per Strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0D16–0F16</td>
<td>○</td>
<td>–</td>
<td>_2,12</td>
<td>–</td>
<td>Implementation dependent^</td>
</tr>
<tr>
<td>1016</td>
<td>● ASI_AS_IF_USER_PRIMARY (ASI_AIUP)</td>
<td>RW2,4,18 (decoded)</td>
<td>T</td>
<td>–</td>
<td>Primary address space, as if user (nonprivileged)</td>
</tr>
<tr>
<td>1116</td>
<td>● ASI_AS_IF_USER_SECONDARY (ASI_AIUS)</td>
<td>RW2,4,18 (decoded)</td>
<td>T</td>
<td>–</td>
<td>Secondary address space, as if user (nonprivileged)</td>
</tr>
<tr>
<td>1216–1316</td>
<td>○</td>
<td>–</td>
<td>_2,12</td>
<td>–</td>
<td>Implementation dependent^</td>
</tr>
<tr>
<td>1416</td>
<td>● ASI_REAL</td>
<td>RW2,4 (decoded)</td>
<td>B</td>
<td>–</td>
<td>Real address</td>
</tr>
<tr>
<td>1516</td>
<td>○ ASI_REAL_IO D (ASI_REAL_L)</td>
<td>RW2,5 (decoded)</td>
<td>B</td>
<td>–</td>
<td>Physical address, noncacheable, with side effect (deprecated)</td>
</tr>
<tr>
<td>1616</td>
<td>○ ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)</td>
<td>RW2,8,14,18 (decoded)</td>
<td>T</td>
<td>–</td>
<td>Primary address space, block load/store, as if user (nonprivileged)</td>
</tr>
<tr>
<td>1716</td>
<td>○ ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS)</td>
<td>RW2,8,14,18 (decoded)</td>
<td>T</td>
<td>–</td>
<td>Secondary address space, block load/store, as if user (nonprivileged)</td>
</tr>
<tr>
<td>1816</td>
<td>● ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)</td>
<td>RW2,4,18 (decoded)</td>
<td>T</td>
<td>–</td>
<td>Primary address space, as if user (nonprivileged), little-endian</td>
</tr>
<tr>
<td>1916</td>
<td>● ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)</td>
<td>RW2,4,18 (decoded)</td>
<td>T</td>
<td>–</td>
<td>Secondary address space, as if user (nonprivileged), little-endian</td>
</tr>
<tr>
<td>1A16–1B16</td>
<td>○</td>
<td>–</td>
<td>_2,12</td>
<td>–</td>
<td>Implementation dependent^</td>
</tr>
<tr>
<td>1C16</td>
<td>○ ASI_REAL_LITTLE (ASI_REAL_L)</td>
<td>RW2,4 (decoded)</td>
<td>B</td>
<td>–</td>
<td>Real address, little-endian</td>
</tr>
<tr>
<td>1D16</td>
<td>○ ASI_REAL_IO_LITTLE D (ASI_REAL_IO_L D)</td>
<td>RW2,5 (decoded)</td>
<td>B</td>
<td>–</td>
<td>Physical address, noncacheable, with side effect, little-endian (deprecated)</td>
</tr>
<tr>
<td>1E16</td>
<td>○ ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUPL)</td>
<td>RW2,8,14,18 (decoded)</td>
<td>T</td>
<td>–</td>
<td>Primary address space, block load/store, as if user (nonprivileged), little-endian</td>
</tr>
<tr>
<td>1F16</td>
<td>○ ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL)</td>
<td>RW2,8,14,18 (decoded)</td>
<td>T</td>
<td>–</td>
<td>Secondary address space, block load/store, as if user (nonprivileged), little-endian</td>
</tr>
<tr>
<td>ASI Value</td>
<td>req’d (●), opt’l (❍)</td>
<td>ASI Name (and Abbreviation)</td>
<td>Access Type(s)</td>
<td>Virtual Address (VA)</td>
<td>T/Non-T</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------</td>
<td>-----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>--------</td>
</tr>
<tr>
<td>20₁₆</td>
<td>●</td>
<td>ASI_SCRATCHPAD</td>
<td>RW²,⁶</td>
<td>(decoded; see below)</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0₁₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8₁₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1₀₁₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1₈₁₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2₀₁₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2₈₁₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3₀₁₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3₈₁₆</td>
<td></td>
</tr>
<tr>
<td>2₁₁₆</td>
<td>●</td>
<td>ASI_MMU_CONTEXTID</td>
<td>RW²,⁶</td>
<td>(decoded; see below)</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8₁₆</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1₀₁₆</td>
<td></td>
</tr>
<tr>
<td>2₂₁₆</td>
<td>●</td>
<td>ASI_LD_TWIXX_AS_IF_USER_ PRIMARY (ASI_LDTX_AIUP)</td>
<td>R²/²,¹¹</td>
<td>(decoded)</td>
<td>T</td>
</tr>
<tr>
<td>2₃₁₆</td>
<td>●</td>
<td>ASI_LD_TWIXX_AS_IF_USER_ SECONDARY (ASI_LDTX_AIUS)</td>
<td>R²/²,¹¹</td>
<td>(decoded)</td>
<td>T</td>
</tr>
<tr>
<td>2₄₁₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</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>T/Non-T</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>2516</td>
<td>◊</td>
<td>ASI_QUEUE</td>
<td>(see below)</td>
<td>N</td>
<td>per strand</td>
</tr>
<tr>
<td></td>
<td>◊</td>
<td></td>
<td>RW^2,6</td>
<td>3C0_{16}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>◊</td>
<td></td>
<td>RW^2,6,17</td>
<td>3C8_{16}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>◊</td>
<td></td>
<td>RW^2,6</td>
<td>3D0_{16}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>◊</td>
<td></td>
<td>RW^2,6,17</td>
<td>3D8_{16}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>◊</td>
<td></td>
<td>RW^2,6</td>
<td>3E0_{16}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>◊</td>
<td></td>
<td>RW^2,6,17</td>
<td>3E8_{16}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>◊</td>
<td></td>
<td>RW^2,6</td>
<td>3F0_{16}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>◊</td>
<td></td>
<td>RW^2,6,17</td>
<td>3F8_{16}</td>
<td></td>
</tr>
<tr>
<td>2616</td>
<td>◊</td>
<td>ASI_LD_TWINX_REAL</td>
<td>R^2,11</td>
<td>B</td>
<td>―</td>
</tr>
<tr>
<td>2716</td>
<td>◊</td>
<td>ASI_LD_TWINX_NUCLEUS</td>
<td>R^2,11</td>
<td>T</td>
<td>―</td>
</tr>
<tr>
<td>28_{16−}</td>
<td>◊</td>
<td></td>
<td>___</td>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>29_{16}</td>
<td>◊</td>
<td></td>
<td>___</td>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>2A_{16}</td>
<td>◊</td>
<td>ASI_LD_TWINX_AS_IF_USER_</td>
<td>R^2,11</td>
<td>T</td>
<td>―</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRIMARY_LITTLE</td>
<td>(ASI_LDTX_AIUPL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2B_{16}</td>
<td>◊</td>
<td>ASI_LD_TWINX_AS_IF_USER_</td>
<td>R^2,11</td>
<td>T</td>
<td>―</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SECONDARY_LITTLE</td>
<td>(ASI_LDTX_AIUS_L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2C_{16}</td>
<td>◊</td>
<td></td>
<td>___</td>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>2D_{16}</td>
<td>◊</td>
<td></td>
<td>___</td>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>2E_{16}</td>
<td>◊</td>
<td>ASI_LD_TWINX_REAL_LITTLE</td>
<td>R^2,11</td>
<td>B</td>
<td>―</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ASI_LDTX_REAL_L)</td>
<td>(ASI_QUAD_LDD_REAL_L)</td>
<td></td>
<td></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>T/ Non-T/ Bypass</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>2F16</td>
<td>○</td>
<td>ASI_LD_TWINX_NUCLEUS_LITTLE (ASI_LDTX_NL)</td>
<td>R27,11 (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>30–7F16</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>_3</td>
<td>—</td>
</tr>
<tr>
<td>4516</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>_3,13</td>
<td>—</td>
</tr>
<tr>
<td>46–4816</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>_3,13</td>
<td>—</td>
</tr>
<tr>
<td>4916</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>_3,13</td>
<td>—</td>
</tr>
<tr>
<td>4A–4B16</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>_3,13</td>
<td>—</td>
</tr>
<tr>
<td>4C16</td>
<td>○</td>
<td>Error Status and Enable Registers</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8016</td>
<td>●</td>
<td>ASI_PRIMARY (ASI_P)</td>
<td>RW4 (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>8116</td>
<td>●</td>
<td>ASI_SECONDARY (ASI_S)</td>
<td>RW4 (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>8216</td>
<td>●</td>
<td>ASI_PRIMARY_NO_FAULT (ASI_PNF)</td>
<td>R7,11 (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>8316</td>
<td>●</td>
<td>ASI_SECONDARY_NO_FAULT (ASI_SNF)</td>
<td>R7,11 (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>84–8716</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>_16</td>
<td>—</td>
</tr>
<tr>
<td>8816</td>
<td>●</td>
<td>ASI_PRIMARY_LITTLE (ASI_PL)</td>
<td>RW4 (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>8916</td>
<td>●</td>
<td>ASI_SECONDARY_LITTLE (ASI_SL)</td>
<td>RW4 (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>8A–8B16</td>
<td>●</td>
<td>ASI_PRIMARY_NO_FAULT_LITTLE (ASI_PNFL)</td>
<td>R7,11 (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>8C–BF16</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>_16</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>T</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>T</td>
<td>—</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>T/ Non-T/ Bypass</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>-----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>C2\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST16_PRIMARY (ASI_PST16_P)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>C3\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST16_SECONDARY (ASI_PST16_S)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>C4\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST32_PRIMARY (ASI_PST32_P)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>C5\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST32_SECONDARY (ASI_PST32_S)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>C6\textsubscript{16}-C7\textsubscript{16}</td>
<td>●</td>
<td>—</td>
<td>_\textsuperscript{15}</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C8\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>C9\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>CA\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>CB\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>CC\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>CD\textsubscript{16}</td>
<td>○</td>
<td>ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL)</td>
<td>W\textsuperscript{R,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>CE\textsubscript{16}-CF\textsubscript{16}</td>
<td>●</td>
<td>—</td>
<td>_\textsuperscript{15}</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>D0\textsubscript{16}</td>
<td>○</td>
<td>ASI_FL8_PRIMARY (ASI_FL8_P)</td>
<td>RW\textsuperscript{,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>D1\textsubscript{16}</td>
<td>○</td>
<td>ASI_FL8_SECONDARY (ASI_FL8_S)</td>
<td>RW\textsuperscript{,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>D2\textsubscript{16}</td>
<td>○</td>
<td>ASI_FL16_PRIMARY (ASI_FL16_P)</td>
<td>RW\textsuperscript{,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>D3\textsubscript{16}</td>
<td>○</td>
<td>ASI_FL16_SECONDARY (ASI_FL16_S)</td>
<td>RW\textsuperscript{,10,14} (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>D4\textsubscript{16}-D7\textsubscript{16}</td>
<td>●</td>
<td>—</td>
<td>_\textsuperscript{15}</td>
<td>—</td>
<td>—</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>T / Non-T / Bypass</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>----------------------------</td>
<td>----------------</td>
<td>----------------------</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>T</td>
<td>—</td>
</tr>
<tr>
<td>D₉₁₆</td>
<td>❍</td>
<td>ASI_FL8_SECONDARY_LITTLE (ASI_FL8_SL)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>D₁₀₁₆</td>
<td>❍</td>
<td>ASI_FL16_PRIMARY_LITTLE (ASI_FL16_PL)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>D₁₁₁₆</td>
<td>❍</td>
<td>ASI_FL16_SECONDARY_LITTLE (ASI_FL16_SL)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>DC₁₆ – DF₁₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>E₀₁₆ – E₁₁₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>E₂₁₆</td>
<td>❍</td>
<td>ASI_LD_TWINX_PRIMARY (ASI_LDTX_P)</td>
<td>R¹⁹ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>E₃₁₆</td>
<td>❍</td>
<td>ASI_LD_TWINX_SECONDARY (ASI_LDTX_S)</td>
<td>R¹⁹ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>E₄₁₆ – E₉₁₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>E₁₀₁₆</td>
<td>❍</td>
<td>ASI_LD_TWINX_PRIMARY_LITTLE (ASI_LDTX_PL)</td>
<td>R¹⁹ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>E₁₁₁₆</td>
<td>❍</td>
<td>ASI_LD_TWINX_SECONDARY_LITTLE (ASI_LDTX_SL)</td>
<td>R¹⁹ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>E₁₂₁₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>F₀₁₆</td>
<td>❍</td>
<td>ASI_BLOCK_PRIMARY (ASI_BLK_P)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>F₁₁₆</td>
<td>❍</td>
<td>ASI_BLOCK_SECONDARY (ASI_BLK_S)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>T</td>
<td>—</td>
</tr>
<tr>
<td>F₂₁₆ – F₇₁₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
TABLE 10-1  UltraSPARC Architecture ASIs  (8 of 8)

<table>
<thead>
<tr>
<th>ASI 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>T/Non-T</th>
<th>Bypass</th>
<th>Shared/per strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8H</td>
<td>● ASI_BLOCK_PRIMARY_LITTLE (ASI_BLK_PL)</td>
<td>RW8,14 (decoded)</td>
<td>T</td>
<td>_</td>
<td>Primary address space, 8x8-byte block load/store, little endian</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F9H</td>
<td>● ASI_BLOCK_SECONDARY_LITTLE (ASI_BLK_SL)</td>
<td>RW8,14 (decoded)</td>
<td>T</td>
<td>_</td>
<td>Secondary address space, 8x8-byte block load/store, little endian</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAH–FFH</td>
<td>●</td>
<td>_</td>
<td>_15</td>
<td>_</td>
<td>_</td>
<td>Implementation dependent1</td>
<td></td>
<td></td>
<td></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 privileged_action exception.

3 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in nonprivileged mode or privileged mode causes a privileged_action exception.

4 May be used with all load alternate, store alternate, atomic alternate and prefetch alternate instructions (CASA, CASXA, LDSTUBA, LDTWA, LDDFA, LDEA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDXA, PREFETCHA, STBA, STTWA, STTWA, STTWA, STTWA, STTWA, SWAPA).

5 May be used with all of the following load alternate and store alternate instructions: LDTWA, LDDFA, LDEA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDXA, STBA, STTWA, STTWA, STTWA, STTWA, STTXA, STXH, STWA, STXH, STWA, STXH, STTXA. Use with an atomic alternate or prefetch alternate instruction (CASA, CASXA, LDSTUBA, SWAPA or PREFETCHA) causes a data_access_exception exception.

6 May only be used in a LDXA or STXA instruction for RW ASIs, LDXA for read-only ASIs and STXH for write-only ASIs. Use of LDXA for write-only ASIs, STXH for read-only ASIs, or any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a data_access_exception exception.

7 May only be used in an LDTXA instruction. Use of this ASI in any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a data_access_exception 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 data_access_exception 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 data_access_exception (privilege violation) exception occurs

---

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 data_access_exception exception.

10 Write(store)-only ASI; an attempted load alternate, atomic alternate, or prefetch alternate instruction to this ASI causes a data_access_exception exception.

11 Read(load)-only ASI; an attempted store alternate or atomic alternate instruction to this ASI causes a data_access_exception exception.

12 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in privileged mode causes a data_access_exception 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 this ASI in any mode causes a data_access_exception exception.

14 An attempted access to this ASI may cause an exception (see Special Memory Access ASIs on page 397 for details).

15 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in any mode causes a data_access_exception exception if this ASI is not implemented by the model dependent implementation.

16 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to a reserved ASI in any mode causes a data_access_exception exception.

17 The Queue Tail Registers (ASI 25_{16}) are read-only. An attempted write to the Queue Tail Registers causes a data_access_exception exception
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 10-2 Privileged ASI_*AS_IF_USER_* ASIs

<table>
<thead>
<tr>
<th>ASI</th>
<th>Names</th>
<th>Addressing</th>
<th>Endianness of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>10₁₆</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>11₁₆</td>
<td>ASI_AS_IF_USER_SECONDARY (ASI_AIUS)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
<tr>
<td>1₆₁₆</td>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)</td>
<td>Virtual (Primary)</td>
<td></td>
</tr>
<tr>
<td>1₇₁₆</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 1₈₁₆, 1₉₁₆, 1E₁₆, and 1F₁₆ (ASI_*AS_IF_USER_*_LITTLE)

These ASIs are little-endian versions of ASIs 1₀₁₆, 1₁₁₆, 1₆₁₆, and 1₇₁₆ (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 data_access_exception (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.
10.4.3 **ASI 14_{16} (ASI_REAL)**

When ASI_REAL is specified in any load alternate, store alternate or prefetch alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a privileged_action exception occurs
- In any other privilege mode:
  - VA is passed through to RA
  - 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_{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 data_access_exception 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.

### 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>18_{16}</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>19_{16}</td>
<td>ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
<tr>
<td>1E_{16}</td>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUP)</td>
<td>Virtual (Primary)</td>
<td></td>
</tr>
<tr>
<td>1F_{16}</td>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
</tbody>
</table>
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 the opposite of that for ASI\_REAL.

10.4.6 ASI 1D\textsubscript{16} (ASI\_REAL\_IO\_LITTLE)

ASI\_REAL\_IO\_LITTLE is a little-endian version of ASI 15\textsubscript{16} (ASI\_REAL\_IO). It operates identically to ASI\_REAL\_IO, except if an access occurs, its endianness the opposite of that for ASI\_REAL\_IO.

10.4.7 ASIs 22\textsubscript{16}, 23\textsubscript{16}, 27\textsubscript{16}, 2A\textsubscript{16}, 2B\textsubscript{16}, 2F\textsubscript{16} (Privileged Load Integer Twin Extended Word)

ASIs 22\textsubscript{16}, 23\textsubscript{16}, 27\textsubscript{16}, 2A\textsubscript{16}, 2B\textsubscript{16} and 2F\textsubscript{16} exist for use with the (nonportable) LDTXA instruction as atomic Load Integer Twin Extended Word operations (see Load Integer Twin Extended Word from Alternate Space on page 250). 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 `data_access_exception` 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 406 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 250). 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 `data_access_exception` 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 406 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 250). These ASIs are distinguished by the address space accessed (Primary or Secondary) and the endianness of the access, as described in Table 10-6.
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 `data_access_exception` 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}`, `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 232 and Block Store on page 312).

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.

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 `data_access_exception` 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 325).

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 **data_access_exception** exception is generated and **mem_address_not_aligned**, **LDDF_mem_address_not_aligned**, and **illegal_instruction** (for \( i = 1 \)) are not generated.

ASIs C0_{16}–C5_{16} and C8_{16}–CD_{16} are only defined for use in Partial Store operations (see page 325). 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 241.

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 on page 236 and Store Floating-Point on page 316).

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 **data_access_exception** 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 2005 ASIs is shown in TABLE 10-1 on page 389.
10.5.1 Privileged Scratchpad Registers

(ASI_SCRATCHPAD) **DI**

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>ASI_SCRATCHPAD</td>
<td>2016</td>
<td>0016 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0816 1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1016 2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1816 3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2016 4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2816 5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3016 6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3816 7</td>
<td>7</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 data_access_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 (ASI_PHYS_USE_EC_L)</td>
<td>ASI_REAL_LITTLE</td>
</tr>
<tr>
<td>1D16</td>
<td>ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE (ASI_PHY_BYPASS_EC_WITH_EBIT_L)</td>
<td>ASI_REAL_IO_LITTLE</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></td>
<td></td>
</tr>
<tr>
<td>2C16</td>
<td>ASI_NUCLEUS_QUAD_LDD_ LITTLE (ASI_NUCLEUS_QUAD_LDD_L)</td>
</tr>
</tbody>
</table>
Performance Instrumentation

(contents to be supplied in a later revision)
A trap is a vectored transfer of control to software running in a privilege mode (see page 410) 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 410.
- Virtual Processor States and Traps on page 412.
- Trap Categories on page 412.
- Trap Control on page 417.
- Trap-Table Entry Addresses on page 418.
- Trap Processing on page 429.
- Exception and Interrupt Descriptions on page 431.
- Register Window Traps on page 436.

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>

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)
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 422 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 419 for details.

The maximum trap level at which privileged software may execute is MAXPTL (which, on an 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, HT indicates a “trap destined for privileged mode” and HY 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>
<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
- Reset trap

12.3.1 Precise Traps

A precise trap is induced by a particular instruction and occurs before any program-visible state has been changed by the trap-inducing instructions. When a precise trap occurs, several conditions must be true:

- The PC saved in TPC[TL] points to the instruction that induced the trap and the NPC saved in TNPC[TL] points to the instruction that was to be executed next.
- All instructions issued before the one that induced the trap have completed execution.
- Any instructions issued after the one that induced the trap remain unexecuted.
Among the actions that trap handler software might take when processing a precise trap are:

- Return to the instruction that caused the trap and reexecute it by executing a RETRY instruction (PC ← old PC, NPC ← old NPC).
- Emulate the instruction that caused the trap and return to the succeeding instruction by executing a DONE instruction (PC ← old NPC, NPC ← old NPC + 4).
- Terminate the program or process associated with the trap.

### 12.3.2 Deferred Traps

A *deferred trap* is also induced by a particular instruction, but unlike a precise trap, a deferred trap may occur after program-visible state has been changed. Such state may have been changed by the execution of either the trap-inducing instruction itself or by one or more other instructions.

There are two classes of deferred traps:

- **Termination deferred traps** — The instruction (usually a store) that caused the trap has passed the retirement point of execution (the TPC has been updated to point to an instruction beyond the one that caused the trap). The trap condition is an error that prevents the instruction from completing and its results becoming globally visible. A termination deferred trap has high trap priority, second only to the priority of resets.

  **Programming Note** Not enough state is saved for execution of the instruction stream to resume with the instruction that caused the trap. Therefore, the trap handler must terminate the process containing the instruction that caused the trap.

- **Restartable deferred traps** — The program-visible state has been changed by the trap-inducing instruction or by one or more other instructions after the trap-inducing instruction.

  **SPARC V9 Compatibility Note** A restartable deferred trap is the “deferred trap” defined in the SPARC V9 specification.

The fundamental characteristic of a *restartable* deferred trap is that the state of the virtual processor on which the trap occurred may not be consistent with any precise point in the instruction sequence being executed on that virtual processor. When a restartable deferred trap occurs, TPC[TL] and TNPC[TL] contain a PC value and an NPC value, respectively, corresponding to a point in the instruction sequence being executed on the virtual processor. 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 $\text{TPC}[\text{TL}]$ refers. By definition, the instruction to which $\text{TPC}[\text{TL}]$ 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 $\text{TPC}[\text{TL}]$, 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 $\text{TPC}[\text{TL}]$ 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 $\text{TPC}[\text{TL}]$ 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 2005 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></td>
<td></td>
<td>Privileged</td>
</tr>
<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>
<tr>
<td></td>
<td></td>
<td>Delivered immediately</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 431 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 415 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 428.

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.aexc fields are updated to their new values.

12.5 Trap-Table Entry Addresses

Traps are delivered to the virtual processor in either privileged mode or hyperprivileged mode, depending on the trap type, the value of TL at the time the trap is taken, and the privilege mode at the time the exception was detected. See TABLE 12-4 on page 422 and TABLE 12-5 on page 426 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>from TBA[63:15] (TBA.tba_high49)</th>
<th>TL&gt;0</th>
<th>TT[TL]</th>
<th>00000</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>15</td>
<td>14</td>
<td>13 5 4 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.

<table>
<thead>
<tr>
<th>Value of TL (before trap)</th>
<th>Software Trap Type</th>
<th>Hardware Trap Type (TT[TL])</th>
<th>Trap Table Offset (from TBA)</th>
<th>Contents of Trap Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>TL = 0</td>
<td>—</td>
<td>000_{16} - 07F_{16}</td>
<td>0_{16} - FE_{016}</td>
<td>Hardware traps</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>080_{16} - OFF_{16}</td>
<td>1000_{16} - 1FE_{016}</td>
<td>Spill / fill traps</td>
</tr>
<tr>
<td></td>
<td>0_{16} - 7F_{16}</td>
<td>100_{16} - 17F_{16}</td>
<td>2000_{16} - 2FE_{016}</td>
<td>Software traps to Privileged level</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>180_{16} - 1FF_{16}</td>
<td>3000_{16} - 3FE_{016}</td>
<td>unassigned</td>
</tr>
<tr>
<td>TL = 1 (TL = MAXPTL-1)</td>
<td>—</td>
<td>000_{16} - 07F_{16}</td>
<td>4000_{16} - 4FE_{016}</td>
<td>Hardware traps</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>080_{16} - OFF_{16}</td>
<td>5000_{16} - 5FE_{016}</td>
<td>Spill / fill traps</td>
</tr>
<tr>
<td></td>
<td>0_{16} - 7F_{16}</td>
<td>100_{16} - 17F_{16}</td>
<td>6000_{16} - 6FE_{016}</td>
<td>Software traps to Privileged level</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>180_{16} - 1FF_{16}</td>
<td>7000_{16} - 7FE_{016}</td>
<td>unassigned</td>
</tr>
</tbody>
</table>

FIGURE 12-3 Privileged-mode Trap Table Layout

The trap table for TL = 0 comprises 512 thirty-two-byte entries; the trap table for TL > 0 comprises 512 more thirty-two-byte entries. Therefore, the total size of a full privileged trap table is \(2 \times 512 \times 32\) bytes (32 Kbytes). However, if privileged software does not use software traps (Tcc instructions) at TL > 0, the table can be made 24 Kbytes long.

12.5.3 Trap 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 419)

TT values 000_{16} - 0FF_{16} are reserved for hardware traps. TT values 100_{16} - 17F_{16} are reserved for software traps (caused by execution of a Tcc instruction) to privileged-mode trap handlers.

**IMPL. DEP. #35-V8-Ce20:** 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 2005. 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 2005, 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-2005</td>
<td>Exception or Interrupt Request</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>● Req’d.</td>
<td>◊ Opt’l</td>
</tr>
<tr>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>—</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>●</td>
<td>instruction_access_exception</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>Reserved</td>
</tr>
<tr>
<td>—</td>
<td>Reserved</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>Reserved</td>
</tr>
<tr>
<td>—</td>
<td>Reserved</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</td>
</tr>
</tbody>
</table>
TABLE 12-4 Exception and Interrupt Requests, by TT Value (2 of 4)

<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>clean_window</td>
<td>024&lt;sub&gt;16&lt;/sub&gt;³</td>
<td>precise</td>
<td>10.1</td>
<td>P (nm)</td>
</tr>
<tr>
<td></td>
<td>025&lt;sub&gt;16&lt;/sub&gt;⁻</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>027&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>division_by_zero</td>
<td>028&lt;sub&gt;16&lt;/sub&gt;</td>
<td>precise</td>
<td>15</td>
<td>P (nm)</td>
</tr>
<tr>
<td></td>
<td>02C&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>02D&lt;sub&gt;16&lt;/sub&gt;⁻</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data_access_exception</td>
<td>030&lt;sub&gt;16&lt;/sub&gt;</td>
<td>precise</td>
<td>12.01</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>032&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mem_address_not_aligned</td>
<td>034&lt;sub&gt;16&lt;/sub&gt;</td>
<td>precise</td>
<td>10.2</td>
<td>H</td>
</tr>
<tr>
<td>LDDF_mem_address_not_aligned</td>
<td>035&lt;sub&gt;16&lt;/sub&gt;</td>
<td>precise</td>
<td>10.1</td>
<td>H</td>
</tr>
<tr>
<td>STDF_mem_address_not_aligned</td>
<td>036&lt;sub&gt;16&lt;/sub&gt;</td>
<td>precise</td>
<td>10.1</td>
<td>H</td>
</tr>
<tr>
<td>privileged_action</td>
<td>037&lt;sub&gt;16&lt;/sub&gt;</td>
<td>precise</td>
<td>11.1</td>
<td>H</td>
</tr>
<tr>
<td>LDQF_mem_address_not_aligned</td>
<td>038&lt;sub&gt;16&lt;/sub&gt;</td>
<td>precise</td>
<td>10.1</td>
<td>H</td>
</tr>
<tr>
<td>STQF_mem_address_not_aligned</td>
<td>039&lt;sub&gt;16&lt;/sub&gt;</td>
<td>precise</td>
<td>10.1</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>03A&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>03B&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>03D&lt;sub&gt;16&lt;/sub&gt;⁻</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interrupt_level_n (n = 1–15)</td>
<td>041&lt;sub&gt;16&lt;/sub&gt;⁻</td>
<td>disrupting</td>
<td>32-π (31 to 17)</td>
<td>P (ie)</td>
</tr>
<tr>
<td></td>
<td>04F&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>050&lt;sub&gt;16&lt;/sub&gt;⁻</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>05D&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>05F&lt;sub&gt;16&lt;/sub&gt;⁻</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>061&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>060&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>062&lt;sub&gt;16&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception or Interrupt Request</td>
<td>TT (Trap Type)</td>
<td>Trap Category</td>
<td>Priority (0 = Highest)</td>
<td>Mode in Which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td>---------------</td>
<td>------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VA_watchpoint</td>
<td>062_{16}</td>
<td>precise</td>
<td>11.2</td>
<td>P (nm)</td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>063_{16}−06C_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Reserved</td>
<td>06D_{16}−06F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
<td>070_{16}−075_{16}</td>
<td>—</td>
<td>V</td>
<td>—</td>
</tr>
<tr>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
<td>077</td>
<td>—</td>
<td>V</td>
<td>—</td>
</tr>
<tr>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
<td>079_{16}−07B_{16}</td>
<td>—</td>
<td>V</td>
<td>—</td>
</tr>
<tr>
<td>Reserved</td>
<td>079_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>cpu_mondo</td>
<td>07C_{16}</td>
<td>disrupting</td>
<td>16.08</td>
<td>P (ie)</td>
</tr>
<tr>
<td>dev_mondo</td>
<td>07D_{16}</td>
<td>disrupting</td>
<td>16.11</td>
<td>P (ie)</td>
</tr>
<tr>
<td>resumable_error</td>
<td>07E_{16}</td>
<td>disrupting</td>
<td>33.3</td>
<td>P (ie)</td>
</tr>
<tr>
<td>implementation_dependent_exception_15 (impl. dep. #35-V8-Cs20)</td>
<td>07F_{16}</td>
<td>—</td>
<td>V</td>
<td>—</td>
</tr>
<tr>
<td>nonresumable_error</td>
<td>07F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>spill_n_normal (n = 0–7)</td>
<td>080_{16}−09C_{16}</td>
<td>precise</td>
<td>9</td>
<td>P (nm)</td>
</tr>
<tr>
<td>(reserved for use by spill_7_normal; see footnote for trap type 09C_{16})</td>
<td>09D_{16}−09F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>spill_n_other (n = 0–7)</td>
<td>0A0_{16}−0BC_{16}</td>
<td>precise</td>
<td>9</td>
<td>P (nm)</td>
</tr>
<tr>
<td>(reserved for use by spill_7_other; see footnote for trap type 0BC_{16})</td>
<td>0BD_{16}−0BF_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>fill_n_normal (n = 0–7)</td>
<td>0C0_{16}−0DC_{16}</td>
<td>precise</td>
<td>9</td>
<td>P (nm)</td>
</tr>
</tbody>
</table>
TABLE 12-4  Exception and Interrupt Requests, by TT Value  (4 of 4)

<table>
<thead>
<tr>
<th>UA-2005</th>
<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>●</td>
<td>(reserved for use by fill_7_normal; see footnote for trap type 0DC16)</td>
<td>0DD16–0DF16</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>●</td>
<td>fill_n_other (n = 0–7)</td>
<td>0E016–0FC16</td>
<td>precise</td>
<td>9</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>●</td>
<td>(reserved for use by fill_7_other; see footnote for trap type 0FC16)</td>
<td>0FD16–0FF16</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>●</td>
<td>trap_instruction</td>
<td>10016–17F16</td>
<td>precise</td>
<td>16.02</td>
<td>P (nm) P (nm)</td>
</tr>
<tr>
<td>●</td>
<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 428), 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.

V The priority of an implementation_dependent_exception_n trap is implementation dependent (impl. dep. # 35-V8-Cs20)

D This exception is deprecated, because the only instructions that can generate it have been deprecated.
## Table 12-5: Exception and Interrupt Requests, by Priority (1 of 2)

<table>
<thead>
<tr>
<th>EA-2005</th>
<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 (and Conditioning Applied), based on Current Privilege Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>instruction_access_exception</td>
<td>00816</td>
<td>precise</td>
<td>3</td>
<td>H H</td>
</tr>
<tr>
<td>●</td>
<td>illegal_instruction</td>
<td>01016</td>
<td>precise</td>
<td>6.2</td>
<td>H H</td>
</tr>
<tr>
<td>●</td>
<td>privileged_opcode</td>
<td>01116</td>
<td>precise</td>
<td>7</td>
<td>P -x-</td>
</tr>
<tr>
<td>●</td>
<td>fp_disabled</td>
<td>02016</td>
<td>precise</td>
<td>8</td>
<td>P (nm) P</td>
</tr>
<tr>
<td>●</td>
<td>spill_n_normal (n = 0–7)</td>
<td>08016–09C16‡</td>
<td>precise</td>
<td></td>
<td>P (nm) P</td>
</tr>
<tr>
<td>●</td>
<td>spill_n_other (n = 0–7)</td>
<td>0A016–0BC16‡</td>
<td>precise</td>
<td></td>
<td>P (nm) P</td>
</tr>
<tr>
<td>●</td>
<td>fill_n_normal (n = 0–7)</td>
<td>0C016–0DC16‡</td>
<td>precise</td>
<td>9</td>
<td>P (nm) P</td>
</tr>
<tr>
<td>●</td>
<td>fill_n_other (n = 0–7)</td>
<td>0E016–0FC16‡</td>
<td>precise</td>
<td></td>
<td>P (nm) P</td>
</tr>
<tr>
<td>●</td>
<td>clean_window</td>
<td>02416†</td>
<td>precise</td>
<td></td>
<td>P (nm) P</td>
</tr>
<tr>
<td>●</td>
<td>LDDF_mem_address_not_aligned</td>
<td>03516</td>
<td>precise</td>
<td>10.1</td>
<td>H H</td>
</tr>
<tr>
<td>●</td>
<td>STDF_mem_address_not_aligned</td>
<td>03616</td>
<td>precise</td>
<td></td>
<td>H H</td>
</tr>
<tr>
<td>○</td>
<td>LDQF_mem_address_not_aligned</td>
<td>03816</td>
<td>precise</td>
<td></td>
<td>H H</td>
</tr>
<tr>
<td>○</td>
<td>STQF_mem_address_not_aligned</td>
<td>03916</td>
<td>precise</td>
<td></td>
<td>H H</td>
</tr>
<tr>
<td>●</td>
<td>mem_address_not_aligned</td>
<td>03416</td>
<td>precise</td>
<td>10.2</td>
<td>H H</td>
</tr>
<tr>
<td>○</td>
<td>fp_exception_other</td>
<td>02216</td>
<td>precise</td>
<td></td>
<td>P (nm) P</td>
</tr>
<tr>
<td>○</td>
<td>fp_exception_ieee_754</td>
<td>02116</td>
<td>precise</td>
<td>11.1</td>
<td>P (nm) P</td>
</tr>
<tr>
<td>●</td>
<td>privileged_action</td>
<td>03716</td>
<td>precise</td>
<td></td>
<td>H H</td>
</tr>
<tr>
<td>○</td>
<td>VA_watchpoint</td>
<td>06216</td>
<td>precise</td>
<td>11.2</td>
<td>P (nm) P</td>
</tr>
</tbody>
</table>
TABLE 12-5 Exception and Interrupt Requests, by Priority (2 of 2)

<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 (and Conditioning Applied), based on Current Privilege Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>● data_access_exception</td>
<td>030_{16}</td>
<td>precise</td>
<td>12.01</td>
<td>H H</td>
</tr>
<tr>
<td>● tag_overflow{D}</td>
<td>023_{16}</td>
<td>precise</td>
<td>14</td>
<td>P P (nm) (nm)</td>
</tr>
<tr>
<td>● division_by_zero</td>
<td>028_{16}</td>
<td>precise</td>
<td>15</td>
<td>P (nm) (nm)</td>
</tr>
<tr>
<td>● trap_instruction</td>
<td>100_{16–17F_{16}}</td>
<td>precise</td>
<td>16.02</td>
<td>-x-</td>
</tr>
<tr>
<td>● htrap_instruction</td>
<td>180_{16–1FF_{16}}</td>
<td>precise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>● cpu_mondo</td>
<td>07C_{16}</td>
<td>disrupting</td>
<td>16.08</td>
<td>P (ie) P (ie)</td>
</tr>
<tr>
<td>● dev_mondo</td>
<td>07D_{16}</td>
<td>disrupting</td>
<td>16.11</td>
<td>P (ie) P (ie)</td>
</tr>
<tr>
<td>● interrupt_level_n (n = 1–15)</td>
<td>041_{16–04F_{16}}</td>
<td>disrupting</td>
<td>32-n (31 to 17)</td>
<td>P (ie) P (ie)</td>
</tr>
<tr>
<td>● resumable_error</td>
<td>07E_{16}</td>
<td>disrupting</td>
<td>33.3</td>
<td>P (ie) P (ie)</td>
</tr>
<tr>
<td>○ implementation_dependent_exception_n</td>
<td>070_{16–075_{16–077_{16–079_{16–07B_{16–07F_{16}}}}}}</td>
<td>—</td>
<td>V</td>
<td>— —</td>
</tr>
<tr>
<td>— nonresumable_error</td>
<td>07F_{16}</td>
<td>—</td>
<td>—</td>
<td>— —</td>
</tr>
</tbody>
</table>

* Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36-V8 on page 428), 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.

V The priority of an implementation_dependent_exception_n trap is implementation dependent (impl. dep. # 35-V8-Cs20)

‡ This exception is deprecated, because the only instructions that can generate it have been deprecated.
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</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:6</td>
<td>spill_or_fill</td>
<td>010₂ for spill traps; 011₂ for fill traps</td>
</tr>
<tr>
<td>5</td>
<td>other</td>
<td>(OTHERWIN ≠ 0)</td>
</tr>
<tr>
<td>4:2</td>
<td>wtype</td>
<td>If (other) then WSTATE.other; else WSTATE.normal</td>
</tr>
</tbody>
</table>

**FIGURE 12-4** Trap Type Encoding for Spill/Fill Traps

12.5.4 Trap Priorities

**TABLE 12-4** on page 422 and **TABLE 12-5** on page 426 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>
</tbody>
</table>
| TL < MAXPTL - 1| TL ← TL + 1

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.
  
  \[
  TL \leftarrow TL + 1
  \]

- Existing state is preserved.

- \(\text{TSTATE}[\text{TL}].\text{gl} \leftarrow \text{GL}\)
- \(\text{TSTATE}[\text{TL}].\text{ccr} \leftarrow \text{CCR}\)
- \(\text{TSTATE}[\text{TL}].\text{asi} \leftarrow \text{ASI}\)
- \(\text{TSTATE}[\text{TL}].\text{pstate} \leftarrow \text{PSTATE}\)
- \(\text{TSTATE}[\text{TL}].\text{cwp} \leftarrow \text{CWP}\)
- \(\text{TPC}[\text{TL}] \leftarrow \text{PC} \quad // \quad (\text{upper 32 bits zeroed if PSTATE.am = 1})\)
- \(\text{TNPC}[\text{TL}] \leftarrow \text{NPC} \quad // \quad (\text{upper 32 bits zeroed if PSTATE.am = 1})\) The trap type is preserved.
  
  \[TT[\text{TL}] \leftarrow \text{the trap type}\]
The Global Level register (GL) is updated. This normally provides access to a fresh set of global registers:

\[ GL \leftarrow \min (GL + 1, \text{MAXPGL}) \]

The PSTATE register is updated to a predefined state:

- PSTATE.mm is unchanged
- PSTATE.pef \(\leftarrow 1\) // if an FPU is present, it is enabled
- PSTATE.am \(\leftarrow 0\) // address masking is turned off
- PSTATE.priv \(\leftarrow 1\) // the virtual processor enters privileged mode
- PSTATE.cle \(\leftarrow\) PSTATE.tle // set endian mode for traps
- PSTATE.ie \(\leftarrow 0\) // interrupts are disabled
- PSTATE.tle is unchanged
- PSTATE.tct \(\leftarrow 0\) // trap on CTI disabled

For a register-window trap (clean_window, window spill, or window fill) only, CWP is set to point to the register window that must be accessed by the trap-handler software, that is:

```
if TT[TL] = 02416 // a clean_window trap
   then CWP \leftarrow CWP + 1
endif
```

```
if (08016 \leq TT[TL] \leq 0BF16) // window spill trap
   then CWP \leftarrow CWP + CANSAVE + 2
endif
```

```
if (0C016 \leq TT[TL] \leq 0FF16) // window fill trap
   then CWP \leftarrow CWP - 1
endif
```

For non-register-window traps, CWP is not changed.

Control is transferred into the trap table:

```
// Note that at this point, TL has already been incremented (above)
if ( (trap is to privileged mode) and (TL \leq \text{MAXPTL}) )
   then
      // the trap is handled in privileged mode
      // Note: The expression “(TL > 1)” below evaluates to the
      // value 0_2 if TL was 0 just before the trap (in which
      // case, TL = 1 now, since it was incremented above,
      // during trap entry). “(TL > 1)” evaluates to 1_2 if
      // TL was > 0 before the trap.
      PC \leftarrow \text{TBA}{63:15} :: (TL > 1) :: TT[TL] :: 0 0000_2
      NPC \leftarrow \text{TBA}{63:15} :: (TL > 1) :: TT[TL] :: 0 0100_2
   else { trap is handled in hyperprivileged mode }
endif
```
Interrupts are ignored as long as $\text{PSTATE.ie} = 0$.

**Programming Note**: State in $\text{TPC}[n]$, $\text{TNPC}[n]$, $\text{TSTATE}[n]$, and $\text{TT}[n]$ is only changed autonomously by the processor when a trap is taken while $\text{TL} = n-1$; however, software can change any of these values with a WRPR instruction when $\text{TL} = n$.

### 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 8, *Instructions*, enumerates which traps can be generated by each instruction.

The following traps are generally expected to be supported in all UltraSPARC Architecture 2005 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** [$\text{TT} = 024_{16}$-$027_{16}$] (Precise) — A SAVE instruction discovered that the window about to be used contains data from another address space; the window must be cleaned before it can be used.

  **IMPL. DEP. #102-V9**: An implementation may choose either to implement automatic cleaning of register windows in hardware or to generate a clean_window trap, when needed, so that window(s) can be cleaned by software. If an implementation chooses the latter option, then support for this trap type is mandatory.

- **cpu_mondo** [$\text{TT} = 07C_{16}$] (Disrupting) — This interrupt is generated when another virtual processor has enqueued a message for this virtual processor. It is used to deliver a trap in privileged mode, to inform privileged software that an interrupt report has been appended to the virtual processor’s CPU mondo queue. A direct message between virtual processors is sent via a CPU mondo interrupt. When the CPU mondo queue has a valid entry, a cpu_mondo exception is sent to the target virtual processor.

- **data_access_exception** [$\text{TT} = 030_{16}$] (Precise) — An exception occurred on an attempted data access.

  The conditions that may cause a data_access_exception exception are:
- **Privilege Violation** — 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.

- **Illegal Access to Noncacheable Page** — An access to a noncacheable page (TTE.cp = 0) was attempted by an atomic load-store instruction (CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA) or an LDTXA instruction.

- **Illegal Access to Page That May Cause Side Effects** — 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.

- **Invalid ASI** — An attempt was made to execute an invalid combination of instruction and ASI. See the instruction descriptions in Chapter 8 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 *data_access_exception* 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 403 and *Partial Store ASIs* on page 404). This includes the following:
    - An attempt to use a Load Twin Extended Word (LDTXA) ASI (see ASIs 1016, 1116, 1616, 1716 and 1816 (ASI_*AS_IF_USER_* on page 397) with any load alternate opcode other than LDTXA’s (which is shared by LDTWA)
    - An attempt to use a nontranslating ASI value with any load or store alternate instruction other than LDXA, LDDFA, STXA, or STDFA
    - An attempt to read from a write-only ASI-accessible register
    - An attempt to write to a read-only ASI-accessible register

- **Illegal Access to Non-Faulting-Only Page** — An attempt was made to access a non-faulting-only page (TTE.nfo = 1) by any type of load, store, or load-store instruction with an ASI other than a nonfaulting ASI (PRIMARY_NO_FAULT[_LITTLE] or SECONDARY_NO_FAULT[_LITTLE]).

---

**Forward Compatibility Note**

The next revision of the UltraSPARC Architecture is expected to replace *data_access_exception* with several more specific exceptions — one for each condition that currently can cause a *data_access_exception*. This will support slightly faster trap handling for these exceptions.

---

**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.ief = 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.flt, 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. Examples: the FPop is
  unimplemented or execution of an FPop requires software assistance to complete.
  The floating-point exception type is encoded in FSR.flt.

- **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 435.

- **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.

  **Note** — An unimplemented FPop instruction generates an
  **illegal_instruction** exception with ftt = 3, instead of an
  **illegal_instruction** exception.

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).
- A non-FPop 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 120.)
- An illegal value is present in an instruction i field.
- An illegal value is present in a field that is explicitly defined for an instruction, such as `cc2`, `cc1`, `cc0`, `fcn`, `impl`, `op2` (IMPDEP2A, IMPDEP2B), `rcond`, or `opf_cc`.
- Illegal register alignment (such as odd `rd` value in a doubleword load instruction).
- Illegal `rd` value for LDXFSR, STXFSR, or the deprecated instructions LDFSR or STFSR.
- ILLTRAP instruction.
- DONE or RETRY when `TL = 0`.

All causes of an illegal_instruction exception are described in individual instruction descriptions in Chapter 8, Instructions.

- **instruction_access_exception** [TT = 00816] (Precise) — An exception occurred on an instruction access. The conditions that may cause an instruction_access_exception exception are:
  - **Privilege Violation** — An attempt to fetch an instruction from a privileged memory page (`TTE.p = 1`) while the virtual processor was executing in nonprivileged mode.
  - **Unauthorized Access** — An attempt to fetch an instruction from a memory page which was missing “execute” permission (`TTE.ep = 0`).
  - **No-Fault Only Access** — An attempt to fetch an instruction from a memory page which was marked for access only by nonfaulting loads (`TTE.nfo = 1`).

- **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`.

  **Implementation Note** interrupt_level_14 can be caused by (1) setting SOFTINT[14] to 1, (2) occurrence of a “TICK match”, or (3) occurrence of a “STICK match” (see SOFTINT Register (ASRs 20, 21, 22) on page 77).

- **LDDF_mem_address_not_aligned** [TT = 03516] (Precise) — An attempt was made to execute an LDDF or LDDFA instruction and the effective address was not doubleword aligned. (impl. dep. #109)

- **mem_address_not_aligned** [TT = 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 397.)

- **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 00\textsubscript{16} to 7F\textsubscript{16} (inclusively).

A data access by nonprivileged or privileged software using a hyperprivileged ASI, that is, an ASI in the range 30\textsubscript{16} to 7F\textsubscript{16} (inclusively).

Execution by nonprivileged software of an instruction with a privileged operand value.

An attempt to read the TICK register by nonprivileged software when TICK.npt = 1.

An attempt to access the PIC register (using RDPIC or WRPIC) while PSTATE.priv = 0 and PCR.priv = 1.

An attempt to execute a nonprivileged instruction with an operand value requiring more privilege than available in the current privilege mode.

privileged_opcode [TT = 011\textsubscript{16}] (Precise) — An attempt was made to execute a privileged instruction while PSTATE.priv = 0.

resumable_error [TT = 07E\textsubscript{16}] (Disrupting) — There is a valid entry in the resumable error queue. This interrupt is used to inform privileged software that an error report has been appended to the resumable error queue, and the current instruction stream is in a consistent state so that execution can be resumed after the error is handled.

spill_n_normal [TT = 080\textsubscript{16}–09F\textsubscript{16}] (Precise)

spill_n_other [TT = 0A0\textsubscript{16}–0BF\textsubscript{16}] (Precise)

A SAVE or FLUSHW instruction has determined that the contents of a register window must be saved to memory.

STDF_mem_address_not_aligned [TT = 036\textsubscript{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\textsubscript{16}] (Precise) (deprecated \textsuperscript{C2}) — A TADDccTV or TSUBccTV instruction was executed, and either 32-bit arithmetic overflow occurred or at least one of the tag bits of the operands was nonzero.

trap_instruction [TT = 100\textsubscript{16}–17F\textsubscript{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\textsubscript{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\textsubscript{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\textsubscript{16}] (Precise) — The virtual processor has detected an attempt to access a virtual address specified by the VA Watchpoint register, while VA watchpoints are enabled and the address is being translated from a virtual address to a physical address. If the load or store address is not being translated.
from a virtual address (for example, the address is being treated as a real address), then a VA\_watchpoint exception will not be generated even if a match is detected between the VA Watchpoint register and a load or store address.

### 12.7.1 SPARC V9 Traps Not Used in UltraSPARC Architecture 2005

The following traps were optional in the SPARC V9 specification and are not used in UltraSPARC Architecture 2005:

- **implementation\_dependent\_exception** [TT = \(077_{16}, 07A_{16}\)] This range of implementation-dependent exceptions has been replaced by a set of architecturally-defined exceptions. (impl.dep. #35-V8-Cs20)

- **LDQF\_mem\_address\_not\_aligned** [TT = \(038_{16}\)] (Precise) — An attempt was made to execute an LDQF instruction and the effective address was word aligned but not quadword aligned. Use of this exception is implementation dependent (impl. dep. #111-V9-Cs10). A separate trap entry for this exception supports fast software emulation of the LDQF instruction when the effective address is word aligned but not quadword aligned. See Load Floating-Point on page 236. (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 316. (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

\[
\text{CLEANWIN} - \text{CANRESTORE}
\]

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 418, 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 $N_{\text{REG\_WINDOWS}}$.

- If the spill trap occurs because of a SAVE instruction (when $\text{CANSAVE} = 0$), there is an overlap window between the CWP and the next register window to be spilled:

$$\text{CWP} \leftarrow (\text{CWP} + 2) \mod N_{\text{REG\_WINDOWS}}$$

If the spill trap occurs because of a FLUSHW instruction, there can be unused windows ($\text{CANSAVE}$) in addition to the overlap window between the CWP and the window to be spilled:

$$\text{CWP} \leftarrow (\text{CWP} + \text{CANSAVE} + 2) \mod N_{\text{REG\_WINDOWS}}$$

Implementation Note | All spill traps can set CWP by using the calculation:

$$\text{CWP} \leftarrow (\text{CWP} + \text{CANSAVE} + 2) \mod N_{\text{REG\_WINDOWS}}$$

since $\text{CANSAVE}$ is 0 whenever a trap occurs because of a SAVE instruction.

- On a fill trap, the window preceding CWP must be filled:

$$\text{CWP} \leftarrow (\text{CWP} - 1) \mod N_{\text{REG\_WINDOWS}}$$

- On a clean_window trap, the window following CWP must be cleaned. Then

$$\text{CWP} \leftarrow (\text{CWP} + 1) \mod N_{\text{REG\_WINDOWS}}$$

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 442).

**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 442.
■ **Software Interrupt Register (SOFTINT)** on page 442.
■ **Interrupt Queues** on page 443.
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 SOFTINT Register (ASRs 20, 21, 22) on page 77 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 443. 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’d into the SOFTINT register. This approach allows the interrupt handler to set one or more bits in the SOFTINT register with a single instruction.

See SOFTINT_SETP Pseudo-Register (ASR 20) on page 78 for a detailed description of the SOFTINT_SET pseudo-register.
13.2.2 Clearing the Software Interrupt Register

When all interrupts scheduled for service at level $n$ have been serviced, kernel software executes a WRSOFTINT_CLR\textsuperscript{P} instruction (WRasr using ASR 21) with a ‘1’ in bit $n$ of the value written, to clear interrupt level $n$ (impl. dep. 34-V8a). The complement of the value written to the SOFTINT CLR register is effectively 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 CLR\textsuperscript{P} Pseudo-Register (ASR 21) on page 79 for a detailed description of the SOFTINT_CLR pseudo-register.

13.3 Interrupt Queues

Interrupts are indicated to privileged mode via circular interrupt queues, each with an associated trap vector. There are 4 interrupt queues, one for each of the following types of interrupts:

- Device mondos\textsuperscript{1}
- CPU mondos
- Resumable errors
- Nonresumable errors

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.

\textsuperscript{1} “mondo” is a historical term, referring to the name of the original UltraSPARC 1 bus transaction in which these interrupts were introduced
The interrupt queue registers are accessed through ASI ASI\_QUEUE (25\(_{16}\)). The ASI and address assignments for the interrupt queue registers are provided in TABLE 13-1.

**TABLE 13-1** Interrupt Queue Register ASI Assignments

<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>25(_{16}) (ASI_QUEUE)</td>
<td>3C016 (\text{RW})</td>
<td></td>
</tr>
<tr>
<td>CPU Mondo Queue Tail</td>
<td>25(_{16}) (ASI_QUEUE)</td>
<td>3C816 (R \text{ or RW}^\dagger)</td>
<td></td>
</tr>
<tr>
<td>Device Mondo Queue Head</td>
<td>25(_{16}) (ASI_QUEUE)</td>
<td>3D016 (\text{RW})</td>
<td></td>
</tr>
<tr>
<td>Device Mondo Queue Tail</td>
<td>25(_{16}) (ASI_QUEUE)</td>
<td>3D816 (R \text{ or RW}^\dagger)</td>
<td></td>
</tr>
<tr>
<td>Resumable Error Queue Head</td>
<td>25(_{16}) (ASI_QUEUE)</td>
<td>3E016 (\text{RW})</td>
<td></td>
</tr>
<tr>
<td>Resumable Error Queue Tail</td>
<td>25(_{16}) (ASI_QUEUE)</td>
<td>3E816 (R \text{ or RW}^\dagger)</td>
<td></td>
</tr>
<tr>
<td>Nonresumable Error Queue Head</td>
<td>25(_{16}) (ASI_QUEUE)</td>
<td>3F016 (\text{RW})</td>
<td></td>
</tr>
<tr>
<td>Nonresumable Error Queue Tail</td>
<td>25(_{16}) (ASI_QUEUE)</td>
<td>3F816 (R \text{ or RW}^\dagger)</td>
<td></td>
</tr>
</tbody>
</table>

^\dagger 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.
13.4 Interrupt Traps

The following interrupt traps are defined in the UltraSPARC Architecture 2005: cpu_mondo, dev_mondo, resumable_error, and nonresumable_error. See Chapter 12, Traps, for details.

UltraSPARC Architecture 2005 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.
Memory Management

An UltraSPARC Architecture Memory Management Unit (MMU) conforms to the requirements set forth in the *SPARC V9 Architecture Manual*. In particular, it supports a 64-bit virtual address space, simplified protection encoding, and multiple page sizes.

In UltraSPARC Architecture 2005, memory management is implementation-specific. Basic concepts are described in this chapter, but see the relevant processor-specific Supplement to this specification for a detailed description of a particular processor’s memory management facilities.

This appendix describes the Memory Management Unit, as observed by privileged software, in these sections:

- **Virtual Address Translation** on page 447.
- **TSB Translation Table Entry (TTE)** on page 448.
- **Translation Storage Buffer (TSB)** on page 451.

### 14.1 Virtual Address Translation

The MMUs may support up to four page sizes: 8 KBytes, 64 KBytes, 4 MBytes, and 256 MBytes. 8-KByte, 64-KByte and 4-MByte page sizes must be supported; other page sizes are optional.

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 14-1. 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.

FIGURE 14-1 Conceptual View of the MMU

14.2 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 14-2 and described in TABLE 14-1.

FIGURE 14-2 Translation Storage Buffer (TSB) Translation Table Entry (TTE)
# Chapter 14 • Memory Management

## Table 14-1

<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. <strong>Programming Note:</strong> 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 data_access_exception. An instruction fetch access to a page with the IMMU TTE.nfo = 1 results in an instruction_access_exception 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>t_addr</td>
<td>Target address from TSB (Real Address (55:13)).</td>
</tr>
</tbody>
</table>

**IMPL. DEP. #224-U3:** Physical address width support by the MMU is implementation dependent in the UltraSPARC Architecture; minimum PA width is 40 bits.

**IMPL. DEP. #238-U3:** When page offset bits for larger page sizes are stored in the TLB, it is implementation dependent whether the data returned from those fields by a Data Access read is zero or the data previously written to them.

**Data – 12**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| ie  |       | Invert Endianness. If ie = 1 for a page, accesses to the page are processed with inverse endianness from that specified by the instruction (big for little, little for big). **Note:** This bit is intended to be set to 1 primarily for noncacheable accesses. The performance of cacheable accesses may be degraded as if the access missed the D-cache. **IMPL. DEP. #.** The ie bit in the IMMU is ignored during ITLB operation. It is implementation dependent if it is implemented and how it is read and written.
Data – 11 e
Side effect. If the side-effect bit is set to 1, loads with ASI_PRIMARY_NO_FAULT, ASI_SECONDARY_NO_FAULT, and their _LITTLE variations will trap for addresses within the page, noncacheable memory accesses other than block loads and stores are strongly ordered against other e-bit accesses, and noncacheable stores are not merged. This bit should be set to 1 for pages that map I/O devices having side effects. Note, also, that the e bit causes the prefetch instruction to be treated as a nop, but does not prevent normal (hardware) instruction prefetching.

**Note:** The e bit does not force a noncacheable access. It is expected, but not required, that the cp and cv bits will be set to 0 when the e bit is set to 1. If both the cp and cv bits are set to 1 along with the e bit, the result is undefined.

**Note:** The e bit and the nfo bit are mutually exclusive; both bits should never be set to 1 in any TTE.

Data – 10 cp, Data – 9 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>I-TLB (Instruction Cache PA-indexed)</th>
<th>D-TLB (Data Cache VA-indexed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00, 01</td>
<td>Noncacheable</td>
<td>Noncacheable</td>
</tr>
<tr>
<td>10</td>
<td>Cacheable L2-cache, I-cache</td>
<td>Cacheable L2-cache</td>
</tr>
<tr>
<td>11</td>
<td>Cacheable L2-cache, I-cache</td>
<td>Cacheable L2-cache, D-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.

Data – 8 p
Privileged. If p = 1, only privileged software can access the page mapped by the TTE. If p = 1 and an access to the page is attempted by nonprivileged mode (PSTATE_priv = 0), then the MMU signals an instruction_access_exception exception or data_access_exception exception.

Data – 7 ep
Executable. If ep = 1, the page mapped by this TTE has execute permission granted. Instructions may be fetched and executed from this page. If ep = 0, an attempt to execute an instruction from this page results in an instruction_access_exception exception.

**IMPL. DEP. #____:** Some UltraSPARC Architecture ITLB implementations may not implement the ep bit, and present the instruction_access_exception exception if there is an attempt to load an ITLB entry with ep = 0 during a hardware tablewalk. In this case, the MMU miss trap handler software must also detect the ep = 0 case when the MMU miss is handled by software.
14.3 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.

14.3.1 TSB Indexing Support

Hardware TSB indexing support via TSB pointers should be provided for the TTEs.

14.3.2 TSB Cacheability

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.

---

**TABLE 14-1** TSB TTE Bit Description (3 of 3)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data – 6</td>
<td>w</td>
<td>IMPL. DEP. #Writable. If w = 1, the page mapped by this TTE has write permission granted. Otherwise, write permission is not granted</td>
</tr>
<tr>
<td>Data – 5:4</td>
<td>soft</td>
<td>Software-defined field, provided for use by the operating system. The soft field can be written with any value in the TSB. Hardware is not required to maintain this field in any TLB (or uTLB), so when it is read from the TLB (or uTLB), it may read as zero.</td>
</tr>
<tr>
<td>Data – 3:0</td>
<td>sz</td>
<td>The page size of this entry, encoded as shown below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>8 Kbyte</td>
</tr>
<tr>
<td>0001</td>
<td>64 Kbyte</td>
</tr>
<tr>
<td>0010</td>
<td>Reserved</td>
</tr>
<tr>
<td>0011</td>
<td>4 Mbyte</td>
</tr>
<tr>
<td>0100</td>
<td>Reserved</td>
</tr>
<tr>
<td>0101</td>
<td>256 Mbyte</td>
</tr>
<tr>
<td>0110</td>
<td>Reserved</td>
</tr>
<tr>
<td>0111</td>
<td>Reserved</td>
</tr>
<tr>
<td>1000-1111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
14.3.3 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 14-3. The constant \( n \) is determined by the size field in the TSB register; it can range from 512 to an implementation-dependent number.

<table>
<thead>
<tr>
<th align="left">Tag1 (8 bytes)</th>
<th align="left">Data1 (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">( \cdots )</td>
<td align="left">( 2^n ) Lines in TSB</td>
</tr>
<tr>
<td align="left">Tag( 2^n ) (8 bytes)</td>
<td align="left">Data( 2^n ) (8 bytes)</td>
</tr>
</tbody>
</table>

FIGURE 14-3 TSB Organization
# Opcode Maps

This appendix contains the UltraSPARC Architecture 2005 instruction opcode maps. Also included are the optional UltraSPARC V instruction opcode maps; UltraSPARC V opcodes are highlighted in bold face.

In this appendix and in Chapter 8, *Instructions*, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE 8-1 on page 124. For preferred substitute instructions for deprecated opcodes, see the individual opcodes in Chapter 8 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 2005 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 <em>illegal_instruction</em> exception.</td>
</tr>
<tr>
<td></td>
<td>An attempt to execute opcode will cause an <em>fp_exception_other</em> exception with FSR.flt = 3 (unimplemented_FPop).</td>
</tr>
</tbody>
</table>

An attempt to execute a reserved opcode behaves as defined in *Reserved Opcodes and Instruction Fields* on page 120.

### TABLE A-1 op(1:0)

<table>
<thead>
<tr>
<th></th>
<th>op (1:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Branches and SETHI (See TABLE A-2)</td>
<td>CALL</td>
</tr>
</tbody>
</table>

### TABLE A-2 op2(2:0) (op = 0)

<table>
<thead>
<tr>
<th></th>
<th>op2 (2:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</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 148.
2. rd = 0, imm22 = 0
### TABLE A-3  op3[5:0] (op = 10_2)  (1 of 2)

<table>
<thead>
<tr>
<th>op3 (3:0)</th>
<th>op3[5:4]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD</td>
<td>ADDcc</td>
<td>TADDcc</td>
<td>WRYD (rd = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>— (rd = 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRCCR (rd = 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRASI (rd = 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>— (rd = 4, 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<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></td>
<td>— (rd = 15 and (rs1 ≠ 0 or i ≠ 1))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>— (rd = 7 – 14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRPRP (rd = 6)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRasrPASS (7 ≤ rd ≤ 14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRPcR (rd = 16)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRPIC (rd = 17)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>— (rd = 18)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRGSR (rd = 19)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRSOFTINT_SETP (rd = 20)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRSOFTINT_CLRP (rd = 21)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRSOFTINTP (rd = 22)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRTICK_CMPRP (rd = 23)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>— (rd = 26 - 31)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>AND</td>
<td>ANDcc</td>
<td>TSUBcc</td>
<td>SAVED (fcn = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RESTORED (fcn = 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ALLCLEAN (fcn = 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OTHERW (fcn = 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NORMALW (fcn = 4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INVALW (fcn = 5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>— (fcn ≥ 6)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OR</td>
<td>ORcc</td>
<td>TADDccTVD</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>XOR</td>
<td>XORcc</td>
<td>TSUBccTVD</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SUB</td>
<td>SUBcc</td>
<td>MULSccTV</td>
<td>FPop1 (See TABLE A-5)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<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>6</td>
<td>ORN</td>
<td>ORNcc</td>
<td>SRL (x = 0), SRLX (x = 1)</td>
<td>IMPDEF1 (VIS) (See TABLE A-12)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>XNOR</td>
<td>XNORcc</td>
<td>SRA (x = 0), SRAX (x = 1)</td>
<td>IMPDEF2</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE A-3  \( \text{op3[5:0] (op = 102)} \)  (2 of 2)

<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>8</td>
<td>ADDC</td>
<td>ADDCc</td>
<td>RDY(^T) (rs1 = 0, i = 0)</td>
<td>JMPL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 1, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDCCR (rs1 = 2, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDASI (rs1 = 3, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDTICK(^{Popt}) (rs1 = 4, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDPC (rs1 = 5, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDFPRS (rs1 = 6, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDas(^{Pcor}) (7 ≤ rd ≤ 14, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MEMBAR (rs1 = 15, rd = 0, i = 1, instruction bit 12 = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 15, rd = 0, i = 1, instruction bit 12 = 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (i = 1, (rs1 = 15 or rd ≠ 0))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>STBAR(^D) (rs1 = 15, rd = 0, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 15 and rd &gt; 0 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDPCCR(^P) (rs1 = 16 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDPIC (rs1 = 17 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 18 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDGSR (rs1 = 19 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 20 or 21 and (i = 0))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDSOFTINT(^P) (rs1 = 22 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDTICK, CMPRP(^P) (rs1 = 23 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDSTICK (rs1 = 24 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDSTICK, CMPRP(^P) (rs1 = 25 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— ((rs1 = 26 – 31 and (i = 0))</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MULX</td>
<td>—</td>
<td>—</td>
<td>RETURN</td>
</tr>
<tr>
<td>A</td>
<td>UMUL(^b)</td>
<td>UMULc(^D)</td>
<td>RDPR(^T) (rs1 = 1–14 or 16)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 15 or 17 – 30)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDPISR (rs1 = 19 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (i = 1, (rs1 = 15 or rd ≠ 0))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>STBAR(^D) (rs1 = 15, rd = 0, i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 15 and rd &gt; 0 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDPCCR(^P) (rs1 = 16 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDPIC (rs1 = 17 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 18 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDGSR (rs1 = 19 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 = 20 or 21 and (i = 0))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDSOFTINT(^P) (rs1 = 22 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDTICK, CMPRP(^P) (rs1 = 23 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDSTICK (rs1 = 24 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RDSTICK, CMPRP(^P) (rs1 = 25 and i = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— ((rs1 = 26 – 31 and (i = 0))</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>SMUL(^D)</td>
<td>SMULc(^D)</td>
<td>FLUSHW</td>
<td>FLUSH</td>
</tr>
<tr>
<td>C</td>
<td>SUBC</td>
<td>SUBCc</td>
<td>MOVcc</td>
<td>SAVE</td>
</tr>
<tr>
<td>D</td>
<td>UDIVX</td>
<td>—</td>
<td>SDIVX</td>
<td>RESTORE</td>
</tr>
<tr>
<td>E</td>
<td>UDIV(^D)</td>
<td>UDIVc(^D)</td>
<td>POPC (rs1 = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (rs1 &gt; 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POPE (fcon = 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RETRY(^P) (fcon = 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (fcon = 2..15)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>— (fcon = 16..31)</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>SDIV(^D)</td>
<td>SDIVc(^D)</td>
<td>MOVr (See TABLE A-8)</td>
<td>—</td>
</tr>
<tr>
<td>op3{5:0}</td>
<td>op3{5:4}</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>LDUW</td>
<td>LDUWA\textsuperscript{PASI}</td>
<td>LDF</td>
<td>LDF\textsuperscript{PASI}</td>
</tr>
<tr>
<td>1</td>
<td>LDUB</td>
<td>LDUBA\textsuperscript{PASI}</td>
<td>LDFSR\textsuperscript{N}, LDXFSR</td>
<td>— (rd &gt; 1)</td>
</tr>
<tr>
<td>2</td>
<td>LDUH</td>
<td>LDUHA\textsuperscript{PASI}</td>
<td>LDQF</td>
<td>LQF\textsuperscript{PASI}</td>
</tr>
<tr>
<td>3</td>
<td>LDTW\textsuperscript{D}</td>
<td>LDTWA\textsuperscript{D, PASI}</td>
<td>LDDF</td>
<td>LDDF\textsuperscript{PASI}</td>
</tr>
<tr>
<td></td>
<td>— (rd odd)</td>
<td>— (rd odd)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>STW</td>
<td>STWA\textsuperscript{PASI}</td>
<td>STF</td>
<td>STF\textsuperscript{PASI}</td>
</tr>
<tr>
<td>5</td>
<td>STB</td>
<td>STBA\textsuperscript{PASI}</td>
<td>STFSR\textsuperscript{N}, STXFSR</td>
<td>— (rd &gt; 1)</td>
</tr>
<tr>
<td>6</td>
<td>STH</td>
<td>STHA\textsuperscript{PASI}</td>
<td>STQF</td>
<td>STQF\textsuperscript{PASI}</td>
</tr>
<tr>
<td>7</td>
<td>STTW\textsuperscript{D}</td>
<td>STTWA\textsuperscript{PASI}</td>
<td>STDF</td>
<td>STDF\textsuperscript{PASI}</td>
</tr>
<tr>
<td></td>
<td>— (rd odd)</td>
<td>— (rd odd)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>LDSW</td>
<td>LDSWA\textsuperscript{PASI}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>LDSB</td>
<td>LDSBA\textsuperscript{PASI}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>LDSH</td>
<td>LDSHA\textsuperscript{PASI}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>LDX</td>
<td>LDXA\textsuperscript{PASI}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>CASA\textsuperscript{PASI}</td>
</tr>
<tr>
<td>D</td>
<td>LDSTUB</td>
<td>LDSTUBA\textsuperscript{PASI}</td>
<td>PREFETCH</td>
<td>PREFETCH\textsuperscript{PASI}</td>
</tr>
<tr>
<td></td>
<td>— (fcn = 5 – 15)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>STX</td>
<td>STXA\textsuperscript{PASI}</td>
<td></td>
<td>CASA\textsuperscript{PASI}</td>
</tr>
<tr>
<td>F</td>
<td>SWAP\textsuperscript{D}</td>
<td>SWAPA\textsuperscript{D, PASI}</td>
<td></td>
<td></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>0016</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0116</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0216</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0316</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0416</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0516</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0616</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0716</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0816</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0916</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0A16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0B16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0C16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0E16–1F16</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>0016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0116</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0216</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0316</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0416</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0516</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0616</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0716</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0816</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0916</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0A16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0B16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0C16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0E16–1F16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opf[8:4]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>---------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>00_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</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>—</td>
<td>—</td>
<td>—</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>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</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>—</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>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</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>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</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}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>11_{16}=17_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>18_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>19_{16}=1F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

† Reserved variation of FMOVR
‡ bit 13 of instruction = 0
### TABLE A-7  cond[3:0]

<table>
<thead>
<tr>
<th>Cond</th>
<th>BPcc op = 0 op2 = 1</th>
<th>Bicc op = 0 op2 = 2</th>
<th>FBPfcc op = 0 op2 = 5</th>
<th>FBfcc^D op = 0 op2 = 6</th>
<th>Tcc op = 2 op3 = 3a16</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>BPLE</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>FBPU</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>FBPU</td>
<td>FBUG^D</td>
<td>TCS</td>
</tr>
<tr>
<td>6</td>
<td>BPNEG</td>
<td>BNEG^D</td>
<td>FBPG</td>
<td>FBG^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>BG^D</td>
<td>FBPU</td>
<td>FBUED</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>BGU^D</td>
<td>FBPGU</td>
<td>FBUG^D</td>
<td>TGU</td>
</tr>
<tr>
<td>D</td>
<td>BPCC</td>
<td>BCC^D</td>
<td>FBPLE</td>
<td>FBLE^D</td>
<td>TCC</td>
</tr>
<tr>
<td>E</td>
<td>BPPOS</td>
<td>BPOS^D</td>
<td>FBPU</td>
<td>FBU^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 rcond[2:0] Instruction Field

<table>
<thead>
<tr>
<th>Rcond {2:0}</th>
<th>BPr op = 0 op2 = 3</th>
<th>MOVr op = 2 op3 = 2F16</th>
<th>FMOVr op = 2 op3 = 3516</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>BRZ</td>
<td>MOVZR</td>
<td>FMOVR&lt;ls</td>
</tr>
<tr>
<td>2</td>
<td>BRLEZ</td>
<td>MOVRLZ</td>
<td>FMOVR&lt;ls</td>
</tr>
<tr>
<td>3</td>
<td>BRLZ</td>
<td>MOVRLZ</td>
<td>FMOVR&lt;ls</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>BRNZ</td>
<td>MOVRNZ</td>
<td>FMOVR&lt;ls</td>
</tr>
<tr>
<td>6</td>
<td>BRGZ</td>
<td>MOVRGZ</td>
<td>FMOVR&lt;ls</td>
</tr>
<tr>
<td>7</td>
<td>BRGEZ</td>
<td>MOVGEZ</td>
<td>FMOVR&lt;ls</td>
</tr>
</tbody>
</table>
### TABLE A-9  \( \text{cc} / \text{opf}_\text{cc} \) Fields (MOVcc and FMOVcc)

<table>
<thead>
<tr>
<th>\text{opf}_\text{cc}</th>
<th>\text{Condition Code Selected}</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc2 cc1 cc0</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>\text{fcc0}</td>
</tr>
<tr>
<td>0 0 1</td>
<td>\text{fcc1}</td>
</tr>
<tr>
<td>0 1 0</td>
<td>\text{fcc2}</td>
</tr>
<tr>
<td>0 1 1</td>
<td>\text{fcc3}</td>
</tr>
<tr>
<td>1 0 0</td>
<td>\text{icc}</td>
</tr>
<tr>
<td>1 0 1</td>
<td>—</td>
</tr>
<tr>
<td>1 1 0</td>
<td>\text{xcc}</td>
</tr>
<tr>
<td>1 1 1</td>
<td>—</td>
</tr>
</tbody>
</table>

### TABLE A-10  \( \text{cc} \) Fields (FBPfcc, FCMP, and FCMPE)

<table>
<thead>
<tr>
<th>cc1 cc0</th>
<th>\text{Condition Code Selected}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>\text{fcc0}</td>
</tr>
<tr>
<td>0 1</td>
<td>\text{fcc1}</td>
</tr>
<tr>
<td>1 0</td>
<td>\text{fcc2}</td>
</tr>
<tr>
<td>1 1</td>
<td>\text{fcc3}</td>
</tr>
</tbody>
</table>

### TABLE A-11  \( \text{cc} \) Fields (BPcc and Tcc)

<table>
<thead>
<tr>
<th>cc1 cc0</th>
<th>\text{Condition Code Selected}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>\text{icc}</td>
</tr>
<tr>
<td>0 1</td>
<td>—</td>
</tr>
<tr>
<td>1 0</td>
<td>\text{xcc}</td>
</tr>
<tr>
<td>1 1</td>
<td>—</td>
</tr>
</tbody>
</table>
### TABLE A-12  IMPDEP1: opf[8:0] for VIS opcodes (op = 10₂, op3 = 36₁₀)

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EDGE8</td>
<td>ARRAY8</td>
<td>FCMPLE16</td>
<td>—</td>
<td>—</td>
<td>FPADD16</td>
<td>FZERO</td>
<td>FAND</td>
</tr>
<tr>
<td>1</td>
<td>EDGE8N</td>
<td>—</td>
<td>—</td>
<td>FMUL 8x16</td>
<td>—</td>
<td>FPADD16S</td>
<td>FZEROS</td>
<td>FANDS</td>
</tr>
<tr>
<td>2</td>
<td>EDGE8L</td>
<td>ARRAY16</td>
<td>FCMPNE16</td>
<td>—</td>
<td>—</td>
<td>FPADD32</td>
<td>FNOR</td>
<td>FXNOR</td>
</tr>
<tr>
<td>3</td>
<td>EDGE8LN</td>
<td>—</td>
<td>—</td>
<td>FMUL 8x16AU</td>
<td>—</td>
<td>FPADD32S</td>
<td>FNORS</td>
<td>FXNORS</td>
</tr>
<tr>
<td>4</td>
<td>EDGE16</td>
<td>ARRAY32</td>
<td>FCMPLE32</td>
<td>—</td>
<td>FPADD16S</td>
<td>FANDNOT2</td>
<td>FSRC1</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>EDGE16N</td>
<td>—</td>
<td>—</td>
<td>FMUL 8x16AL</td>
<td>—</td>
<td>FPADD16S</td>
<td>FANDNOT2S</td>
<td>FSRC1S</td>
</tr>
<tr>
<td>6</td>
<td>EDGE16L</td>
<td>—</td>
<td>FCMPNE32</td>
<td>—</td>
<td>FPADD32</td>
<td>FNOT2</td>
<td>FORNOT2</td>
<td>—</td>
</tr>
<tr>
<td>7</td>
<td>EDGE16LN</td>
<td>—</td>
<td>—</td>
<td>FMUL 8Ux16</td>
<td>—</td>
<td>FPADD32</td>
<td>FNOT2S</td>
<td>FORNOT2S</td>
</tr>
<tr>
<td>8</td>
<td>EDGE32</td>
<td>ALIGN ADDRESS</td>
<td>FCMPGT16</td>
<td>—</td>
<td>FPADD32</td>
<td>FZEROS</td>
<td>FERS</td>
<td>—</td>
</tr>
<tr>
<td>9</td>
<td>EDGE32N</td>
<td>BMASK</td>
<td>—</td>
<td>FMULD 8Ux16</td>
<td>—</td>
<td>FPADD32</td>
<td>FZEROS</td>
<td>FERS</td>
</tr>
<tr>
<td>A</td>
<td>EDGE32L</td>
<td>ALIGN ADDRESS _LITTLE</td>
<td>FCMPGT16</td>
<td>—</td>
<td>FPADD16S</td>
<td>FZEROS</td>
<td>FERS</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>EDGE32LN</td>
<td>—</td>
<td>—</td>
<td>FPADD16S</td>
<td>FZEROS</td>
<td>FAND</td>
<td>FONE</td>
<td>—</td>
</tr>
<tr>
<td>C</td>
<td>—</td>
<td>—</td>
<td>FCMPGT32</td>
<td>—</td>
<td>RSHUFFLE</td>
<td>—</td>
<td>FXOR</td>
<td>FOR</td>
</tr>
<tr>
<td>D</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>FPADD32</td>
<td>—</td>
<td>FPADD16S</td>
<td>FZEROS</td>
<td>FAND</td>
</tr>
<tr>
<td>E</td>
<td>—</td>
<td>—</td>
<td>FCMPGT32</td>
<td>—</td>
<td>RSHUFFLE</td>
<td>—</td>
<td>FXOR</td>
<td>FOR</td>
</tr>
<tr>
<td>F</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>opf (3:0)</td>
<td>00–07</td>
<td>08–1F</td>
<td>09–1F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>2</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>
</tr>
<tr>
<td>4</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>
</tr>
<tr>
<td>6</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>
</tr>
<tr>
<td>8</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>
</tr>
<tr>
<td>A</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>
</tr>
<tr>
<td>C</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>
</tr>
<tr>
<td>E</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>
</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 465.

The appendix contains these sections:
- Definition of an Implementation Dependency on page 463.
- Hardware Characteristics on page 464.
- Implementation Dependency Categories on page 464.
- List of Implementation Dependencies on page 465.

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 2005 Compliance with SPARC V9 Architecture* on page 23.

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>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whether an instruction complies with UltraSPARC Architecture 2005 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>24, 48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An UltraSPARC Architecture implementation may contain from 72 to 640 general-purpose 64-bit <em>R</em> registers. This corresponds to a grouping of the registers into ( \text{MAXPGL} + 1 ) sets of global <em>R</em> registers plus a circular stack of ( \text{N_REG_WINDOWS} ) sets of 16 registers each, known as register windows. The number of register windows present ( \text{(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>119</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 <em>fp_exception_other</em> exception with <em>FSR.ftt</em> = unfinished_FPop or <em>FSR.ftt</em> = unimplemented_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>
</tbody>
</table>

APPENDIX B • Implementation Dependencies 465
### Table B-1 SPARC V9 Implementation Dependencies (2 of 9)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-V8</td>
<td>f</td>
<td>I/O registers privileged status</td>
<td>27</td>
</tr>
<tr>
<td>7-V8</td>
<td>t</td>
<td>I/O register definitions</td>
<td>27</td>
</tr>
<tr>
<td>8-V8-</td>
<td>t</td>
<td>RDAsr/WRasr target registers</td>
<td>29, 67, 285, 353</td>
</tr>
<tr>
<td>Cs20</td>
<td>Cs20</td>
<td>Ancillary state registers (ASRs) in the range 0–27 that are not defined in UltraSPARC Architecture 2005 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>29, 67, 285, 353</td>
</tr>
<tr>
<td>Cs20</td>
<td>Cs20</td>
<td>Whether each of the implementation-dependent read/write ancillary state register instructions (for ASRs 28–31) is privileged 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>
<tr>
<td>18-V8-</td>
<td>f</td>
<td>Nonstandard IEEE 754-1985 results</td>
<td>60</td>
</tr>
<tr>
<td>V8-</td>
<td></td>
<td>UltraSPARC Architecture 2005 implementations do not implement a nonstandard floating-point mode. FSR.ns is a reserved bit; it always reads as 0 and writes to it are ignored.</td>
<td></td>
</tr>
<tr>
<td>Ms10</td>
<td>Ms10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19-V8</td>
<td>a</td>
<td>FPU version, FSR.ver</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits 19:17 of the FSR, FSR.ver, identify one or more implementations of the FPU architecture.</td>
<td></td>
</tr>
<tr>
<td>20-V8-21-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>22-V8</td>
<td>f</td>
<td>FPU tem, cexc, and aexc</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An UltraSPARC Architecture implementation implements the tem, cexc, and aexc fields in hardware, conformant to IEEE Std 754-1983.</td>
<td></td>
</tr>
<tr>
<td>23-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>24-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>25-V8</td>
<td>f</td>
<td>RDPR of FQ with nonexistent FQ</td>
<td>63, 289</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>26-V8-28-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>29-V8</td>
<td>t</td>
<td>Address space identifier (ASI) definitions</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 388 and Block Load and Store ASIs on page 403 for details.</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE B-1  SPARC V9 Implementation Dependencies (3 of 9)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>30-V8</td>
<td>f</td>
<td>ASI address decoding</td>
<td>109</td>
</tr>
<tr>
<td>Cu3</td>
<td>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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31-V8</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>Ms10</td>
<td>Whether any restartable deferred traps (and associated deferred-trap queues) are present is implementation dependent.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33-V8</td>
<td>f</td>
<td>Trap precision</td>
<td>417</td>
</tr>
<tr>
<td>Cs10</td>
<td>In an UltraSPARC Architecture implementation, all exceptions that occur as the result of program execution are precise.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34-V8</td>
<td>f</td>
<td>Interrupt clearing</td>
<td>443</td>
</tr>
<tr>
<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 443).</td>
<td></td>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<td>35-V8</td>
<td>t</td>
<td>Implementation-dependent traps</td>
<td>420</td>
</tr>
<tr>
<td>Cs20</td>
<td>Trap type (TT) values 060_{16}–07F_{16} were reserved for implementation_dependent_exception_n exceptions in SPARC V9 but are now all defined as standard UltraSPARC Architecture exceptions.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36-V8</td>
<td>f</td>
<td>Trap priorities</td>
<td>428</td>
</tr>
<tr>
<td>a:</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>
<td></td>
</tr>
<tr>
<td>b:</td>
<td>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) are undefined or are guaranteed to remain unchanged.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44-V8</td>
<td>f</td>
<td>Data access FPU trap</td>
<td>238</td>
</tr>
<tr>
<td>Cs10</td>
<td>a:</td>
<td>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) are undefined or are guaranteed to remain unchanged.</td>
<td></td>
</tr>
<tr>
<td>b:</td>
<td>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>
<td></td>
</tr>
<tr>
<td>45-V8–46-V8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

APPENDIX B • Implementation Dependencies 467
RDasr instructions with \( r_d \) in the range 28–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For an RDasr instruction with \( r_s1 \) 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)
- whether an attempt to execute the instruction causes an \texttt{illegal_instruction} exception

WRasr instructions with \( r_d \) in the range 26–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRasr instruction with \( r_d \) in the range 26–31, the following are implementation dependent:

- the interpretation of bits 18:0 in the instruction
- the operation(s) performed (for example, \texttt{xor}) to generate the value written to the ASR
- whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20)
- whether an attempt to execute the instruction causes an \texttt{illegal_instruction} exception

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| 47-V8 Cs20 | t | RDasr instructions with \( r_d \) in the range 28–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For an RDasr instruction with \( r_s1 \) 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)  
- whether an attempt to execute the instruction causes an \texttt{illegal_instruction} exception | 256 |
| 48-V8 Cs20 | t | WRasr instructions with \( r_d \) in the range 26–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRasr instruction with \( r_d \) in the range 26–31, the following are implementation dependent:  
- the interpretation of bits 18:0 in the instruction  
- the operation(s) performed (for example, \texttt{xor}) to generate the value written to the ASR  
- whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20)  
- whether an attempt to execute the instruction causes an \texttt{illegal_instruction} exception | 354 |
| 55-V8 Cs10 | f | Tininess detection  
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. | 66 |
| 56–100 | Reserved |  |
| 101-V9 CS10 | v | Maximum trap level (\texttt{MAXPTL})  
The architectural parameter \( \texttt{MAXPTL} \) 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 \( \texttt{MAXPTL} = \texttt{MAXPGL} \) (see impl. dep. #401-S10). Architecturally, \( \texttt{MAXPTL} \) must be \( \geq 2 \). | 94, 96 |
| 102-V9 | f | Clean windows trap  
An implementation may choose either to implement automatic “cleaning” of register windows in hardware or to generate a \texttt{clean_window} trap, when needed, for window(s) to be cleaned by software. | 431 |
The following aspects of the PREFETCH and PREFETCHA instructions are implementation dependent:

- The attributes of the block of memory prefetched: its size (minimum = 64 bytes) and its alignment (minimum = 64-byte alignment)
- Whether each defined prefetch variant is implemented (1) as a NOP, (2) with its full semantics, or (3) with common-case prefetching semantics
- Whether and how variants 16, 18, 19 and 24–31 are implemented; if not implemented, a variant must execute as a NOP

The following aspects of the PREFETCH and PREFETCHA instructions used to be (but are no longer) implementation dependent:

- While in nonprivileged mode, an attempt to reference an ASI in the range $0_{16}..7F_{16}$ by a PREFETCHA instruction executes as a NOP; specifically, it does not cause a privileged_action exception.
- PREFETCH and PREFETCHA have no observable effect in privileged code
- While in privileged mode, an attempt to reference an ASI in the range $30_{16}..7F_{16}$ by a PREFETCHA instruction executes as a NOP (specifically, it does not cause a privileged_action exception)

The following aspects of the Prefetch instructions are implementation dependent:

- If an accurate count cannot always be returned when TICK is read, any inaccuracy should be small, bounded, and documented.
- An implementation may implement fewer than 63 bits in TICK.counter; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any upper bits not implemented must read as 0.

The IMPDEP2A instructions are completely implementation dependent.

Implementation-dependent aspects include their operation, the interpretation of bits 29:25 and 18:0 in their encodings, and which (if any) exceptions they may cause.

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.

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.

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.

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.
TABLE B-1  SPARC V9 Implementation Dependencies  (6 of 9)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| 109-| V9-Cs10  | **LDDF**<sub>(A)_mem_address_not_aligned</sub>  
LDDF requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid \((i = 1\) or instruction bits 12:5 = 0) LDDF instruction may cause an \(LDDF\_mem\_address\_not\_aligned\) exception. In this case, the trap handler software shall emulate the LDDF instruction and return.  
(In an UltraSPARC Architecture processor, the \(LDDF\_mem\_address\_not\_aligned\) exception occurs in this case and trap handler software emulates the LDDF instruction) | 102, 237, 434 |
| 110-| V9-Cs10  | **STDF**<sub>(A)_mem_address_not_aligned</sub>  
STDF requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid \((i = 1\) or instruction bits 12:5 = 0) STDF instruction may cause an \(STDF\_mem\_address\_not\_aligned\) exception. In this case, the trap handler software must emulate the STDF instruction and return.  
(In an UltraSPARC Architecture processor, the \(STDF\_mem\_address\_not\_aligned\) exception occurs in this case and trap handler software emulates the STDF instruction) | 102, 317, 435 |
|     |          | **LDDF**<sub>(A)_mem_address_not_aligned</sub>  
LDDFA requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid \((i = 1\) or instruction bits 12:5 = 0) LDDFA instruction may cause an \(LDDF\_mem\_address\_not\_aligned\) exception. In this case, the trap handler software shall emulate the LDDFA instruction and return.  
(In an UltraSPARC Architecture processor, the \(LDDF\_mem\_address\_not\_aligned\) exception occurs in this case and trap handler software emulates the LDDFA instruction) | 240 |
|     |          | **STDF**<sub>(A)_mem_address_not_aligned</sub>  
STDFA requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid \((i = 1\) or instruction bits 12:5 = 0) STDFA instruction may cause an \(STDF\_mem\_address\_not\_aligned\) exception. In this case, the trap handler software must emulate the STDFA instruction and return.  
(In an UltraSPARC Architecture processor, the \(STDF\_mem\_address\_not\_aligned\) exception occurs in this case and trap handler software emulates the STDFA instruction) | 320 |
APPENDIX B • Implementation Dependencies

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-f</td>
<td>V9-Cs10</td>
<td><strong>LDQF(A)_mem_address_not_aligned</strong></td>
<td>103, 102, 237, 436</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 <strong>LDQF_mem_address_not_aligned</strong> exception. In this case, the trap handler software must emulate the LDQF instruction and return. (In an UltraSPARC Architecture processor, the <strong>LDQF_mem_address_not_aligned</strong> exception occurs in this case and trap handler software emulates the LDQF instruction) (this exception does not occur in hardware on UltraSPARC Architecture 2005 implementations, because they do not implement the LDQF instruction in hardware)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 <strong>LDQF_mem_address_not_aligned</strong> exception. In this case, the trap handler software must emulate the LDQF instruction and return. (In an UltraSPARC Architecture processor, the <strong>LDQF_mem_address_not_aligned</strong> exception occurs in this case and trap handler software emulates the LDQFA instruction) (this exception does not occur in hardware on UltraSPARC Architecture 2005 implementations, because they do not implement the LDQFA instruction in hardware)</td>
<td></td>
</tr>
<tr>
<td>112-f</td>
<td>V9-Cs10</td>
<td><strong>STQF(A)_mem_address_not_aligned</strong></td>
<td>103, 317, 436</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 <strong>STQF_mem_address_not_aligned</strong> exception. In this case, the trap handler software must emulate the STQF instruction and return. (In an UltraSPARC Architecture processor, the <strong>STQF_mem_address_not_aligned</strong> exception occurs in this case and trap handler software emulates the STQF instruction) (this exception does not occur in hardware on UltraSPARC Architecture 2005 implementations, because they do not implement the STQF instruction in hardware)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 <strong>STQF_mem_address_not_aligned</strong> exception. In this case, the trap handler software must emulate the STQFA instruction and return. (In an UltraSPARC Architecture processor, the <strong>STQF_mem_address_not_aligned</strong> exception occurs in this case and trap handler software emulates the STQFA instruction) (this exception does not occur in hardware on UltraSPARC Architecture 2005 implementations, because they do not implement the STQFA instruction in hardware)</td>
<td></td>
</tr>
</tbody>
</table>
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.

Coherence and atomicity of memory operations

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent.

Implementation-dependent memory model

An implementation may choose to identify certain addresses and use an implementation-dependent memory model for references to them.

FLUSH latency

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.

Input/output (I/O) semantics

The semantic effect of accessing I/O registers is implementation dependent.

Implicit ASI when TL > 0

In SPARC V9, when TL > 0, the implicit ASI for instruction fetches, loads, and stores is implementation dependent. In all UltraSPARC Architecture implementations, when TL > 0, the implicit ASI for instruction fetches is ASI_NUCLEUS; loads and stores will use ASI_NUCLEUS if PSTATE.cle = 0 or ASI_NUCLEUS_LITTLE if PSTATE.cle = 1.

Address masking

(1) When PSTATE.am = 1, only the less-significant 32 bits of the PC register are stored in the specified destination register(s) in CALL, JMPL, and RDPC instructions, while the more-significant 32 bits of the destination registers(s) are set to 0. (2) When PSTATE.am = 1, during a trap, only the less-significant 32 bits of the PC and NPC are stored (respectively) to TPC[TL] and TNPC[TL]: the more-significant 32 bits of TPC[TL] and TNPC[TL] are set to 0.
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.

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 2005 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.

### UltraSPARC Architecture Implementation Dependencies

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>126-V9</td>
<td>Register Windows State registers width</td>
<td>82</td>
</tr>
<tr>
<td>Ms10</td>
<td>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 Management Instructions on page 116. 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 2005 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.</td>
<td></td>
</tr>
<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>303</td>
</tr>
<tr>
<td>204-U3-CS10</td>
<td>DCR bits 5:3 and 0</td>
<td>75</td>
</tr>
<tr>
<td>205-U3-Cs10</td>
<td>Instruction Trap Register</td>
<td></td>
</tr>
</tbody>
</table>
### Software intervention after instruction-induced error

Precision of the trap to signal an instruction-induced error of which recovery requires software intervention is implementation dependent.

### Error logging registers’ information

The information that the error logging registers preserves beyond the reset induced by an ERROR signal is implementation dependent.

### Trap with fatal error

This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.

### AFSR .priv

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.

### Base address generation

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.

### data_access_exception trap

The causes of a data_access_exception trap are implementation dependent in UltraSPARC Architecture 2005.
Data Watchpoint Reliability

Data Watchpoint traps are completely implementation-dependent in UltraSPARC Architecture processors.

This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.

Conditions for \textit{fp\_exception\_other} with \textit{unfinished\_FPop}

The conditions under which an \textit{fp\_exception\_other} exception with floating-point trap type of \textit{unfinished\_FPop} can occur are implementation dependent. An implementation may cause \textit{fp\_exception\_other} with \textit{unfinished\_FPop} under a different (but specified) set of conditions.

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[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 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.

PCR accessibility when \textit{PSTATE.priv} = 0

In an UltraSPARC Architecture implementation, PCR is never accessible to nonprivileged software. Specifically, when a virtual processor is operating in nonprivileged mode (\textit{PSTATE.priv} = 0), an attempt to access PCR (using an RDPCR or a WRPCR instruction) results in a \textit{privileged\_opcode} exception.

Reserved.

This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.

Reserved.

LDDFA with ASI C0\textsubscript{16}–C5\textsubscript{16} or C8\textsubscript{16}–CD\textsubscript{16} and misaligned memory address

If an LDDFA opcode is used with an ASI of C0\textsubscript{16}–C5\textsubscript{16} or C8\textsubscript{16}–CD\textsubscript{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 exception generated is \textit{data\_access\_exception, mem\_address\_not\_aligned}, or \textit{LDDF\_mem\_address\_not\_aligned}.

Reserved.

Attempted access to ASI registers with LDTWA

If an LDTWA instruction referencing a non-memory ASI is executed, it generates a \textit{data\_access\_exception} exception.

Attempted access to ASI registers with STTWA

If an STTWA instruction referencing a non-memory ASI is executed, it generates a \textit{data\_access\_exception} exception.
### TABLE B-2  UltraSPARC Architecture Implementation Dependencies  (4 of 6)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| 302-U4-Cs10 | Scratchpad registers  
An UltraSPARC Architecture processor includes eight privileged Scratchpad registers (64 bits each, read/write accessible). | 405  |
| 303-U4-CS10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005. | —    |
| 305-U4-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005. | —    |
| 306-U4-Cs10 | Trap type generated upon attempted access to noncacheable page with LDTXA  
When an LDTXA instruction attempts access from an address that is not mapped to cacheable memory space, a *data_access_exception* exception is generated. | 251  |
| 307-U4-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005. | —    |
| 308-U3-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005. | —    |
| 309-U4-Cs10 | Reserved. | —    |
| 311–319 | Reserved. | —    |
| 327–399 | Reserved. | —    |
| 400-S10 | Global Level register (GL) implementation  
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. | 96   |
| 401-S10 | Maximum Global Level (MAXPGL)  
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 \(\geq 2\). | 94, 96|
| 403-S10 | Setting of “dirty” bits in FPRS  
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. The specific conditions under which a dirty bit is set are implementation dependent. | 74, 74|
| 404-S10 | Scratchpad registers 4 through 7  
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 *data_access_exception* exception). | 405  |
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 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 $2^{64} - 1$.

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.

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 LDBLOCKFs to non-cacheable locations are (a) strictly ordered, (b) not strictly ordered and cause an illegal_instruction exception, or (c) not strictly ordered and silently execute without causing an exception (option (c) is strongly discouraged)
- Whether the MMU ignores the side-effect bit (TTE.e) for LDBLOCKF accesses (in which case, LDBLOCKFs behave as if TTE.e = 0)
- 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

<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>26</td>
</tr>
<tr>
<td>409-S10-Cs20</td>
<td>FLUSH instruction and memory consistency</td>
<td>175</td>
</tr>
<tr>
<td>410-S10</td>
<td>Block Load behavior</td>
<td>233</td>
</tr>
</tbody>
</table>

**TABLE B-2 UltraSPARC Architecture Implementation Dependencies (5 of 6)**
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 314).
- 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 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).
- Whether the MMU ignores the side-effect bit (TTE.e) for STBLOCKF accesses (in which case, STBLOCKFs behave as if TTE.e = 0)
- Any other restrictions on the behavior of STBLOCKF, as described in implementation-specific documentation.

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 LDTXA instruction (the recommended behavior) or only on accesses to the first 8 bytes.

If (1) TSTATE[TL].pstate.am = 1 and (2) a DONE or RETRY instruction is executed (which sets PSTATE.am to ‘1’ by restoring the value from TSTATE[TL].pstate.am to PSTATE.am), it is implementation dependent whether the DONE or RETRY instruction masks (zeroes) the more-significant 32 bits of the values it places into PC and NPC.

Reserved for UltraSPARC Architecture 2005

Reserved for UltraSPARC Architecture 2006

Reserved.
Assembly Language Syntax

This appendix supports Chapter 8, Instructions. Each instruction description in Chapter 8 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 479.
- **Syntax Design** on page 485.
- **Synthetic Instructions** on page 486.

## C.1 Notation Used

The notations defined here are also used in the assembly language syntax descriptions in Chapter 8, 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, “immiasi” would be replaced by a number in the range 0 to 255 (the value of the immiasi 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, \texttt{reg}\textsubscript{rs2} is a \texttt{reg} (register name) whose binary value will be placed in the \texttt{rs2} field of the resulting instruction.

### C.1.1 Register Names

**\texttt{reg}**. A \texttt{reg} is an integer register name. It can have any of the following values:
\begin{itemize}
  \item \%r0–\%r31
  \item \%g0–\%g7 (global registers; same as \%r0–\%r7)
  \item \%o0–\%o7 (out registers; same as \%r8–\%r15)
  \item \%l0–\%l7 (local registers; same as \%r16–\%r23)
  \item \%i0–\%i7 (in registers; same as \%r24–\%r31)
  \item \%fp (frame pointer; conventionally same as \%i6)
  \item \%sp (stack pointer; conventionally same as \%o6)
\end{itemize}

Subscripts identify the placement of the operand in the binary instruction as one of the following:
\begin{itemize}
  \item \texttt{reg}\textsubscript{rs1} (rs1 field)
  \item \texttt{reg}\textsubscript{rs2} (rs2 field)
  \item \texttt{reg}\textsubscript{rd} (rd field)
\end{itemize}

**\texttt{freg}**. An \texttt{freg} is a floating-point register name. It may have the following values:
\%f0, \%f1, \%f2–\%f63

See Floating-Point Registers on page 52.

Subscripts further identify the placement of the operand in the binary instruction as one of the following:
\begin{itemize}
  \item \texttt{freg}\textsubscript{rs1} (rs1 field)
  \item \texttt{freg}\textsubscript{rs2} (rs2 field)
  \item \texttt{freg}\textsubscript{rs3} (rs3 field)
  \item \texttt{freg}\textsubscript{rd} (rd field)
\end{itemize}

**\texttt{asr\_reg}**. An \texttt{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:
\begin{itemize}
  \item \texttt{asr\_reg}\textsubscript{rs1} (rs1 field)
  \item \texttt{asr\_reg}\textsubscript{rd} (rd field)
\end{itemize}

\footnote{In actual usage, the \%sp, \%fp, \%gn, \%or, \%ln, and \%in forms are preferred over \%rn.}
\textit{i_or_x_cc.} An \textit{i_or_x_cc} specifies a set of integer condition codes, those based on either the 32-bit result of an operation (\textit{icc}) or on the full 64-bit result (\textit{xcc}). It may have either of the following values:
\begin{itemize}
  \item \%icc
  \item \%xcc
\end{itemize}

\textit{fccn.} An \textit{fccn} specifies a set of floating-point condition codes. It can have any of the following values:
\begin{itemize}
  \item \%fcc0
  \item \%fcc1
  \item \%fcc2
  \item \%fcc3
\end{itemize}

\section*{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:
\begin{itemize}
  \item \%asi \hspace{1cm} Address Space Identifier (ASI) register
  \item \%canrestore \hspace{1cm} Restorable Windows register
  \item \%cansave \hspace{1cm} Savable Windows register
  \item \%ccr \hspace{1cm} Condition Codes register
  \item \%cleanwin \hspace{1cm} Clean Windows register
  \item \%cwp \hspace{1cm} Current Window Pointer (CWP) register
  \item \%fprs \hspace{1cm} Floating-Point Registers State (FPRS) register
  \item \%fsr \hspace{1cm} Floating-Point State register
  \item \%gsr \hspace{1cm} General Status Register (GSR)
  \item \%otherwin \hspace{1cm} Other Windows (OTHERWIN) register
  \item \%pc \hspace{1cm} Program Counter (PC) register
  \item \%pcr \hspace{1cm} Performance Control Register (PCR)
  \item \%pic \hspace{1cm} Performance Instrumentation Counters
  \item \%pil \hspace{1cm} Processor Interrupt Level register
  \item \%pstate \hspace{1cm} Processor State register
  \item \%softint \hspace{1cm} Soft Interrupt register
  \item \%softint_clr \hspace{1cm} Soft Interrupt register (clear selected bits)
  \item \%softint_set \hspace{1cm} Soft Interrupt register (set selected bits)
  \item \%sys_tick \hspace{1cm} System Timer (STICK) register
  \item \%sys_tick_cmpr \hspace{1cm} System Timer Compare (STICK_CMPR) register
  \item \%tba \hspace{1cm} Trap Base Address (TBA) register
  \item \%tick \hspace{1cm} Cycle count (TICK) register
\end{itemize}
%tick_cmpr  Timer Compare (TICK_CMPR) register
tl          Trap Level (TL) register
%tnpc       Trap Next Program Counter (TNPC) register
%tpc        Trap Program Counter (TPC) register
%tstate     Trap State (TSTATE) register
%tt         Trap Type (TT) register
%wstate     Window State register
%y          Y register

The following special symbol names are unary operators that perform the functions described:

%uhi         Extracts bits 63:42 (high 22 bits of upper word) of its operand
%ulo or %hm  Extracts bits 41:32 (low-order 10 bits of upper word) of its operand
%hi or %lm   Extracts bits 31:10 (high-order 22 bits of low-order word) of its operand
%lo          Extracts bits 9:0 (low-order 10 bits) of its operand

Certain predefined value names appear in the syntax table in typewriter font. They must be written exactly as they are shown, including the leading sharp sign (#). The value names and the 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>#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</td>
<td>(1116)</td>
</tr>
<tr>
<td>#n_reads_strong</td>
<td>20</td>
<td>(1416)</td>
</tr>
<tr>
<td>#one_read_strong</td>
<td>21</td>
<td>(1516)</td>
</tr>
<tr>
<td>#n_writes_strong</td>
<td>22</td>
<td>(1616)</td>
</tr>
<tr>
<td>#one_write_strong</td>
<td>23</td>
<td>(1716)</td>
</tr>
</tbody>
</table>

for MEMBAR instruction “mmask” field

| #LoadLoad                      | 0116  |                        |
| #StoreLoad                     | 0216  |                        |
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
- immasi: 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:

<table>
<thead>
<tr>
<th>Value Name in Assembly Language</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>#LoadStore</td>
<td>0416</td>
<td>for MEMBAR instruction “cmask” field</td>
</tr>
<tr>
<td>#StoreStore</td>
<td>0816</td>
<td></td>
</tr>
<tr>
<td>#Lookaside</td>
<td>1016</td>
<td></td>
</tr>
<tr>
<td>#MemIssue</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>#Sync</td>
<td>4016</td>
<td></td>
</tr>
</tbody>
</table>
\begin{align*}
reg_{rs1} & \quad \text{(equivalent to } reg_{rs1} + \%g0) \\
reg_{rs1} + \text{simm13} & \\
reg_{rs1} - \text{simm13} & \\
\text{simm13} & \quad \text{(equivalent to } \%g0 + \text{simm13)} \\
\text{simm13} + reg_{rs1} & \quad \text{(equivalent to } reg_{rs1} + \text{simm13)}
\end{align*}

**address** Can be any of the following:

\begin{align*}
reg_{rs1} & \quad \text{(equivalent to } reg_{rs1} + \%g0) \\
reg_{rs1} + \text{simm13} & \\
reg_{rs1} - \text{simm13} & \\
\text{simm13} & \quad \text{(equivalent to } \%g0 + \text{simm13)} \\
\text{simm13} + reg_{rs1} & \quad \text{(equivalent to } reg_{rs1} + \text{simm13)} \\
reg_{rs1} + reg_{rs2} &
\end{align*}

**membar_mask** Is the following:

\begin{itemize}
\item \text{const7} \quad \text{A constant that can be represented in 7 bits. Typically, this is an expression involving the logical OR of some combination of} \\
\text{#Lookaside, #MemIssue, #Sync, #StoreStore, #LoadStore,} \\
\text{#StoreLoad,} \text{ and #LoadLoad.}
\end{itemize}

**prefetchFcn** (**prefetch function**) Can be any of the following:

\begin{itemize}
\item \text{#n_reads}
\item \text{#one_read}
\item \text{#n_writes}
\item \text{#one_write}
\item \text{#page}
\end{itemize}

\begin{itemize}
\item \text{0–31}
\end{itemize}

**regaddr** (**register-only address**) Can be any of the following:

\begin{align*}
reg_{rs1} & \quad \text{(equivalent to } reg_{rs1} + \%g0) \\
reg_{rs1} + reg_{rs2} &
\end{align*}

**regOrImm** (**register or immediate value**) Can be either of:

\begin{itemize}
\item \text{reg}_{rs2}
\item \text{simm13}
\end{itemize}

**regOrImm10** (**register or immediate value**) Can be either of:
\[ \text{reg} \text{or}\_\text{imm11} \]  (register or immediate value) Can be either of:

\[ \text{reg} \text{or}\_\text{imm11} \]

\[ \text{reg} \text{or}\_\text{shcnt} \]  (register or shift count value) Can be any of:

\[ \text{reg} \text{or}\_\text{shcnt} \]

\[ \text{software\_trap\_number} \]  Can be any of the following:

\[ \text{reg}\_\text{rs1} \]  (equivalent to \( \text{reg}\_\text{rs1} + \%g0 \))

\[ \text{reg}\_\text{rs1} + \text{reg}\_\text{rs2} \]

\[ \text{reg}\_\text{rs1} + \text{simm8} \]

\[ \text{reg}\_\text{rs1} - \text{simm8} \]  (equivalent to \( \%g0 + \text{simm8} \))

\[ \text{simm8} \]  (equivalent to \( \%g0 + \text{simm8} \))

\[ \text{simm8} + \text{reg}\_\text{rs1} \]  (equivalent to \( \text{reg}\_\text{rs1} + \text{simm8} \))

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 (in a Load, Store, CASA, CASXA, LDSTUB[A], or SWAP[A] instruction) is always indicated by square brackets ([]); a reference to the address of a memory location (such as in a JMPL, CALL, or SETHI) is specified directly, without square brackets.

C.3 Synthetic Instructions

TABLE C-2 describes the mapping of a set of synthetic (or “pseudo”) instructions to actual instructions. These synthetic instructions are provided by the SPARC V9 assembler for the convenience of assembly language programmers.

**Note:** Synthetic instructions should not be confused with “pseudo ops,” which typically provide information to the assembler but do not generate instructions. Synthetic instructions always generate instructions; they provide more mnemonic syntax for standard SPARC V9 instructions.

**TABLE C-2** Mapping Synthetic to SPARC V9 Instructions  (1 of 3)

<table>
<thead>
<tr>
<th>Synthetic Instruction</th>
<th>SPARC V9 Instruction(s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>reg$<em>{rs1}$, reg$</em>{or_imm}$</td>
<td>subcc reg$<em>{rs1}$, reg$</em>{or_imm}$, %g0</td>
</tr>
<tr>
<td>jmp</td>
<td>address</td>
<td>jmpl address, %g0</td>
</tr>
<tr>
<td>call</td>
<td>address</td>
<td>jmpl address, %o7</td>
</tr>
<tr>
<td>iprefetch label</td>
<td>bn,a,pt %xcc,label</td>
<td>Instruction prefetch.</td>
</tr>
<tr>
<td>tst</td>
<td>reg$_{rs1}$</td>
<td>orcc %g0, reg$_{rs1}$, %g0</td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td>jmpl %i7+8, %g0</td>
</tr>
<tr>
<td>retl</td>
<td></td>
<td>jmpl %o7+8, %g0</td>
</tr>
<tr>
<td>restore</td>
<td></td>
<td>restore %g0, %g0, %g0</td>
</tr>
<tr>
<td>save</td>
<td></td>
<td>save %g0, %g0, %g0</td>
</tr>
<tr>
<td>setuw</td>
<td>value, reg$_{rd}$</td>
<td>sethi %hi(value), reg$_{rd}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— or —</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— or —</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sethi %hi(value), reg$_{rdi}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or reg$<em>{rd}$, %lo(value), reg$</em>{rd}$</td>
</tr>
<tr>
<td>set</td>
<td>value, reg$_{rd}$</td>
<td>sethi %hi(value), reg$_{rd}$</td>
</tr>
<tr>
<td>setsw</td>
<td>value, reg$_{rd}$</td>
<td>sethi %hi(value), reg$_{rd}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— or —</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>or %g0, value, reg</td>
<td>or %g0, value, reg</td>
<td>(When (4096 \leq \text{value} \leq 4095)).</td>
</tr>
<tr>
<td>sra reg, %g0, reg</td>
<td>sra reg, %g0, reg</td>
<td>(Otherwise, if (\text{value} &lt; 0) and ((\text{value} &amp; 3FF_{16}) = 0))</td>
</tr>
<tr>
<td>sethi %hi(value), reg</td>
<td>sethi %hi(value), reg</td>
<td>(Otherwise, if (\text{value} \leq 0)).</td>
</tr>
<tr>
<td>or reg, %lo(value), reg</td>
<td>or reg, %lo(value), reg</td>
<td></td>
</tr>
<tr>
<td>sethi %hi(value), reg</td>
<td>sethi %hi(value), reg</td>
<td></td>
</tr>
<tr>
<td>or reg, %lo(value), reg</td>
<td>or reg, %lo(value), reg</td>
<td></td>
</tr>
<tr>
<td>sra reg, %g0, reg</td>
<td>sra reg, %g0, reg</td>
<td>Warning: do not use \text{setsw} in the delay slot of a CTI.</td>
</tr>
<tr>
<td>setx value, reg, reg</td>
<td>setx value, reg, reg</td>
<td>Create 64-bit constant.</td>
</tr>
<tr>
<td>or %lo(value), reg</td>
<td>or %lo(value), reg</td>
<td></td>
</tr>
<tr>
<td>sllx reg, 32, reg</td>
<td>sllx reg, 32, reg</td>
<td></td>
</tr>
<tr>
<td>sethi %hi(value), reg</td>
<td>sethi %hi(value), reg</td>
<td></td>
</tr>
<tr>
<td>or reg, %lo(value), reg</td>
<td>or reg, %lo(value), reg</td>
<td></td>
</tr>
<tr>
<td>signx reg, reg</td>
<td>sra reg, %g0, reg</td>
<td>Sign-extend 32-bit value to 64 bits.</td>
</tr>
<tr>
<td>neg reg, reg</td>
<td>neg reg, reg</td>
<td></td>
</tr>
<tr>
<td>cas [reg1], reg, reg</td>
<td>cas [reg1], reg, reg</td>
<td></td>
</tr>
<tr>
<td>casl [reg1], reg, reg</td>
<td>casl [reg1], reg, reg</td>
<td></td>
</tr>
<tr>
<td>casxl [reg1], reg, reg</td>
<td>casxl [reg1], reg, reg</td>
<td></td>
</tr>
<tr>
<td>inc reg, reg</td>
<td>add reg, 1, reg</td>
<td>Increment by 1.</td>
</tr>
<tr>
<td>inc const13, reg</td>
<td>add reg, const13, reg</td>
<td>Increment by (\text{const13}).</td>
</tr>
<tr>
<td>incc reg, reg</td>
<td>addcc reg, 1, reg</td>
<td>Incre by (\text{const13}); set (\text{icc} &amp; \text{xcc}).</td>
</tr>
<tr>
<td>incc reg, reg</td>
<td>addcc reg, 1, reg</td>
<td></td>
</tr>
<tr>
<td>dec reg, reg</td>
<td>sub reg, 1, reg</td>
<td>Decrement by 1.</td>
</tr>
<tr>
<td>dec const13, reg</td>
<td>sub reg, const13, reg</td>
<td>Decrement by (\text{const13}).</td>
</tr>
<tr>
<td>dec reg, reg</td>
<td>sub reg, 1, reg</td>
<td></td>
</tr>
<tr>
<td>dec const13, reg</td>
<td>sub reg, const13, reg</td>
<td></td>
</tr>
<tr>
<td>decc reg, reg</td>
<td>subcc reg, 1, reg</td>
<td>Decrement by 1; set (\text{icc} &amp; \text{xcc}).</td>
</tr>
<tr>
<td>Synthetic Instruction</td>
<td>SPARC V9 Instruction(s)</td>
<td>Comment</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>decc</td>
<td>const13, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>Decr by const13; set ICC &amp; XCC.</td>
</tr>
<tr>
<td>bst</td>
<td>reg_or_imm, reg&lt;sub&gt;rs1&lt;/sub&gt;</td>
<td>Bit test.</td>
</tr>
<tr>
<td>bset</td>
<td>reg_or_imm, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>Bit set.</td>
</tr>
<tr>
<td>bc1r</td>
<td>reg_or_imm, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>Bit clear.</td>
</tr>
<tr>
<td>bto1g</td>
<td>reg_or_imm, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>Bit toggle.</td>
</tr>
<tr>
<td>clr</td>
<td>reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>Clear (zero) register.</td>
</tr>
<tr>
<td>clr&lt;sub&gt;b&lt;/sub&gt;</td>
<td>[address]</td>
<td>Clear byte.</td>
</tr>
<tr>
<td>clr&lt;sub&gt;r&lt;/sub&gt;</td>
<td>[address]</td>
<td>Clear half-word.</td>
</tr>
<tr>
<td>clr&lt;sub&gt;x&lt;/sub&gt;</td>
<td>[address]</td>
<td>Clear word.</td>
</tr>
<tr>
<td>clruw</td>
<td>reg&lt;sub&gt;rs1&lt;/sub&gt;, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>Clear extended word.</td>
</tr>
<tr>
<td>clruw&lt;sub&gt;u&lt;/sub&gt;</td>
<td>reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>Copy and clear upper word.</td>
</tr>
<tr>
<td>mov</td>
<td>reg_or_imm, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>Clear upper word.</td>
</tr>
<tr>
<td>mov&lt;sub&gt;y&lt;/sub&gt;</td>
<td>%y, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>mov&lt;sub&gt;asr&lt;/sub&gt;</td>
<td>%asr, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>reg_or_imm, %y</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>reg_or_imm, %asr</td>
<td></td>
</tr>
</tbody>
</table>
Index

A
a (annul) instruction field
branch instructions, 142, 143, 145, 148, 163, 165
accesses
cacheable, 367
I/O, 367
restricted ASI, 371
with side effects, 367, 378
accumulated exception (aexc) field of FSR register, 63, 418, 466
ADD instruction, 134
ADDC instruction, 134
ADDCcc instruction, 134, 306
ADDCcc instruction, 134
address
operand syntax, 484
space identifier (ASI), 387
address mask (am) field of PSTATE register
description, 92
address space, 20
address space identifier (ASI), 366
appended to memory address, 25, 100
architecturally specified, 371
bypass, 388
bypassing, 93
changed in, 406
changed in UA
ASI_LD_TWIX_NUCLEUS_LITTLE, 406
ASI_LDTX_N, 406
ASI_LDTX_NL, 406
ASI_REAL, 406
ASI_REAL_IO, 406
ASI_REAL_IO_LITTLE, 406
ASI_REAL_LITTLE, 406
definition, 7
electing address space information, 101
explicit, 108
explicitly specified in instruction, 108
implicit, See implicit ASIs
nontranslating, 12, 257, 333
nontranslating ASIs, 388
with prefetch instructions, 279
restricted, 371, 387
privileged, 371
restriction indicator, 71
SPARC V9 address, 369
translating ASIs, 388
unrestricted, 371, 387
address space identifier (ASI) register
for load/store alternate instructions, 71
address for explicit ASI, 108
and LDDA instruction, 239, 255
and LDSTUBA instruction, 248
load integer from alternate space
instructions, 229
with prefetch instructions, 279
for register-immediate addressing, 371
restoring saved state, 154, 294
saving state, 409
and STDA instruction, 332
store floating-point into alternate space
instructions, 319
store integer to alternate space instructions, 308
and SWAPA instruction, 337
after trap, 30
and TSTATE register, 88
and write state register instructions, 354
addressing modes, 20
ADDX instruction (SPARC V8), 134
ADDXcc instruction (SPARC V8), 134
alias
  floating-point registers, 52
aliased, 7
ALIGNADDRESS instruction, 135
ALIGNADDRESS_LITTLE instruction, 135
alignment
  data (load/store), 26, 102, 369
doubleword, 26, 102, 369
extended-word, 102
halfword, 26, 102, 369
instructions, 26, 102, 369
integer registers, 254, 256
memory, 369, 434
quadword, 26, 102, 369
word, 26, 102, 369
ALLCLEAN instruction, 136
alternate space instructions, 27, 71
ancillary state registers (ASRs)
  access, 67
  assembly language syntax, 480
  I/O register access, 27
  possible registers included, 286, 355
  privileged, 29, 466
  reading/writing implementation-dependent
  processor registers, 29, 466
writing to, 354
AND instruction, 137
ANDcc instruction, 137
ANDN instruction, 137
ANDNcc instruction, 137
annul bit
  in branch instructions, 148
  in conditional branches, 163
  annulled branches, 148
application program, 7, 67
architectural direction note, 4
architecture, meaning for SPARC V9, 19
arithmetic overflow, 70
ARRAY16 instruction, 138
ARRAY32 instruction, 138
ARRAY8 instruction, 138
ASI
  invalid, and data_access_exception, 432
ASI register, 67
ASI. See address space identifier (ASI)
ASI_AIUP, 390, 399
ASI_AL, 390, 399
ASI_ALUP, 390, 399
ASI_ALUS, 390, 399
ASI_ALUSL, 390, 399
ASI_AS_IF_USER*, 92
ASI_AS_IF_USER_NONFAULT_LITTLE, 372
ASI_AS_IF_USER_PRIMARY, 390, 398
ASI_AS_IF_USER_PRIMARY_LITTLE, 372, 390, 399, 432
ASI_AS_IF_USER_SECONDARY, 372, 390, 398, 432
ASI_AS_IF_USER_SECONDARY_LITTLE, 372, 390, 399, 432
ASI_AS_IF_USER_SECONDARY_NOFAULT_LITTLE, 372
ASI_BLK_AIUP, 390, 399
ASI_BLK_AIUPL, 390, 399
ASI_BLK_AIUS, 390, 399
ASI_BLK_AIUSL, 390, 399
ASI_BLK_P, 395
ASI_BLK_PL, 396
ASI_BLK_S, 395
ASI_BLK_SL, 396
ASI_BLOCK_AS_IF_USER_PRIMARY, 390, 398
ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE, 390, 399
ASI_BLOCK_AS_IF_USER_SECONDARY, 390, 398
ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE, 390, 399
ASI_BLOCK_PRIMARY, 395
ASI_BLOCK_PRIMARY_LITTLE, 396
ASI_BLOCK_SECONDARY, 395
ASI_BLOCK_SECONDARY_LITTLE, 396
ASI_FL16_P, 394
ASI_FL16_PL, 395
ASI_FL16_PRIMARY, 394
ASI_FL16_PRIMARY_LITTLE, 395
ASI_FL16_S, 394
ASI_FL16_SL, 395
ASI_FL16_SECONDARY, 394
ASI_FL16_SECONDARY_LITTLE, 395
ASI_FL8_P, 394
ASI_FL8_PL, 395
ASI_FL8_PRIMARY, 394
ASI_FL8_PRIMARY_LITTLE, 395
ASI_FL8_S, 394
ASI_FL8_SL, 395
ASI_FL8_SECONDARY, 394
ASI_FL8_SECONDARY_LITTLE, 395
ASI_LD_TWINX_AS_IF_USER_PRIMARY, 391,
Index

ASI_LD_TWINX_AS_IF_USER_PRIMARY_LITTLE, 392, 401
ASI_LD_TWINX_AS_IF_USER_SECONDARY, 391, 401
ASI_LD_TWINX_AS_IF_USER_SECONDARY_LITTLE, 392, 401
ASI_LD_TWINX_NUCLEUS, 392, 401, 406
ASI_LD_TWINX_NUCLEUS_L, 369
ASI_LD_TWINX_NUCLEUS_LITTLE, 393, 401, 406
ASI_LD_TWINX_PRIMARY, 395, 403
ASI_LD_TWINX_PRIMARY_LITTLE, 395, 403
ASI_LD_TWINX_REAL, 392, 402
ASI_LD_TWINX_REAL_LITTLE, 392, 402
ASI_LD_TWINX_REAL_REAL_L, 369
ASI_LD_TWINX_SECONDARY, 395, 403
ASI_LD_TWINX_SECONDARY_LITTLE, 395, 403
ASI_LDTX_AIU, 250, 391, 401
ASI_LDTX_AIU_L, 250, 401
ASI_LDTX_AIUPL, 392
ASI_LDTX_AIUS, 250, 401
ASI_LDTX_AIUS_L, 392, 401
ASI_LDTX_N, 250, 392, 406
ASI_LDTX_NL, 250, 393, 401, 406
ASI_LDTX_P, 250, 395
ASI_LDTX_PL, 250, 395
ASI_LDTX_R, 402
ASI_LDTX_REAL, 250, 392
ASI_LDTX_REAL_L, 392, 402
ASI_LDTX_S, 250, 395
ASI_LDTX_SL, 250, 395
ASI_MMU_CONTEXTID, 391
ASI_N, 389
ASI_NL, 389
ASI_NUCLEUS, 108, 389
ASI_NUCLEUS_LITTLE, 108, 389
ASI_NUCLEUS_QUAD_LDD, 406
ASI_NUCLEUS_QUAD_LDD_L, 406
ASI_NUCLEUS_QUAD_LDD_LITTLE, 406
ASI_P, 393
ASI_PHY_BYPASS_EC_WITH_EBIT_L, 406
ASI_PHY_BYPASS_EC_WITH_EBIT_LITTLE, 406
ASI_PHY_BYPASS_EC_WITH_EBIT_LITTLE, 406
ASI_PHYS_USE_EC, 406
ASI_PHYS_USE_EC_L, 406
ASI_PHYS_USE_EC_LITTLE, 406
ASI_PL, 393
ASI_PNF, 393
ASI_PNFL, 393
ASI_PRIMARY, 108, 371, 372, 393
ASI_PRIMARY_LITTLE, 108, 371, 393
ASI_PRIMARY_NO_FAULT, 368, 384, 393, 432
ASI_PRIMARY_NO_FAULT_LITTLE, 368, 384, 393, 432
ASIPRIMARY_NO_FAULT, 368, 384
ASI PRIMARY_NO_FAULT_LITTLE, 368, 384
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASIPRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASIPRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI_PRIMARY_NO_FAULT_LITTLE, 368
ASI_PRIMARY_NO_FAULT, 368
ASI.Primary_NO_FAULT_LITTLE, 368
atomic
  memory operations, 251, 380, 381
  store doubleword instruction, 330, 332
  store instructions, 307, 308
atomic load-store instructions
  compare and swap, 151
  load-store unsigned byte, 247, 337
  load-store unsigned byte to alternate space, 248
  simultaneously addressing doublewords, 336
  swap R register with alternate space
    memory, 337
  swap R register with memory, 151, 336
atomicity, 368, 472

B
BA instruction, 142, 143, 459
BCC instruction, 142, 459
bclrg synthetic instruction, 488
BCS instruction, 142, 459
BE instruction, 142, 459
Berkeley RISCs, 22
BG instruction, 142, 459
BGE instruction, 142, 459
BGU instruction, 142, 459
Bicc instructions, 142, 453
big-endian, 7
big-endian byte order, 26, 90, 103
binary compatibility, 22
BL instruction, 459
BLD, See LDBLOCKF instruction
BLE instruction, 142, 459
BLEU instruction, 142, 459
block load instructions, 53, 232, 403
block store instructions, 53, 312, 403
blocked byte formatting, 139
BMASK instruction, 144
BN instruction, 142, 459
BNE instruction, 142, 459
BNEG instruction, 142, 459
BP instruction, 459
BPA instruction, 145, 459
BPCC instruction, 145, 459
BPcc instructions, 70, 71, 145, 460
BPCS instruction, 145, 459
BPE instruction, 145, 459
BPG instruction, 145, 459
BPGE instruction, 145, 459
BPGU instruction, 145, 459
BPL instruction, 145, 459
BPLE instruction, 145, 459
BPLEU instruction, 145, 459
BPN instruction, 145, 459
BPNE instruction, 145, 459
BPNEG instruction, 145, 459
BPOS instruction, 142, 459
BPPOS instruction, 145, 459
BP instructions, 148, 459
BPVC instruction, 145, 459
BPVS instruction, 145, 459
branch
  annulled, 148
  delayed, 99
  elimination, 115, 116
  fcc-conditional, 163, 165
  icc-conditional, 143
  instructions
    on floating-point condition codes, 162
    on floating-point condition codes with prediction, 164
    on integer condition codes with prediction (BPcc), 145
    on integer condition codes, See Bicc instructions
      when contents of integer register match
        condition, 148
        prediction bit, 148
        unconditional, 142, 146, 163, 165
        with prediction, 20
    BRGEZ instruction, 148
    BRGZ instruction, 148
    BRLEZ instruction, 148
    BRLZ instruction, 148
    BRNZ instruction, 148
    BRZ instruction, 148
      bset synthetic instruction, 488
      BSHUFFLE instruction, 144
      BST, See STBLOCKF instruction
      btog synthetic instruction, 488
      btst synthetic instruction, 488
    BVC instruction, 142, 459
    BVS instruction, 142, 459
    bypass ASIs, 388
    byte, 7
      addressing, 108
      data format, 33
      order, 26
order, big-endian, 26
order, little-endian, 26
byte order
    big-endian, 90
    implicit, 90
    in trap handlers, 417
little-endian, 90

C

cache
    coherency protocol, 367
data, 375
instruction, 375
miss, 284
    nonconsistent instruction cache, 375
cacheable accesses, 366
caching, TSB, 451
CALL instruction
    description, 150
displacement, 29
does not change CWPL, 50
    and JMPL instruction, 226
writing address into \( R_{15} \), 52
call synthetic instruction, 486
CANRESTORE (restorable windows) register, 83
    and clean_window exception, 117
    and CLEANWIN register, 83, 85, 437
counting windows, 85
decmented by RESTORE instruction, 290
decmented by SAVED instruction, 300
detecting window underflow, 50
if registered window was spilled, 291
incremented by SAVE instruction, 298
modified by NORMALW instruction, 272
modified by OTHERW instruction, 274
range of values, 82, 473
RESTORE instruction, 117
specification for RDPR instruction, 288
specification for WRPR instruction, 356
window underflow, 437
CANSAVE (savable windows) register, 83
decmented by SAVE instruction, 298
detecting window overflow, 50
FLUSHW instruction, 177
if equals zero, 116
incremented by RESTORE, 290
incremented by SAVED instruction, 300
range of values, 82, 473
SAVE instruction, 438
specification for RDPR instruction, 288
specification for WRPR instruction, 356
window overflow, 436
CAS synthetic instruction, 381
CASA instruction, 151
    32-bit compare-and-swap, 380
    alternate space addressing, 27
    and \textit{data access exception} (noncacheable page)
    exception, 432
    atomic operation, 247
    hardware primitives for mutual exclusion of
    CASA, 379
    in multiprocessor system, 248, 336, 337
    \( R \) register use, 101
    word access (memory), 102
cash synthetic instructions, 487
CASX synthetic instruction, 380, 381
CASXA instruction, 151
    64-bit compare-and-swap, 380
    alternate space addressing, 27
    and \textit{data access exception} (noncacheable page)
    exception, 432
    atomic operation, 248
doubleword access (memory), 102
    hardware primitives for mutual exclusion of
    CASA, 379
    in multiprocessor system, 247, 248, 336, 337
    \( R \) register use, 101
catastrophic error exception, 410
cc0 instruction field
    branch instructions, 145, 165
    floating point compare instructions, 169
    move instructions, 264, 460
cc1 instruction field
    branch instructions, 145, 165
    floating point compare instructions, 169
    move instructions, 264, 460
cc2 instruction field
    move instructions, 264, 460
CCR (condition codes) register, 69
    32-bit operation (icc) bit of condition field, 70, 71
    64-bit operation (xicc) bit of condition field, 70, 71
ADD instructions, 134
ASR for, 67
carry (c) bit of condition fields, 70
icc field, \textit{See} CCR.icc field
MULScc instruction, 268
negative (n) bit of condition fields, 70
overflow bit (v) in condition fields, 70
restored by RETRY instruction, 154, 294
saved after trap, 409
saving after trap, 30
TSTATE register, 88
write instructions, 354
xcc field, See CCR.xcc field
zero (z) bit of condition fields, 70

CCR.xcc field
add instructions, 134, 339
bit setting for signed division, 350
bit setting for signed/unsigned multiply, 351
bit setting for unsigned division, 349
branch instructions, 143, 146, 264
integer subtraction instructions, 335
logical operation instructions, 137, 273, 358
MULScc instruction, 268
Tcc instruction, 343

CCR.xcc field
add instructions, 134, 339
bit setting for signed/unsigned multiply, 351
bit setting for unsigned division, 349
branch instructions, 143, 146, 264
logical operation instructions, 137, 273, 358
subtract instructions, 335
Tcc instruction, 343

clean register window, 298, 431
clean window, 8
and window traps, 86, 436
CLEANWIN register, 85
definition, 437
number is zero, 117
trap handling, 438
clean_window exception, 83, 117, 299, 431, 437, 468

CLEANWIN (clean windows) register, 83
CANSAVE instruction, 117
clean window counting, 83
incremented by trap handler, 438
range of values, 82, 473
specification for RDPR instruction, 288
specification for WRPR instruction, 356
specifying number of available clean windows, 437
value calculation, 85
clock cycle, counts for virtual processor, 72
clock tick registers, See TICK and STICK registers
clock-tick register (TICK), 435
cmn synthetic instruction, 488
cmp synthetic instruction, 335, 486
coherence, 8
between processors, 472
data cache, 375
domain, 367
memory, 368
unit, memory, 369
compare and swap instructions, 151
comparison instruction, 110, 335
compatibility note, 4
completed (memory operation), 8
compliant SPARC V9 implementation, 23
cond instruction field
branch instructions, 143, 145, 163, 165
floating point move instructions, 180
move instructions, 264
condition codes
adding, 339
effect of compare-and-swap instructions, 152
extended integer (xcc), 71
floating-point, 163
icc field, 70
integer, 69
results of integer operation (icc), 71
subtracting, 335, 345
trapping on, 343
xcc field, 70
condition codes register, See CCR register
conditional branches, 143, 163, 165
conditional move instructions, 30
conforming SPARC V9 implementation, 23
const22 instruction field of ILLTRAP instruction, 222
constants, generating, 302
context, 8
nucleus, 176
cell identifier, 370
control transfer
pseudo-control-transfer via WRPR to
PSTATE.am, 93
cell-transfer instructions (CTIs), 28, 154, 294
conventions
font, 2
notational, 2
conversion
between floating-point formats instructions, 218
floating-point to integer instructions, 216, 363
integer to floating-point instructions, 173, 221
planar to packed, 206
copyback, 8
CPI, 8
CPU, pipeline draining, 82, 86
cpu_mondo exception, 431
cross-call, 8
CTI, 8, 16
current exception (cexc) field of FSR register, 64, 119, 466
current window, 8
current window pointer register, See CWP register
current_little_endian (cle) field of PSTATE register, 90, 371
CWP (current window pointer) register and instructions
   CALL and JMPL instructions, 50
   FLUSHW instruction, 177
   RDPR instruction, 288
   RESTORE instruction, 117, 290
   SAVE instruction, 116, 290, 298
   WRPR instruction, 356
   and traps
   after spill trap, 438
   after spill/fill trap, 30
   on window trap, 438
   saved by hardware, 409
CWP (current window pointer) register, 82
clean windows, 84
definition, 8
incremented/decremented, 49, 290, 298
overlapping windows, 49
range of values, 82, 473
restored during RETRY, 154, 294
specifying windows for use without cleaning, 437
and TSTATE register, 88

D
D superscript on instruction name, 124
d16hi instruction field
   branch instructions, 148
d16lo instruction field
   branch instructions, 148
data
   access, 8
   cache coherence, 375
   conversion between SIMD formats, 41
   flow order constraints
   memory reference instructions, 374
   register reference instructions, 373
formats
   byte, 33
   doubleword, 33
   halfword, 33
   Int16 SIMD, 42
   Int32 SIMD, 42
   quadword, 33
   tagged word, 33
   Uint8 SIMD, 42
   word, 33
types
   floating-point, 33
   signed integer, 33
   unsigned integer, 33
   width, 33
Data Cache Unit Control register, See DCU CR
data_access_exception (invalid ASI) exception
   with load alternate instructions, 230
data_access_exception exception, 431
   with compare-and-swap instructions, 153
   with LD instructions, 228
   with LSHORTF instructions, 231, 234
   with LDTXA instructions, 252
   with load instructions, 238, 254, 257
   with load instructions and ASIs, 241, 401, 402, 403, 404, 405
   with store instructions and ASIs, 241, 401, 402, 403, 404, 405
   with STPARTIALF instructions, 327
   with SWAPA instruction, 338
DCTI couple, 115
DCTI instructions, 8
   behavior, 99
   RETURN instruction effects, 296
dec synthetic instructions, 487
deccccg synthetic instructions, 487
defered trap, 413
   distinguishing from disrupting trap, 415
   floating-point, 289
   restartable
      implementation dependency, 414
      software actions, 414
delay instruction
   and annul field of branch instruction, 163
   annulling, 29
   conditional branches, 165
   DONE instruction, 154
executed after branch taken, 148
following delayed control transfer, 29
RETRY instruction, 294
RETURN instruction, 296
unconditional branches, 165
with conditional branch, 146
delayed branch, 99
delayed control transfer, 148
delayed CTI, See DCTI
denormalized number, 8
deprecated, 8
deprecated exceptions
tag_overflow, 435
deprecated instructions
FBA, 162
FBE, 162
FBG, 162
FBGE, 162
FBL, 162
FBLE, 162
FBLG, 162
FBN, 162
FNBE, 162
FBU, 162
FBU, 162
FBU, 162
FBUE, 162
FBUGE, 162
FBUG, 162
FBUGL, 162
LDFSR, 243
LDTW, 253
LDTWA, 255
MULScc, 69, 268
RDY, 67, 69, 285
SDIV, 69, 348
SDIVcc, 69, 348
SMUL, 69, 351
SMULLcc, 69, 351
STFSR, 323
STTW, 330
STTW, 332
SWAP, 336
SWAPA, 337
TADDCcTV, 340
TSU BecTV, 346
UDIV, 69, 348
UDIVcc, 69, 348
UMUL, 69, 351
UMULLcc, 69, 351

WRY, 67, 69, 353
dev_mon
dispt9 instruction field
branch instructions, 145, 165
disp22 instruction field
branch instructions, 142, 163
disp30 instruction field
word displacement (CALL), 150
dispatch, 9
disrupting trap, 415
divide instructions, 28, 270, 348
division_by_zero exception, 111, 270, 433
division-by-zero bits of FSR.aexc/FSR.cexc fields, 66
DONE instruction, 154
effect on TNPC register, 87
effect on TSTATE register, 88
generating illegal_instruction exception, 434
modifying CCCR.xcX conditioXcodes, 70
return from trap, 409
return from trap handler with different GL value, 97
target address, 29
double, 9
doubleword, 9
addressing, 106
alignment, 26, 102, 369
data format, 33
definition, 9

E
EDGE16 instruction, 156
EDGE16L instruction, 156
EDGE16LN instruction, 158
EDGE16N instruction, 158
EDGE32 instruction, 156
EDGE32L instruction, 156
EDGE32N instruction, 158
EDGE8 instruction, 156
EDGE8L instruction, 156
EDGE8LN instruction, 158
EDGE8N instruction, 158
emulating multiple unsigned condition codes, 116
enable floating-point
See FPRS register, fef field
See PSTATE register, pef field
even parity, 9
exception, 9
exceptions
  See also individual exceptions
catastrophic error, 410
causing traps, 409
clean_window, 431, 468
cpu_mondo, 431
data_access_exception, 431
definition, 410
dev_mondo, 432
division_by_zero, 433
fill_n_normal, 433
fill_n_other, 433
fp_disabled
  and GSR, 76
fp_disabled, 433
fp_exception_ieee_754, 433
fp_exception_other, 433
htrap_instruction, 433
illegal_instruction, 433
instruction_access_exception, 434, 434
interrupt_level_14
  and SOFTINT.int_level, 78
  and STICK_CMPR.stick_cmpr, 81
  and TICK_CMPR.tick_cmpr, 80
interrupt_level_14, 434
interrupt_level_15
  and SOFTINT.int_level, 78
interrupt_level_n
  and SOFTINT.register, 77
  and SOFTINT.int_level, 78
interrupt_level_n, 416, 434
LDDF_mem_address_notAligned, 434
LDQF_mem_address_notAligned, 436
mem_address_notAligned, 434
nonresumable_error, 434
pending, 31
privileged_action, 434
privileged_opcode
  and access to register-window PR state registers, 81, 86, 95, 97
  and access to SOFTINT, 77
  and access to SOFTINT_CLR, 79
  and access to SOFTINT_SET, 78
  and access to STICK_CMPR, 81
  and access to TICK_CMPR, 79
privileged_opcode, 435
resumable_error, 435
spill_n_normal, 299, 435
spill_n_other, 299, 435
STDF_mem_address_notAligned, 435
STQF_mem_address_notAligned, 436
tag_overflow (deprecated), 436
trap_instruction, 435
unimplemented_LDTW, 435
unimplemented_STTW, 435
VA_watchpoint, 435
execute_unit, 373
execute_state
  trap processing, 429
explicit ASI, 9, 108, 389
extended word, 9
  addressing, 106
F
  F registers, 9, 24, 119, 359, 418
  FABSd instruction, 159, 457, 458
  FABSq instruction, 159, 457, 458
  FABSs instruction, 159
  FADD, 160
  FADDd instruction, 160
  FADDq instruction, 160
  FADDs instruction, 160
  FALIGNDATA instruction, 161
  FAND instruction, 214
  FANDNOT1 instruction, 214
  FANDNOT1S instruction, 214
  FANDNOT2 instruction, 214
  FANDNOT2S instruction, 214
  FANDS instruction, 214
  FB instruction, 162, 163, 459
  FBE instruction, 162, 459
  FBcc instructions, 58, 162, 433, 453, 459
  FBG instruction, 162, 459
  FBGE instruction, 162, 459
  FBPL instruction, 162, 459
  FBLE instruction, 162, 459
  FBLG instruction, 162, 459
  FBNE instruction, 162, 163, 459
  FBNE instruction, 162, 459
  FBO instruction, 162, 459
  FBN instruction, 162, 163, 459
  FBP instruction, 164, 165, 459
  FBPE instruction, 164, 459
  FBPcc instructions, 58, 164, 453, 459, 460
  FBPG instruction, 164, 459
  FBPG instruction, 164, 459
  FBPL instruction, 164, 459

Index  9
FBPLE instruction, 164, 459
FBPLG instruction, 164, 459
FBPN instruction, 164, 165, 459
FBPNE instruction, 164, 459
FBPO instruction, 164, 459
FBPU instruction, 164, 459
FBPU instruction, 164, 459
FBPU instruction, 164, 459
FBPU instruction, 164, 459
FBPUE instruction, 164, 459
FBPUG instruction, 164, 459
FBPUGE instruction, 164, 459
FBPUL instruction, 164, 459
FBPULE instruction, 164, 459
FBU instruction, 162, 459
FBUE instruction, 162, 459
FBUG instruction, 162, 459
FBUGE instruction, 162, 459
FBUL instruction, 162, 459
FBULE instruction, 162, 459
fcc-conditional branches, 163, 165
FCC instruction field
DONE instruction, 154
PREFETCH, 278
RETRY instruction, 294
FDIVd instruction, 171
FDIVq instruction, 171
FDIVq instructions, 171
FDIVq instruction, 171
FDIVq instruction, 194
FDTOi instruction, 216, 363
FDTOq instruction, 218
FDTOs instruction, 218
FDTOx instruction, 218, 458
fef field of FPRS register, 73
and access to GSR, 76
and fp_disabled exception, 433
branch operations, 163, 165
byte permutation, 144
cmparison operations, 167, 170
data movement operations, 265
enabling FPU, 92
floating-point operations, 159, 160, 171, 173, 178,
183, 186, 194, 196, 215, 216, 218, 220, 221, 237,
239, 243, 245
integer arithmetic operations, 205, 210
logical operations, 211, 212, 214
memory operations, 234
read operations, 287, 304, 314
special addressing operations, 135, 161, 317, 323,
327, 329, 355
fef, See FPRS register, fef field
FEXPAND instruction, 172
FEXPAND operation, 172
fill handler, 291
fill register window, 433
overflow/underflow, 50
RESTORE instruction, 85, 290, 437
RESTORED instruction, 118, 292, 438
RETRY instruction, 438
selection of, 437
trap handling, 437, 438
trap vectors, 291
window state, 85
fill_n_normal exception, 291, 297, 433, 433
fill_n_other exception, 291, 297, 433
FIToD instruction, 173
FIToQ instruction, 173
FIToQ instruction, 173
fixed values, 223
fixed-point scaling, 189
floating point
absolute value instructions, 159
add instructions, 160
compare instructions, 58, 59, 169, 169, 361
condition code bits, 163
condition codes (fcc) fields of FSR register, 61,
163, 165, 169
data type, 33
defered-trap queue (FQ), 289
divide instructions, 171
exception, 9
exception, encoding type, 60
FPRS register, 354
FSR condition codes, 59
move instructions, 178
multiply instructions, 194
negate instructions, 196
operate (FPop) instructions, 9, 30, 60, 64, 119, 243
registers
destination F, 359
FPRS, See FPRS register
FSR, See FSR register
programming, 56
rounding direction, 59
square root instructions, 215
subtract instructions, 220
trap types, 9
IEEE_754_exception, 61, 62, 64, 67, 360
invalid_fp_register, 159, 160, 220
results after recovery, 62
traps
defered, 289
precise, 289
floating-point condition codes (fcc) fields of FSR register, 418
floating-point operate (FPop) instructions, 433
floating-point trap types
IEEE_754_exception, 418, 433
floating-point unit (FPU), 9, 24
FLUSH instruction, 175
memory ordering control, 260
FLUSH instruction
memory/instruction synchronization, 174
FLUSH instruction, 174, 383
data access, 8
immediacy of effect, 176
in multiprocessor system, 174
in self-modifying code, 175
latency, 472
flush instruction memory, See FLUSH instruction
flush register windows instruction, 177
FLUSHW instruction, 177, 435
effect, 30
management by window traps, 86, 436
spill exception, 118, 177, 438
FMOVcc instructions
conditionally moving floating-point register contents, 71
conditions for copying floating-point register contents, 115
copying a register, 58
encoding of opf<84> bits, 458
encoding of opf_cc instruction field, 460
encoding of rcond instruction field, 459
floating-point moves, 180
FPop instruction, 119
used to avoid branches, 184, 264
FMOVccd instruction, 458
FMOVccq instruction, 458
FMOVd instruction, 178, 457, 458
FMOVDFcc instructions, 180
FMOVdGEZ instruction, 185
FMOVdGZ instruction, 185
FMOVdIcc instructions, 180
FMOVdLEZ instruction, 185
FMOVdLZ instruction, 185
FMOVdNZ instruction, 185
FMOVdZ instruction, 185
FMOVq instruction, 178, 457, 458
FMOVQcc instructions, 180, 183
FMOVqGEZ instruction, 185
FMOVqGZ instruction, 185
FMOVqIcc instructions, 180, 183
FMOVqLEZ instruction, 185
FMOVqLZ instruction, 185
FMOVqNZ instruction, 185
FMOVqZ instruction, 185
FMOVr instructions, 119, 459
FMOVRq instructions, 186
FMOVRsGZ instruction, 185
FMOVRsLEZ instruction, 185
FMOVRsLZ instruction, 185
FMOVRsNZ instruction, 185
FMOVRsZ instruction, 185
FMOVs instruction, 178
FMOVSc instruction, 182
FMOVSFcc instructions, 180
FMOVsGEZ instruction, 185
FMOVSiccc instructions, 180
FMOVScxcc instructions, 180
FMOVxcc instructions, 180, 183
FMULSUSx16 instruction, 188, 191
FMUL8ULx16 instruction, 188, 191
FMUL8x16 instruction, 188, 189
FMUL8x16AL instruction, 188, 190
FMUL8x16AU instruction, 188, 190
FMULd instruction, 194
FMULD8SUx16 instruction, 188, 192
FMULD8ULx16 instruction, 188, 193
FMULq instruction, 194
FMULs instruction, 194
FNAND instruction, 214
FNANDS instruction, 214
FNEG instructions, 196
FNEGd instruction, 196
FNEGq instruction, 196, 457, 458
FNEGq instruction, 196, 457, 458
FNEGs instruction, 196
FNOR instruction, 214
FNORS instruction, 214
FNOT1 instruction, 212
FNOT1S instruction, 212
FNOT2 instruction, 212
FNOT2S instruction, 212
FONE instruction, 211
FONES instruction, 211
FOR instruction, 214
formats, instruction, 100
FORNOT1 instruction, 214
FORNOT1S instruction, 214
FORNOT2 instruction, 214
FORNOT2S instruction, 214
FORS instruction, 214

**fp_disabled** exception, 433

- absolute value instructions, 159, 160, 220
- cause encoded in FSR.flt, 61
- FADDq instruction, 160, 220
- FCMP[E]q instructions, 170
- FDIVq instruction, 171
- FdTOq, FqTOd instructions, 219
- FiTOq instruction, 173
- FMOVcc instruction, 184
- FMOVq instruction, 178
- FMOVrQ instruction, 187
- FMULq, FdMUlq instructions, 195
- FNEGq instruction, 196
- FqTOx, FqTOi instructions, 217
- FSQRT instructions, 215
- FxTOq instruction, 221
- incorrect IEEE Std 754-1985 result, 119, 465
- occurrence, 133
- supervisor handling, 360
- trap type of unfinished_FPop, 62
- unimplemented_FPop for quad FPop, 57
- when quad FPop unimplemented in hardware, 63
- with floating-point arithmetic instructions, 171, 195
- FPACK instruction, 77
- FPACK instructions, 197-201
- FPACK16 instruction, 197, 198
- FPACK16 operation, 198
- FPACK32 instruction, 197, 199
- FPACK32 operation, 199
- FPACKFIR instruction, 197, 201
- FPACKFIR operation, 201
- FPADD16 instruction, 203
- FPADD16S instruction, 203
- FPADD32 instruction, 203

and tem bit of FSR, 60

cause encoded in FSR.flt, 61

FSR.aexc, 64

FSR.cexc, 65

FSR.flt, 64
generated by FCMP or FCMPE, 59

and IEEE 754 overflow/underflow conditions, 64, 65

trap handler, 360

when FSR.tem = 0, 418

when FSR.tem =1, 418

with floating-point arithmetic instructions, 160, 171, 195, 220

**fp_exception_other** exception, 67, 433

absolute value instructions, 159

cause encoded in FSR.flt, 61

FADDq instruction, 160, 220

FCMP[E]q instructions, 170

FDIVq instruction, 171

FdTOq, FqTOd instructions, 219

FiTOq instruction, 173

FMOVcc instruction, 184

FMOVq instruction, 178

FMOVrQ instruction, 187

FMULq, FdMUlq instructions, 195

FNEGq instruction, 196

FqTOx, FqTOi instructions, 217

FSQRT instructions, 215

FxTOq instruction, 221

incorrect IEEE Std 754-1985 result, 119, 465

occurrence, 133

supervisor handling, 360

trap type of unfinished_FPop, 62

unimplemented_FPop for quad FPop, 57

when quad FPop unimplemented in hardware, 63

with floating-point arithmetic instructions, 171, 195

FPACK instruction, 77

FPACK instructions, 197-201

FPACK16 instruction, 197, 198

FPACK16 operation, 198

FPACK32 instruction, 197, 199

FPACK32 operation, 199

FPACKFIR instruction, 197, 201

FPACKFIR operation, 201

FPADD16 instruction, 203

FPADD16S instruction, 203

FPADD32 instruction, 203

fp_exception exception, 64

fp_exception_iae 754 “invalid” exception, 216

fp_exception_iae 754 exception, 433
FPADD32S instruction, 203
FPMERGE instruction, 206
FPop, 9
FPop instruction
unimplemented, 433
FPop, See floating-point operate (FPop) instructions
FPRS register
See also floating-point registers state (FPRS) register
FPRS register, 73
ASR summary, 68
definition, 9
fef field, 119, 417
RDFPRS instruction, 286
FPRS register fields
dl (dirty lower fp registers), 74
du (dirty upper fp registers, 74
fef, 73
fef, See also fef field of FPRS register
FPSUB16 instruction, 208
FPSUB16S instruction, 208
FPSUB32 instruction, 208
FPSUB32S instruction, 208
FPU, 10
FqTOd instruction, 218
FqTOi instruction, 216, 363
FqTOs instruction, 218
FqTOx instruction, 216, 457, 458
freg, 480
FsMULd instruction, 194
FSQRTd instruction, 215
FSQRTq instruction, 215
FSQRTs instruction, 215
FSR (floating-point state) register fields
aexc (accrued exception), 61, 62, 63, 64, 360
aexc (accrued exceptions)
in user-mode trap handler, 360
-- dza (division by zero) bit of aexc, 66
-- nxa (rounding) bit of aexc, 67
cexc (current exception), 59, 61, 62, 64, 65, 360, 433
cexc (current exceptions)
in user-mode trap handler, 360
-- dzc (division by zero) bit of cexc, 66
-- nxc (rounding) bit of cexc, 67
fcc (condition codes), 58, 61, 62, 360, 481
fccn, 59
fft (floating-point trap type), 60, 64, 119, 316, 323, 433
in user-mode trap handler, 360
not modified by LDFSR/LDXFSR instructions, 58
qne (queue not empty), 63
in user-mode trap handler, 360
rd (rounding), 59
tem (trap enable mask), 59, 63, 65, 433
ver, 60
FSR (floating-point state) register, 58
after floating-point trap, 360
compliance with IEEE Std 754-1985, 67
LDFSR instruction, 243
reading/writing, 58
values in fft field, 61
writing to memory, 316, 323
FSRC1 instruction, 212
FSRC1S instruction, 212
FSRC2 instruction, 212
FSRC2S instruction, 212
FsTOd instruction, 218
FsTOi instruction, 216, 363
FsTOq instruction, 218
FsTOx instruction, 216, 457, 458
FSUBd instruction, 220
FSUBq instruction, 220
FSUBs instruction, 220
functional choice, implementation-dependent, 465
FXNOR instruction, 214
FXNORS instruction, 214
FXOR instruction, 214
FXORS instruction, 214
FxTOd instruction, 221, 458
FxTOq instruction, 221, 458
FxTOs instruction, 221, 458
FZERO instruction, 211
FZEROS instruction, 211
G
General Status register, See GSR
generating constants, 302
GL register, 96
access, 97
during trap processing, 429
function, 96
reading with RDPR instruction, 288, 356
relationship to TL, 97
restored during RETRY, 154, 294
SPARC V9 compatibility, 94
and TSTATE register, 88
value restored from TSTATE[TL], 97
writing to, 97
global level register, See GL register
global registers, 20, 24, 46, 48, 465
Graphics Status register, See GSR
GSR (general status) register
fields
  align, 77
  im (interval mode) field, 77
  irnd (rounding), 77
  mask, 77
  scale, 77
GSR (general status) register
ASR summary, 68

H

halfword, 10
  alignment, 26, 102, 369
data format, 33
hardware
dependency, 464
traps, 420
hardware trap stack, 30
htrap_instruction exception, 344, 433

I

i (integer) instruction field
  arithmetic instructions, 268, 270, 273, 348, 351
  floating point load instructions, 236, 239, 243
  flush memory instruction, 174
  flush register instruction, 177
  jump-and-link instruction, 226
  load instructions, 227, 247, 248, 253, 255
  logical operation instructions, 137, 273, 358
  move instructions, 264, 266
  POPC, 276
  PREFETCH, 278
  RETURN, 296
I/O
  access, 367
  memory, 366
  memory-mapped, 367
IEEE 754, 10
IEEE_754_exception floating-point trap type, 10, 61, 62, 64, 67, 360, 418, 433
IEEE-754 exception, 10
IER register (SPARC V8), 355
illegal_instruction
  and OTHERW instruction, 303
illegal_instruction exception, 177, 433
  attempt to write in nonprivileged mode, 80
  DONE/RETRY, 155, 295, 296
  ILLTRAP, 222
  instruction not specifically defined in architecture, 120
  not implemented in hardware, 133
  POPC, 277
  PREFETCH, 284
  RETURN, 297
  with BPr instruction, 149
  with branch instructions, 146, 149
  with CASA and CASXA instructions, 152, 273
  with CASXA instruction, 153
  with DONE instruction, 154
  with FMOV instructions, 178
  with FMOVcc instructions, 184
  with load instructions, 52, 234, 238, 254, 256, 404
  with move instructions, 265, 267
  with read hyperprivileged register instructions, 288
  with read instructions, 286, 287, 288, 357, 468
  with store instructions, 317, 324, 330, 331, 333
  with STQFA instruction, 321
  with Tcc instructions, 344
  with TPC register, 86
  with TSTATE register, 88
  with write instructions, 355, 357
  write to ASR 5, 73
  write to STICK register, 80
ILLTRAP instruction, 222, 433
imm_asi instruction field
  explicit ASI, providing, 108
  floating point load instructions, 239
  load instructions, 248, 253, 255
  PREFETCH, 278
immediate CTI, 99
I-MMU
  and instruction prefetching, 368
IMPDEPI instruction, 224
IMPDEPI instructions, 223, 461, 462
IMPDEP2A instructions, 223, 434, 469
IMPDEP2B instructions, 120, 223, 434
implementation, 10
implementation dependency, 463
implementation dependent, 10
implementation note, 4
implementation-dependent functional choice, 465
implementation-dependent instructions, See IMPDEP2A instructions
implicit ASI, 10, 108, 388
implicit ASI memory access
  LDFSR, 243
  LDSTUB, 247
load fp instructions, 236
load integer doubleword instructions, 253
load integer instructions, 227
STD, 330
STFSR, 323
store floating-point instructions, 316
store integer instructions, 307
SWAP, 336
implicit byte order, 90
in registers, 46, 49, 298
incc synthetic instructions, 487
inexact accrued (nxa) bit of eaxc field of FSR register, 363
inexact current (nxc) bit of cxec field of FSR register, 363
inexact quotient, 348, 349
infinity, 363
initiated, 10
input/output (I/O) locations
  access by nonprivileged code, 466
  behavior, 366
  contents and addresses, 466
  identifying, 472
  order, 366
  semantics, 472
  value semantics, 366
instruction fields, 10
  See also individual instruction fields
definition, 10
instruction group, 10
instruction MMU, See I-MMU
instruction prefetch buffer, invalidation, 175
instruction set architecture (ISA), 10, 10, 21
instruction_access_exception exception, 434
instructions
  32-bit wide, 20
  alignment, 102
  alignment, 26, 135, 369
arithmetic, integer
  addition, 134, 339
  division, 28, 270, 348
  multiplication, 28, 268, 270, 351
  subtraction, 335, 345
tagged, 28
array addressing, 138
atomic
  CASA/CASXA, 151
load twin extended word from alternate space, 250
load-store, 101, 151, 247, 248, 336, 337
load-store unsigned byte, 247, 248
successful loads, 227, 229, 254, 256
successful stores, 307, 308
branch
  branch if contents of integer register match condition, 148
  branch on floating-point condition codes, 162, 164
  branch on integer condition codes, 142, 145
cache, 375
casuing illegal instruction, 222
compare and swap, 151
comparison, 110, 335
conditional move, 30
tcontrol-transfer (CTIs), 28, 154, 294
conversion
  convert between floating-point formats, 218
  convert floating-point to integer, 216
  convert integer to floating-point, 173, 221
  floating-point to integer, 363
count of number of bits, 276
edge handling, 156
fetches, 102
floating point
  compare, 58, 59, 169
  floating-point add, 160
  floating-point compare, 361
  floating-point divide, 171
  floating-point load, 101, 236
  floating-point load from alternate space, 239
  floating-point move, 178, 180, 185
  floating-point operate (FPop), 30, 243
  floating-point square root, 215
  floating-point store, 101, 316
  floating-point store to alternate space, 319
  floating-point subtract, 220
  operate (FPop), 60, 64
short floating-point load, 245
short floating-point store, 328
status of floating-point load, 243
flush instruction memory, 174
flush register windows, 177
formats, 100
implementation-dependent, See IMPDEP2A
instructions
jump and link, 29, 226
loads
block load, 232
floating point, See instructions: floating point integer, 101
simultaneously addressing doublewords, 336
unsigned byte, 151, 247
unsigned byte to alternate space, 248
logical operations
64-bit/32-bit, 212, 214
AND, 137
logical 1-operand ops on F registers, 211
logical 2-operand ops on F registers, 212
logical 3-operand ops on F registers, 214
logical XOR, 358
OR, 273
memory, 383
moves
floating point, See instructions: floating point integer register, 262, 266
on condition, 20
ordering MEMBAR, 110
permuting bytes specified by GSR[mask], 144
pixel component distance, 275, 275
pixel formatting (PACK), 197
prefetch data, 278
read privileged register, 288
read state register, 29, 285
register window management, 30
reordering, 373
reserved, 120
reserved fields, 133
RETRY
and restartable deferred traps, 414
RETURN vs. RESTORE, 296
sequencing MEMBAR, 110
set high bits of low word, 302
set interval arithmetic mode, 304
setting GSR[mask] field, 144
shift, 28
shift count, 305
shut down to enter power-down mode, 303
SIMD, 15
simultaneous addressing of doublewords, 337
stores
block store, 312
floating point, See instructions: floating point integer, 101, 307
integer (except doubleword), 307
integer into alternate space, 308
partial, 325
unsigned byte, 151
unsigned byte to alternate space, 248
unsigned bytes, 247
swap R register, 336, 337
synthetic (for assembly language programmers), 486–488
tagged addition, 339
test-and-set, 380
timing, 133
trap on integer condition codes, 342
write privileged register, 356
write state register, 354
integer unit (IU)
condition codes, 71
definition, 10
description, 24
interrupt
enable (ie) field of PSTATE register, 416, 417
level, 95
request, 10, 31, 409
interrupt_level_14 exception, 78, 434
and SOFTINT.int_level, 78
and STICK_CMPR.stick_cmpr, 81
and TICK_CMPR.tick_cmpr, 80
interrupt_level_15 exception
and SOFTINT.int_level, 78
interrupt_level_n exception, 416, 434
and SOFTINT.register, 77
and SOFTINT.int_level, 78
inter-strand operation, 11
intra-strand operation, 11
invalid accrued (nva) bit of ae[x] field of FSR register, 66
invalid ASI
and data_access_exception, 432
invalid current (nv[x]) bit of cex[x] field of FSR register, 66, 363
invalid_exception exception, 216
invalid_fp_register floating-point trap type, 159, 160, 170, 171, 173, 178, 184, 187, 215, 220
INVALW instruction, 225
iprefetch synthetic instruction, 486
ISA, 11
ISA, See instruction set architecture
issue unit, 373, 373
issued, 11
italic font, in assembly language syntax, 479
IU, 11
ixc synthetic instructions, 487
IXX>data_access_exception (invalid ASI)
with load alternate instructions, 256

J
jmp synthetic instruction, 486
JMPL instruction, 226
  computing target address, 29
does not change CWP, 50
  mem_address_not_aligned exception, 434
  reexecuting trapped instruction, 296
jump and link, See JMPL instruction

L
LD instruction (SPARC V8), 227
LDBLOCKF instruction, 232, 403
LDD instruction (SPARC V8 and V9), 254
LDFA instruction, 402
LDDA instruction (SPARC V8 and V9), 256
LDFF instruction, 102, 236, 434
LDF_mem_address_not_aligned exception, 434
  address not doubleword aligned, 470
  address not quadword aligned, 471
  load instruction with partial store ASI and
  misaligned address, 241
  with load instructions, 237, 240, 404
  with store instructions, 320, 404
LDFF_mem_not_aligned exception, 57
LDFFA instruction, 239, 327
  alignment, 102
  ASIs for fp load operations, 404
  behavior with partial store ASIs, 237–??, 241,
  241–??, 404–??
  causing LDFF_mem_address_not_aligned
  exception, 102, 434
  for block load operations, 403
  used with ASIs, 403
LDF instruction, 57, 236
LDFA instruction, 57, 239
LDFFR instruction, 58, 60, 61, 243, 434
LDQF instruction, 236, 436
LDQF_mem_address_not_aligned exception, 436
  address not quadword aligned, 471
  LDQF/LDQFA instruction, 103
  with load instructions, 240
LDQFA instruction, 239
LDSB instruction, 227
LDSBA instruction, 229
LDSH instruction, 227
LDSHA instruction, 229
LDSHARF instruction, 245
LDSTUB instruction, 101, 247, 248, 380, 381
  and data_access_exception (noncacheable page)
  exception, 432
  hardware primitives for mutual exclusion of
  LDSTUB, 379
LDSTUBA instruction, 247, 248
  alternate space addressing, 27
  and data_access_exception exception, 432
  hardware primitives for mutual exclusion of
  LDSTUBA, 379
LDSW instruction, 227
LDSWA instruction, 229
LDTW instruction, 52, 102
LDTW instruction (deprecated), 253
LDTWA instruction, 52, 102
LDTWA instruction (deprecated), 255
LDTX instruction, 400
LDTXA instruction, 104, 106, 250, 401
  access alignment, 102
  access size, 102
  and data_access_exception (noncacheable page)
  exception, 432
LDUB instruction, 227
LDUBA instruction, 229
LDUH instruction, 227
LDUAH instruction, 229
LDUW instruction, 227
LDUWA instruction, 229
LDX instruction, 227
LDXA instruction, 229, 257, 378
LDXFSR instruction, 58, 60, 61, 236, 243, 300, 434
leaf procedure
  modifying windowed registers, 117
little-endian byte order, 11, 26, 90
load
  block, See block load instructions
floating-point from alternate space
  instructions, 239
floating-point instructions, 236, 243
  from alternate space, 237, 243
instructions, 11
instructions accessing memory, 101
nonfaulting, 372
short floating-point, See short floating-point load
instructions
LoadLoad MEMBAR relationship, 259
LoadLoad MEMBAR relationship, 382
LoadLoad predefined constant, 484
loads
  nonfaulting, 384
load-store alignment, 26, 102, 369
load-store instructions
  compare and swap, 151
definition, 11
load-store unsigned byte, 151, 247, 336, 337
load-store unsigned byte to alternate space, 248
memory access, 25
swap R register with alternate space
  memory, 337
swap R register with memory, 151, 336
LoadStore MEMBAR relationship, 259, 382
LoadStore predefined constant, 484
local registers, 46, 49, 290
logical XOR instructions, 358
Lookaside predefined constant, 484
LSTPARTIALF instruction, 404

M
MAXPGL, 24, 46, 48, 94, 96, 97, 476
MAXPTL
  and MAXPGL, 97
    instances of TNPC register, 87
    instances of TPC register, 86
    instances of TSTATE register, 88
    instances of TT register, 89
may (keyword), 11
mem_address_not_aligned exception, 434
JMPL instruction, 226
LDTXA, 401, 402, 403
load instruction with partial store ASI and
  misaligned address, 241
RETURN, 297
when recognized, 153
with CASA instruction, 152
with compare instructions, 153
with load instructions, 102–103, 227, 228, 230,
  237, 243, 254, 256, 257, 403, 404
with store instructions, 102–103, 307, 308, 310,
  321, 324, 331, 333, 403, 404
with swap instructions (deprecated), 336, 338
MEMBAR
  #Sync
    semantics, 261
instruction
    atomic operation ordering, 381
    FLUSH instruction, 174, 383
    functions, 258, 381–383
    memory ordering, 260
    memory synchronization, 110
    side-effect accesses, 368
    STBAR instruction, 260
mask encodings
  #LoadLoad, 259, 382
  #LoadStore, 259, 382
  #Lookaside, 259, 383
  #MemIssue, 259, 383
  #StoreLoad, 259, 382
  #StoreStore, 259, 382
  #Sync, 259, 383
predefined constants
  #LoadLoad, 484
  #LoadStore, 484
  #Lookaside, 484
  #MemIssue, 484
  #StoreLoad, 484
  #StoreStore, 484
  #Sync, 484
MEMBAR
  #Lookaside, 378
  #StoreLoad, 378
membar_mask, 484
MemIssue predefined constant, 484
memory
    access instructions, 25, 101
    alignment, 369
    atomic operations, 380
    atomicity, 472
    cached, 366
    coherence, 368, 472
    coherency unit, 369
    data, 383
instruction, 383
location, 366
models, 365
ordering unit, 369
real, 366
reference instructions, data flow order
constraints, 374
synchronization, 260
virtual address, 366
virtual address 0, 385
Memory Management Unit
definition, 11
Memory Management Unit, See MMU
memory model
mode control, 377
partial store order (PSO), 376
relaxed memory order (RMO), 260, 376
sequential consistency, 377
strong, 376
total store order (TSO), 260, 376, 377
weak, 376
memory model (mm) field of PSTATE register, 91
memory order
pending transactions, 375
program order, 373
memory_model (mm) field of PSTATE register, 377
memory-mapped I/O, 367
mmask instruction field
store instructions, 311
MMU
definition, 11
page sizes, 447
mode
nonprivileged, 22
privileged, 24, 86, 371
motion estimation, 275
MOVA instruction, 262
MOVCC instruction, 262
MOVcc instructions, 262
conditionally moving integer register
contents, 71
conditions for copying integer register
contents, 115
copying a register, 58
encoding of cond field, 459
encoding of opf_cc instruction field, 460
used to avoid branches, 184, 264
MOVCS instruction, 262
move floating-point register if condition is true, 180
move floating-point register if contents of integer
register satisfy condition, 185
MOVE instruction, 262
move integer register if condition is satisfied
instructions, 262
move integer register if contents of integer register
satisfies condition instructions, 266
move on condition instructions, 20
MOVFA instruction, 263
MOVFE instruction, 263
MOVFG instruction, 263
MOVFGE instruction, 263
MOVFL instruction, 263
MOVFLG instruction, 263
MOVFN instruction, 263
MOVFNE instruction, 263
MOVFO instruction, 263
MOVFU instruction, 263
MOVFUE instruction, 263
MOVFUG instruction, 263
MOVFUGE instruction, 263
MOVFUL instruction, 263
MOVGF instruction, 263
MOVGF instruction, 263
MOVGE instruction, 262
MOVGE instruction, 263
MOVG instruction, 262
MOVG instruction, 263
MOVGE instruction, 263
MOVG instruction, 262
MOVGE instruction, 263
MOVGU instruction, 262
MOVL instruction, 262
MOVLE instruction, 262
MOVL instruction, 262
MOVLEU instruction, 262
MOVN instruction, 262
move synthetic instructions, 488
MOVNE instruction, 262
MOVNEG instruction, 262
MOVPO instruction, 262
MOvr instructions, 116, 266, 459
MOVRGEZ instruction, 266
MOVRGZ instruction, 266
MOVRLEZ instruction, 266
MOVRLZ instruction, 266
MOVRLEZ instruction, 266
MOVRNZ instruction, 266
MOVRLEZ instruction, 266
multiple unsigned condition codes, emulating, 116
multiply instructions, 28, 270, 351
multiprocessor synchronization instructions, 151,
336, 337
multiprocessor system, 12, 174, 283, 336, 337, 375,
MULX instruction, 270
must (keyword), 12

N
N superscript on instruction name, 124
N_REG_WINDOWS, 12
integer unit registers, 24, 465
RESTORE instruction, 290
SAVE instruction, 298
value of, 46, 82
NaN (not-a-number)
conversion to integer, 363
converting floating-point to integer, 216
quiet, 169, 170, 361
signalling, 59, 169, 170, 218, 361
transformation, 361
neg synthetic instructions, 487
negative infinity, 363
nested traps, 21
next program counter register, See NPC register
NFO, 12
noncacheable
accesses, 366
nonfaulting load, 12, 372
nonfaulting loads
behavior, 384
use by optimizer, 385
nonleaf routine, 226
nonprivileged, 12
mode, 7, 12, 22, 24, 61
software, 73
nonprivileged trap (npt) field of TICK register, 72, 287
nonresumable_error exception, 434
nonstandard floating-point, See floating-point status
register (FSR) NS field
nontranslating ASI, 12, 257, 333
nontranslating ASIs, 388
nonvirtual memory, 283
NOP instruction, 142, 163, 165, 271, 279, 343
normal traps, 420
NORMALW instruction, 272
not synthetic instructions, 487
note
architectural direction, 4
compatibility, 4
general, 4
implementation, 4
programming, 4
NPC (next program counter) register, 73
control flow alteration, 16
definition, 12
DONE instruction, 154
instruction execution, 99
relation to TNPC register, 87
RETURN instruction, 294
saving after trap, 30
npt, 12
nucleus context, 176
nucleus software, 12
NUMA, 12
NWIN, See N_REG_WINDOWS

O
octlet, 12
odd parity, 13
op3 instruction field
arithmetic instructions, 134, 146, 149, 151, 268, 270, 348, 351
floating point load instructions, 236, 239, 243
flush instructions, 174, 177
jump-and-link instruction, 226
load instructions, 227, 247, 248, 253, 255
logical operation instructions, 137, 273, 358
PREFETCH, 278
RETURN, 296
opcode
definition, 13
format, 224
opf instruction field
floating point arithmetic instructions, 160, 171, 194, 215
floating point compare instructions, 169
floating point conversion instructions, 216, 218, 221
floating point instructions, 159
floating point integer conversion, 173
floating point move instructions, 178
floating point negate instructions, 196
opf_cc instruction field
floating point move instructions, 180
move instructions, 460
opf_low instruction field, 180
optional, 13
OR instruction, 273
ORcc instruction, 273
ordering MEMBAR instructions, 110
ordering unit, memory, 369
ORN instruction, 273
ORNcc instruction, 273
OTHERW instruction, 274
OTHERWIN (other windows) register, 84
FLUSHW instruction, 177
keeping consistent state, 85
modified by OTHERW instruction, 274
partitioned, 85
range of values, 82, 473
rd designation for WRPR instruction, 356
rs1 designation for RDPR instruction, 288
SAVE instruction, 299
zeroed by INVALW instruction, 225
zeroed by NORMALW instruction, 272
OTHERWIN register trap vectors
fill/spill traps, 437
handling spill/fill traps, 437
selecting spill/fill vectors, 437
out register #7, 52
out registers, 46, 49, 298
overflow
bits
(v) in condition fields of CCR, 111
accrued (ofa) in aexec field of FSR register, 66
current (ofc) in cexec field of FSR register, 66
causing spill trap, 436
tagged add/subtract instructions, 111
P (predict) instruction field of branch instructions, 145, 148, 149, 165
P superscript on instruction name, 124
packed-to-planar conversion, 206
packing instructions, See FPACK instructions
page fault, 283
page table entry (PTE), See translation table entry (TTE)
parity, even, 9
parity, odd, 13
partial store instructions, 325, 404
partial store order (PSO) memory model, 376, 376
partitioned
additions, 203
subtracts, 208
P_AS superscript on instruction name, 124
PASR superscript on instruction name, 124
PC (program counter) register, 14, 68, 72
after instruction execution, 99
CALL instruction, 150
changed by NOP instruction, 271
copied by JMPL instruction, 226
saving after trap, 30
set by DONE instruction, 154
set by RETRY instruction, 294
Trap Program Counter register, 86
PCR
ASR summary, 68
PCR register fields
priv, 75
sl (select lower bits of PIC), 75
st (system trace enable), 75
su (select upper bits of PIC), 75
ut (user trace enable), 75
PDIST instruction, 275
pef field of PSTATE register
and access to GSR, 76
and fp_disabled exception, 433
and FPop instructions, 119
branch operations, 163, 165
byte permutation, 144
comparison operations, 167, 170
data movement operations, 265
 enabling FPU, 73
integer arithmetic operations, 205, 210
logical operations, 211, 212, 214
memory operations, 234
read operations, 287, 304, 314
special addressing operations, 135, 161, 317, 323, 327, 329, 355
trap control, 417
pef, See PSTATE, pef field
Performance Control register, See PCR
Performance instrumentation counter register, See PIC register
PIC (performance instrumentation counter) register, 13, 75
accessing, 435
ASR summary, 68
and PCR, 74
picl field, 76
picu field, 76
PIL (processor interrupt level) register, 95
interrupt conditioning, 416
interrupt request level, 418
interrupt_level_n, 434
specification of register to read, 288
specification of register to write, 356
trap processing control, 417
pipeline, 13
pipeline draining of CPU, 82, 86
pixel instructions
compare, 166
component distance, 275, 275
formatting, 197
pixel registers for storing values, 223
planar-to-packed conversion, 206
Pnp superscript on instruction name, 124
POPC instruction, 276
POR, 13
positive infinity, 363
Ppic superscript on instruction name, 124
precise floating-point traps, 289
precise trap, 412
conditions for, 412
software actions, 413
vs. disrupting trap, 415
predefined constants
LoadLoad, 484
lookaside, 484
MemIssue, 484
StoreLoad, 484
StoreStore, 484
Sync, 484
predict bit, 149
prefetch
for one read, 282
for one write, 283
for several reads, 282
for several writes, 282
page, 283
prefetch data instruction, 278
PREFETCH instruction, 101, 278, 469
prefetch_fn, 484
PREFETCHA instruction, 278, 469
and invalid ASI or VA, 432
prefetchable, 13
priority of traps, 417, 428
privilege violation
and data_access_exception, 432, 434
privileged, 13
mode, 24, 86
registers, 86
software, 23, 50, 61, 92, 109, 177, 420, 469
privileged (priv) field of PCR register, 287
privileged (priv) field of PSTATE register, 94, 152,
154, 155, 230, 234, 239, 240, 248, 256, 308, 314,
privileged mode, 13
privileged_action exception, 434
accessing restricted ASIs, 371
PIC access, 75
restricted ASI access attempt, 109, 388
TICK register access attempt, 71
with CASA instruction, 152
with compare instructions, 153
with load alternate instructions, 230, 234, 240,
248, 256, 308, 314, 320, 333, 338, 355
with load instructions, 239
with RDAsr instructions, 287
with read instructions, 287
with store instructions, 322
with swap instructions, 338
privileged_opcode exception, 435
DONE instruction, 155
RETRY instruction, 295
SAVED instruction, 300
with DONE instruction, 155, 288, 295, 357
with write instructions, 357
processor, 13
execute unit, 373
issue unit, 373, 373
 privilege-mode transition diagram, 411
reorder unit, 373
self-consistency, 373
processor cluster, See processor module
processor interrupt level register, See PIL register
processor state register, See PSTATE register
processor states
execute_state, 429
program counter register, See PC register
program counters, saving, 409
program order, 373, 373
programming note, 4
PSO, See partial store order (PSO) memory model
PSR register (SPARC V8), 355
PSTATE register
fields
priv
and access to PCR, 74

PSTATE register
- entering privileged execution mode, 409
- restored by RETRY instruction, 154, 294
- saved after trap, 409
- saving after trap, 30
- specification for RDPR instruction, 288
- specification for WRPR instruction, 356
- and TSTATE register, 88

PSTATE register fields
- ag
  - unimplemented, 94
- am
  - CALL instruction, 150
  - description, 92
  - masked/unmasked address, 154, 226, 294, 296
- cle
  - and implicit ASIs, 108
  - and PSTATE.tle, 90
  - description, 90
- ie
  - description, 94
  - enabling disrupting traps, 416
  - interrupt conditioning, 416
  - masking disrupting trap, 421
- mm
  - description, 91
  - implementation dependencies, 91, 376, 472
  - reserved values, 91
  - pef and FPRS.pes, 92
  - description, 92
  - See also pef field of PSTATE register
- priv
  - access to register-window PR state registers, 86
  - accessing restricted ASIs, 371
  - description, 94
  - determining mode, 12, 13, 450
- tle
  - description, 90

PTE (page table entry), See translation table entry (TTE)

data format, 33
quiet NaN (not-a-number), 59, 169, 170, 361

R
- R register, 14
  - #15, 52
    - special-purpose, 52
    - alignment, 254, 256
  - rational quotient, 348
- R-A-W, See read-after-write memory hazard
- rcond
  - instruction field
    - branch instructions, 148
    - encoding of, 459
    - move instructions, 266
- rd (rounding), 14
  - rd instruction field, 15
    - arithmetic instructions, 134, 146, 149, 151, 268, 270, 348, 351
    - floating point arithmetic, 160
    - floating point arithmetic instructions, 171, 194, 215
    - floating point conversion instructions, 216, 218, 221
    - floating point integer conversion, 173
    - floating point load instructions, 236, 239, 243
    - floating point move instructions, 178, 180
    - floating point negate instructions, 196
    - floating-point instructions, 159
    - jump-and-link instruction, 226
    - load instructions, 227, 247, 248, 253, 255
    - logical operation instructions, 137, 273, 358
    - move instructions, 264, 266
    - POPC, 276
- RDASI instruction, 67, 71, 285
- RDAsr instruction, 285
  - accessing I/O registers, 27
  - implementation dependencies, 286, 468
  - reading ASRs, 67
- RDCCR instruction, 67, 69, 285, 285
- RDFPRS instruction, 68, 73, 285
- RDGSR instruction, 68, 76, 285
- RDPC instruction, 68, 285
  - reading PC register, 73
- RDPCR instruction, 68, 285
- RDPIC instruction, 68, 285, 435
- RDPR instruction, 14, 68, 288
  - accessing GL register, 97
  - accessing non-register-window PR state
registers, 86
accessing register-window PR state registers, 81
and register-window PR state registers, 81
effect on TNPC register, 87
effect on TPC register, 87
effect on TSTATE register, 88
effect on T state, 89
reading privileged registers, 86
reading PSTATE register, 90
reading the TICK register, 72
registers read, 288
RDSOFTINT instruction, 68, 77, 285
RDSTICK instruction, 68, 80, 285
RDSTICK_CMPR instruction, 68, 285
RDTICK instruction, 68, 72, 285
RDTICK_CMPR instruction, 68, 285
RDY instruction, 69
read ancillary state register (RDasr) instructions, 285
read state register instructions, 29
read-after-write memory hazard, 373, 374
real memory, 366
reference MMU, 479
reg, 480
reg_or_imm, 484, 485
reg_plus_imm, 483
regaddr, 484
register reference instructions, data flow order constraints, 373
register window, 46, 48
register window management instructions, 30
register windows clean, 83, 85, 86, 117, 431, 436, 437, 438
fill, 50, 85, 117, 118, 291, 292, 300, 433, 437, 438
management of, 22
overlapping, 49–51
spill, 50, 85, 116, 118, 299, 300, 435, 436, 437, 438
registers See also individual register (common) names
address space identifier (ASI), 371
ASI (address space identifier), 71
chip-level multithreading, See CMT
clean windows (CLEANWIN), 83
clock-tick (TICK), 435
current window pointer (CWP), 82
F (floating point), 359, 418
floating-point, 24
programming, 56
floating-point registers state (FPRS), 73
floating-point state (FSR), 58
general status (GSR), 76
global, 20, 24, 46, 48, 48, 465
global level (GL), 96
IER (SPARC V8), 355
in, 46, 49, 298
local, 46, 49
next program counter (NPC), 73
other windows (OTHERWIN), 84
out, 46, 49, 298
out #7, 52
performance control (PCR), 74
performance instrumentation counter (PIC), 75
pixel storage registers, 223
processor interrupt level (PIL)
and PIC, 76
and PIC counter overflow, 76
and SOFTINT, 78
and STICK_CMPR, 81
and TICK_CMPR, 80
processor interrupt level (PIL), 95
program counter (PC), 72
PSR (SPARC V8), 355
R register #15, 52
renaming mechanism, 374
restorable windows (CANRESTORE), 83, 83
savable windows (CANSAVE), 83
scratchpad privileged, 405
SOFTINT, 68
SOFTINT_CLR pseudo-register, 68, 79
SOFTINT_SET pseudo-register, 68, 78
STICK, 80
STICK_CMPR
ASR summary, 68
int_dis field, 78, 81
stick_cmp field, 81
and system software trapping, 81
TBR (SPARC V8), 355
TICK, 71
TICK_CMPR
int_dis field, 78, 80
tick_cmp field, 80
TICK_CMPR, 68, 79
trap base address (TBA), 89
trap base address, See registers: TBA
trap level (TL), 94
trap level, See registers: TL
trap next program counter (TNPC), 87

24 UltraSPARC Architecture 2005 • Draft D0.8.7, 27 Mar 2006
trap next program counter, See registers: TNPC
trap program counter (TPC), 86
trap program counter, See registers: TPC
trap state (TSTATE), 88
trap state, See registers: TSTATE
trap type (TT), 89, 420
trap type, See registers: TT
VA_WATCHPOINT, 435
visible to software in privileged mode, 86–97
WIM (SPARC V8), 355
window state (WSTATE), 84
window state, See registers: WSTATE
Y (32-bit multiply/divide), 69

relaxed memory order (RMO) memory model, 260, 376
renaming mechanism, register, 374
reorder unit, 373
reordering instruction, 373
reserved, 14
fields in instructions, 133
register field, 46
reset
reset trap, 415
restartable deferred trap, 413
restorable windows register, See CANRESTORE register
RESTORE instruction, 50, 290–291
actions, 117
and current window, 52
decrementing CWP register, 49
fill trap, 433, 437
followed by SAVE instruction, 50
managing register windows, 30
operation, 290
performance trade-off, 290, 298
and restorable windows (CANRESTORE) register, 83
restoring register window, 290
role in register state partitioning, 85
reset
restored synthetic instruction, 486
RESTORED instruction, 118, 292
creating inconsistent window state, 292
fill handler, 291
fill trap handler, 118, 438
register window management, 30
restricted, 14
restricted address space identifier, 109
restricted ASI, 371, 387
resumable_error exception, 435
ret/ret1 synthetic instructions, 486
RETRY instruction, 294
and restartable deferred traps, 414
effect on TNPC register, 87
effect on TPC register, 87
effect on TSTATE register, 88
generating illegal_instruction exception, 434
modifying CCR.xcc, 70
reexecuting trapped instruction, 438
restoring gl value in GL, 97
return from trap, 409
returning to instruction after trap, 416
target address, return from privileged traps, 29
RETURN instruction, 296–297
computing target address, 29
fill trap, 433
mem_address_not_aligned exception, 434
operation, 296
reexecuting trapped instruction, 296
RETURN vs. RESTORE instructions, 296
RMO, 15
RMO, See relaxed RESTORE instructions, 296
rounding
for floating-point results, 59
in signed division, 349
rounding direction (rd) field of FSR register, 160, 171, 194, 215, 216, 218, 220, 221
routine, nonleaf, 226
rs1 instruction field, 15
arithmetic instructions, 134, 146, 149, 151, 268, 270, 348, 351
branch instructions, 148
floating point arithmetic instructions, 160, 171, 194
floating point compare instructions, 169
floating point load instructions, 236, 239, 243
flush memory instruction, 174
jump-and-link instruction, 226
load instructions, 227, 247, 248, 253, 255
logical operation instructions, 137, 273, 358
move instructions, 266
PREFETCH, 278
RETURN, 296
rs2 instruction field, 15
arithmetic instructions, 134, 146, 149, 151, 268, 270, 273, 348, 351
floating point arithmetic instructions, 160, 171, 194, 215
floating point compare instructions, 169
floating point conversion instructions, 216, 218, 221
floating point instructions, 159
floating point integer conversion, 173
floating point load instructions, 236, 239, 243
floating point move instructions, 178, 180
floating point negate instructions, 196
flush memory instruction, 174
jump-and-link instruction, 226
load instructions, 227, 253, 255
logical operation instructions, 137, 358
move instructions, 264, 266
POPC, 276
PREFETCH, 278
RTO, 15
RTS, 15

S
savable windows register, See CANSAVE register
SAVE instruction, 49, 298
actions, 116
after RESTORE instruction, 296
clean_window exception, 431, 437
and current window, 52
decrementing CWP register, 49
effect on privileged state, 299
leaf procedure, 226
and local/out registers of register window, 50
managing register windows, 30
no clean window available, 84
number of usable windows, 83
operation, 298
performance trade-off, 298
role in register state partitioning, 85
and savable windows (CANSAVE) register, 83
spill trap, 435, 436, 438
save synthetic instructions, 486
SAVED instruction, 118, 300
creating inconsistent window state, 300
register window management, 30
spill handler, 299, 300
spill trap handler, 118, 438
scaling of the coefficient, 189
scratchpad registers
privileged, 405
SDIV instruction, 69, 348
SDIVcc instruction, 69, 348
SDIVX instruction, 270
self-consistency, processor, 373
self-modifying code, 174, 175
sequencing MEMBAR instructions, 110
sequential consistency memory model, 377
SETHI instruction, 110, 302
creating 32-bit constant in r register, 27
and NOP instruction, 271
with rd = 0, 302
set r synthetic instructions, 486
shall (keyword), 15
shared memory, 365
shift count encodings, 305
shift instructions, 28
shift instructions, 110, 305
short floating-point load and store instructions, 404
short floating-point load instructions, 245
short floating-point store instructions, 328
should (keyword), 15
SHUTDOWN instruction, 303
SIAM instruction, 304
side effect
accesses, 367
definition, 15
I/O locations, 366
instruction prefetching, 368
real memory storage, 366
visible, 367
signalling NaN (not-a-number), 59, 169, 170, 218, 361
signed integer data type, 33
signx synthetic instructions, 487
SIMD, 15
instruction data formats, 41–43
simm10 instruction field
move instructions, 266
simm11 instruction field
move instructions, 264
simm13 instruction field
floating point
load instructions, 236
simm13 instruction field
arithmetic instructions, 268, 270, 273, 348, 351
floating point load instructions, 239, 243
flush memory instruction, 174
jump-and-link instruction, 226
load instructions, 227, 247, 248, 253, 255
logical operation instructions, 137, 358
POPC, 276
PREFETCH, 278
RETURN, 296
single instruction/multiple data, See SIMD
SLL instruction, 305
SLLX instruction, 305
SMUL instruction, 69, 351
SMULcc instruction, 69, 351
SOFTINT register, 68, 77
  clearing, 443
  clearing of selected bits, 79
  communication from nucleus code to kernel
code, 442
  scheduling interrupt vectors, 441, 442
setting, 442
SOFTINT register fields
  int_level, 78
  sm (stick_int), 78
  tm (tick_int), 78, 80
SOFTINT_CLR pseudo-register, 68, 79
SOFTINT_SET pseudo-register, 68, 78, 79
software
  nucleus, 12
  software translation table, 447
  software trap, 343, 420
software trap number (SWTN), 343
  software, nonprivileged, 73
software trap_number, 485
source operands, 203, 208
SPA
  ASI_TWIN_DW_NUCLEUS, 406
SPARC V8 compatibility
  LD, LDUW instructions, 227
  operations to I/O locations, 368
  read state register instructions, 286
  STA instruction renamed, 309
  STBAR instruction, 260, 311
  STD instruction, 331
  STDA instruction, 333
  tagged subtract instructions, 347
  UNIMP instruction renamed, 222
  window_overflow exception superseded, 433
  write state register instructions, 355
SPARC V9
  compliance, 13
  features, 20
SPARC V9 Application Binary Interface (ABI), 22
speculative load, 15
spill register window, 435
  FLUSH instruction, 118
overflow/underflow, 50
RESTORE instruction, 117
SAVE instruction, 85, 116, 298, 436
SAVED instruction, 118, 300, 438
selection of, 437
  trap handling, 438
  trap vectors, 299, 438
window state, 85
spill_n_normal exception, 299, 435
  and FLUSHW instruction, 177
spill_n_other exception, 299, 435
  and FLUSHW instruction, 177
SRA instruction, 305
SRAX instruction, 305
SRL instruction, 305
SRLX instruction, 305
stack frame, 298
state registers (ASRs), 67–81
STB instruction, 307
STBA instruction, 308
STBAR instruction, 260
STBAR instruction, 286, 354, 374, 381
STBLOCKF instruction, 312, 403
STDF instruction, 102, 316, 435
STDF_mem_address_not_aligned exception, 435
  and store instructions, 317, 321
STDF/STDFA instruction, 102
STDFA instruction, 319
alignment, 102
ASIs for fp store operations, 404
causing data_access_exception exception, 404
causing mem_address_not_aligned or
illegal_instruction exception, 404
causing STDF_mem_address_not_aligned
exception, 102, 435
for block load operations, 403
for partial store operations, 404
used with ASIs, 403
STF instruction, 316
STFA instruction, 319
STFSR instruction, 58, 60, 61, 434
STH instruction, 307
STHA instruction, 308
STICK register, 68, 72, 80
  counter field, 80
  npt field, 72, 80
RDSTICK instruction, 285
STICK_CMPR register, 68, 81
  int_dis field, 78, 81
RDSTICK_CMPR instruction, 285
stick_cmpr field, 81
store
  block, See block store instructions
  partial, See partial store instructions
  short floating-point, See short floating-point store instructions
store buffer
  merging, 367
store floating-point into alternate space instructions, 319
store instructions, 15, 101
StoreLoad MEMBAR relationship, 259, 382
StoreLoad predefined constant, 484
stores to alternate space, 27, 71, 108
StoreStore MEMBAR relationship, 259, 382
StoreStore predefined constant, 484
STPARTIALF instruction, 325
STQF instruction, 103, 316, 436
STQF_mem_address_not_aligned exception, 436
STQF/STQFA instruction, 103
strand, 15
strong consistency memory model, 377
strong ordering, 377
Strong Sequential Order, 378
strongly ordered page, illegal access to, 432
STSHORTF instruction, 328
STTW instruction, 52, 102
STTW instruction (deprecated), 330
STTWA instruction, 52, 102
STTWA instruction (deprecated), 332
STW instruction, 307
STWA instruction, 308
STX instruction, 307
STXA instruction, 308
  accessing nontranslating ASIs, 333
  mem_address_not_aligned exception, 308
  referencing internal ASIs, 378
STXFSR instruction, 58, 60, 61, 316, 434
SUB instruction, 335, 335
SUBC instruction, 335, 335
SUBcc instruction, 110, 335, 335
SUBCcc instruction, 335, 335
subnormal number, 16
subtract instructions, 335
superscalar, 16
supervisor software
  accessing special protected registers, 26
definition, 16
SWAP instruction, 336
  accessing doubleword simultaneously with other instructions, 337
  and data_access_exception (noncacheable page) exception, 432
  hardware primitive for mutual exclusion, 379, 380
  identification of R register to be exchanged, 101
  in multiprocessor system, 247, 248
  memory accessing, 336
  ordering by MEMBAR, 381
swap R register
  bit contents, 151
  with alternate space memory instructions, 337
  with memory instructions, 336
SWAPA instruction, 337
  accessing doubleword simultaneously with other instructions, 337
  alternate space addressing, 27
  and data_access_exception (noncacheable page) exception, 432
  hardware primitive for mutual exclusion, 379
  in multiprocessor system, 247, 248
  ordering by MEMBAR, 381
SWTN (software trap number), 343
Sync predefined constant, 484
synchronization, 261
synchronization, 16
synthetic instructions
  mapping to SPARC V9 instructions, 486–488
  for assembly language programmers, 486
  mapping
    bclrg, 488
    bset, 488
    btog, 488
    btst, 488
    call, 486
    cmn, 487
    clr, 488
    cmp, 486
    dec, 487
    deccc, 487
    inc, 487
    inccc, 487
    iprefetch, 486
    jmp, 486
    movn, 488
    neg, 487
not, 487
restore, 486
ret/ret1, 486
save, 486
setn, 486
signx, 487
tst, 486
vs. pseudo ops, 486
system clock-tick register (STICK), 80
system software
accessing memory space by server program, 370
ASIs allowing access to memory space, 372
FLUSH instruction, 176, 384
processing exceptions, 370
trap types from which software must recover, 61
System Tick Compare register, See STICK_CMPR
register
System Tick register, See STICK register
T
TA instruction, 342, 459
TADDc instruction, 111, 339
TADDcTV instruction, 111, 435
tag overflow, 111
tag _ overflow exception, 111, 339, 340, 341, 345, 347
tag _ overflow exception (deprecated), 435
tagged arithmetic, 111
tagged arithmetic instructions, 28
tagged word data format, 33
tagged words, 33
TBA (trap base address) register, 89, 411
establishing table address, 30, 409
initialization, 419
specification for RDPR instruction, 288
specification for WRPR instruction, 356
trap behavior, 16
TBR register (SPARC V8), 355
TCC instruction, 342
Tcc instructions, 342
at TL > 0, 420
causing trap, 409
causing trap to privileged trap handler, 420
CCR register bits, 70
generating htrap _ instruction exception, 433
generating illegal _ instruction exception, 433
generating trap _ instruction exception, 435
opcode maps, 455, 459, 460
programming uses, 344
trap table space, 30
vector through trap table, 409
TCS instruction, 342, 459
TE instruction, 342, 459
termination deferred trap, 413
test-and-set instruction, 380
TG instruction, 342, 459
TGE instruction, 342, 459
TGU instruction, 342, 459
thread, 16
TICK register, 68
controlling access to timing information, 72
counter field, 72, 469
inaccuracies between two readings of, 469
npt field, 72
specification for RDPR instruction, 288
TICK_CMPR register, 68, 79
int_dis field, 78, 80
tick_cmpr field, 80
timer registers, See TCK register and STICK register
timing of instructions, 133
tininess (floating-point), 66
TL (trap level) register, 94, 411
affect on privilege level to which a trap is
delivered, 418
and implicit ASIs, 108
displacement in trap table, 409
executing RESTORED instruction, 292
executing SAVED instruction, 300
indexing for WRPR instruction, 356
indexing privileged register after RDPR, 288
setting register value after WRPR, 356
and TBA register, 419
and TPC register, 86
and TSTATE register, 88
and TT register, 89
use in calculating privileged trap vector address, 419
and WSTATE register, 84
TL instruction, 342, 459
TLB
and 3-dimensional arrays, 141
miss
reloading TLB, 447, 451
TLE instruction, 342, 459
TLEU instruction, 342, 459
TN instruction, 342, 459
TNE instruction, 342, 459
TNEG instruction, 342, 459
TPC (trap program counter) register, 16, 86
address of trapping instruction, 289
number of instances, 86
specification for RDPR instructions, 288
specification for WRPR instruction, 356
TPOS instruction, 342, 459
translating ASIs, 388
Translation Table Entry, See TTE
trap
See also exceptions and traps
noncacheable accesses, 368
when taken, 16
trap enable mask (tem) field of FSR register, 417, 418, 466
trap handler
privileged mode, 420
regular/nonfaulting loads, 12
returning from, 154, 294
user, 62, 362
trap level register, See TL register
trap next program counter register, See TNPC register
trap on integer condition codes instructions, 342
trap program counter register, See TPC register
trap state register, See TSTATE register
trap type (TT) register, 420
trap type register, See TT register
trap_instruction (ISA) exception, 343, 344, 435
trap_little_endian (tle) field of PSTATE register, 90
traps, 16
See also exceptions and individual trap names categories
defered, 412, 413, 415
disrupting, 412, 415
precise, 412, 412, 415
priority, 417, 428
reset, 412, 415
restartable
implementation dependency, 414
restartable deferred, 413
termination deferred, 413
caused by undefined feature/behavior, 17
causes, 31, 31
definition, 30, 410
hardware, 420
hardware stack, 21
level specification, 94
model stipulations, 417
nested, 21
normal, 420
processing, 429
software, 343, 420
stack, 429
vector address, specifying, 89
TSB, 16, 451
cacheability, 451
caching, 451
indexing support, 451
organization, 452
TSO, 16
TSO, See total store order (TSO) memory model
tst, synthetic instruction, 486
TSTATE (trap state) register, 88
DONE instruction, 154, 294
registers saved after trap, 30
restoring GL value, 97
specification for RDPR instruction, 288
specification for WRPR instruction, 356
tstate, See trap state (TSTATE) register
TSUBcc instruction, 111, 345
TSUBccTV instruction, 111, 435
TT (trap type) register, 89
and privileged trap vector address, 419
reserved values, 467
specification for RDPR instruction, 288
specification for WRPR instruction, 356
and Tcc instructions, 344
transferring trap control, 420
window spill/fill exceptions, 84
WRPR instruction, 356
TTE, 16
context ID field, 449
cp (cacheability) field, 366
cp field, 432, 450, 450
cv field, 450, 450
e field, 367, 384, 432, 450
ie field, 449
indexing support, 451
nfo field, 384, 432, 449, 450
p field, 432, 450
size field, 451  
soft2 field, 449  
SPARC V8 equivalence, 448  
taddr field, 449  
v field, 449  
va_tag field, 449  
w field, 451  
TVC instruction, 342, 459  
TVS instruction, 342, 459  
typewriter font, in assembly language syntax, 479

U
UDIV instruction, 69, 348  
UDIVcc instruction, 69, 348  
UDIVX instruction, 270  
UltraSPARC, previous ASIs  
ASI_NUCLEUS_QUAD_LDD, 406  
ASI_NUCLEUS_QUAD_LDD_L, 406  
ASI_NUCLEUS_QUAD_LDD_LITTLE, 406  
ASI_PHYS_BYPASS_EC_WITH_EBIT, 406  
ASI_PHYS_BYPASS_EC_WITH_EBIT_L, 406  
ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE, 406  
ASI_PHYS_USE_EC, 406  
ASI_PHYS_USE_EC_L, 406  
ASI_PHYS_USE_EC_LITTLE, 406  
UMUL instruction, 69, 351  
UMULcc instruction, 69, 351  
unassigned, 17  
unconditional branches, 142, 146, 163, 165  
undefined, 17  
underflow  
bits of FSR register  
accrued (ufa) bit of aexc field, 66, 363  
current (ufc) bit of ecx, 66  
current (ufc) bit of ecx field, 363  
mask (ufm) bit of fcsr Jrn, 66  
mask (ufm) bit of tcm field, 362  
detection, 50  
ocurrence, 437  
handling, 67  
result after recovery, 62  
unimplemented_LDTW exception, 254, 435  
unimplemented_STTW exception, 331, 435  
uniprocessor system, 17  
unrestricted, 17  
unrestricted ASI, 387  
unsigned integer data type, 33  
user application program, 17  
user trap handler, 62, 362

V
VA, 17  
VA_WATCHPOINT exception, 435  
VA_WATCHPOINT register, 435  
value clipping, See FPACK instructions  
value semantics of input/output (I/O) locations, 366  
VER (version) register fields  
impl, 60  
virtual  
address, 366  
address 0, 385  
virtual address, 17  
virtual core, 17  
virtual memory, 283  
VIS, 17  
VIS instructions  
encoding, 461, 462  
impliedly referencing GSR register, 76  
Visual Instruction Set, See VIS instructions

W
W-A-R, See write-after-read memory hazard  
watchpoint comparator, 93  
W-A-W, See write-after-write memory hazard  
WM register (SPARC V8), 355  
window fill exception, See also fill_n_normal exception  
window fill trap handler, 30  
window overflow, 50, 436  
window spill exception, See also spill_n_normal exception  
window spill trap handler, 30  
window state register, See WSTATE register
window underflow, 437
window, clean, 298
window_fill exception, 84, 117
RETURN, 296
window_spill exception, 84
word, 17
alignment, 26, 102, 369
data format, 33
WRASI instruction, 67, 71, 353
WRasr instruction, 353
accessing I/O registers, 27
attempt to write to ASR 5 (PC), 73
cannot write to PC register, 73
implementation dependencies, 468
writing ASRs, 67
WRCCR instruction, 67, 69, 70, 353
WRFPRS instruction, 68, 73, 353
WRGSR instruction, 68, 76, 353
WRIER instruction (SPARC V8), 355
write ancillary state register (WRasr) instructions, 353
write ancillary state register instructions, See WRasr instruction
write privileged register instruction, 356
write-after-read memory hazard, 374
write-after-write memory hazard, 373, 374
WRPCR instruction, 68, 353
WRPIC instruction, 68, 353, 435
WRPR instruction, 18
accessing non-register-window PR state registers, 86
accessing register-window PR state registers, 81
and register-window PR state registers, 81
effect on TNPC register, 87
effect on TPC register, 87
effect on TSTATE register, 88
effect on TT register, 89
writing to Gl register, 97
writing to PSTATE register, 90
WRPSR instruction (SPARC V8), 355
WRSOFTINT instruction, 68, 77, 353
WRSOFTINT_CLR instruction, 68, 77, 79, 353, 443
WRSOFTINT_SET instruction, 68, 77, 78, 353, 442
WRSTICK_CMPR instruction, 68, 353
WRTBR instruction (SPARC V8), 355
WRTICK_CMP instruction, 68, 353
WRWIM instruction (SPARC V8), 355
WRY instruction, 67, 69, 353
WSTATE (window state) register
description, 84
and fill/spill exceptions, 437
normal field, 437
other field, 437
overview, 81
reading with RDPR instruction, 288
spill exception, 177
spill trap, 299
writing with WRPR instruction, 356

X
XNOR instruction, 358
XNORcc instruction, 358
XOR instruction, 358
XORcc instruction, 358

Y
Y register, 67, 69
after multiplication completed, 268
content after divide operation, 348
divide operation, 348
multiplication, 268
unsigned multiply results, 351
WRY instruction, 354
Y register (deprecated), 69

Z
zero virtual address, 385