Oracle SPARC Architecture 2011

One Architecture
... Multiple Innovative Implementations

Draft D1.0.0, 12 Jan 2016

Privilege Levels: Privileged and Nonprivileged

Distribution: Public
## Contents

Preface ................................................................. i

1 Document Overview .............................................. 1
   1.1 Navigating *Oracle SPARC Architecture 2011* .................. 1
   1.2 Fonts and Notational Conventions ................................. 2
       1.2.1 Implementation Dependencies ............................... 3
       1.2.2 Notation for Numbers ....................................... 3
       1.2.3 Informational Notes ....................................... 3
   1.3 Reporting Errors in this Specification ........................... 4

2 Definitions ....................................................... 5

3 Architecture Overview ............................................ 13
   3.1 The Oracle SPARC Architecture 2011 ............................ 13
       3.1.1 Features .................................................. 13
       3.1.2 Attributes ................................................. 14
           3.1.2.1 Design Goals ....................................... 15
           3.1.2.2 Register Windows .................................... 15
       3.1.3 System Components ....................................... 15
           3.1.3.1 Binary Compatibility ................................ 15
           3.1.3.2 Oracle SPARC Architecture 2011 MMU ............... 15
           3.1.3.3 Privileged Software ................................ 15
       3.1.4 Architectural Definition .................................. 16
       3.1.5 Oracle SPARC Architecture 2011 Compliance with SPARC V9 Architecture .............. 16
       3.1.6 Implementation Compliance with Oracle SPARC Architecture 2011 ...................... 16
   3.2 Processor Architecture ....................................... 16
       3.2.1 Integer Unit (IU) ......................................... 16
       3.2.2 Floating-Point Unit (FPU) ................................ 17
   3.3 Instructions .................................................... 17
       3.3.1 Memory Access ............................................. 17
           3.3.1.1 Memory Alignment Restrictions ....................... 18
           3.3.1.2 Addressing Conventions ............................. 18
           3.3.1.3 Addressing Range ................................... 18
           3.3.1.4 Load/Store Alternate ................................ 18
           3.3.1.5 Separate Instruction and Data Memories ............. 19
           3.3.1.6 Input/Output (I/O) .................................. 19
           3.3.1.7 Memory Synchronization ............................. 19
       3.3.2 Integer Arithmetic / Logical / Shift Instructions ............ 19
       3.3.3 Control Transfer .......................................... 20
       3.3.4 State Register Access ..................................... 20
           3.3.4.1 Ancillary State Registers ............................ 20
           3.3.4.2 PR State Registers .................................. 20
5 Registers ................................................................. 31
  5.1 Reserved Register Fields ........................................ 32
  5.2 General-Purpose R Registers .................................. 32
    5.2.1 Global R Registers ........................................ 33
    5.2.2 Windowed R Registers ..................................... 34
    5.2.3 Special R Registers ........................................ 37
  5.3 Floating-Point Registers ....................................... 38
    5.3.1 Floating-Point Register Number Encoding ............... 40
    5.3.2 Double and Quad Floating-Point Operands ............... 41
  5.4 Floating-Point State Register (FSR) ......................... 42
    5.4.1 Floating-Point Condition Codes (fcc0, fcc1, fcc2, fcc3) 42
    5.4.2 Rounding Direction (rd) ................................. 43
    5.4.3 Trap Enable Mask (tem) ................................... 43
    5.4.4 Nonstandard Floating-Point (ns) .......................... 43
    5.4.5 FPU Version (ver) ......................................... 43
    5.4.6 Floating-Point Trap Type (ftt) ........................... 44
    5.4.7 Accrued Exceptions (aexc) ............................... 46
    5.4.8 Current Exception (cexc) ................................. 46
    5.4.9 Floating-Point Exception Fields ......................... 47
    5.4.10 FSR Conformance ......................................... 48
  5.5 Ancillary State Registers ..................................... 48
    5.5.1 32-bit Multiply/Divide Register (Y) (ASR 0) .......... 50
    5.5.2 Integer Condition Codes Register (CCR) (ASR 2) .... 50
      5.5.2.1 Condition Codes (CCR.xcc and CCR.icc) .......... 50
    5.5.3 Address Space Identifier (ASI) Register (ASR 3) .... 51
    5.5.4 Tick (TICK) Register (ASR 4) ........................... 52
    5.5.5 Program Counters (PC, NPC) (ASR 5) ................... 53
    5.5.6 Floating-Point Registers State (FPRS) Register (ASR 6) 53
    5.5.7 General Status Register (GSR) (ASR 19) .............. 54

4 Data Formats .......................................................... 23
  4.1 Integer Data Formats ........................................... 24
    4.1.1 Signed Integer Data Types ............................... 24
      4.1.1.1 Signed Integer Byte, Halfword, and Word .......... 25
      4.1.1.2 Signed Integer Doubleword (64 bits) ............. 25
      4.1.1.3 Signed Integer Extended-Word (64 bits) .......... 25
    4.1.2 Unsigned Integer Data Types ............................ 25
      4.1.2.1 Unsigned Integer Byte, Halfword, and Word ....... 26
      4.1.2.2 Unsigned Integer Doubleword (64 bits) .......... 26
      4.1.2.3Unsigned Extended Integer (64 bits) ............... 26
    4.1.3 Tagged Word (32 bits) .................................... 26
  4.2 Floating-Point Data Formats .................................. 27
    4.2.1 Floating Point, Single Precision (32 bits) .......... 27
    4.2.2 Floating Point, Double Precision (64 bits) .......... 27
    4.2.3 Floating Point, Quad Precision (128 bits) .......... 28
    4.2.4 Floating-Point Data Alignment in Memory and Registers 29
  4.3 SIMD Data Formats .............................................. 29
    4.3.1 Uint8 SIMD Data Format ................................. 30
    4.3.2 Int16 SIMD Data Formats ................................. 30
    4.3.3 Int32 SIMD Data Format ................................. 30
  5 Registers ............................................................. 31
  5.1 Reserved Register Fields ...................................... 32
  5.2 General-Purpose R Registers .................................. 32
    5.2.1 Global R Registers ....................................... 33
    5.2.2 Windowed R Registers ..................................... 34
    5.2.3 Special R Registers ...................................... 37
  5.3 Floating-Point Registers ...................................... 38
    5.3.1 Floating-Point Register Number Encoding .............. 40
    5.3.2 Double and Quad Floating-Point Operands .............. 41
  5.4 Floating-Point State Register (FSR) .......................... 42
    5.4.1 Floating-Point Condition Codes (fcc0, fcc1, fcc2, fcc3) 42
    5.4.2 Rounding Direction (rd) ............................... 43
    5.4.3 Trap Enable Mask (tem) .................................. 43
    5.4.4 Nonstandard Floating-Point (ns) ......................... 43
    5.4.5 FPU Version (ver) ....................................... 43
    5.4.6 Floating-Point Trap Type (ftt) .......................... 44
    5.4.7 Accrued Exceptions (aexc) ............................. 46
    5.4.8 Current Exception (cexc) ............................... 46
    5.4.9 Floating-Point Exception Fields ...................... 47
    5.4.10 FSR Conformance ....................................... 48
  5.5 Ancillary State Registers .................................... 48
    5.5.1 32-bit Multiply/Divide Register (Y) (ASR 0) ......... 50
    5.5.2 Integer Condition Codes Register (CCR) (ASR 2) .... 50
      5.5.2.1 Condition Codes (CCR.xcc and CCR.icc) ......... 50
    5.5.3 Address Space Identifier (ASI) Register (ASR 3) ... 51
    5.5.4 Tick (TICK) Register (ASR 4) .......................... 52
    5.5.5 Program Counters (PC, NPC) (ASR 5) ................. 53
    5.5.6 Floating-Point Registers State (FPRS) Register (ASR 6) 53
    5.5.7 General Status Register (GSR) (ASR 19) ............ 54
6 Instruction Set Overview

6.1 Instruction Execution ........................................ 75
6.2 Instruction Formats ........................................ 76
6.3 Instruction Categories ..................................... 76

6.3.1 Memory Access Instructions ............................ 77
6.3.1.1 Memory Alignment Restrictions .................. 77
6.3.1.2 Addressing Conventions .......................... 78
6.3.1.3 Address Space Identifiers (ASIs) ............... 81
6.3.1.4 Separate Instruction Memory ..................... 82

6.3.2 Memory Synchronization Instructions ................. 83

6.3.3 Integer Arithmetic and Logical Instructions ........... 83
6.3.3.1 Setting Condition Codes ......................... 83
6.3.3.2 Shift Instructions ................................. 83
6.3.3.3 Set High 22 Bits of Low Word .................. 83
6.3.3.4 Integer Multiply/Divide ......................... 83
6.3.3.5 Tagged Add/Subtract ............................ 83

6.3.4 Control-Transfer Instructions (CTIs) ................. 84
6.3.4.1 Conditional Branches ............................ 85
6.3.4.2 Unconditional Branches .......................... 86
6.3.4.3 CALL and JMPL Instructions .................... 86
6.3.4.4 RETURN Instruction .............................. 86
6.3.4.5 DONE and RETRY Instructions .................. 86
6.3.4.6 Trap Instruction (Tcc) ........................... 86
6.3.4.7 DCTI Couples ................................... 87

6.3.5 Conditional Move Instructions ......................... 87

6.3.6 Register Window Management Instructions .......... 88
6.3.6.1 SAVE Instruction ................................ 88
6.3.6.2 RESTORE Instruction ........................... 88

5.5.8 SOFTINTP Register (ASRs 20, 21, 22) .................. 55
5.5.8.1 SOFTINT_SETP Pseudo-Register (ASR 20) ........ 55
5.5.8.2 SOFTINT_CLR Pseudo-Register (ASR 21) ........ 56

5.5.9 System Tick (STICK) Register (ASR 24) ............... 56

5.5.10 System Tick Compare (STICK_CMPRP) Register (ASR 25) ... 57

5.5.11 Compatibility Feature Register (CFR) (ASR 26) ....... 59

5.5.12 Pause Count (PAUSE) Register (ASR 27) ............. 60

5.6 Register-Window PR State Registers ..................... 61
5.6.1 Current Window Pointer (CP) (PR 9) .................. 61
5.6.2 Savable Windows (CANSVP) Register (PR 10) ........ 62
5.6.3 Restorable Windows (CANRESTOREP) Register (PR 11) .. 62
5.6.4 Clean Windows (CLEANWINP) Register (PR 12) ...... 62
5.6.5 Other Windows (OTHERWINP) Register (PR 13) ...... 63
5.6.6 Window State (WSTATEP) Register (PR 14) .......... 63
5.6.7 Register Window Management .......................... 64
5.6.7.1 Register Window State Definition ................. 64
5.6.7.2 Register Window Traps ............................ 64

5.7 Non-Register-Window PR State Registers ................. 65
5.7.1 Trap Program Counter (TPCP) Register (PR 0) ........ 65
5.7.2 Trap Next PC (TNPCP) Register (PR 1) ............... 65
5.7.3 Trap State (TSTATEP) Register (PR 2) ............... 66
5.7.4 Trap Type (TP) Register (PR 3) ....................... 67
5.7.5 Tick (TICK) Register (PR 4) ........................ 67
5.7.6 Trap Base Address (TBA) Register (PR 5) .......... 67
5.7.7 Processor State (PSTATEP) Register (PR 6) ......... 68
5.7.8 Trap Level Register (TL) (PR 7) ....................... 71
5.7.9 Processor Interrupt Level (PILP) Register (PR 8) .... 72
5.7.10 Global Level Register (GLP) (PR 16) ............... 73

6 Instruction Set Overview .................................... 75

6.1 Instruction Execution ........................................ 75
6.2 Instruction Formats ........................................ 76
6.3 Instruction Categories ..................................... 76

6.3.1 Memory Access Instructions ............................ 77
6.3.1.1 Memory Alignment Restrictions .................. 77
6.3.1.2 Addressing Conventions .......................... 78
6.3.1.3 Address Space Identifiers (ASIs) ............... 81
6.3.1.4 Separate Instruction Memory ..................... 82

6.3.2 Memory Synchronization Instructions ................. 83

6.3.3 Integer Arithmetic and Logical Instructions ........... 83
6.3.3.1 Setting Condition Codes ......................... 83
6.3.3.2 Shift Instructions ................................. 83
6.3.3.3 Set High 22 Bits of Low Word .................. 83
6.3.3.4 Integer Multiply/Divide ......................... 83
6.3.3.5 Tagged Add/Subtract ............................ 83

6.3.4 Control-Transfer Instructions (CTIs) ................. 84
6.3.4.1 Conditional Branches ............................ 85
6.3.4.2 Unconditional Branches .......................... 86
6.3.4.3 CALL and JMPL Instructions .................... 86
6.3.4.4 RETURN Instruction .............................. 86
6.3.4.5 DONE and RETRY Instructions .................. 86
6.3.4.6 Trap Instruction (Tcc) ........................... 86
6.3.4.7 DCTI Couples ................................... 87

6.3.5 Conditional Move Instructions ......................... 87

6.3.6 Register Window Management Instructions .......... 88
6.3.6.1 SAVE Instruction ................................ 88
6.3.6.2 RESTORE Instruction ........................... 88
7 Instructions .............................................. 93
  7.45.1 FMUL8x16 Instruction .......................... 181
  7.45.2 FMUL8x16AU Instruction ....................... 182
  7.45.3 FMUL8x16AL Instruction ......................... 182
  7.45.4 FMUL8SUx16 Instruction ......................... 183
  7.45.5 FMUL8ULx16 Instruction ......................... 183
  7.45.6 FMULD8SUx16 Instruction ....................... 184
  7.45.7 FMULD8ULx16 Instruction ....................... 184
  7.51.1 FPACK16 ........................................ 193
  7.51.2 FPACK32 ......................................... 194
  7.51.3 FPACKFIX ....................................... 195
  7.87.1 Memory Synchronization ......................... 257
  7.87.2 Synchronization of the Virtual Processor .......... 258
  7.87.3 TSO Ordering Rules affecting Use of MEMBAR .......... 258
  7.104.1 Exceptions .................................... 290
  7.104.2 Weak versus Strong Prefetches ................. 291
  7.104.3 Prefetch Variants ............................. 291
    7.104.3.1 Prefetch for Several Reads (fcn = 0, 20(1416)) .... 292
    7.104.3.2 Prefetch for One Read (fcn = 1, 21(1516)) ........ 292
    7.104.3.3 Prefetch for Several Writes (and Possibly Reads) (fcn = 2, 22(1616)) .... 292
    7.104.3.4 Prefetch for One Write (fcn = 3, 23(1716)) .... 292
    7.104.3.5 Prefetch Page (fcn = 4) ........................ 293
    7.104.3.6 Prefetch to Nearest Unified Cache (fcn = 17(1116)) .... 293
  7.104.4 Implementation-Dependent Prefetch Variants (fcn = 16, 18, 19, and 24–31) .... 293
  7.104.5 Additional Notes ................................ 293
8 IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011 .................... 365
  8.1 Traps Inhibiting Results ............................ 365
  8.2 Underflow Behavior ................................ 366
    8.2.1 Trapped Underflow Definition (ufm = 1) ............ 367
    8.2.2 Untrapped Underflow Definition (ufm = 0) ........... 367
  8.3 Integer Overflow Definition ........................ 367
  8.4 Floating-Point Nonstandard Mode .................... 367
  8.5 Arithmetic Result Tables ........................... 368
    8.5.1 Floating-Point Add (FADD) and Add and Halve (FHADD) 369
    8.5.2 Floating-Point Negative Add (FNADD) and Negative Add and Halve (FNHADD) 370
    8.5.3 Floating-Point Subtract (FSUB) and Subtract and Halve (FHSUB) 370
    8.5.4 Floating-Point Multiply .......................... 371
    8.5.5 Floating-Point Negative Multiply (FNMUL) ............ 371
    8.5.6 Floating-Point Multiply-Add (FMAD) .................. 372
    8.5.7 Floating-Point Negative Multiply-Add (FNMADD) .... 372
    8.5.8 Floating-Point Multiply-Subtract (FMSUB) ............ 373
    8.5.9 Floating-Point Negative Multiply-Subtract (FNMSUB) .... 374
    8.5.10 Floating-Point Divide (FDIV) ........................ 376
    8.5.11 Floating-Point Square Root (FSQRT) .................. 376
9 Memory ........................................................................... 381
  9.1 Memory Location Identification ........................................ 381
  9.2 Memory Accesses and Cacheability .................................... 381
    9.2.1 Coherence Domains .............................................. 382
      9.2.1.1 Cacheable Accesses ....................................... 382
      9.2.1.2 Noncacheable Accesses ................................. 382
      9.2.1.3 Noncacheable Accesses with Side-Effect ............. 382
  9.3 Memory Addressing and Alternate Address Spaces ................ 383
    9.3.1 Memory Addressing Types ....................................... 384
    9.3.2 Memory Address Spaces ........................................ 384
    9.3.3 Address Space Identifiers ...................................... 384
  9.4 SPARC V9 Memory Model ............................................. 386
    9.4.1 SPARC V9 Program Execution Model ....................... 386
    9.4.2 Virtual Processor/Memory Interface Model ................. 388
  9.5 The Oracle SPARC Architecture Memory Model — TSO ........... 389
    9.5.1 Memory Model Selection ...................................... 389
    9.5.2 Programmer-Visible Properties of the Oracle SPARC Architecture TSO Model 390
    9.5.3 TSO Ordering Rules ............................................ 391
    9.5.4 Hardware Primitives for Mutual Exclusion ................. 392
      9.5.4.1 Compare-and-Swap (CASA, CASXA) ................. 392
      9.5.4.2 Swap (SWAP) ............................................ 392
      9.5.4.3 Load Store Unsigned Byte (LDSTUB) ................. 392
    9.5.5 Memory Ordering and Synchronization ....................... 392
      9.5.5.1 Ordering MEMBAR Instructions ........................ 393
      9.5.5.2 Sequencing MEMBAR Instructions ..................... 393
      9.5.5.3 Synchronizing Instruction and Data Memory .......... 394
  9.6 Nonfaulting Load ..................................................... 395
  9.7 Store Coalescing ..................................................... 396
10 Address Space Identifiers (ASIs) .......................................... 397
  10.1 Address Space Identifiers and Address Spaces ..................... 397
  10.2 ASI Values .......................................................... 397
  10.3 ASI Assignments .................................................... 398
    10.3.1 Supported ASIs ............................................... 398
    10.4 Special Memory Access ASIs ..................................... 406
      10.4.1 ASIs 10_{16}, 11_{16}, 16_{16}, and 17_{16} (ASI_*AS_IF_USER_*) .................. 406
      10.4.2 ASIs 18_{16}, 19_{16}, 1E_{16}, and 1E_{16} (ASI_*AS_IF_USER_*_LITTLE) ....... 407
      10.4.3 ASI 14_{16} (ASI_REAL) ..................................... 407
      10.4.4 ASI 15_{16} (ASI_REAL_IO) ................................ 408
      10.4.5 ASI 1C_{16} (ASI_REAL_LITTLE) .......................... 408
      10.4.6 ASI 1D_{16} (ASI_REAL_IO_LITTLE) ....................... 408
      10.4.7 ASIs 22_{16}, 23_{16}, 26_{16}, 27_{16}, 2A_{16}, 2B_{16}, 2E_{16} 2F_{16} (Privileged Load Integer Twin Extended Word) 408
      10.4.8 ASIs 26_{16} and 2E_{16} (Privileged Load Integer Twin Extended Word, Real Addressing) 409
      10.4.9 ASIs E2_{16}, E3_{16}, EA_{16}, EB_{16} (Nonprivileged Load Integer Twin Extended Word) 410
      10.4.10 Block Load and Store ASIs .................................. 410
      10.4.11 Partial Store ASIs ......................................... 411
      10.4.12 Short Floating-Point Load and Store ASIs .............. 411
    10.5 ASI-Accessible Registers ........................................ 411
## 11 Performance Instrumentation

- **11.1 High-Level Requirements**
  - **11.1.1 Usage Scenarios**
  - **11.1.2 Metrics**
  - **11.1.3 Accuracy Requirements**
- **11.2 Performance Counters and Controls**
  - **11.2.1 Counter Overflow**
  - **11.2.2 Metrics**
  - **11.2.3 Conditioning of Performance Counters**
  - **11.2.4 Trap Handler Actions for Disrupting Traps**
- **11.3 Interrupt Packets**
- **11.4 Interrupt Queues**
- **11.5 Privileged Scratchpad Registers (ASI_SCRATCHPAD)**
- **11.6 ASI Changes in the Oracle SPARC Architecture**

## 12 Traps

- **12.1 Virtual Processor Privilege Modes**
- **12.2 Virtual Processor States and Traps**
  - **12.2.0.1 Usage of Trap Levels**
  - **12.2.0.2 Trap-Table Entry Addresses**
  - **12.2.0.3 Trap-Table Entry Address to Privileged Mode**
  - **12.2.0.4 Privileged Trap Table Organization**
  - **12.2.0.5 Trap Type (TT)**
    - **12.2.0.5.1 Trap Type for Spill/Fill Traps**
  - **12.2.0.6 Trap Priorities**
  - **12.2.0.7 Window Trap Handlers**
  - **12.2.0.8 Window Spill and Fill Traps**
  - **12.2.0.9 Vectoring of Fill/Spill Traps**
  - **12.2.0.10 CWP on Window Traps**
  - **12.2.0.11 Window Trap Handlers**
- **12.3 Trap Categories**
  - **12.3.1 Precise Traps**
  - **12.3.2 Deferred Traps**
  - **12.3.3 Disrupting Traps**
    - **12.3.3.1 Disrupting versus Precise and Deferred Traps**
    - **12.3.3.2 Causes of Disrupting Traps**
    - **12.3.3.3 Conditioning of Disrupting Traps**
    - **12.3.3.4 Trap Handler Actions for Disrupting Traps**
  - **12.3.4 Uses of the Trap Categories**
  - **12.3.5 Trap Control**
    - **12.3.5.1 PIL Control**
    - **12.3.5.2 FSR.tem Control**
  - **12.3.6 Trap Table-Entry Addresses**
    - **12.3.6.1 Trap-Table Entry Address to Privileged Mode**
    - **12.3.6.2 Privileged Trap Table Organization**
    - **12.3.6.3 Trap Type (TT)**
      - **12.3.6.3.1 Trap Type for Spill/Fill Traps**
    - **12.3.6.4 Trap Priorities**
    - **12.3.6.5 Window Trap Handlers**
    - **12.3.6.6 Window Spill and Fill Traps**
    - **12.3.6.7 Vectoring of Fill/Spill Traps**
    - **12.3.6.8 CWP on Window Traps**
    - **12.3.6.9 Window Trap Handlers**
- **12.4 Trap Processing**
  - **12.4.1 Normal Trap Processing**
- **12.5 Exception and Interrupt Descriptions**
  - **12.5.1 SPARC V9 Traps Not Used in Oracle SPARC Architecture 2011**
- **12.6 Register Window Traps**
  - **12.6.1 Window Spill and Fill Traps**
  - **12.6.2 clean_window Trap**
- **12.7 Exception and Interrupt Descriptions**
  - **12.7.1 SPARC V9 Traps Not Used in Oracle SPARC Architecture 2011**
- **12.8 CWP on Window Traps**
- **12.8.1 Window Spill and Fill Traps**
- **12.8.2 clean_window Trap**
- **12.8.3 Vectoring of Fill/Spill Traps**
- **12.8.4 CWP on Window Traps**
- **12.8.5 Window Trap Handlers**

## 13 Interrupt Handling

- **13.1 Interrupt Packets**
- **13.2 Software Interrupt Register (SOFTINT)**
  - **13.2.1 Setting the Software Interrupt Register**
  - **13.2.2 Clearing the Software Interrupt Register**
- **13.3 Interrupt Queues**
  - **13.3.1 Interrupt Queue Registers**
- **13.4 Interrupt Traps**

## 14 Memory Management

- **14.1 Virtual Address Translation**
14.2 Context ID ................................................................. 448
14.3 TSB Translation Table Entry (TTE) ................................. 449
14.4 Translation Storage Buffer (TSB) .................................... 453
   14.4.1 TSB Cacheability and Consistency ........................... 453
   14.4.2 TSB Organization .............................................. 453
14.5 ASI Value, Context ID, and Endianness Selection for Translation ........................................ 454
14.6 SPARC V9 “MMU Attributes” ....................................... 456
   14.6.1 Accessing MMU Registers .................................... 456
   14.6.2 Context ID Registers ......................................... 457
A Opcode Maps ............................................................... 459
B Implementation Dependencies .......................................... 471
   B.1 Definition of an Implementation Dependency .................. 471
   B.2 Hardware Characteristics ....................................... 472
   B.3 Implementation Dependency Categories ....................... 472
   B.4 List of Implementation Dependencies ......................... 472
C Assembly Language Syntax ............................................ 485
   C.1 Notation Used .................................................... 485
      C.1.1 Register Names .............................................. 485
      C.1.2 Special Symbol Names ...................................... 486
      C.1.3 Values ....................................................... 488
      C.1.4 Labels ........................................................ 488
      C.1.5 Other Operand Syntax ...................................... 488
      C.1.6 Comments .................................................. 490
   C.2 Syntax Design ..................................................... 490
   C.3 Synthetic Instructions .......................................... 490

................................. Index1
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. The UltraSPARC Architecture 2005 specification (predecessor to this Oracle SPARC Architecture 2011) provided the first significant update in over 10 years to Oracle’s SPARC processor architecture.

What’s New?

Oracle SPARC Architecture 2011 pulls together in one document all parts of the architecture:

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

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

■ the VIS™ 1 and VIS 2 instruction set extensions and the associated GSR register
■ multiple levels of global registers, controlled by the GL register
■ Sun’s 64-bit MMU architecture
■ privileged instructions ALLCLEAN, OTHERW, NORMALW, and INVALW
■ access to the VER register is now hyperprivileged
■ the SIR instruction is now hyperprivileged

Oracle SPARC Architecture 2011 includes the following changes since UltraSPARC Architecture 2007:

■ the VIS 3 instructions set extensions

In addition, architectural features are now 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 Oracle SPARC Architecture processor implementations. This information provides guidance that should be particularly helpful to programmers who write in assembly language or those who write tools that generate SPARC instructions. It also provides the infrastructure for defining clear procedures for adding and removing features from the architecture over time, with minimal software disruption.

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
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 Oracle SPARC Architecture 2011 specification are the result of years of deliberation, review, and feedback from readers of earlier revisions. I would particularly like to acknowledge the following people for their key contributions:

- The Oracle SPARC 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 his work on UltraSPARC Architecture 2005, including 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 the UltraSPARC Architecture 2005 specification
- The working group responsible for developing the system of Software Classes and Implementation Classes, comprising: Steve Chessin, Yuan Chou, Peter Damron, Q. Jacobson, Nicolai Kosche, Bob Maier, Ashley Saulsbury, Lawrence Spracklen, and David Weaver.
- Lawrence Spracklen, for his advice and numerous contributions regarding descriptions of VIS instructions
- Tom Webber, for providing descriptions of several new features in Oracle SPARC Architecture 2011
- Al Martin, for providing meticulously detailed floating-point exception tables, which have been integrated into Chapter 8, IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011.

I hope you find the Oracle SPARC Architecture 2011 specification more complete, accurate, and readable than its predecessors.

— David Weaver
Oracle SPARC Architecture Sr. Principal Engineer and specification editor
Corrections and other comments regarding this specification can be emailed to:
David.Weaver@Oracle.com
Document Overview

This chapter discusses:
- Fonts and Notational Conventions on page 2.
- Reporting Errors in this Specification on page 4.

1.1 Navigating Oracle SPARC Architecture 2011

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

If you are familiar with the SPARC V9 architecture but not Oracle SPARC Architecture 2011, note that Oracle SPARC Architecture 2011 conforms to the SPARC V9 Level 1 architecture (and most of Level 2), with numerous extensions — particularly with respect to VIS instructions.

This specification is structured as follows:
- Chapter 2, Definitions, which defines key terms used throughout the specification
- Chapter 3, Architecture Overview, provides an overview of Oracle SPARC Architecture 2011
- Chapter 4, Data Formats, describes the supported data formats
- Chapter 5, Registers, describes the register set
- Chapter 6, Instruction Set Overview, provides a high-level description of the Oracle SPARC Architecture 2011 instruction set
- Chapter 7, Instructions, describes the Oracle SPARC Architecture 2011 instruction set in great detail
- Chapter 8, IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011, describes the trap model
- Chapter 9, Memory describes the supported memory model
- Chapter 10, Address Space Identifiers (ASIs), provides a complete list of supported ASIs
- Chapter 11, Performance Instrumentation describes the architecture for performance monitoring hardware
- Chapter 12, Traps, describes the trap model
- Chapter 13, Interrupt Handling, describes how interrupts are handled
- Chapter 14, Memory Management, describes MMU operation
- Appendix A, Opcode Maps, provides the overall picture of how the instruction set is mapped into opcodes
- Appendix B, Implementation Dependencies, describes all implementation dependencies
Appendix C, Assembly Language Syntax, describes extensions to the SPARC assembly language syntax; in particular, synthetic instructions are documented in this appendix.

### 1.2 Fonts and Notational Conventions

Fonts are used as follows:

- **Italic** font is used for emphasis, book titles, and the first instance of a word that is defined.
- **Italic** font is also used for terms where substitution is expected, for example, "fccn", "virtual processor n", or "reg_plus_imm".
- **Italic sans serif** font is used for exception and trap names. For example, "The privileged_action exception....."
- **lowercase helvetica** font is used for register field names (named bits) and instruction field names, for example: "The rs1 field contains...."
- **UPPERCASE HELVETICA** font is used for register names; for example, FSR.
- **TYPEWRITER** (Courier) font is used for literal values, such as code (assembly language, C language, ASI names) and for state names. For example: %f0, ASI_PRIMARY, execute_state.

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

- **UPPERCASE words** are acronyms or instruction names. Some common acronyms appear in the glossary in Chapter 2, Definitions. **Note:** Names of some instructions contain both upper- and lower-case letters.
- An underscore character joins words in register, register field, exception, and trap names. **Note:** Such words may be split across lines at the underline 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 inclusive 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”.

An 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.

A slash ( / ) is used to separate paired or complementary values in a list, for example, “the LDBLOCKF/STBLOCKF instruction pair ....”

The double colon (::) is an operator that indicates concatenation (typically, of bit vectors). Concatenation strictly strings the specified component values into a single longer string, in the order specified. The concatenation operator performs no arithmetic operation on any of the component values.

1.2.1 Implementation Dependencies

Implementors of Oracle SPARC Architecture 2011 processors are allowed to resolve some aspects of the architecture in machine-dependent ways.

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

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

1.2.2 Notation for Numbers

Numbers throughout this specification are decimal (base-10) unless otherwise indicated. Numbers in other bases are followed by a numeric subscript indicating their base (for example, 10012, FFFF 000016). Long binary and hexadecimal numbers within the text have spaces inserted every four characters to improve readability. Within C language or assembly language examples, numbers may be preceded by “0x” to indicate base-16 (hexadecimal) notation (for example, 0xFFFF0000).

1.2.3 Informational Notes

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

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

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

- **Implementation Note**: An Implementation Note contains incidental information, describing how an Oracle SPARC Architecture 2011 processor might implement an architectural feature.
1.3 Reporting Errors in this Specification

This specification has been reviewed for completeness and accuracy. Nonetheless, as with any document this size, errors and omissions may occur, and reports of such are welcome. Please send “bug reports” and other comments on this document to email address:
David.Weaver@Oracle.com
Definitions

This chapter defines concepts and terminology common to all implementations of Oracle SPARC Architecture 2011.

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

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

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

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

ASI Address space identifier.

ASR Ancillary State register.

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

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

CCR Abbreviation for Condition Codes Register.

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

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

completed (memory operation) Said of a memory transaction when an idealized memory has executed the transaction with respect to all processors. A load is considered completed when no subsequent memory transaction can affect the value returned by the load. A store is considered completed when no subsequent load can return the value that was overwritten by the store.

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

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

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

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

cross-call An interprocessor call in a system containing multiple virtual processors.
CTI  Abbreviation for control-transfer instruction.

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

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

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

DCTI  Delayed control transfer instruction.

denormalized number  Synonym for subnormal number.

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

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

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

exception  A condition that makes it impossible for the processor to continue executing the current instruction stream. Some exceptions may be masked (that is, trap generation disabled — for example, floating-point exceptions masked by FSR.tem) so that the decision on whether or not to apply special processing can be deferred and made by software at a later time. See also trap.

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

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

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

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

floating-point exception  An exception that occurs during the execution of a floating-point operate (FPop) instruction. The exceptions fp_exception_ieee_754 and fp_exception_other.

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

floating-point operate instructions  Instructions that perform floating-point calculations, as defined in Floating-Point Operate (FPop) Instructions on page 90. FPop instructions do not include FBfcc instructions, loads and stores between memory and the F registers, or non-floating-point operations that read or write F registers.

floating-point trap type  The specific type of a floating-point exception, encoded in the FSR.ftt field.
floating-point unit A processing unit that contains the floating-point registers and performs floating-point operations, as defined by this specification.

FPop Abbreviation for floating-point operate (instructions).

FPRS Floating-Point Register State register.

FPU Floating-Point Unit.

FSR Floating-Point Status register.

GL Global Level register.

GSR General Status register.

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

hyperprivileged An adjective that describes:
(1) the state of the processor when the processor is in hyperprivileged mode;
(2) processor state that is only accessible to software while the processor is in hyperprivileged mode


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

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

implementation dependent An aspect of the Oracle SPARC 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 Oracle SPARC Architecture 2011 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.
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 \textit{initiated}.

**IU**

\textit{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 \textit{Load} includes loads into integer or floating-point registers, block loads, and alternate address space variants of those instructions. See also \textit{load-store} and \textit{store}, the definitions of which are mutually exclusive with \textit{load}.

**load-store**

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

**may**

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

**Memory Management Unit (MMU)**

The address translation hardware in an Oracle SPARC Architecture implementation that translates 64-bit virtual address into underlying hardware addresses. The MMU is composed of the ASRs and ASI registers used to manage address translation. See also \textit{context real address}, and \textit{virtual address}.

**microcore**

Synonym for \textit{pipeline}.

**MMU**

Abbreviation for \textit{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 Oracle SPARC Architecture-compliant products. Synonym for \textit{shall}.

**next program counter**

Conceptually, a register that contains the address of the instruction to be executed next if a trap does not occur.

**NFO**

Nonfault access only.

**nonfaulting load**

A load operation that behaves identically to a normal load operation, except when supplied an invalid effective address by software. In that case, a regular load triggers an exception whereas a nonfaulting load appears to ignore the exception and loads its destination register with a value of zero (on an Oracle SPARC Architecture processor, hardware treats regular and nonfaulting loads identically; the distinction is made in trap handler software). Contrast with \textit{speculative load}.

**nonprivileged**

An adjective that describes

(1) the state of the virtual processor when \texttt{PSTATE.priv = 0}, that is, when it is in nonprivileged mode;

(2) virtual processor state information that is accessible to software regardless of the current privilege mode; for example, nonprivileged registers, nonprivileged ASRs, or, in general, nonprivileged state;

(3) an instruction that can be executed in any privilege mode (privileged or nonprivileged).

**nonprivileged mode**

The mode in which a virtual processor is operating when executing application software (at the lowest privilege level). Nonprivileged mode is defined by \texttt{PSTATE.priv = 0}. See also \textit{privileged} and \textit{hyperprivileged}.

**nontranslating ASI**

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

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

NUMA  Nonuniform memory access.

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

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

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

opcode  A bit pattern that identifies a particular instruction.

optional  A feature not required for Oracle SPARC Architecture 2011 compliance.

PC  Program counter.

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

PIL  Processor Interrupt Level register.

pipeline  Refers to an execution pipeline, the basic collection of hardware needed to execute instructions. *Synonym for microcore*. See also processor, strand, thread, and virtual processor.

prefetchable  (1) An attribute of a memory location that indicates to an MMU that PREFETCH operations to that location may be applied.

(2) A memory location condition for which the system designer has determined that no undesirable effects will occur if a PREFETCH operation to that location is allowed to succeed. Typically, normal memory is prefetchable.

Nonprefetchable locations include those that, when read, change state or cause external events to occur. For example, some I/O devices are designed with registers that clear on read; others have registers that initiate operations when read. See also *side effect*.

privileged  An adjective that describes:

(1) the state of the virtual processor when PSTATE.priv = 1,

that is, when the virtual processor is in privileged mode;

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

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

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

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

processor core  *Synonym for physical core*.

processor module  *Synonym for processor*.

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

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

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

RA  Real address.
RAS Reliability, Availability, and Serviceability

RAW Read After Write (hazard)

rd Rounding direction.

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

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

A reserved instruction field must read as 0, unless the implementation supports extended instructions within the field. The behavior of an Oracle SPARC Architecture 2011 virtual processor when it encounters a nonzero value in a reserved instruction field is as defined in Reserved Opcodes and Instruction Fields on page 91.

A reserved bit combination within an instruction field is defined in Chapter 7, Instructions. In all cases, an Oracle SPARC Architecture 2011 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 52).

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

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

retired An instruction is said to be “retired” when one of the following two events has occurred:

1. A precise trap has been taken, with TPC containing the instruction’s address (the instruction has not changed architectural state in this case).
2. The instruction’s execution has progressed to a point at which architectural state affected by the instruction has been updated such that all three of the following are true:
   - The PC has advanced beyond the instruction.
   - Except for deferred trap handlers, no consumer in the same instruction stream can see the old values and all consumers in the same instruction stream will see the new values.
   - Stores are visible to all loads in the same instruction stream, including stores to noncacheable locations.

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

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

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

shall Synonym for must.

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

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

SIMD Single Instruction/Multiple Data; a class of instructions that perform identical operations on multiple data contained (or “packed”) in each source operand.
speculative load  A load operation that is issued by a virtual processor speculatively, that is, before it is known whether the load will be executed in the flow of the program. Speculative accesses are used by hardware to speed program execution and are transparent to code. An implementation, through a combination of hardware and system software, must nullify speculative loads on memory locations that have side effects; otherwise, such accesses produce unpredictable results. Contrast with nonfaulting load.

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

strand  The hardware state that must be maintained in order to execute a software thread. See also pipeline, processor, thread, and virtual processor.

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

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

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

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

system  A set of virtual processors that share a common hardware memory address space.

taken  A control-transfer instruction (CTI) is taken when the CTI writes the target address value into NPC. A trap is taken when the control flow changes in response to an exception, reset, Tcc instruction, or interrupt. An exception must be detected and recognized before it can cause a trap to be taken.

TBA  Trap base address.

thread  A software entity that can be executed on hardware. See also pipeline, processor, strand, and virtual processor.

TNPC  Trap-saved next program counter.

TPC  Trap-saved program counter.

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

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

TSO  Total Store Order (a memory model).

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

UA-2011  Oracle SPARC Architecture 2011

unassigned  A value (for example, an ASI number), the semantics of which are not architecturally mandated and which may be determined independently by each implementation within any guidelines given.
undefined An aspect of the architecture that has deliberately been left unspecified. Software should have no expectation of, nor make any assumptions about, an undefined feature or behavior. Use of such a feature can deliver unexpected results and may or may not cause a trap. An undefined feature may vary among implementations, and may also vary over time on a given implementation.

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

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

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 Abbreviation for virtual address.

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

virtual core, virtual processor core Synonyms for virtual processor.

virtual processor The term virtual processor, or virtual processor core, is used to identify each strand in a processor. At any given time, an operating system can have a different thread scheduled on each virtual processor. See also pipeline, processor, strand, and thread.

VIS Abbreviation for VISTM Instruction Set.

VP 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.
Architecture Overview

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

The Oracle SPARC Architecture 2011 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 Oracle SPARC Architecture 2011 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 Oracle SPARC Architecture 2011 processors may not apply to SPARC V9 processors produced by other vendors.

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

3.1 The Oracle SPARC Architecture 2011

This section briefly describes features, attributes, and components of the Oracle SPARC Architecture 2011 and, further, describes correct implementation of the architecture specification and SPARC V9-compliance levels.

3.1.1 Features

The Oracle SPARC Architecture 2011, like its ancestor SPARC V9, includes the following principal features:

- A linear 64-bit address space with 64-bit addressing.
- 32-bit wide instructions — These are aligned on 32-bit boundaries in memory. Only load and store instructions access memory and perform I/O.
Few addressing modes — A memory address is given as either “register + register” or “register + immediate”.

Triadic register addresses — Most computational instructions operate on two register operands or one register and a constant and place the result in a third register.

A large windowed register file — At any one instant, a program sees 8 global integer registers plus a 24-register window of a larger register file. The windowed registers can be used as a cache of procedure arguments, local values, and return addresses.

Floating point — The architecture provides an IEEE 754-compatible floating-point instruction set, operating on a separate register file that provides 32 single-precision (32-bit), 32 double-precision (64-bit), and 16 quad-precision (128-bit) overlayed registers.

Fast trap handlers — Traps are vectored through a table.

Multiprocessor synchronization instructions — Multiple variations of atomic load-store memory operations are supported.

Predicted branches — The branch with prediction instructions allows the compiler or assembly language programmer to give the hardware a hint about whether a branch will be taken.

Branch elimination instructions — Several instructions can be used to eliminate branches altogether (for example, Move on Condition). Eliminating branches increases performance in superscalar and superpipelined implementations.

Hardware trap stack — A hardware trap stack is provided to allow nested traps. It contains all of the machine state necessary to return to the previous trap level. The trap stack makes the handling of faults and error conditions simpler, faster, and safer.

In addition, Oracle SPARC Architecture 2011 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, Oracle SPARC Architecture 2011 provides multiple sets; typically, one set is used at each trap level.

Extended instruction set — Oracle SPARC Architecture 2011 provides many instruction set extensions, including the VIS instruction set for “vector” (SIMD) data operations.

More detailed, specific instruction descriptions — Oracle SPARC Architecture 2011 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 — Oracle SPARC Architecture 2011 provides a blueprint for the software view of the UltraSPARC MMU (TTEs and TSBs).

3.1.2 Attributes

Oracle SPARC Architecture 2011 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, Oracle SPARC Architecture 2011 allows for a spectrum of processor and system implementations at a variety of price/performance points for a range of applications, including scientific/engineering, programming, real-time, and commercial applications.
3.1.2.1 Design Goals

The Oracle SPARC Architecture 2011 architecture is designed to be a target for optimizing compilers and high-performance hardware implementations. This specification documents the Oracle SPARC Architecture 2011 and provides a design spec against which an implementation can be verified, using appropriate verification software.

3.1.2.2 Register Windows

The Oracle SPARC Architecture 2011 architecture is derived from the SPARC architecture, which was formulated at Sun Microsystems in 1984 through 1987. The SPARC architecture is, in turn, based on the RISC I and II designs engineered at the University of California at Berkeley from 1980 through 1982. The SPARC “register window” architecture, pioneered in the UC Berkeley designs, allows for straightforward, high-performance compilers and a reduction in memory load/store instructions.

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

3.1.3 System Components

The Oracle SPARC Architecture 2011 allows for a spectrum of subarchitectures, such as cache system.

3.1.3.1 Binary Compatibility

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

Although different Oracle SPARC Architecture 2011 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, Oracle SPARC Architecture 2011 is binary upward-compatible from SPARC V9 for applications running in nonprivileged mode that conform to the SPARC V9 ABI and upward-compatible from SPARC V8 for applications running in nonprivileged mode that conform to the SPARC V8 ABI.

3.1.3.2 Oracle SPARC Architecture 2011 MMU

Although the SPARC V9 architecture allows its implementations freedom in their MMU designs, Oracle SPARC Architecture 2011 defines a common MMU architecture (see Chapter 14, Memory Management) with some specifics left to implementations (see processor implementation documents).

3.1.3.3 Privileged Software

Oracle SPARC Architecture 2011 does not assume that all implementations must execute identical privileged software (operating systems). Thus, certain traits that are visible to privileged software may be tailored to the requirements of the system.
3.1.4 Architectural Definition

The Oracle SPARC Architecture 2011 is defined by the chapters and appendixes of this specification. A correct implementation of the architecture interprets a program strictly according to the rules and algorithms specified in the chapters and appendixes.

Oracle SPARC Architecture 2011 defines a set of implementations that conform to the SPARC V9 architecture, Level 1.

3.1.5 Oracle SPARC Architecture 2011 Compliance with SPARC V9 Architecture

Oracle SPARC Architecture 2011 fully complies with SPARC V9 Level 1 (nonprivileged). It partially complies with SPARC V9 Level 2 (privileged).

3.1.6 Implementation Compliance with Oracle SPARC Architecture 2011

Compliant implementations must not add to or deviate from this standard except in aspects described as implementation dependent. Appendix B, Implementation Dependencies lists all Oracle SPARC Architecture 2011, SPARC V9, and SPARC V8 implementation dependencies. Documents for specific Oracle SPARC Architecture 2011 processor implementations describe the manner in which implementation dependencies have been resolved in those implementations.

IMPL. DEP. #1-V8: Whether an instruction complies with Oracle SPARC Architecture 2011 by being implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.

3.2 Processor Architecture

An Oracle SPARC 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 Oracle SPARC Architecture virtual processor can run in nonprivileged mode, privileged mode, or in mode(s) of greater privilege. In privileged mode, the processor can execute nonprivileged and privileged instructions. In nonprivileged mode, the processor can only execute nonprivileged instructions. In nonprivileged or privileged mode, an attempt to execute an instruction requiring greater privilege than the current mode causes a trap.

3.2.1 Integer Unit (IU)

An Oracle SPARC Architecture 2011 implementation’s integer unit contains the general-purpose registers and controls the overall operation of the virtual processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU.
IMPL. DEP. #2-V8: An Oracle SPARC Architecture implementation may contain from 72 to 640
general-purpose 64-bit \( R \) registers. This corresponds to a grouping of the registers into \( \text{MAXPGL} + 1 \)
sets of global \( R \) registers plus a circular stack of \( N \_ \text{REG} \_ \text{WINDOWS} \) sets of 16 registers each, known as
register windows. The number of register windows present \( N \_ \text{REG} \_ \text{WINDOWS} \) is implementation
dependent, within the range of 3 to 32 (inclusive).

3.2.2 Floating-Point Unit (FPU)

An Oracle SPARC Architecture 2011 implementation’s FPU has thirty-two 32-bit (single-precision)
floating-point registers, thirty-two 64-bit (double-precision) floating-point registers, and sixteen 128-
bit (quad-precision) floating-point registers, some of which overlap.

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

If the FPU is not enabled, then an attempt to execute a floating-point instruction generates an
\( \text{fp\_disabled} \) trap and the \( \text{fp\_disabled} \) trap handler software must either
- Enable the FPU (if present) and reexecute the trapping instruction, or
- Emulate the trapping instruction in software.

3.3 Instructions

Instructions fall into the following basic categories:
- Memory access
- Integer arithmetic / logical / shift
- Control transfer
- State register access
- Floating-point operate
- Conditional move
- Register window management
- SIMD (single instruction, multiple data) instructions

These classes are discussed in the following subsections.

3.3.1 Memory Access

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

The destination field of the load/store instruction specifies either one or two \( R \) registers or one, two,
or four \( F \) registers that supply the data for a store or that receive the data from a load.

Integer load and store instructions support byte, halfword (16-bit), word (32-bit), and extended-word
(64-bit) accesses. There are versions of integer load instructions that perform either sign-extension or
zero-extension on 8-bit, 16-bit, and 32-bit values as they are loaded into a 64-bit destination register.
Floating-point load and store instructions support word, doubleword, and quadword\(^1\) memory
accesses.

\(^1\) No Oracle SPARC Architecture processor currently implements the LDQF instruction in hardware; it generates an exception and is
emulated in software running at a higher privilege level.
CASA, CASXA, and LDSTUB are special atomic memory access instructions that concurrent processes use for synchronization and memory updates.

\textbf{Note} The SWAP instruction is also specified, but it is deprecated and should not be used in newly developed software.

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

### 3.3.1.1 Memory Alignment Restrictions

A memory access on an Oracle SPARC 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 \textit{Memory Alignment Restrictions} on page 77.

### 3.3.1.2 Addressing Conventions

The Oracle SPARC 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 Oracle SPARC Architecture also supports little-endian byte order for data accesses only: the address of a quadword, doubleword, word, or halfword is the address of its least significant byte. Increasing the address means increasing the significance of the data unit being accessed.

Addressing conventions are illustrated in FIGURE 6-2 on page 79 and FIGURE 6-3 on page 81.

### 3.3.1.3 Addressing Range

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

### 3.3.1.4 Load/Store Alternate

Variants of load/store instructions, the \textit{load/store alternate} instructions, can specify an arbitrary 8-bit address space identifier for the load/store data access.

Access to alternate spaces \( 00_{16} \)–\( 2F_{16} \) is restricted to privileged software, access to alternate spaces \( 30_{16} \)–\( 7F_{16} \) is restricted to hyperprivileged software, and access to alternate spaces \( 80_{16} \)–\( FF_{16} \) is unrestricted. Some of the ASIs are available for implementation-dependent uses. Privileged software can use the implementation-dependent ASIs to access special protected registers, such as cache control registers, virtual processor state registers, and other processor-dependent or system-dependent values. See \textit{Address Space Identifiers (ASIs)} on page 81 for more information.

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

\textbf{Note} The SWAPA instruction is also specified, but it is deprecated and should not be used in newly developed software.
3.3.1.5 Separate Instruction and Data Memories

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

In such split-memory systems, the coherency mechanism may be split, so a write\(^1\) into data memory is not immediately reflected in instruction memory. For this reason, programs that modify their own instruction stream (self-modifying code\(^2\)) and that wish to be portable across all Oracle SPARC Architecture (and SPARC V9) processors must issue FLUSH instructions, or a system call with a similar effect, to bring the instruction and data caches into a consistent state.

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

3.3.1.6 Input/Output (I/O)

The Oracle SPARC 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).

\textbf{IMPL. DEP. #123-V9:} The semantic effect of accessing input/output (I/O) locations is implementation dependent.

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

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

3.3.1.7 Memory Synchronization

Two instructions are used for synchronization of memory operations: FLUSH and MEMBAR. Their operation is explained in \textit{Flush Instruction Memory} on page 166 and \textit{Memory Barrier} on page 256, respectively.

\textbf{Note} STBAR is also available, but it is deprecated and should not be used in newly developed software.

3.3.2 Integer Arithmetic / Logical / Shift Instructions

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

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

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

\(^2\) this is practiced, for example, by software such as debuggers and dynamic linkers
3.3.3 Control Transfer

Control-transfer instructions (CTIs) include PC-relative branches and calls, register-indirect jumps, and conditional traps. Most of the control-transfer instructions are delayed; that is, the instruction immediately following a control-transfer instruction in logical sequence is dispatched before the control transfer to the target address is completed. Note that the next instruction in logical sequence may not be the instruction following the control-transfer instruction in memory.

The instruction following a delayed control-transfer instruction is called a delay instruction. Setting the annul bit in a conditional delayed control-transfer instruction causes the delay instruction to be annulled (that is, to have no effect) if and only if the branch is not taken. Setting the annul bit in an unconditional delayed control-transfer instruction (“branch always”) causes the delay instruction to be always annulled.

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

Branch and CALL instructions use PC-relative displacements. The jump and link (JMPL) and return (RETURN) instructions use a register-indirect target address. They compute their target addresses either as the sum of two R registers or as the sum of an R register and a 13-bit signed immediate value. The “branch on condition codes without prediction” instruction provides a displacement of ±8 Mbytes; the “branch on condition codes with prediction” instruction provides a displacement of ±1 Mbyte; the “branch on register contents” instruction provides a displacement of ±128 Kbytes; and the CALL instruction’s 30-bit word displacement allows a control transfer to any address within ±2 gigabytes (±231 bytes).

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

3.3.4 State Register Access

3.3.4.1 Ancillary State Registers

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

IMPL. DEP. #8-V8-Cs20: Ancillary state registers (ASRs) in the range 0–27 that are not defined in Oracle SPARC Architecture 2011 are reserved for future architectural use.

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

3.3.4.2 PR State Registers

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

Floating-point operate (FPop) instructions perform all floating-point calculations; they are register-to-register instructions that operate on the floating-point registers. FPops compute a result that is a function of one, two, or three source operands. The groups of instructions that are considered FPops are listed in Floating-Point Operate (FPop) Instructions on page 90.

3.3.6 Conditional Move

Conditional move instructions conditionally copy a value from a source register to a destination register, depending on an integer or floating-point condition code or on the contents of an integer register. These instructions can be used to reduce the number of branches in software.

3.3.7 Register Window Management

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

3.3.8 SIMD

Oracle SPARC Architecture 2011 includes SIMD (single instruction, multiple data) instructions, also known as "vector" instructions, which allow a single instruction to perform the same operation on multiple data items, totalling 64 bits, such as eight 8-bit, four 16-bit, or two 32-bit data items. These operations are part of the "VIS" extensions.

3.4 Traps

A trap is a vectored transfer of control to privileged software through a trap table that may contain the first 8 instructions (32 for some frequently used traps) of each trap handler. The base address of the table is established by software in a state register (the Trap Base Address register, TBA). The displacement within the table is encoded in the type number of each trap and the level of the trap. Part of the trap table is reserved for hardware traps, and part of it is reserved for software traps generated by trap (Tcc) instructions.

A trap causes the current PC and NPC to be saved in the TPC and TNPC registers. It also causes the CCR, ASI, PSTATE, and CWP registers to be saved in TSTATE. TPC, TNPC, and TSTATE are entries in a hardware trap stack, where the number of entries in the trap stack is equal to the number of supported trap levels. A trap also sets bits in the PSTATE register and typically increments the GL register. Normally, the CWP is not changed by a trap; on a window spill or fill trap, however, the CWP is changed to point to the register window to be saved or restored.

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

See Chapter 12, Traps, for a complete description of traps.
Data Formats

The Oracle SPARC Architecture recognizes these fundamental data types:

- Signed integer: 8, 16, 32, and 64 bits
- Unsigned integer: 8, 16, 32, and 64 bits
- SIMD data formats: Uint8 SIMD (32 bits), Int16 SIMD (64 bits), and Int32 SIMD (64 bits)
- Floating point: 32, 64, and 128 bits

The widths of the data types are as follows:

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

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

Data formats are described in these sections:

- **Integer Data Formats** on page 24.
- **Floating-Point Data Formats** on page 27.
- **SIMD Data Formats** on page 29.

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

- **Signed Integer Doubleword (64 bits)** on page 25.
- **Unsigned Integer Doubleword (64 bits)** on page 26.
- **Floating Point, Double Precision (64 bits)** on page 27.
- **Floating Point, Quad Precision (128 bits)** on page 28.
4.1 Integer Data Formats

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

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

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

**TABLE 4-2** Integer Doubleword/Extended-word 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>SD-0</td>
<td>signed_dbl_integer(63:32)</td>
<td>$n \pmod{8} = 0$</td>
<td>$r \pmod{2} = 0$</td>
</tr>
<tr>
<td>SD-1</td>
<td>signed_dbl_integer(31:0)</td>
<td>$(n + 4) \pmod{8} = 4$</td>
<td>$(r + 1) \pmod{2} = 1$</td>
</tr>
<tr>
<td>SX</td>
<td>signed_ext_integer(63:0)</td>
<td>$n \pmod{8} = 0$</td>
<td>—</td>
</tr>
<tr>
<td>UD-0</td>
<td>unsigned_dbl_integer(63:32)</td>
<td>$n \pmod{8} = 0$</td>
<td>$r \pmod{2} = 0$</td>
</tr>
<tr>
<td>UD-1</td>
<td>unsigned_dbl_integer(31:0)</td>
<td>$(n + 4) \pmod{8} = 4$</td>
<td>$(r + 1) \pmod{2} = 1$</td>
</tr>
<tr>
<td>UX</td>
<td>unsigned_ext_integer(63:0)</td>
<td>$n \pmod{8} = 0$</td>
<td>—</td>
</tr>
</tbody>
</table>

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

The data types are illustrated in the following subsections.

4.1.1 Signed Integer Data Types

Figures in this section illustrate the following signed data types:

- Signed integer byte
- Signed integer halfword
- Signed integer word
- Signed integer doubleword
- Signed integer extended-word
4.1.1.1 Signed Integer Byte, Halfword, and Word

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

```
SB
S |
7 6 0
```

```
SH
S |
15 14 0
```

```
SW
S |
31 30 0
```

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

4.1.1.2 Signed Integer Doubleword (64 bits)

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

```
SD–0
S |
31 30 0
```

```
signed_int_doubleword{62:32}
```

```
SD–1
S |
31 0
```

```
signed_int_doubleword{31:0}
```

FIGURE 4-2 Signed Integer Double Data Format

4.1.1.3 Signed Integer Extended-Word (64 bits)

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

```
SX
S |
63 62 0
```

```
signed_int_extended
```

FIGURE 4-3 Signed Integer Extended-Word Data Format

4.1.2 Unsigned Integer Data Types

Figures in this section illustrate the following unsigned data types:

- Unsigned integer byte
- Unsigned integer halfword
- Unsigned integer word
- Unsigned integer doubleword
- Unsigned integer extended-word
4.1.2.1 Unsigned Integer Byte, Halfword, and Word

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

![Unsigned Integer Byte](image1)

4.1.2.2 Unsigned Integer Doubleword (64 bits)

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

![Unsigned Integer Double Data Format](image2)

4.1.2.3 Unsigned Extended Integer (64 bits)

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

![Unsigned Extended Integer Data Format](image3)

4.1.3 Tagged Word (32 bits)

FIGURE 4-7 illustrates the tagged word data format.

![Tagged Word Data Format](image4)
4.2 Floating-Point Data Formats

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

4.2.1 Floating Point, Single Precision (32 bits)

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

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\exp{7:0}$</th>
<th>$\text{fraction}{22:0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>23 22</td>
</tr>
</tbody>
</table>

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

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

- Normalized value ($0 < e < 255$): $(-1)^s \times 2^{e-127} \times 1.f$
- Subnormal value ($e = 0$): $(-1)^s \times 2^{-126} \times 0.f$
- Zero ($e = 0, f = 0$): $(-1)^s \times 0$
- Signalling NaN: $s = u; e = 255$ (max); $f = .0uu--uu$
  (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); $f = .000--00$

4.2.2 Floating Point, Double Precision (64 bits)

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

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\exp{10:0}$</th>
<th>$\text{fraction}{51:32}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>20 19</td>
</tr>
</tbody>
</table>

**FIGURE 4-9** Floating-Point Double-Precision Data Format
4.2.3 Floating Point, Quad Precision (128 bits)

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

![Figure 4-10: Floating-Point Quad-Precision Data Format](image)

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

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

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

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

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

<table>
<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 (\mod 8) (n)</td>
<td>0 mod 2 (f)</td>
</tr>
<tr>
<td>FD-1</td>
<td>fraction[31:0]</td>
<td>0 mod 4 (\mod 8) (n + 4)</td>
<td>1 mod 2 (f + 1^0)</td>
</tr>
<tr>
<td>FQ-0</td>
<td>s:exp[14:0]:fraction[111:96]</td>
<td>0 mod 4 (\mod 8) (n)</td>
<td>0 mod 4 (f)</td>
</tr>
<tr>
<td>FQ-1</td>
<td>fraction[95:64]</td>
<td>0 mod 4 (\mod 8) (n + 4)</td>
<td>1 mod 4 (f + 1^0)</td>
</tr>
<tr>
<td>FQ-2</td>
<td>fraction[63:32]</td>
<td>0 mod 4 (\mod 8) (n + 8)</td>
<td>2 mod 4 (f + 2)</td>
</tr>
<tr>
<td>FQ-3</td>
<td>fraction[31:0]</td>
<td>0 mod 4 (\mod 8) (n + 12)</td>
<td>3 mod 4 (f + 3^0)</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 double-word-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 \(\leq 31\)).

4.3 SIMD Data Formats

SIMD (single instruction/multiple data) instructions perform identical operations on multiple data contained (“packed”) in each source operand. This section describes the data formats used by SIMD instructions.

Conversion between the different SIMD data formats can be achieved through SIMD multiplication or by the use of the SIMD data formatting instructions.
4.3.1 Uint8 SIMD Data Format

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

![Uint8 SIMD](image)

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

4.3.2 Int16 SIMD Data Formats

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

![Int16 SIMD](image)

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

4.3.3 Int32 SIMD Data Format

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

![Int32 SIMD](image)

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

---

**Programming Note**

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

The following registers are described in this chapter:
- **General-Purpose R Registers** on page 32.
- **Floating-Point Registers** on page 38.
- **Floating-Point State Register (FSR)** on page 42.
- **Ancillary State Registers** on page 48. The following registers are included in this category:
  - 32-bit Multiply/Divide Register (Y) (ASR 0) on page 50.
  - Integer Condition Codes Register (CCR) (ASR 2) on page 50.
  - Address Space Identifier (ASI) Register (ASR 3) on page 51.
  - Tick (TICK) Register (ASR 4) on page 52.
  - Program Counters (PC, NPC) (ASR 5) on page 53.
  - Floating-Point Registers State (FPRS) Register (ASR 6) on page 53.
  - General Status Register (GSR) (ASR 19) on page 54.
  - SOFTINT® Register (ASRs 20, 21, 22) on page 55.
  - SOFTINT_SET® Pseudo-Register (ASR 20) on page 55.
  - SOFTINT_CLR® Pseudo-Register (ASR 21) on page 56.
  - System Tick (STICK) Register (ASR 24) on page 56.
  - System Tick Compare (STICK_CMPR®) Register (ASR 25) on page 57.
  - Compatibility Feature Register (CFR) (ASR 26) on page 59.
  - Compatibility Feature Register (CFR) (ASR 26) on page 59.
- **Register-Window PR State Registers** on page 61. The following registers are included in this subcategory:
  - Current Window Pointer (CWP®) Register (PR 9) on page 61.
  - Savable Windows (CANSAVE®) Register (PR 10) on page 62.
  - Restorable Windows (CANRESTORE®) Register (PR 11) on page 62.
  - Clean Windows (CLEANWIN®) Register (PR 12) on page 62.
  - Other Windows (OTHERWIN®) Register (PR 13) on page 63.
  - Window State (WSTATE®) Register (PR 14) on page 63.
- **Non-Register-Window PR State Registers** on page 65. The following registers are included in this subcategory:
  - Trap Program Counter (TPC®) Register (PR 0) on page 65.
  - Trap Next PC (TNPC®) Register (PR 1) on page 65.
  - Trap State (TSTATE®) Register (PR 2) on page 66.
  - Trap Type (TT®) Register (PR 3) on page 67.
  - Trap Base Address (TBA®) Register (PR 5) on page 67.
  - Processor State (PSTATE®) Register (PR 6) on page 68.
  - Trap Level Register (TL®) (PR 7) on page 71.
  - Processor Interrupt Level (PIL®) Register (PR 8) on page 72.
  - Global Level Register (GL®) (PR 16) on page 73.

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

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

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

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

**Programming Notes**

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

- If software writes a reserved register field to any value other than (1) zero or (2) a value previously read from that field, it is considered a software error. Such an error:
  - may or may not be detected or reported (for example, by a trap) by Oracle SPARC Architecture 2011 processors (and software should not expect that it will be)
  - may cause a trap or cause other unintended behavior when executed on future Oracle SPARC Architecture processors

When a register is read, software should not assume that register fields reserved in Oracle SPARC Architecture 2011 will read as 0 or any other particular value, either now or in the future.

**Implementation Notes**

When a register is read by software, an Oracle SPARC Architecture 2011 virtual processor should return a value of zero for any bits reserved in Oracle SPARC Architecture 2011.

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

5.2 General-Purpose R Registers

An Oracle SPARC Architecture virtual processor contains an array of general-purpose 64-bit R registers. The array is partitioned into \textit{MAXPGL} + 1 sets of eight global registers, plus \textit{N \_ REG \_ WINDOWS} groups of 16 registers each. The value of \textit{N \_ REG \_ WINDOWS} in an Oracle SPARC Architecture implementation falls within the range 3 to 32 (inclusive).
One set of 8 global registers is always visible. At any given time, a group of 24 registers, known as a register window, is also visible. A register window comprises the 16 registers from the current 16-register group (referred to as 8 in registers and 8 local registers), plus half of the registers from the next 16-register group (referred to as 8 out registers). See FIGURE 5-1.

SPARC instructions use 5-bit fields to reference R registers. That is, 32 R registers are visible to software at any moment. Which 32 out of the full set of R registers are visible is described in the following sections. The visible 32 R registers are named R[0] through R[31], illustrated in FIGURE 5-1.

### 5.2.1 Global R Registers

Registers R[0]–R[7] refer to a set of eight registers called the global registers (labelled g0 through g7). At any time, one of MAXPGL +1 sets of eight registers is enabled and can be accessed as the current set of global registers. The currently enabled set of global registers is selected by the GL register. See Global Level Register (GLP) (PR 16) on page 73.

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

A set of 24 R registers that is visible as R[8]–R[31] at any given time is called a “register window”. The registers that become R[8]–R[15] in a register window are called the out registers of the window. Note that the in registers of a register window become the out registers of an adjacent register window. See TABLE 5-1 and FIGURE 5-2.

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

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

V9 Compatibility Notes

In the SPARC V9 architecture, the number of 16-register windowed register sets, N_REG_WINDOWS, ranges from 3† to 32 (impl. dep. #2-V8).

The maximum global register set index in the Oracle SPARC Architecture, MAXPGL, ranges from 2 to 15. The number of implemented global register sets is MAXPGL + 1.

The total number of R registers in a given Oracle SPARC Architecture implementation is:

\[(N\_REG\_WINDOWS \times 16) + ((MAXPGL + 1) \times 8)\]

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

† The controlling equation for register window operation, as described in 5.6.7.1 on page 64, is:

\[CANSAVE + CANRESTORE + OTHERWIN = N\_REG\_WINDOWS – 2\]

Since N_REG_WINDOWS cannot be negative, the minimum number of implemented register windows is “2”. However, since the SAVED and RESTORED instructions increment CANSAVE and CANRESTORE, the minimum value of N_REG_WINDOWS in practice increases to “3”. An implementation with N_REG_WINDOWS = 2 would not be able to support use of the SAVED and RESTORED instructions — in such an implementation, a spill trap handler would have to emulate the SAVE instruction (the one that caused the spill trap) in its entirety (including its addition semantics) and the spill handler would have to end with a DONE instruction instead of RETRY.
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 \textit{ins} with one adjacent window and its \textit{outs} with another. The \textit{outs} of the CWP – 1 (\textit{modulo} \textit{N\_REG\_WINDOWS}) window are addressable as the \textit{ins} of the current window, and the \textit{outs} in the current window are the \textit{ins} of the CWP + 1 (\textit{modulo} \textit{N\_REG\_WINDOWS}) window. The \textit{locals} are unique to each window.

Register address \( o \), where \( 8 \leq o \leq 15 \), refers to exactly the same \textit{out} register before the register window is advanced by a SAVE instruction (CWP is incremented by 1 (\textit{modulo} \textit{N\_REG\_WINDOWS})) as does register address \( o + 16 \) after the register window is advanced. Likewise, register address \( i \), where \( 24 \leq i \leq 31 \), refers to exactly the same \textit{in} register before the register window is restored by a RESTORE instruction (CWP is decremented by 1 (\textit{modulo} \textit{N\_REG\_WINDOWS})) as does register address \( i – 16 \) after the window is restored. See FIGURE 5-2 on page 35 and FIGURE 5-3 on page 37.

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

#### Programming Note

Since the procedure call instructions (CALL and JMPL) do not change the CWP, a procedure can be called without changing the window. S

---

Since CWP arithmetic is performed modulo \textit{N\_REG\_WINDOWS}, the highest-numbered implemented window overlaps with window 0. The \textit{outs} of window \textit{N\_REG\_WINDOWS} – 1 are the \textit{ins} of window 0. Implemented windows are numbered contiguously from 0 through \textit{N\_REG\_WINDOWS} – 1.
Because the windows overlap, the number of windows available to software is 1 less than the number of implemented windows; that is, \( N_{\text{REG\_WINDOWS}} - 1 \). When the register file is full, the \textit{outs} of the newest window are the \textit{ins} of the oldest window, which still contains valid data.

Window overflow is detected by the \texttt{CANSAVE} register, and window underflow is detected by the \texttt{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 \texttt{SAVE} instruction, the \textit{local} and \textit{out} registers are guaranteed to contain either zeroes or valid data from the current context. If software executes a \texttt{RESTORE} and later executes a \texttt{SAVE}, then the contents of the resulting window’s \textit{local} and \textit{out} registers are not guaranteed to be preserved between the \texttt{RESTORE} and the \texttt{SAVE}. Those registers may even have been written with “dirty” data, that is, data created by software running in a different context. However, if the clean window protocol is being used, system software must guarantee that registers in the current window after a \texttt{SAVE} always contain only zeroes or valid data from that context. See \texttt{Clean Windows (CLEANWIN\textsuperscript{P}) Register (PR 12)} on page 62, \texttt{Savable Windows (CANSAVE\textsuperscript{P}) Register (PR 10)} on page 62, and \texttt{Restorable Windows (CANRESTORE\textsuperscript{P}) Register (PR 11)} on page 62.

\textbf{Implementation Note} An Oracle SPARC 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.

\textit{Register Window Management Instructions} on page 88 describes how the windowed integer registers are managed.

\footnote{For example, any of those 16 registers might be altered due to the occurrence of a trap between the \texttt{RESTORE} and the \texttt{SAVE}, or might be altered during the \texttt{RESTORE} operation due to the way that register windows are implemented. After a \texttt{RESTORE} instruction executes, software must assume that the values of the affected 16 registers from before the \texttt{RESTORE} are unrecoverable.}
FIGURE 5-3 Windowed R Registers for $N_{REG\_WINDOWS} = 8$

In FIGURE 5-3, $N_{REG\_WINDOWS} = 8$. The eight *global* registers are not illustrated. CWP = 0, CANSAVE = 4, OTHERWIN = 1, and CANRESTORE = 1. If the procedure using window $w_0$ 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.

### 5.2.3 Special R Registers

The use of two of the R registers is fixed, in whole or in part, by the architecture:
- The value of $R[0]$ is always zero; writes to it have no program-visible effect.
- The **CALL** instruction writes its own address into register $R[15]$ (*out* register 7).
Register-Pair Operands. LDTW, LDTWA, STTW, and STTWA instructions access a pair of words (“twin words”) in adjacent R registers and require even-odd register alignment. The least significant bit of an R register number in these instructions is unused and must always be supplied as 0 by software.

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

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

5.3 Floating-Point Registers

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

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

The floating-point registers are arranged so that some of them overlap, that is, are aliased. The layout and numbering of the floating-point registers are shown in TABLE 5-2. Unlike the windowed R registers, all of the floating-point registers are accessible at any time. The floating-point registers can be read and written by floating-point operate (FPop1/FPop2 format) instructions, by load/store single/double/quad floating-point instructions, by VIS™ instructions, and by block load and block store instructions.

<table>
<thead>
<tr>
<th>Register</th>
<th>Single Precision (32-bit)</th>
<th>Assembly Language</th>
<th>Double Precision (64-bit)</th>
<th>Assembly Language</th>
<th>Quad Precision (128-bit)</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_S[0] )</td>
<td>%f0</td>
<td>63:32</td>
<td>( F_D[0] )</td>
<td>%d0</td>
<td>127:64</td>
<td>( F_Q[0] )</td>
</tr>
<tr>
<td>( F_S[1] )</td>
<td>%f1</td>
<td>31:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F_S[3] )</td>
<td>%f3</td>
<td>31:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F_S[5] )</td>
<td>%f5</td>
<td>31:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F_S[7] )</td>
<td>%f7</td>
<td>31:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F_S[8] )</td>
<td>%f8</td>
<td>63:32</td>
<td>( F_D[8] )</td>
<td>%d8</td>
<td>127:64</td>
<td>( F_Q[8] )</td>
</tr>
<tr>
<td>( F_S[9] )</td>
<td>%f9</td>
<td>31:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F_S[10] )</td>
<td>%f10</td>
<td>63:32</td>
<td>( F_D[10] )</td>
<td>%d10</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>( F_S[11] )</td>
<td>%f11</td>
<td>31:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Assembly Language</td>
<td>Bits</td>
<td>Register</td>
<td>Assembly Language</td>
<td>Bits</td>
<td>Register</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-------------------</td>
<td>-------</td>
<td>-----------</td>
</tr>
<tr>
<td>F_S[15]</td>
<td>%f15</td>
<td>31:0</td>
<td>F_D[18]</td>
<td>%d18</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>F_S[17]</td>
<td>%f17</td>
<td>31:0</td>
<td>F_D[22]</td>
<td>%d22</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>F_S[19]</td>
<td>%f19</td>
<td>31:0</td>
<td>F_D[26]</td>
<td>%d26</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>F_S[22]</td>
<td>%f22</td>
<td>63:32</td>
<td>F_D[32]</td>
<td>%d32</td>
<td>127:64</td>
<td></td>
</tr>
<tr>
<td>F_S[23]</td>
<td>%f23</td>
<td>31:0</td>
<td>F_D[34]</td>
<td>%d34</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>F_S[25]</td>
<td>%f25</td>
<td>31:0</td>
<td>F_D[38]</td>
<td>%d38</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>F_S[26]</td>
<td>%f26</td>
<td>63:32</td>
<td>F_D[40]</td>
<td>%d40</td>
<td>127:64</td>
<td></td>
</tr>
<tr>
<td>F_S[27]</td>
<td>%f27</td>
<td>31:0</td>
<td>F_D[42]</td>
<td>%d42</td>
<td>63:0</td>
<td></td>
</tr>
<tr>
<td>F_S[28]</td>
<td>%f28</td>
<td>63:32</td>
<td>F_D[44]</td>
<td>%d44</td>
<td>127:64</td>
<td></td>
</tr>
<tr>
<td>F_S[29]</td>
<td>%f29</td>
<td>31:0</td>
<td>F_D[46]</td>
<td>%d46</td>
<td>63:0</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 5-2**  Floating-Point Registers, with Aliasing  
(2 of 3)
5.3.1 Floating-Point Register Number Encoding

Register numbers for single, double, and quad registers are encoded differently in the 5-bit register number field of a floating-point instruction. If the bits in a register number field are labelled $b_5 \ldots b_0$ (where $b_5$ is the most significant bit of the register number), the encoding of floating-point register numbers into 5-bit instruction fields is as given in TABLE 5-3.

TABLE 5-3 Floating-Point Register Number Encoding

<table>
<thead>
<tr>
<th>Register Operand Type</th>
<th>Full 6-bit Register Number</th>
<th>Encoding in a 5-bit Register Field in an Instruction</th>
</tr>
</thead>
</table>

**SPARC V8 Compatibility Note**

In the SPARC V8 architecture, bit 0 of double and quad register numbers encoded in instruction fields was required to be zero. Therefore, all SPARC V8 floating-point instructions can run unchanged on an Oracle SPARC Architecture virtual processor, using the encoding in TABLE 5-3.
5.3.2 Double and Quad Floating-Point Operands

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

**Programming Note**

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

1. Whenever possible, align floating-point data in memory on proper address boundaries. If access to a datum is required to be atomic, the datum must be properly aligned.

2. If a double- or quad-precision datum is not properly aligned in memory or is still aligned on a 4-byte boundary, and access to the datum in memory is not required to be atomic, then software should attempt to allocate a register for it in the lower half of the floating-point register file so that the datum can be loaded with multiple LDF[A] instructions.

3. If the only available registers for such a datum are located in the upper half of the floating-point register file and access to the datum in memory is not required to be atomic, the word-aligned datum can be loaded into them by one of two methods:
   - Load the datum into an upper register by using multiple LDF[A] instructions to first load it into a double- or quad-precision register in the lower half of the floating-point register file, then copy that register to the desired destination register in the upper half.

   Use an LDDF[A] or LDQF[A] instruction to perform the load directly into the upper floating-point register, understanding that use of these instructions on poorly aligned data can cause a trap (LDDF_mem_not_aligned or LDQF_mem_not_aligned) on some implementations, possibly slowing down program execution significantly.

**Programming Note**

If an Oracle SPARC Architecture 2011 implementation does not implement a particular quad floating-point arithmetic operation in hardware and an invalid quad register operand is specified, the illegal_instruction trap occurs because it has higher priority.

**Implementation Note**

Oracle SPARC Architecture 2011 implementations do not implement any quad floating-point arithmetic operations in hardware. Therefore, an attempt to execute any of them results in a trap on the illegal_instruction exception.
5.4 Floating-Point State Register (FSR)

The Floating-Point State register (FSR) fields, illustrated in FIGURE 5-4, contain FPU mode and status information. The lower 32 bits of the FSR are read and written by the (deprecated) STFSR and LDFSR instructions, respectively. The 64-bit FSR register is read by the STXFSR instruction and written by the LDXFSR instruction. The ver, ftt, qne, unimplemented (for example, ns), and reserved (“—”) fields of FSR are not modified by either LDFSR or LDXFSR. The LDXEFSR instruction can be used to write all implemented fields of FSR (notably ftt), but not the ver, qne, and reserved fields.

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

#### Programming Note

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

The subsections on pages 42 through 48 describe the remaining fields in the FSR.

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

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

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

In TABLE 5-4, f_{rs1} and f_{rs2} correspond to the single, double, or quad values in the floating-point registers specified by a floating-point compare instruction’s rs1 and rs2 fields. The question mark (?) indicates an unordered relation, which is true if either f_{rs1} or f_{rs2} is a signalling NaN or a quiet NaN. If FCMP or FCMPE generates an \textit{fp\_exception\_ieee\_754} exception, then fccn is unchanged.

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

<table>
<thead>
<tr>
<th>Indicated Relation (FCMP*, FCMPE*)</th>
<th>Content of fccn</th>
</tr>
</thead>
<tbody>
<tr>
<td>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] \neq F[rs2] (unordered)</td>
<td>3</td>
</tr>
</tbody>
</table>
5.4.2 Rounding Direction (rd)

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

<table>
<thead>
<tr>
<th>Indicated Relation (FLCMP*)</th>
<th>Content of fccn</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F[rs1] ≥ F[rs2], neither operand is NaN</td>
<td>F[rs1] &lt; F[rs2], neither operand is NaN</td>
<td>F[rs1] is NaN, F[rs2] is NaN</td>
<td>F[rs2] is NaN, regardless of F[rs1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the interval mode bit of the General Status register has a value of 1 (GSR.im = 1), then the value of FSR.rd is ignored and floating-point results are instead rounded according to GSR.irnd. See General Status Register (GSR) (ASR 19) on page 54 for further details.

5.4.3 Trap Enable Mask (tem)

Bits 27 through 23 are enable bits for each of the five IEEE-754 floating-point exceptions that can be indicated in the current_exception field (cexc). See FIGURE 5-6 on page 47. If a floating-point instruction generates one or more exceptions and the tem bit corresponding to any of the exceptions is 1, then this condition causes an fp_exception_ieee_754 trap. A tem bit value of 0 prevents the corresponding IEEE 754 exception type from generating a trap.

5.4.4 Nonstandard Floating-Point (ns)

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

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

For detailed requirements for the case when an Oracle SPARC Architecture processor elects to implement floating-point nonstandard mode, see Floating-Point Nonstandard Mode on page 367.

5.4.5 FPU Version (ver)

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

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

FSR.ver = 7 is reserved to indicate that no hardware floating-point controller is present.
5.4.6 Floating-Point Trap Type (ftt)

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. FSR.ftt can be directly written to a specific value by the LDXEFSR instruction.

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

FSR.ftt encodes the primary condition (“floating-point trap type”) that caused the generation of an fp_exception_other or fp_exception_ieee_754 exception. It is possible for more than one such condition to occur simultaneously; in such a case, only the highest-priority condition will be encoded in FSR.ftt. The conditions leading to fp_exception_other and fp_exception_ieee_754 exceptions, their relative priorities, and the corresponding FSR.ftt values are listed in TABLE 5-6. Note that the FSR.ftt values 4 and 5 were defined in the SPARC V9 architecture but are not currently in use, and that the value 7 is reserved for future architectural use.

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

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

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

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

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

The foregoing describes the result seen by a user trap handler if an IEEE exception is signalled, either immediately from an \( fp\_exception\_ieee\_754 \) exception or after recovery from an unfinished_FPop. In either case, \( cexc \) as seen by the trap handler reflects the exception causing the trap.

In the cases of an \( fp\_exception\_other \) exception with a floating-point trap type of unfinished_FPop that does not subsequently generate an IEEE trap, the recovery software should set \( cexc, aexc, \) and the destination register or \( fccn \), as appropriate.

### ftt = 1 (IEEE_754_exception)

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

### ftt = 2 (unfinished_FPop)

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

**Implementation Note**

Implementations are encouraged to support standard IEEE 754 floating-point arithmetic with reasonable performance (that is, without generating \( fp\_exception\_other \) with \( FSR.ftt=\text{unfinished\_FPop} \)) in all cases, even if some cases are slower than others.

**IMPL. DEP. #248-U3**

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

### ftt = 3 (Reserved)

**SPARC V9 Compatibility Note**

In SPARC V9, FSR.ftt = 3 was defined to be "unimplemented_FPop". All conditions which used to cause \( fp\_exception\_other \) with \( FSR.ftt=3 \) now cause an \textit{illegal\_instruction} exception, instead. FSR.ftt = 3 is now reserved and available for other future uses.

### ftt = 4 (Reserved)

**SPARC V9 Compatibility Note**

In the SPARC V9 architecture, FSR.ftt = 4 was defined to be "sequence_error", for use with certain error conditions associated with a floating-point queue (FQ). Since Oracle SPARC 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 Oracle SPARC 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.
46 Oracle SPARC Architecture 2011 • Draft D1.0.0, 12 Jan 2016

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

| Implementation Note | If an Oracle SPARC Architecture 2011 processor does not implement a particular quad FPop in hardware, that FPop generates an `illegal_instruction` exception instead of `fp_exception_other` with FSR.ftt = 6 (invalid_fp_register), regardless of the specified F registers. |

### 5.4.7 Accrued Exceptions (aexc)

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

After an FPop completes with ftt = 0, the tem and cexc fields are logically anded together. If the result is nonzero, aexc is left unchanged and an `fp_exception_ieee_754` trap is generated; otherwise, the new cexc field is ored into the aexc field and no trap is generated. Thus, while (and only while) traps are masked, exceptions are accumulated in the aexc field.

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

### 5.4.8 Current Exception (cexc)

FSR.cexc (FSR[4:0]) indicates whether one or more IEEE 754 floating-point exceptions were generated by the most recently executed FPop instruction. The absence of an exception causes the corresponding bit to be cleared (set to 0). See FIGURE 5-6 on page 47.

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

The cexc bits are set as described in **Floating-Point Exception Fields** on page 47, by the execution of an FPop that either does not cause a trap or causes an `fp_exception_ieee_754` exception with FSR.ftt = IEEE_754_exception. An IEEE 754 exception that traps shall cause exactly one bit in FSR.cexc to be set, corresponding to the detected IEEE Std 754-1985 exception.

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

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

- If an IEEE 754 underflow condition occurs:
  - if FSR.tem.ufm = 0 and FSR.tem.nxm = 0, the FSR.cexc.ufc and FSR.cexc.nxc bits are both set to 1, the other three bits of FSR.cexc are set to 0, and an `fp_exception_ieee_754` trap does not occur.
  - if FSR.tem.ufm = 0 and FSR.tem.nxm = 1, the FSR.cexc.nxc bit is set to 1, the other four bits of FSR.cexc are set to 0, and an `fp_exception_ieee_754` trap does occur.
if FSR.tem.ufm = 1, the FSR.cexc.ufc bit is set to 1, the other four bits of FSR.cexc are set to 0, and an \texttt{fp\_exception\_ieee\_754} trap does occur.

The above behavior is summarized in TABLE 5-7 (where “✔” indicates “exception was detected” and “x” indicates “don’t care”):

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

Notes: 1 When the underflow trap is disabled (FSR.tem.ufm = 0) underflow is always accompanied by inexact. 2 Overflow is always accompanied by inexact.

If the execution of an FPop causes a trap other than \texttt{fp\_exception\_ieee\_754}, FSR.cexc is left unchanged.

### 5.4.9 Floating-Point Exception Fields

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

<table>
<thead>
<tr>
<th>FSR.tem</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvm ofm uf m dzm nxm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27 26 25 24 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>FSR.aexc</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>nva ofa ufa dza nx a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 8 7 6 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>FSR.cexc</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvc ofc ufc dzc nxc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 3 2 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 5-8** Current Exception Bits (aexc) Fields of FSR
Invalid (**nv**, **nva**). An operand is improper for the operation to be performed. For example, 0.0 ÷ 0.0 and ∞ − ∞ are invalid; 1 = invalid operand(s), 0 = valid operand(s).

Overflow (**ofc**, **ofa**). The result, rounded as if the exponent range were unbounded, would be larger in magnitude than the destination format’s largest finite number; 1 = overflow, 0 = no overflow.

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

Underflow is never indicated when the correct unrounded result is 0. Otherwise, when the correct unrounded result is not 0:

- If FSR.tem.ufm = 0: Underflow occurs if a nonzero result is tiny and a loss of accuracy occurs.
- If FSR.tem.ufm = 1: Underflow occurs if a nonzero result is tiny.

The SPARC V9 architecture allows tininess to be detected either before or after rounding. However, in all cases and regardless of the setting of FSR.tem.ufm, an Oracle SPARC Architecture strand detects tininess before rounding (impl. dep. #55-V8-Cs10). See Trapped Underflow Definition (**ufm** = 1) on page 367 and Untrapped Underflow Definition (**ufm** = 0) on page 367 for additional details.

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

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

### 5.4.10 FSR Conformance

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

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

### 5.5 Ancillary State Registers

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

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

The SPARC V9 architecture left ASRs numbered 16–31 available for implementation-dependent uses. Oracle SPARC Architecture virtual processors implement the ASRs summarized in TABLE 5-8 and defined in the following subsections.
Each virtual processor contains its own set of ASRs; ASRs are not shared among virtual processors.

**TABLE 5-8**

<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&lt;sup&gt;D&lt;/sup&gt;</td>
<td>Y register (deprecated)</td>
<td>RDY&lt;sup&gt;D&lt;/sup&gt;</td>
<td>WRY&lt;sup&gt;D&lt;/sup&gt;</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CCR</td>
<td>Condition Codes register</td>
<td>RDCCCR</td>
<td>WRCCCR</td>
</tr>
<tr>
<td>3</td>
<td>ASI</td>
<td>ASI register</td>
<td>RDASI</td>
<td>WRASI</td>
</tr>
<tr>
<td>4</td>
<td>TICK&lt;sub&gt;Pdis,Hdis&lt;/sub&gt;</td>
<td>TICK register</td>
<td>RDTICK&lt;sub&gt;Pdis,Hdis&lt;/sub&gt;, RDPR&lt;sup&gt;P&lt;/sup&gt; (TICK)</td>
<td>WRPR&lt;sup&gt;P&lt;/sup&gt; (TICK)</td>
</tr>
<tr>
<td>5</td>
<td>PC</td>
<td>Program Counter (PC)</td>
<td>RDPC</td>
<td>(all instructions)</td>
</tr>
<tr>
<td>6</td>
<td>FPRS</td>
<td>Floating-Point Registers Status register</td>
<td>RDFPRS</td>
<td>WRFPRS</td>
</tr>
<tr>
<td>7–12</td>
<td>(7–0C&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>(0D&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>(0E&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>(0F&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved (used for MEMBAR; see text under MEMBAR [p. 257] or RDasr [p. 297] instructions)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>(10&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>(11&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16–18</td>
<td>(10&lt;sub&gt;16&lt;/sub&gt;–12&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>(13&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>GSR</td>
<td>General Status register (GSR)</td>
<td>RDGSR, FALIGNDATA, many VIS and floating-point instructions</td>
</tr>
<tr>
<td>20</td>
<td>(14&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>SOFTINT_SET&lt;sup&gt;P&lt;/sup&gt;</td>
<td>(pseudo-register, for &quot;Write 1s Set&quot; to SOFTINT register, ASR 22)</td>
<td>WRSOFTINT_SET&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>21</td>
<td>(15&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>SOFTINT_CLR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>(pseudo-register, for &quot;Write 1s Clear&quot; to SOFTINT register, ASR 22)</td>
<td>WRSOFTINT_CLR&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>22</td>
<td>(16&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>SOFTINT&lt;sup&gt;P&lt;/sup&gt;</td>
<td>per-virtual processor Soft Interrupt register</td>
<td>RD SOFTINT&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>23</td>
<td>(17&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>(18&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>STICK&lt;sub&gt;Pdis,Hdis&lt;/sub&gt;</td>
<td>System Tick register</td>
<td>RDSTICK&lt;sub&gt;Pdis,Hdis&lt;/sub&gt;</td>
</tr>
<tr>
<td>25</td>
<td>(19&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>STICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>System Tick Compare register</td>
<td>RDSTICK_CMPR&lt;sup&gt;P&lt;/sup&gt;</td>
</tr>
<tr>
<td>26</td>
<td>(1A&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>CFR</td>
<td>Compatibility Feature register</td>
<td>RDCF</td>
</tr>
<tr>
<td>27</td>
<td>(1B&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>PAUSE</td>
<td>Pause Count register</td>
<td>PAUSE (WRasr 27)</td>
</tr>
<tr>
<td>28</td>
<td>(1C&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>(1D&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>(1E&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>(1F&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>(1F&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.5.1 32-bit Multiply/Divide Register (Y) (ASR 0)

The Y register is deprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC V9 software. It is recommended that all instructions that reference the Y register (that is, SMUL, SMULcc, UML, UMUL, UMULcc, SDIV, SDIVcc, UDIV, UDIVcc, RDY, and WRKY) be avoided. For suitable substitute instructions, see the following pages: for the multiply instructions, see pages 318 and page 355; for division instructions, see pages 310 and 353; for the read instruction, see page 296; and for the write instruction, see page 358.

The low-order 32 bits of the Y register, illustrated in FIGURE 5-9, contain the more significant word of the 64-bit product of an integer multiplication, as a result of a 32-bit integer multiply (SMUL, SMULcc, UML, UMULcc) 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 WRKY instructions, respectively.

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

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

5.5.2.1 Condition Codes (CCR.xcc and CCR.icc)

All instructions that set integer condition codes set both the xcc and icc fields. The xcc condition codes indicate the result of an operation when viewed as a 64-bit operation. The icc condition codes indicate the result of an operation when viewed as a 32-bit operation. For example, if an operation results in the 64-bit value 0000 0000 FFFF FFFF₁₆, the 32-bit result is negative (icc.n is set to 1) but the 64-bit result is nonnegative (xcc.n is set to 0).

Each of the 4-bit condition-code fields is composed of four 1-bit subfields, as shown in FIGURE 5-11.

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

The z bits indicate whether the ALU result was zero for the last instruction that modified the integer condition codes; 1 = zero, 0 = nonzero.
The \( v \) bits signify whether the ALU result was within the range of (was representable in) 64-bit (\( xcc \)) or 32-bit (\( icc \)) 2’s-complement notation for the last instruction that modified the integer condition codes; \( 1 = \) overflow, \( 0 = \) no overflow. The \( v \) bits may be used to test for signed overflow.

The \( c \) bits indicate whether a 2’s-complement carry (or borrow) occurred during the last instruction that modified the integer condition codes. Carry is set on addition if there is a carry out of bit 63 (\( xcc \)) or bit 31 (\( icc \)). Carry is set on subtraction if there is a borrow into bit 63 (\( xcc \)) or bit 31 (\( icc \)); \( 1 = \) borrow, \( 0 = \) no borrow (see TABLE 5-9). The \( c \) bits may be used to test for unsigned overflow.

**TABLE 5-9** Setting of Carry (Borrow) bits for Subtraction That Sets CCs

<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] \geq R[rs2][31:0] )</td>
<td>( CCR.icc.c \leftarrow 0 )</td>
</tr>
<tr>
<td>( R[rs1][31:0] &lt; R[rs2][31:0] )</td>
<td>( CCR.icc.c \leftarrow 1 )</td>
</tr>
<tr>
<td>( R[rs1][63:0] \geq R[rs2][63:0] )</td>
<td>( CCR.xcc.c \leftarrow 0 )</td>
</tr>
<tr>
<td>( R[rs1][63:0] &lt; R[rs2][63:0] )</td>
<td>( CCR.xcc.c \leftarrow 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 also be modified by a DONE or RETRY instruction, which replaces these bits with the contents of TSTATE.ccr. The behavior of the following instructions are conditioned by the contents of CCR.icc or CCR.xcc:

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

**Extended (64-bit) integer condition codes (\( xcc \)).** Bits 7 through 4 are the IU condition codes, which indicate the results of an integer operation, with both of the operands and the result considered to be 64 bits wide.

**32-bit Integer condition codes (\( icc \)).** Bits 3 through 0 are the IU condition codes, which indicate the results of an integer operation, with both of the operands and the result considered to be 32 bits wide.

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

The Address Space Identifier register (FIGURE 5-12) specifies the address space identifier to be used for load and store alternate instructions that use the “\( rs1 + simm13 \)” addressing form.

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

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

**FIGURE 5-12** Address Space Identifier Register
5.5.4 Tick (TICK) Register (ASR 4)  

FIGURE 5-13 illustrates the TICK register.

![TICK Register Diagram]

The counter field of the TICK register is a 63-bit counter that counts virtual processor clock cycles. Bit 63 of the TICK register reads as 0.

Privileged software can read the TICK register with either the RDPR or RDTICK instruction, but only when privileged access to TICK is enabled by hyperprivileged software. An attempt by privileged software to read the TICK register when privileged access is disabled causes a privileged_action exception.

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

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

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

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

The foregoing description regarding access to the TICK register is summarized in TABLE 5-10 and TABLE 5-11.

TABLE 5-10  Access to TICK register through ASR 4 (RDTICK instruction)

<table>
<thead>
<tr>
<th>Privilege Mode</th>
<th>Read Access Using RDTICK Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonprivileged</td>
<td>if allowed by hyperprivileged software, TICK may be read with RDTICK otherwise, privileged_action exception</td>
</tr>
<tr>
<td>Privileged</td>
<td>if allowed by hyperprivileged software, TICK may be read with RDTICK otherwise privileged_action exception</td>
</tr>
</tbody>
</table>

TABLE 5-11  Access to TICK register through 105-V9h PR 4 (RDPR and WRPR instructions)

<table>
<thead>
<tr>
<th>Privilege Mode</th>
<th>Read Access Using RDPR TICK instruction</th>
<th>Write Access Using WRPR TICK instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonprivileged</td>
<td>privileged_opcode exception</td>
<td>privileged_opcode exception</td>
</tr>
<tr>
<td>Privileged</td>
<td>if allowed by hyperprivileged software, TICK may be read with RDPR TICK otherwise privileged_action exception</td>
<td>illegal_instruction exception</td>
</tr>
</tbody>
</table>

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.
(a) If an accurate count cannot always be returned when \textsc{TICK} is read, any inaccuracy should be small, bounded, and documented.

(b) An implementation may implement fewer than 63 bits in \textsc{TICK}; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any upper bits not implemented must read as zero.

5.5.5 Program Counters (PC, NPC) (ASR 5)\textsuperscript{A1}

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

The \textsc{PC} can be read directly with the RDPC instruction. \textsc{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 \textit{illegal_instruction} exception.

The Next Program Counter, \textsc{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 \textsc{NPC} always contain zeroes.

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

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

See Chapter 6, \textit{Instruction Set Overview}, for details on how \textsc{PC} and \textsc{NPC} are used.

5.5.6 Floating-Point Registers State (FPRS) Register (ASR 6)\textsuperscript{A1}

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

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{fprs_register}
\caption{Floating-Point Registers State Register}
\end{figure}

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

\textbf{Enable FPU (fef).} Bit 2, \textit{fef}, determines whether the FPU is enabled. If it is disabled, executing a floating-point instruction causes an \textit{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 \textit{fp_disabled} exception; that is, both FPRS.fef and PSTATE.pef must be set to 1 to enable floating-point operations.

\begin{table}[h]
\centering
\begin{tabular}{l|c|c}
\textbf{Programming Note} & FPRS.fef & FPRS.du
\end{tabular}
\caption{Floating-Point Registers State Register}
\end{table}

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).

\textbf{Dirty Upper Registers (du).} Bit 1 is the “dirty” bit for the upper half of the floating-point registers; that is, F[32]–F[62]. It is set to 1 whenever any of the upper floating-point registers is modified. The du bit is cleared only by software.
An Oracle SPARC Architecture 2011 virtual processor may set \texttt{FPRS.du} pessimistically; that is, it may be set whenever an FPop executes, even though an exception may occur that prevents the instruction from completing so no destination \texttt{F} register was actually modified (impl. dep. \#403-S10). Note that if the FPop triggers \texttt{fp\_disabled}, \texttt{FPRS.du} is \textit{not} modified.

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

An Oracle SPARC Architecture 2011 virtual processor may set \texttt{FPRS.dl} pessimistically; that is, it may be set whenever an FPop executes, even though an exception may occur that prevents the instruction from completing so no destination \texttt{F} register was actually modified (impl. dep. \#403-S10). Note that if the FPop triggers \texttt{fp\_disabled}, \texttt{FPRS.dl} is \textit{not} modified.

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

The General Status Register\(^1\) (GSR) is a nonprivileged read/write register that is implicitly referenced by many VIS instructions. The GSR can be read by the RDGSR instruction (see Read Ancillary State Register on page 295) and written by the WRGSR instruction (see Write Ancillary State Register on page 357).

If the FPU is disabled (\texttt{PSTATE.pef} = 0 or \texttt{FPRS.fef} = 0), an attempt to access this register using an otherwise-valid RDGSR or WRGSR instruction causes an \texttt{fp\_disabled} trap.

The GSR is illustrated in FIGURE 5-15 and described in TABLE 5-12.

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

#### TABLE 5-12 GSR Bit Description

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>\texttt{mask} 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>Reserved.</td>
</tr>
<tr>
<td>27</td>
<td>\texttt{im} Interval Mode: If GSR.im = 0, rounding is performed according to FSR.rd; if GSR.im = 1, rounding is performed according to GSR.imd.</td>
</tr>
<tr>
<td>26:25</td>
<td>\texttt{irnd} IEEE Std 754-1985 rounding direction to use in Interval Mode (GSR.im = 1), as follows:</td>
</tr>
<tr>
<td></td>
<td>\texttt{irnd} Round toward ...</td>
</tr>
<tr>
<td></td>
<td>0 Nearest (even, if tie)</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
</tr>
<tr>
<td></td>
<td>2 + ∞</td>
</tr>
<tr>
<td></td>
<td>3 − ∞</td>
</tr>
<tr>
<td>24:8</td>
<td>Reserved.</td>
</tr>
<tr>
<td>7:3</td>
<td>\texttt{scale} 5-bit shift count in the range 0–31, used by the FPACK instructions for formatting.</td>
</tr>
<tr>
<td>2:0</td>
<td>\texttt{align} Least three significant bits of the address computed by the last-executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction.</td>
</tr>
</tbody>
</table>

\(^1\) This register was (inaccurately) referred to as the "Graphics Status Register" in early UltraSPARC implementations.
5.5.8 SOFTINT\textsuperscript{P} Register (ASRs 20 \(A2\), 21 \(A2\), 22 \(A1\))

Software uses the privileged, read/write SOFTINT register (ASR 22) to schedule interrupts (via \textit{interrupt\_level\_n} exceptions).

SOFTINT \(A1\) can be read with a RDSOFTINT instruction (see \textit{Read Ancillary State Register} on page 295) and written with a WRSOFTINT, WRSOFTINT\_SET, or WRSOFTINT\_CLR instruction (see \textit{Write Ancillary State Register} on page 357). An attempt to access to this register in nonprivileged mode causes a \texttt{privileged\_opcode} exception.

<table>
<thead>
<tr>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>To atomically modify the set of pending software interrupts, use of the SOFTINT_SET and SOFTINT_CLR ASRs is recommended.</td>
</tr>
</tbody>
</table>

The SOFTINT register is illustrated in FIGURE 5-16 and described in TABLE 5-13.

![FIGURE 5-16 SOFTINT Register (ASR 22)](image)

**TABLE 5-13** SOFTINT Bit Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>sm</td>
<td>When the STICK_CMPR (ASR 25) register’s int_dis (interrupt disable) field is 0 (that is, System Tick Compare is enabled) and its stick_cmp field matches the value in the STICK register, then SOFTINT.sm (“STICK match”) is set to 1 and a level 14 interrupt (interrupt_level_14) is generated. See System Tick Compare (STICK_CMPR) Register (ASR 25) on page 57 for details. SOFTINT.sm can also be directly written to 1 by software.</td>
</tr>
<tr>
<td>15:1</td>
<td>int_level</td>
<td>When SOFTINT.int_level([n-1]) (SOFTINT([n])) is set to 1, an interrupt_level_n exception is generated.</td>
</tr>
</tbody>
</table>

**Notes:**
- A level-14 interrupt (interrupt\_level\_14) can be triggered by SOFTINT.sm or a write to SOFTINT.int\_level\[13\] (SOFTINT\[14\]).
- A level-15 interrupt (interrupt\_level\_15) can be triggered by a write to SOFTINT.int\_level\[14\] (SOFTINT\[15\]), or possibly by other implementation-dependent mechanisms.
- An interrupt\_level\_n exception will only cause a trap if (PIL < n) and (PSTATE.ie = 1).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Setting either SOFTINT.sm or SOFTINT.int\_level\[13\] (SOFTINT\[14\]) to 1 causes a level-14 interrupt (interrupt\_level\_14). However, those two bits are independent; setting one of them does not affect the other.

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

5.5.8.1 SOFTINT\_SET\textsuperscript{P} Pseudo-Register (ASR 20) \(A2\)

A Write State register instruction to ASR 20 (WRSOFTINT\_SET) atomically sets selected bits in the privileged SOFTINT Register (ASR 22) (see page 55). That is, bits 16:0 of the write data are \texttt{ored} into SOFTINT; any ‘1’ bit in the write data causes the corresponding bit of SOFTINT to be set to 1. Bits 63:17 of the write data are ignored.
Access to ASR 20 is privileged and write-only. There is no instruction to read this pseudo-register. An attempt to write to ASR 20 in non-privileged mode, using the WRasr instruction, causes a privilegedOpcode exception.

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

FIGURE 5-17 illustrates the SOFTINT_SET pseudo-register.

![FIGURE 5-17 SOFTINT_SET Pseudo-Register (ASR 20)](image)

### 5.5.8.2 SOFTINT_CLR P Pseudo-Register (ASR 21) (A2)

A Write State register instruction to ASR 21 (WRSOFTINT_CLR) atomically clears selected bits in the privileged SOFTINT register (ASR 22) (see page 55). That is, bits 16:0 of the write data are inverted and anded into SOFTINT; any ‘1’ bit in the write data causes the corresponding bit of SOFTINT to be set to 0. Bits 63:17 of the write data are ignored.

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

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

FIGURE 5-18 illustrates the SOFTINT_CLR pseudo-register.

![FIGURE 5-18 SOFTINT_CLR Pseudo-Register (ASR 21)](image)

### 5.5.9 System Tick (STICK) Register (ASR 24) (A1)

The System Tick (STICK) register provides a counter that consistently measures time across all virtual processors (strands) of a system. The 63-bit counter field of the STICK register automatically increments at a fixed frequency of 1.0 GHz, therefore counter bit \( n \) will be observed to increment at a frequency of \( 1.0 \text{ GHz} \div 2^n \).

The STICK register is illustrated in FIGURE 5-19 and described below.

**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.

Bit 63 of the STICK register reads as 0.

The counter field spans bits 62:\( m \) of the STICK register. \( m \) is implementation-dependent (impl.dep. #442-S10(c)), but \( m \) must be less than or equal to 4 (STICK granularity of 16ns or better); \( m \) can be 0.
The counter must increment 1 billion ± 25,000 times every second (an error rate of no more than 25 parts per million). This means that the counter must not gain or lose more than 2.16 seconds per day.

**IMPL. DEP. #442-S10:**

(a) If an accurate count cannot always be returned when STICK is read, any inaccuracy should be small, bounded, and documented.

(b) (no longer applies)

(c) The bit number $m$ of the least significant implemented bit of the counter field of the STICK register is implementation-dependent, but must be in range 4 to 0, inclusively (that is, $4 \geq m \geq 0$). Any low-order bits not implemented must read as zero.

Writes to unimplemented bits of STICK are ignored. Any implementation that does not implement all bits of STICK must ensure that comparisons to STICK_CMPR correctly account for the absence of the unimplemented bits.

At least one STICK register must be implemented per system. However, multiple STICK registers per system may be implemented (for example, STICK may be implemented per-core or per-strand). No more than one STICK may be implemented per strand. An implementation must document how many STICKs are implemented per system and their relationships to strands and other structures.

Privileged software can read the STICK register with the RDSTICK instruction, but only when privileged access to STICK is enabled by hyperprivileged software. An attempt by privileged software to read the STICK register when privileged access is disabled causes a privileged_action exception.

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 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.

The difference between two values read from the STICK register at different times reflects the amount of time that has passed between the reads; $(\text{value}_1 - \text{value}_0)$ yields the number of nanoseconds that elapsed between the two reads.

**Note**

If STICK begins counting at 0, it will overflow in approximately 34 years.

### 5.5.10 System Tick Compare (STICK_CMPR<sup>P</sup>) Register (ASR 25)

The privileged STICK_CMPR register allows system software to cause an interrupt_level_14 trap when the STICK register reaches the value specified in STICK_CMPR. Nonprivileged accesses to this register cause a privileged_opcode exception (see Exception and Interrupt Descriptions on page 434).

The System Tick Compare Register is illustrated in FIGURE 5-20 and described in TABLE 5-14.
The value of \( n \) (as shown in FIGURE 5-20) is implementation-dependent, but must be between 0 and 8, inclusively \((0 \leq n \leq 8)\).

The value of \( n \) (as shown in FIGURE 5-20) is implementation-dependent, but must be between 0 and 9, inclusively \((0 \leq n \leq 9)\).

### TABLE 5-14 STICK_CMPR Register Description

<table>
<thead>
<tr>
<th>Bits</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>62:61</td>
<td>stick_cmpr</td>
<td>System Tick Compare Field.</td>
</tr>
<tr>
<td>(n−1):0</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

A *stick_match* exception occurs when all three of the following conditions are met:

- **STICK_CMPR.int_dis = 0**
- A transition occurs from
  
    \[
    \text{(STICK.counter} \ll m) < \text{(STICK_CMPR.stick_cmpr} \ll n) 
    \]
  
    in one cycle to
  
    \[
    \text{(STICK.counter} \ll m) \geq \text{(STICK_CMPR.stick_cmpr} \ll n) 
    \]
  
    in the following cycle, where:
  
    - \( m \) is the bit number of the least-significant implemented bit of \( \text{STICK.counter} \) (see page 56)
    - \( n \) is the bit number of the least-significant implemented bit of \( \text{STICK_CMPR.stick_cmpr} \)
    - \( \ll \) is the arithmetic left-shift operator
    - \( < \) and \( \geq \) are unsigned comparisons

- The above state transition occurred due to incrementing \( \text{STICK} \) and not due to an explicit write to \( \text{STICK} \) or \( \text{STICK_CMPR} \)

When a *stick_match* exception occurs, SOFTINT[16] (SOFTINT.sm) is set to 1. This has the effect of posting an interrupt_level_14 trap request to the virtual processor, which causes an interrupt_level_14 trap when \((\text{PIL} < 14) \text{ and } (\text{PSTATE.ie} = 1)\). The interrupt_level_14 trap handler must check SOFTINT[14] and SOFTINT[16] (SOFTINT.sm) to determine the cause of the interrupt_level_14 trap.

An implementation may compare \((\text{STICK_CMPR.stick_cmpr} \ll n)\) to \( \text{STICK} \) periodically, instead of comparing every cycle.

The implementation must ensure that if \( \text{STICK_CMPR.int_dis = 0} \) and a 63-bit value written to \( \text{STICK_CMPR.stick_cmpr} \) is greater than \( \text{STICK} \), that the implementation will detect a *stick_match* exception at a future time.

**Implementation Note**

If an implementation does not compare against \( \text{STICK_CMPR.stick_cmpr} \) every cycle, it is recommended that the implementation not implement the less-significant bits of \( \text{STICK_CMPR.stick_cmpr} \) that correspond to the time interval between comparisons of \( \text{STICK_CMPR.stick_cmpr} \).

In addition, if a write to \( \text{STICK_CMPR.stick_cmpr} \) has nonzero unimplemented lower bits and if the 63-bit value written to \( \text{STICK_CMPR.stick_cmpr} \) is greater than \( \text{STICK} \), then the implementation must ensure that the *stick_match* exception is not lost. This may be accomplished by incrementing the value of the implemented bits by an amount sufficient to ensure that a *stick_match* exception will occur at a future time, or by posting the *stick_match* exception to take effect at some later time, or by other means.

Any alterations performed by virtual processor hardware of the values written to \( \text{STICK_CMPR.stick_cmpr} \) must be specified in that processor’s Implementation Supplement to this document.
5.5.11 Compatibility Feature Register (CFR) (ASR 26)

Each virtual processor has a Compatibility Feature Register (CFR). The CFR is read-only.

The format of the CFR is shown in TABLE 5-15.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:22</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>21</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>20:15</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>14:12</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>crc32c</td>
<td>If set, the processor supports the CRC32C opcode. If not set, an attempt to execute a CRC32C instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>10</td>
<td>montsqr</td>
<td>If set, the processor supports the MONTSQR opcode. If not set, an attempt to execute a MONTSQR instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>9</td>
<td>montmul</td>
<td>If set, the processor supports the MONTMUL opcode. If not set, an attempt to execute a MONTMUL instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>8</td>
<td>mpmul</td>
<td>If set, the processor supports the MPMUL opcode. If not set, an attempt to execute an MPMUL instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>7</td>
<td>sha512</td>
<td>If set, the processor supports the SHA512 opcode. If not set, an attempt to execute a SHA512 instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>6</td>
<td>sha256</td>
<td>If set, the processor supports the SHA256 opcode. If not set, an attempt to execute a SHA256 instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>5</td>
<td>sha1</td>
<td>If set, the processor supports the SHA1 opcode. If not set, an attempt to execute a SHA1 instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>4</td>
<td>md5</td>
<td>If set, the processor supports the MD5 opcode. If not set, an attempt to execute an MD5 instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>3</td>
<td>camellia</td>
<td>If set, the processor supports Camellia opcodes (CAMELLIA_E, CAMELLIA_FL, and CAMELLIA_FLI). If not set, an attempt to execute a Camellia instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>2</td>
<td>des</td>
<td>If set, the processor supports DES opcodes (DES_ROUND, DES_IP, DES_IP, and DES_KEXPAND). If not set, an attempt to execute a DES instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>1</td>
<td>aes</td>
<td>If set, the processor supports AES opcodes (AES_EROUND01, AES_EROUND23, AES_DROUND01, AES_DROUND23, AES_EROUND_01_LAST, AES_EROUND_23_LAST, AES_DROUND_01_LAST, AES_DROUND_23_LAST, AES_KEXPAND0, AES_KEXPAND1, and AES_KEXPAND2). If not set, an attempt to execute an AES instruction results in a compatibility_feature trap.</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The CFR enumerates the capabilities that the virtual processor supports. Software can use the CFR to determine whether particular features (such as instruction opcodes or registers) are available for use on the current virtual processor. If a virtual processor executes an opcode associated with a bit in CFR but that bit is 0, that feature is not present and a compatibility_feature trap occurs.

**Programming Note**: For correctness and optimal performance, prior to using any feature associated with a bit in CFR, an application or library must first check the CFR to ensure that the desired feature is actually supported by the underlying hardware.
5.5.12 Pause Count (PAUSE) Register (ASR 27)

The nonprivileged PAUSE register provides a means for software to voluntarily and temporarily pause execution, freeing up processor resources for use by other threads of execution. When PAUSE = 0, it has no effect on execution. When PAUSE ≠ 0, execution is paused (temporarily suspended) until PAUSE = 0 again.

Software initiates a pause for \( n \) virtual processor cycles by writing the value \( n \) to the PAUSE register, using the WRPAUSE instruction. PAUSE is write-only and cannot be read by software; in particular, there is no RDPAUSE instruction.

When PAUSE ≠ 0, it is automatically decremented once every virtual processor cycle. When PAUSE = 0, no decrementation occurs.

![Figure 5-21: PAUSE Register](image)

**Implementation Note**

An implementation should implement enough bits in the PAUSE register to satisfy the equation:

\[
\frac{\left(2^B - (S \times 4)\right)}{2^B} \geq 0.99
\]

where:

- \( B \) is the number of bits that must be implemented in the PAUSE register
- \( S \) is the number of strands that share an instruction pipeline, and
- 4 represents the number of slots in the pipeline needed to execute the four instructions in a minimal busy-wait loop (load, compare, branch, PAUSE).

Solving this equation for \( B \) yields:

\[
B \geq \text{ceiling}(\log_2(100 \times S)) + 2
\]

**Implementation Note**

While PAUSE ≠ 0, the strand should suspend forward progress and should release all shared hardware resources to other strands.

When the value of PAUSE reaches 0, a paused strand resumes forward progress and begins using shared processor resources again.

The PAUSE register will be zero after the number of cycles specified in the PAUSE instruction have elapsed (that is, PAUSE has auto-decremented down to 0) or when any of the following events occur, all of which cause PAUSE to be zeroed:

- an unmasked (PSTATE.\( \text{ie} = 1 \)) disrupting trap request
- a deferred trap request

A masked trap request that occurs while a strand is paused has no effect on the suspension of the strand or on the contents of the PAUSE register. In particular, a masked trap request does not zero PAUSE.

---

\( \text{B} \)
\( \text{S} \)
\( \text{4} \)
\( \text{ceiling} \)
\( \text{PSTATE} \)
\( \text{ie} \)
\( \text{PAUSE} \)
5.6 Register-Window PR State Registers

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

Privileged registers CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN contain values in the range 0..N_REG_WINDOWS − 1. An attempt to write a value greater than N_REG_WINDOWS − 2 to CANSAVE, CANRESTORE, or OTHERWIN violates the register window state definition in Register Window State Definition on page 64. All five of these registers should have the same width.

$$\left\lceil \log_2(N_{\text{REG WINDOWS}}) \right\rceil$$

Although the architectural width of each of these five registers is 5 bits, their actual implemented width is implementation dependent and shall be between $$\left\lceil \log_2(N_{\text{REG WINDOWS}}) \right\rceil$$ 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.

For details of how the window-management registers are used, see Register Window Management Instructions on page 88.

5.6.1 Current Window Pointer (CWP) Register (PR 9)

The privileged CWP register, shown in FIGURE 5-22, is a counter that identifies the current window into the array of integer registers. See Register Window Management Instructions on page 88 and Chapter 12, Traps, for information regarding how hardware manipulates the CWP register.
5.6.2 Savable Windows (CANSAVE\textsuperscript{P}) Register (PR 10)\textsuperscript{A1}

The privileged CANSAVE register, shown in FIGURE 5-23, contains the number of register windows following CWP that are not in use and are, hence, available to be allocated by a SAVE instruction without generating a window spill exception.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig5_23.png}
\caption{CANSAVE Register, Figure 5-24, page 88}
\end{figure}

Programming Note
According to the Register Window State Definition on page 64, the maximum valid value of CANSAVE is $N_{\text{REG \_ WINDOWS}} - 2$. However, implementations are not required to prevent out-of-range values (notably, $N_{\text{REG \_ WINDOWS}} - 1$) from being written to CANSAVE, so software should take care to never write such values to it.

5.6.3 Restorable Windows (CANRESTORE\textsuperscript{P}) Register (PR 11)\textsuperscript{A1}

The privileged CANRESTORE register, shown in FIGURE 5-24, contains the number of register windows preceding CWP that are in use by the current program and can be restored (by the RESTORE instruction) without generating a window fill exception.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig5_24.png}
\caption{CANRESTORE Register}
\end{figure}

Programming Note
According to the Register Window State Definition on page 64, the maximum valid value of CANRESTORE is $N_{\text{REG \_ WINDOWS}} - 2$. However, implementations are not required to prevent out-of-range values (notably, $N_{\text{REG \_ WINDOWS}} - 1$) from being written to CANRESTORE, so software should take care to never write such values to it.

5.6.4 Clean Windows (CLEANWIN\textsuperscript{P}) Register (PR 12)\textsuperscript{A1}

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

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig5_25.png}
\caption{CLEANWIN Register}
\end{figure}
The CLEANWIN register counts the number of register windows that are “clean” with respect to the current program; that is, register windows that contain only zeroes, valid addresses, or valid data from that program. Registers in these windows need not be cleaned before they can be used. The count includes the register windows that can be restored (the value in the CANRESTORE register) and the register windows following CWP that can be used without cleaning. When a clean window is requested (by a SAVE instruction) and none is available, a clean_window exception occurs to cause the next window to be cleaned.

5.6.5 Other Windows (OTHERWIN\textsuperscript{P}) Register (PR 13) \textsuperscript{A1}

The privileged OTHERWIN register, shown in FIGURE 5-26, contains the count of register windows that will be spilled/filled by a separate set of trap vectors based on the contents of WSTATE\_other. If OTHERWIN is zero, register windows are spilled/filled by use of trap vectors based on the contents of WSTATE\_normal.

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

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

The privileged WSTATE register, shown in FIGURE 5-27, specifies bits that are inserted into TT\_TL\_[4:2] on traps caused by window spill and fill exceptions. These bits are used to select one of eight different window spill and fill handlers. If OTHERWIN = 0 at the time a trap is taken because of a window spill or window fill exception, then the WSTATE\_normal bits are inserted into TT\_TL. Otherwise, the WSTATE\_other bits are inserted into TT\_TL. See Register Window State Definition, below, for details of the semantics of OTHERWIN.
5.6.7 Register Window Management

The state of the register windows is determined by the contents of the set of privileged registers described in Register-Window PR State Registers on page 61. Those registers are affected by the instructions described in Register Window Management Instructions on page 88. Privileged software can read/write these state registers directly by using RDPR/WRPR instructions.

5.6.7.1 Register Window State Definition

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

$$\text{CANSAVE} + \text{CANRESTORE} + \text{OTHERWIN} = N_{\text{REG WINDOWS}} - 2$$

FIGURE 5-3 on page 37 shows how the register windows are partitioned to obtain the above equation. The partitions are as follows:

- The current window plus the window that must not be used because it overlaps two other valid windows. In FIGURE 5-3, these are windows 0 and 5, respectively. They are always present and account for the “2” subtracted from $N_{\text{REG WINDOWS}}$ 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 5-3) are counted in CANSAVE.
- Windows that have valid contents for the current address space and that can be used (through the RESTORE instruction) without causing a fill trap. These windows (window 7 in FIGURE 5-3) are counted in CANRESTORE.
- Windows that have valid contents for an address space other than the current address space. An attempt to use these windows through a SAVE (RESTORE) instruction results in a spill (fill) trap to a separate set of trap vectors, as discussed in the following subsection. These windows (window 6 in FIGURE 5-3) are counted in OTHERWIN.

In addition,

$$\text{CLEANWIN} \geq \text{CANRESTORE}$$

since CLEANWIN is the sum of CANRESTORE and the number of clean windows following CWP.

For the window-management features of the architecture described in this section to be used, the state of the register windows must be kept consistent at all times, except within the trap handlers for window spilling, filling, and cleaning. While window traps are being handled, the state may be inconsistent. Window spill/fill trap handlers should be written so that a nested trap can be taken without destroying state.

**Programming Note** System software is responsible for keeping the state of the register windows consistent at all times. Failure to do so will cause undefined behavior. For example, CANSAVE, CANRESTORE, and OTHERWIN must never be greater than or equal to $N_{\text{REG WINDOWS}} - 1$.

5.6.7.2 Register Window Traps

Window traps are used to manage overflow and underflow conditions in the register windows, support clean windows, and implement the FLUSHW instruction.

See Register Window Traps on page 439 for a detailed description of how fill, spill, and clean_window traps support register windowing.
5.7 Non-Register-Window PR State Registers

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

5.7.1 Trap Program Counter (TPC\(^P\)) Register (PR 0) \(^{A1}\)

The privileged Trap Program Counter register (TPC; FIGURE 5-28) contains the program counter (PC) from the previous trap level. There are \(\text{MAXPTL}\) instances of the TPC, but only one is accessible at any time. The current value in the TL register determines which instance of the TPC[TL] register is accessible. An attempt to read or write the TPC register when TL = 0 causes an illegal_instruction exception.

![FIGURE 5-28 Trap Program Counter Register Stack]

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

**TABLE 5-16** 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) ← value</td>
</tr>
</tbody>
</table>

5.7.2 Trap Next PC (TNPC\(^P\)) Register (PR 1) \(^{A1}\)

The privileged Trap Next Program Counter register (TNPC; FIGURE 5-28) is the next program counter (NPC) from the previous trap level. There are \(\text{MAXPTL}\) instances of the TNPC, but only one is accessible at any time. The current value in the TL register determines which instance of the TNPC register is accessible. An attempt to read or write the TNPC register when TL = 0 causes an illegal_instruction exception.

During normal operation, the value of TNPC\(_n\), where \(n\) is greater than the current trap level (\(n > \text{TL}\)), is undefined.
TABLE 5-17 lists the events that cause TNPC to be read or written.

TABLE 5-17   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] ← value</td>
</tr>
</tbody>
</table>

5.7.3 Trap State (TSTATE^P) Register (PR 2)  

The privileged Trap State register (TSTATE; FIGURE 5-30) contains the state from the previous trap level, comprising the contents of the GL, CCR, ASI, PSTATE, and CWP registers from the previous trap level. There are 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 TL = 0 causes an illegal_instruction exception.

V9 Compatibility Note

Because there are more bits in the Oracle SPARC Architecture’s PSTATE register than in a SPARC V9 PSTATE register, a 13-bit PSTATE value is stored in TSTATE instead of the 10-bit value specified in the SPARC V9 architecture.
TABLE 5-18 lists the events that cause TSTATE to be read or written.

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

<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>

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

The privileged Trap Type register (TT; see FIGURE 5-31) contains the trap type of the trap that caused entry to the current trap level. There are MAXPTL instances of the TT register, but only one is accessible at a time. The current value in the TL register determines which instance of the TT register is accessible. An attempt to read or write the TT register when TL = 0 causes an *illegal_instruction* exception.

\[\begin{array}{ll}
\text{TT}_1^P & \text{Trap type from trap while TL} = 0 \\
\text{TT}_2^P & \text{Trap type from trap while TL} = 1 \\
\vdots
\end{array}\]

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

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

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

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

#### 5.7.5 Tick (TICK) Register (PR 4)

PR 4 and ASR 4 refer to the same physical TICK register. See *Tick (TICK) Register (ASR 4)* on page 52 for a description of that register and how it can be accessed by the RDasr, RDPR, and WRPR instructions.

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

The privileged Trap Base Address register (TBA), shown in FIGURE 5-32, provides the upper 49 bits (bits 63:15) of the virtual address used to select the trap vector for a trap that is to be delivered to privileged mode. The lower 15 bits of the TBA always read as zero, and writes to them are ignored.
Details on how the full address for a trap vector is generated, using TBA and other state, are provided in Trap-Table Entry Address to Privileged Mode on page 424.

5.7.7 Processor State (PSTATE) Register (PR 6)

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

The following subsections describe the fields of the PSTATE register.

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

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

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

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

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

For the purposes of the control_transfer_instruction exception, a discontinuity in instruction-fetch addresses caused by a WRPR to PSTATE that changes the value of PSTATE.am (and thus, potentially the more-significant 32 bits of the address of the next instruction; see page 71) is not considered a control transfer. Only explicit CTIs can generate a control_transfer_instruction exception.
Current Little Endian (cle). This bit affects the endianness of data accesses performed using an implicit ASI. When PSTATE.cle = 1, all data accesses using an implicit ASI are performed in little-endian byte order. When PSTATE.cle = 0, all data accesses using an implicit ASI are performed in big-endian byte order. Specific ASIs used are shown in TABLE 6-3 on page 81. Note that the endianness of a data access may be further affected by TTE.ie used by the MMU.

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

Trap Little Endian (tle). When a trap is taken, the current PSTATE register is pushed onto the trap stack. During a virtual processor trap to privileged mode, the PSTATE.tle bit is copied into PSTATE.cle in the new PSTATE register. This behavior allows system software to have a different implicit byte ordering than the current process. Thus, if PSTATE.tle is set to 1, data accesses using an implicit ASI in the trap handler are little-endian.

The original state of PSTATE.cle is restored when the original PSTATE register is restored from the trap stack.

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

<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 01, 10, or 11 to PSTATE.mm because they are implementation-dependent or reserved for future extensions to the architecture, and in any case not currently portable across implementations.

- Total Store Order (TSO) — Loads are ordered with respect to earlier loads. Stores are ordered with respect to earlier loads and stores. Thus, loads can bypass earlier stores but cannot bypass earlier loads; stores cannot bypass earlier loads or stores.

IMPL. DEP. #113-V9-Ms10: Whether memory models represented by PSTATE.mm = 10 or 11 are supported in an Oracle SPARC Architecture processor is implementation dependent. If the 10 model is supported, then when PSTATE.mm = 10 the implementation must correctly execute software that adheres to the RMO model described in The SPARC Architecture Manual-Version 9. If the 11 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 Oracle SPARC 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 Oracle SPARC Architecture 2011 implementations. All software written to run correctly under RMO will run correctly under TSO on an Oracle SPARC Architecture 2011 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 \textit{fp_disabled} trap.

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

Address Mask (am). The PSTATE.am bit is provided to allow 32-bit SPARC software to run correctly on a 64-bit SPARC processor. When PSTATE.am = 1, bits 63:32 of virtual addresses are masked out (treated as 0).

PSTATE.am = PSTATE.am = PSTATE.am only affects virtual addresses (including those referenced using ASI_AS_IF_USER$^*$ ASIs); it does not affect real addresses.

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

When an MMU is disabled, PSTATE.am has no effect on (does not cause masking of) addresses.

\begin{tabular}{|p{1.5in}|p{4.5in}|}
\hline
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.
\end{tabular}

Misuse of the PSTATE.am bit can result in undesirable behavior. In particular, PSTATE.am should \textit{not} be set to 1 in privileged mode.

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

Instances in which the more-significant 32 bits of a virtual address are \textit{masked} when PSTATE.am = 1 include:

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

\begin{tabular}{|p{1.5in}|p{4.5in}|}
\hline
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.
\end{tabular}

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

- When a target address is written to NPC by a control transfer instruction
- When NPC is incremented to NPC + 4 during execution of an instruction that is not a taken control transfer
When the address is a nontranslating address (an address that is never translated by an MMU, such as the address of an internal register accessed using an ASI)

- When a WRPR instruction writes to TPC[TL] or TNPC[TL]
  
  **Programming Note**: Since writes to PSTATE are nondelayed (see page 68), a change to PSTATE.am can affect which instruction is executed immediately after the write to PSTATE.am. Specifically, if a WRPR to the PSTATE register changes the value of PSTATE.am from '0' to '1', and NPC[63:32] when the WRPR began execution was nonzero, then the next instruction executed after the WRPR will be from the address indicated in NPC[31:0] (with the more-significant 32 address bits set to zero).

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

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

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

**Programming Note**: PSTATE.am affects the operation of the edge-handling instructions, EDGE<8|16|32>[L]*. See Edge Handling Instructions on page 146 and Edge Handling Instructions (no CC) on page 147.

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

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

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

Outstanding disrupting exceptions that are destined for privileged mode can only cause a trap when the virtual processor is in nonprivileged or privileged mode and PSTATE.ie = 1. At all other times, they are held pending. For more details, see Conditioning of Disrupting Traps on page 422.

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

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

The privileged Trap Level register (TL; FIGURE 5-34) specifies the current trap level. TL = 0 is the normal (nontrap) level of operation. TL > 0 implies that one or more traps are being processed.

---

1. which sets PSTATE.am to '1', by restoring the value from TSTATE[TL].pstate.am to PSTATE.am
The maximum valid value that the TL register may contain is \( \text{MAXPTL} \), which is always equal to the number of supported trap levels beyond level 0.

**IMPL. DEP. #101-V9-CS10:** The architectural parameter \( \text{MAXPTL} \) 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 \( \text{MAXPTL} = \text{MAXPGL} \) (see impl dep. #401-S10). Architecturally, \( \text{MAXPTL} \) must be \( \geq 2 \).

In an Oracle SPARC Architecture 2011 implementation, \( \text{MAXPTL} = 2 \). 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 5-21.

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

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

**Programming Note**
An Oracle SPARC Architecture implementation only needs to implement sufficient bits in the TL register to encode the maximum trap level value. In an implementation where \( \text{MAXPTL} \leq 3 \), bits 63:2 of data written to the TL register using the WRPR instruction are ignored; only the least-significant two bits (bits 1:0) of TL are actually written. For example, if \( \text{MAXPTL} = 2 \), writing a value of \( 05_{16} \) to the TL register causes a value of \( 1_{16} \) to actually be stored in TL.

**Implementation Note**
\( \text{MAXPTL} = 2 \) for all Oracle SPARC Architecture 2011 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\(^\dagger\), an Oracle SPARC Architecture virtual processor’s behavior when executing with \( TL > 0 \) in nonprivileged mode is undefined.

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

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

The privileged Processor Interrupt Level register (\( \text{PIL}^P \); see FIGURE 5-35) specifies the interrupt level above which the virtual processor will accept an \( \text{interrupt_level}\_n \) interrupt. Interrupt priorities are mapped so that interrupt level 2 has greater priority than interrupt level 1, and so on. See TABLE 12-4 on page 427 for a list of exception and interrupt priorities.
5.7.10 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 5-36 illustrates the Global Level register.

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

The valid range of values that the GL register may contain is 0 to \(\text{MAXPGL}\), where \(\text{MAXPGL}\) is one fewer than the number of global register sets available to the virtual processor.

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

In all Oracle SPARC Architecture 2011 implementations, \(\text{MAXPGL} = 2\) (impl. dep. #401-S10).

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

GL operates similarly to TL, in that it increments during entry to a trap, but the values of GL and TL are independent. That is, 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, \(\text{MAXPTL}\) and \(\text{MAXPGL}\) are not necessarily equal.

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

In privileged mode, attempting to write a value greater than \(\text{MAXPGL}\) to the GL register causes \(\text{MAXPGL}\) to be written to GL.
The effect of writing to GL with a WRPR instruction is summarized in TABLE 5-22.

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

<table>
<thead>
<tr>
<th>Value $x$ Written with WRPR</th>
<th>Privilege Level when WRPR Is Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leq \text{MAXPG}L$</td>
<td>Privileged: $GL \leftarrow x$</td>
</tr>
<tr>
<td>$x &gt; \text{MAXPG}L$</td>
<td>Nonprivileged: privileged_opcode exception, $GL \leftarrow \text{MAXPG}L$ (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 6

Instruction Set Overview

Instructions are fetched by the virtual processor from memory and are executed, annulled, or trapped. Instructions are encoded in 4 major formats and partitioned into 11 general categories. Instructions are described in the following sections:

- Instruction Execution on page 75.
- Instruction Formats on page 76.
- Instruction Categories on page 76.

6.1 Instruction Execution

The instruction at the memory location specified by the program counter is fetched and then executed. Instruction execution may change program-visible virtual processor and/or memory state. As a side effect of its execution, new values are assigned to the program counter (PC) and the next program counter (NPC).

An instruction may generate an exception if it encounters some condition that makes it impossible to complete normal execution. Such an exception may in turn generate a precise trap. Other events may also cause traps: an exception caused by a previous instruction (a deferred trap), an interrupt or asynchronous error (a disrupting trap), or a reset request (a reset trap). If a trap occurs, control is vectored into a trap table. See Chapter 12, Traps, for a detailed description of exception and trap processing.

If a trap does not occur and the instruction is not a control transfer, the next program counter is copied into the PC, and the NPC is incremented by 4 (ignoring arithmetic overflow if any). There are two types of control-transfer instructions (CTIs): delayed and immediate. For a delayed CTI, at the end of the execution of the instruction, NPC is copied into the PC and the target address is copied into NPC. For an immediate CTI, at the end of execution, the target is copied to PC and target + 4 is copied to NPC. In the SPARC instruction set, many CTIs do not transfer control until after a delay of one instruction, hence the term “delayed CTI” (DCTI). Thus, the two program counters provide for a delayed-branch execution model.

For each instruction access and each normal data access, an 8-bit address space identifier (ASI) is appended to the 64-bit memory address. Load/store alternate instructions (see Address Space Identifiers (ASIs) on page 81) can provide an arbitrary ASI with their data addresses or can use the ASI value currently contained in the ASI register.
6.2 Instruction Formats

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

\[ \text{op} = 00_2: \text{SETHI, Branches, and ILLTRAP} \]

\[
\begin{array}{cccc}
00 & \text{rd} & \text{op2} & \text{imm22} \\
00 & a & \text{cond} & \text{op2} & \text{disp22} \\
00 & a & \text{cond} & \text{op2} & \text{cc1|cc0} & p & \text{disp19} \\
00 & a & 0 & \text{rcond} & \text{op2} & \text{d16hi} & p & \text{rs1} & \text{d16lo} \\
\end{array}
\]

\[ \text{op} = 01_2: \text{CALL} \]

\[
\begin{array}{c}
01 & \text{disp30} \\
\end{array}
\]

\[ \text{op} = 10_2 \text{ or } 11_2: \text{Arithmetic, Logical, Moves, Tcc, Loads, Stores, Prefetch, and Misc} \]

\[
\begin{array}{cccc}
1x & \text{rd} & \text{op3} & \text{rs1} & =0 & \text{imm asi} & \text{rs2} \\
1x & \text{rd} & \text{op3} & \text{rs1} & =1 & \text{simm13} & \\
\end{array}
\]

FIGURE 6-1 Summary of Instruction Formats

6.3 Instruction Categories

Oracle SPARC Architecture instructions can be grouped into the following categories:

- Memory access
- Memory synchronization
- Integer arithmetic
- Control transfer (CTI)
- Conditional moves
- Register window management
- State register access
- Privileged register access
- Floating-point operate
- Implementation dependent
- Reserved

These categories are described in the following subsections.
6.3.1 Memory Access Instructions

Load, store, load-store, and PREFETCH instructions are the only instructions that access memory. All of the memory access instructions except CASA, CASXA, and Partial Store use either two R registers or an R register and \texttt{simm13} to calculate a 64-bit byte memory address. For example, Compare and Swap uses a single R register to specify a 64-bit byte memory address. To this 64-bit address, an ASI is appended that encodes address space information.

The destination field of a memory reference instruction specifies the R or F register(s) that supply the data for a store or that receive the data from a load or LDSTUB. For SWAP, the destination register identifies the R register to be exchanged atomically with the calculated memory location. For Compare and Swap, an R register is specified, the value of which is compared with the value in memory at the computed address. If the values are equal, then the destination field specifies the R register that is to be exchanged atomically with the addressed memory location. If the values are unequal, then the destination field specifies the R register that is to receive the value at the addressed memory location; in this case, the addressed memory location remains unchanged. LDFS/LDFESR/LDXFSR and STFSR/STXFSR are special load and store instructions that load or store the floating-point status register, FSR, instead of acting on an R or F register.

The destination field of a PREFETCH instruction (\texttt{fctn}) 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.

6.3.1.1 Memory Alignment Restrictions

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

A floating-point doubleword access (LDDF, LDDFA, STDF, STDFA) should be aligned on an 8-byte boundary, but is only required to be aligned on a word (4-byte) boundary. A floating-point doubleword access to an address that is 4-byte aligned but not 8-byte aligned may result in less efficient and nonatomic access (causes a trap and is emulated in software (impl. dep. #109-V9-Cs10)), so 8-byte alignment is recommended.

A floating-point quadword access (LDQF, LDQFA, STQF, STQFA) should be aligned on a 16-byte boundary, but is only required to be aligned on a word (4-byte) boundary. A floating-point quadword access to an address that is 4-byte aligned but not 8-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 \texttt{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 \texttt{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 \texttt{STDF_mem_address_not_aligned} exception (impl. dep. #110-V9-Cs10).
An LDQF or LDQFA instruction accessing an address that is word aligned but not quadword aligned may cause an **LDQF_mem_address_not_aligned** exception (impl. dep. #111-V9-Cs10a).

**Implementation Note** Although the architecture provides for the **LDQF_mem_address_not_aligned** exception, Oracle SPARC Architecture 2011 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, Oracle SPARC Architecture 2011 implementations do not currently generate it.

### 6.3.1.2 Addressing Conventions

An Oracle SPARC Architecture virtual processor uses big-endian byte order for all instruction accesses and, by default, for data accesses. It is possible to access data in little-endian format by use of selected ASIs. It is also possible to change the default byte order for implicit data accesses. See *Processor State (PSTATE)* Register (PR 6) on page 68 for more information.

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.</td>
</tr>
<tr>
<td>halfword</td>
<td>For a load/store halfword instruction, two bytes are accessed. The most significant byte (bits 15–8) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 1.</td>
</tr>
<tr>
<td>word</td>
<td>For a load/store word instruction, four bytes are accessed. The most significant byte (bits 31–24) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 3.</td>
</tr>
<tr>
<td>doubleword or 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>
</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.
<table>
<thead>
<tr>
<th>Type</th>
<th>Address Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>\begin{align*} 7 &amp; 0 \end{align*}</td>
</tr>
<tr>
<td>Halfword</td>
<td>\begin{align*} 0 &amp; 1 \ 15 &amp; 8 &amp; 7 &amp; 0 \end{align*}</td>
</tr>
<tr>
<td>Word</td>
<td>\begin{align*} 00 &amp; 01 &amp; 10 &amp; 11 \ 31 &amp; 24 &amp; 23 &amp; 16 &amp; 15 &amp; 8 &amp; 7 &amp; 0 \end{align*}</td>
</tr>
<tr>
<td>Doubleword</td>
<td>\begin{align*} 000 &amp; 001 &amp; 010 &amp; 011 \ 63 &amp; 56 &amp; 55 &amp; 48 &amp; 47 &amp; 40 &amp; 39 &amp; 32 \end{align*}</td>
</tr>
<tr>
<td>Extended word</td>
<td>\begin{align*} 0000 &amp; 0001 &amp; 0010 &amp; 0011 \ 127 &amp; 120 &amp; 119 &amp; 112 &amp; 111 &amp; 104 &amp; 103 &amp; 96 \end{align*}</td>
</tr>
<tr>
<td>Quadword</td>
<td>\begin{align*} 0100 &amp; 0101 &amp; 0110 &amp; 0111 \ 95 &amp; 88 &amp; 87 &amp; 80 &amp; 79 &amp; 72 &amp; 71 &amp; 64 \end{align*}</td>
</tr>
<tr>
<td></td>
<td>\begin{align*} 1000 &amp; 1001 &amp; 1010 &amp; 1011 \ 63 &amp; 56 &amp; 55 &amp; 48 &amp; 47 &amp; 40 &amp; 39 &amp; 32 \end{align*}</td>
</tr>
<tr>
<td></td>
<td>\begin{align*} 1100 &amp; 1101 &amp; 1110 &amp; 1111 \ 31 &amp; 24 &amp; 23 &amp; 16 &amp; 15 &amp; 8 &amp; 7 &amp; 0 \end{align*}</td>
</tr>
</tbody>
</table>

FIGURE 6-2  Big-endian Addressing Conventions
Little-endian Addressing Convention. Within a multiple-byte integer, the byte with the smallest address is the least significant; a byte’s significance increases as its address increases. The little-endian addressing conventions are defined in TABLE 6-2 and illustrated in FIGURE 6-3.

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

<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>
</tbody>
</table>
| doubleword or extended word | For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 63–56) is accessed at the address + 7.  
For the deprecated integer load/store 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).  
†Note that the LDTXA instruction, which is not an LDTWA operation but does share LDTWA’s opcode, is not deprecated. |
| quadword              | For a load/store quadword instruction, 16 bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 127–120) is accessed at the address + 15. |
6.3.1.3 Address Space Identifiers (ASIs)

Alternate-space load, store, and load-store instructions specify an explicit ASI to use for their data access; when \( i = 0 \), the explicit ASI is provided in the instruction's \texttt{imm}_{asi} \texttt{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 \texttt{PSTATE.cle}. Instruction fetches use an implicit ASI that depends only on the current trap level. The cases are enumerated in TABLE 6-3.

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

<table>
<thead>
<tr>
<th>Access Type</th>
<th>TL</th>
<th>\texttt{PSTATE.cle}</th>
<th>ASI Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>( = 0 )</td>
<td>any</td>
<td>\texttt{ASI PRIMARY}</td>
</tr>
<tr>
<td></td>
<td>( &gt; 0 )</td>
<td>any</td>
<td>\texttt{ASI NUCLEUS}^{*}</td>
</tr>
</tbody>
</table>

FIGURE 6-3 Little-endian Addressing Conventions

Byte

<table>
<thead>
<tr>
<th>Address</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
</table>

Halfword

\begin{array}{cccc}
\text{Address}(0) & = & 7 & 0 & 15 & 8 \\
\end{array}

Word

\begin{array}{ccccccc}
\text{Address}(1:0) & = & 7 & 0 & 15 & 8 & 23 & 16 & 31 & 24 \\
\end{array}

Doubleword / Extended word

\begin{array}{cccccc}
\text{Address}(2:0) & = & 000 & 001 & 010 & 011 & 7 & 0 & 15 & 8 & 23 & 16 & 31 & 24 \\
\text{Address}(2:0) & = & 100 & 101 & 110 & 111 & 39 & 32 & 47 & 40 & 55 & 48 & 63 & 56 \\
\end{array}

Quadword

\begin{array}{ccccccc}
\text{Address}(3:0) & = & 0000 & 0001 & 0010 & 0011 & 7 & 0 & 15 & 8 & 23 & 16 & 31 & 24 \\
\text{Address}(3:0) & = & 0100 & 0101 & 0110 & 0111 & 39 & 32 & 47 & 40 & 55 & 48 & 63 & 56 \\
\text{Address}(3:0) & = & 1000 & 1001 & 1010 & 1011 & 71 & 64 & 79 & 72 & 87 & 80 & 95 & 88 \\
\text{Address}(3:0) & = & 1100 & 1101 & 1110 & 1111 & 103 & 96 & 111 & 104 & 119 & 112 & 127 & 120 \\
\end{array}
See also *Memory Addressing and Alternate Address Spaces* on page 383.

ASIs 0016–7F16 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 8016 through FF16 are unrestricted; software is allowed to access them regardless of the virtual processor’s privilege mode, as summarized in TABLE 6-4.

### TABLE 6-4 Allowed Accesses to ASIs

<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>0016–7F16</td>
<td>Restricted</td>
<td>Nonprivileged (0)</td>
<td><code>privileged_action</code> exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Privileged (1)</td>
<td>Valid access</td>
</tr>
<tr>
<td>8016–FF16</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 Oracle SPARC Architecture 2011 ASIs are implementation dependent. See TABLE 10-1 on page 399 for details.

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

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

**V9 Compatibility Note** In SPARC V9, an implementation could choose to decode only a subset of the 8-bit ASI specifier.

### 6.3.1.4 Separate Instruction Memory

A SPARC V9 implementation may choose to access instruction and data through the same address space and use hardware to keep data and instruction memory consistent at all times. It may also choose to overload independent address spaces for data and instructions and allow them to become inconsistent when data writes are made to addresses shared with the instruction space.

**Programming Note** A SPARC V9 program containing self-modifying code should use FLUSH instruction(s) after executing stores to modify instruction memory and before executing the modified instruction(s), to ensure the consistency of program execution.
6.3.2 Memory Synchronization Instructions

Two forms of memory barrier (MEMBAR) instructions allow programs to manage the order and completion of memory references. Ordering MEMBARs induce a partial ordering between sets of loads and stores and future loads and stores. Sequencing MEMBARs exert explicit control over completion of loads and stores (or other instructions). Both barrier forms are encoded in a single instruction, with subfunctions bit-encoded in cmask and mmask fields.

6.3.3 Integer Arithmetic and Logical Instructions

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

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

6.3.3.1 Setting Condition Codes

Most integer arithmetic instructions have two versions: one sets the integer condition codes (icc and xcc) as a side effect; the other does not affect the condition codes. A special comparison instruction for integer values is not needed since it is easily synthesized with the “subtract and set condition codes” (SUBcc) instruction. See Synthetic Instructions on page 490 for details.

6.3.3.2 Shift Instructions

Shift instructions shift an R register left or right by a constant or variable amount. None of the shift instructions change the condition codes.

6.3.3.3 Set High 22 Bits of Low Word

The “set high 22 bits of low word of an R register” instruction (SETHI) writes a 22-bit constant from the instruction into bits 31 through 10 of the destination register. It clears the low-order 10 bits and high-order 32 bits, and it does not affect the condition codes. Its primary use is to construct constants in registers.

6.3.3.4 Integer Multiply/Divide

The integer multiply instruction performs a $64 \times 64 \rightarrow 64$-bit operation; the integer divide instructions perform $64 \div 64 \rightarrow 64$-bit operations. For compatibility with SPARC V8 processors, $32 \times 32 \rightarrow 64$-bit multiply instructions, $64 \div 32 \rightarrow 32$-bit divide instructions, and the Multiply Step instruction are provided. Division by zero causes a division_by_zero exception.

6.3.3.5 Tagged Add/Subtract

The tagged add/subtract instructions assume tagged-format data, in which the tag is the two low-order bits of each operand. If either of the two operands has a nonzero tag or if 32-bit arithmetic overflow occurs, tag overflow is detected. If tag overflow occurs, then TADDcc and TSUBcc set the CCR.icc.v bit; if 64-bit arithmetic overflow occurs, then they set the CCR.xcc.v bit.

The trapping versions (TADDccTV, TSUBccTV) of these instructions are deprecated. See Tagged Add on page 346 and Tagged Subtract on page 351 for details.
6.3.4 Control-Transfer Instructions (CTIs)

The basic control-transfer instruction types are as follows:

- Conditional branch (Bicc, BPcc, BPr, FBfcc, FBPfcc)
- Compare and Branch (C*Bcond)
- Unconditional branch
- Call and link (CALL)
- Jump and link (JMPL, RETURN)
- Return from trap (DONE, RETRY)
- Trap (Tcc)

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

Some control transfer instructions (branches) can optionally annul, that is, not execute, the instruction in the delay slot, based on the setting of an annul bit in the instruction. The effect of the annul bit depends upon whether the transfer is taken or not taken and whether the branch is conditional or unconditional. Annulled delay instructions neither affect the program-visible state, nor can they cause a trap.

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 Oracle SPARC Architecture implementation, the only benefit of annulled branches might be a slight reduction in code size. Therefore, the use of annulled branch instructions is no longer encouraged.

TABLE 6-5 defines the value of the program counter and the value of the next program counter after execution of each instruction. Conditional branches have two forms: branches that test a condition (including branch-on-register), represented in the table by Bcc, and branches that are unconditional, that is, always or never taken, represented in the table by BA and BN, respectively. The effect of an annulled branch is shown in the table through explicit transfers of control, rather than by fetching and annulling the instruction.

<table>
<thead>
<tr>
<th>Instruction Group</th>
<th>Address Form</th>
<th>Delayed?</th>
<th>Taken?</th>
<th>Annul Bit?</th>
<th>New PC</th>
<th>New NPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-CTIs</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>NPC</td>
<td>NPC + 4</td>
</tr>
<tr>
<td>Bcc</td>
<td>PC-relative</td>
<td>Yes</td>
<td>Yes</td>
<td>0</td>
<td>NPC</td>
<td>EA</td>
</tr>
<tr>
<td>Bcc</td>
<td>PC-relative</td>
<td>Yes</td>
<td>No</td>
<td>0</td>
<td>NPC</td>
<td>NPC + 4</td>
</tr>
<tr>
<td>Bcc</td>
<td>PC-relative</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>NPC</td>
<td>EA</td>
</tr>
</tbody>
</table>
The effective address, “EA” in TABLE 6-5, specifies the target of the control-transfer instruction. The effective address is computed in different ways, depending on the particular instruction.

- **PC-relative effective address** — A PC-relative effective address is computed by sign extending the instruction’s immediate field to 64-bits, left-shifting the word displacement by 2 bits to create a byte displacement, and adding the result to the contents of the PC.

- **Register-indirect effective address** — If \( i = 0 \), a register-indirect effective target address is \( R[rs1] + R[rs2] \). If \( i = 1 \), a register-indirect effective target address is \( R[rs1] + \text{sign_ext}(\text{simm13}) \).

- **Trap vector effective address** — A trap vector effective address first computes the software trap number as the least significant 7 or 8 bits of \( R[rs1] + R[rs2] \) if \( i = 0 \), or as the least significant 7 or 8 bits of \( R[rs1] + \text{imm}_\text{trap#} \) if \( i = 1 \). Whether 7 or 8 bits are used depends on the privilege level — 7 bits are used in nonprivileged mode and 8 bits are used in privileged mode. The trap level, TL, is incremented. The hardware trap type is computed as 256 + the software trap number and stored in \( TT[TL] \). The effective address is generated by combining the contents of the TBA register with the trap type and other data; see *Trap Processing* on page 432 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].

---

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

<table>
<thead>
<tr>
<th>Instruction Group</th>
<th>Address Form</th>
<th>Delayed?</th>
<th>Taken?</th>
<th>Annul Bit?</th>
<th>New PC</th>
<th>New NPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>C*Bcond</td>
<td>PC-relative</td>
<td>No</td>
<td>Yes</td>
<td>—</td>
<td>EA</td>
<td>EA + 4</td>
</tr>
<tr>
<td>C*Bcond</td>
<td>PC-relative</td>
<td>No</td>
<td>No</td>
<td>—</td>
<td>NPC</td>
<td>NPC + 4</td>
</tr>
<tr>
<td>JMPL, RETURN</td>
<td>Register-indirect</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>NPC</td>
<td>EA</td>
</tr>
<tr>
<td>DONE</td>
<td>Trap state</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>TNPC[TL]</td>
<td>TNPC[TL] + 4</td>
</tr>
<tr>
<td>RETRY</td>
<td>Trap state</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>TPC[TL]</td>
<td>TNPC[TL]</td>
</tr>
<tr>
<td>Tcc</td>
<td>Trap vector</td>
<td>No</td>
<td>Yes</td>
<td>—</td>
<td>EA</td>
<td>EA + 4</td>
</tr>
<tr>
<td>Tcc</td>
<td>Trap vector</td>
<td>No</td>
<td>No</td>
<td>—</td>
<td>NPC</td>
<td>NPC + 4</td>
</tr>
</tbody>
</table>

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

---

### 6.3.4.1 Conditional Branches

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

Note: The annuling behavior of a taken conditional branch is different from that of an unconditional branch.
6.3.4.2 Unconditional Branches

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

Note: The annul behavior of an unconditional branch is different from that of a taken conditional branch.

6.3.4.3 CALL and JMPL Instructions

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

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

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

6.3.4.4 RETURN Instruction

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

6.3.4.5 DONE and RETRY Instructions

The DONE and RETRY instructions are used by privileged software to return from a trap. These instructions restore the machine state to values saved in the TSTATE register stack.

RETRY returns to the instruction that caused the trap in order to reexecute it. DONE returns to the instruction pointed to by the value of NPC associated with the instruction that caused the trap, that is, the next logical instruction in the program. DONE presumes that the trap handler did whatever was requested by the program and that execution should continue.

6.3.4.6 Trap Instruction (Tcc)

The Tcc instruction initiates a trap if the condition specified by its cond field matches the current state of the condition code 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.
6.3.4.7 DCTI Couples

A delayed control transfer instruction (DCTI) in the delay slot of another DCTI is referred to as a “DCTI couple”. The use of DCTI couples is deprecated in the Oracle SPARC Architecture; no new software should place a DCTI in the delay slot of another DCTI, because on future Oracle SPARC Architecture implementations DCTI couples may execute either slowly or differently than the programmer assumes it will.

SPARC V8 and SPARC V9 Compatibility Note
The SPARC V8 architecture left behavior undefined for a DCTI couple. The SPARC V9 architecture defined behavior in that case, but as of UltraSPARC Architecture 2005, use of DCTI couples was deprecated. Software should not expect high performance from DCTI couples, and performance of DCTI couples should be expected to decline further in future processors.

Programming Note
As noted in TABLE 6-5 on page 84, an annulled branch-always (branch-always with a = 1) instruction is not architecturally a DCTI. However, since not all implementations make that distinction, for optimal performance, a DCTI should not be placed in the instruction word immediately following an annulled branch-always instruction (BA,A or BPA,A).

6.3.5 Conditional Move Instructions

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

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

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

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

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

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

can be coded as

```
cmp %i0, %i2 ! (A > B)
or %g0, 0, %i3 ! set X = 0
movg %xcc, 1, %i3 ! overwrite X with 1 if A > B
```
to eliminate the need for a branch.
**MOVr and FMOVr Instructions.** The MOVr and FMOVr instructions allow the contents of any integer or floating-point register to be moved to a destination integer or floating-point register if the contents of a register satisfy a specified condition. The conditions to test are enumerated in **TABLE 6-6.**

**TABLE 6-6 MOVR and FMOVr Test Conditions**

<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,

```plaintext
movrnz %i2, %l4, %l6
```

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

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

### 6.3.6 Register Window Management Instructions

This subsection describes the instructions that manage register windows in the Oracle SPARC Architecture. The privileged registers affected by these instructions are described in *Register-Window PR State Registers* on page 61.

#### 6.3.6.1 SAVE Instruction

The SAVE instruction allocates a new register window and saves the caller’s register window by incrementing the CWP register.

If CANSAVE = 0, then execution of a SAVE instruction causes a window spill exception, that is, one of the `spill_n_<normal|other>` exceptions.

If CANSAVE ≠ 0 but the number of clean windows is zero, that is, `(CLEANWIN - CANRESTORE) = 0`, then SAVE causes a `clean_window` exception.

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

#### 6.3.6.2 RESTORE Instruction

The RESTORE instruction restores the previous register window by decrementing the CWP register.

If CANRESTORE = 0, execution of a RESTORE instruction causes a window fill exception, that is, one of the `fill_n_<normal|other>` exceptions.

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

SAVED is a privileged instruction used by a spill trap handler to indicate that a window spill has completed successfully. It increments CANSAVE and decrements either OTHERWIN or CANRESTORE, depending on the conditions at the time SAVED is executed.

See SAVED on page 309 for details.

6.3.6.4 RESTORED Instruction

RESTORED is a privileged instruction, used by a fill trap handler to indicate that a window has been filled successfully. It increments CANRESTORE and decrements either OTHERWIN or CANSAVE, depending on the conditions at the time RESTORED is executed. RESTORED also manipulates CLEANWIN, which is used to ensure that no address space’s data become visible to another address space through windowed registers.

See RESTORED on page 302 for details.

6.3.6.5 Flush Windows Instruction

The FLUSHW instruction flushes all of the register windows, except the current window, by performing repetitive spill traps. The FLUSHW instruction causes a spill trap if any register window (other than the current window) has valid contents. The number of windows with valid contents is computed as:

\[ N_{\text{REG WINDOWS}} - 2 - \text{CANSAVE} \]

If this number is nonzero, the FLUSHW instruction causes a spill trap. Otherwise, FLUSHW has no effect. If the spill trap handler exits with a RETRY instruction, the FLUSHW instruction continues causing spill traps until all the register windows except the current window have been flushed.

Programming Note

This note describes a common convention for use of register windows, SAVE, RESTORE, CALL, and JMPL instructions.

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

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

The SAVE and RESTORE instructions can be used to atomically establish a new memory stack pointer in an R register and switch to a new or previous register window.
6.3.7 Ancillary State Register (ASR) Access

The read/write state register instructions access program-visible state and status registers. These instructions read/write the state registers into/from R registers. A read/write Ancillary State register instruction is privileged only if the accessed register is privileged.

The supported RDAsr and WRAsr instructions are described in Ancillary State Registers on page 48.

6.3.8 Privileged Register Access

The read/write privileged register instructions access state and status registers that are visible only to privileged software. These instructions read/write privileged registers into/from R registers. The read/write privileged register instructions are privileged.

6.3.9 Floating-Point Operate (FPop) Instructions

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

The term “FPop” refers to instructions in the FPop1, FMAf, and FPop2 opcode spaces. FPop instructions do not include FBfcc instructions, loads and stores between memory and the F registers, or non-floating-point operations that read or write F registers.

The FMOVcc instructions function for the floating-point registers as the MOVcc instructions do for the integer registers. See MOVcc and FMOVcc Instructions on page 87.

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 88.

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. In this case, software running in a mode with greater privileges must emulate any functionality not present in the hardware.

See ftt = 2 (unfinished_FPop) on page 45 to see which instructions can produce an fp_exception_other exception (with FSR.ftt = unfinished_FPop).

6.3.10 Implementation-Dependent Instructions

The SPARC V9 architecture provided two instruction spaces that are entirely implementation dependent: IMPDEP1 and IMPDEP2.
In the Oracle SPARC Architecture, the IMPDEP1 opcode space is used by many VIS instructions. The IMPDEP2B opcode space is primarily used for implementation of floating-point multiply-add/multiply-subtract instructions. The remaining opcodes in IMPDEP1 and IMPDEP2 are now marked as reserved opcodes.

6.3.11 Reserved Opcodes and Instruction Fields

If a conforming Oracle SPARC Architecture 2011 implementation attempts to execute an instruction bit pattern that is not specifically defined in this specification, it behaves as follows:

- If the instruction bit pattern encodes an implementation-specific extension to the instruction set, that extension is executed.
- If the instruction does not encode an extension to the instruction set, then the instruction bit pattern is invalid and causes an *illegal_instruction* exception.

See Appendix A, *Opcode Maps*, for an enumeration of the reserved instruction bit patterns (opcodes).

*Programming Note* For software portability, software (such as assemblers, static compilers, and dynamic compilers) that generates SPARC instructions must always generate zeroes in instruction fields marked “reserved” (“—”).
Oracle SPARC Architecture 2011 extends the standard SPARC V9 instruction set with additional classes of instructions:

- **Enhanced functionality:**
  - Instructions for alignment (*Align Address* on page 114)
  - Array handling (*Three-Dimensional Array Addressing* on page 117)
  - Byte-permutation instructions (*Byte Mask and Shuffle* on page 122 and *CMASK* on page 136)
  - Edge handling (*Edge Handling Instructions* on pages 146 and 147)
  - Logical operations on floating-point registers (*F Register Logical Operate (1 operand)* on page 213)
  - Partitioned arithmetic (*Partitioned Add* on page 196, *Partitioned Add with Saturation* on page 199, or *PSTATE.psf = 0* *Partitioned Subtract* on page 207, and *Partitioned Subtract with Saturation* on page 210)
  - Pixel manipulation (*FEXPAND* on page 160, *FPACK* on page 192, and or *PSTATE.psf = 0* *FPMERGE* on page 206)

- **Efficient memory access**
  - Partial store (*DAE_invalid.asi* *DAE_invalid.asi* *Store Partial Floating-Point* on page 332)
  - Short floating-point loads and stores (*Store Short Floating-Point* on page 335)
  - Block load and store (*Block Load* on page 230 and *Block Store* on page 322)
  - Efficient interval arithmetic: SIAM (*Set Interval Arithmetic Mode* on page 315) and all instructions that reference GSR.im
  - Floating-point Multiply-Add and Multiply-Subtract (FMA) instructions (*Floating-Point Multiply-Add and Multiply-Subtract (fused)* on page 170)
  - Direct moves between integer and floating-point registers: *Move Floating-Point Register to Integer Register* on page 271 and *Move Integer Register to Floating-Point Register* on page 272

TABLE 7-2 provides a quick index of instructions, alphabetically by architectural instruction name.

TABLE 7-4 summarizes the instruction set, listed within functional categories.
Within these tables and throughout the rest of this chapter, and in Appendix A, *Opcode Maps*, certain opcodes are marked with mnemonic superscripts. The superscripts and their meanings are defined in TABLE 7-1.

**TABLE 7-1** Instruction Superscripts

<table>
<thead>
<tr>
<th>Superscript</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Deprecated instruction (do not use in new software)</td>
</tr>
<tr>
<td>H&lt;sub&gt;dis&lt;/sub&gt;</td>
<td>Privileged action if in nonprivileged or privileged mode and access is disabled</td>
</tr>
<tr>
<td>N</td>
<td>Nonportable instruction</td>
</tr>
<tr>
<td>P</td>
<td>Privileged instruction</td>
</tr>
<tr>
<td>P&lt;sub&gt;ASI&lt;/sub&gt;</td>
<td>Privileged action if bit 7 of the referenced ASI is 0</td>
</tr>
<tr>
<td>P&lt;sub&gt;ASR&lt;/sub&gt;</td>
<td>Privileged instruction if the referenced ASR register is privileged</td>
</tr>
<tr>
<td>P&lt;sub&gt;dis&lt;/sub&gt;</td>
<td>Privileged action if in nonprivileged mode (PSTATE,priv = 0) and nonprivileged access is disabled</td>
</tr>
<tr>
<td>Page</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>107</td>
<td>ADD (ADDcc)</td>
</tr>
<tr>
<td>107</td>
<td>ADDC (ADDCCc)</td>
</tr>
<tr>
<td>108</td>
<td>ADDXC (ADDXCCc) N</td>
</tr>
<tr>
<td>108</td>
<td>ADDXC (ADDXCCc) N</td>
</tr>
<tr>
<td>109</td>
<td>AES_DROUND&lt;01</td>
</tr>
<tr>
<td>109</td>
<td>AES_EROUND&lt;01</td>
</tr>
<tr>
<td>112</td>
<td>AES_KEXPAND0</td>
</tr>
<tr>
<td>112</td>
<td>AES_KEXPAND1</td>
</tr>
<tr>
<td>112</td>
<td>AES_KEXPAND2</td>
</tr>
<tr>
<td>114</td>
<td>ALIGNADDRESS[_LITTLE]</td>
</tr>
<tr>
<td>115</td>
<td>ALLOCLEAN</td>
</tr>
<tr>
<td>115</td>
<td>AND (ANDcc)</td>
</tr>
<tr>
<td>115</td>
<td>ANDN (ANDNcc)</td>
</tr>
<tr>
<td>117</td>
<td>ARRAY&lt;8</td>
</tr>
<tr>
<td>120</td>
<td>Bicc</td>
</tr>
<tr>
<td>120</td>
<td>BMASK</td>
</tr>
<tr>
<td>123</td>
<td>BPcc</td>
</tr>
<tr>
<td>125</td>
<td>BPir</td>
</tr>
<tr>
<td>127</td>
<td>CALL</td>
</tr>
<tr>
<td>128</td>
<td>CAMELLIA_FN</td>
</tr>
<tr>
<td>130</td>
<td>CAMELLIA_FLN</td>
</tr>
<tr>
<td>130</td>
<td>CAMELLIA_FLIN</td>
</tr>
<tr>
<td>131</td>
<td>CASA^PASI</td>
</tr>
<tr>
<td>131</td>
<td>CASX^PASI</td>
</tr>
<tr>
<td>133</td>
<td>Cbcond</td>
</tr>
<tr>
<td>136</td>
<td>CMASK&lt;8</td>
</tr>
<tr>
<td>138</td>
<td>CRC32C^N</td>
</tr>
<tr>
<td>142</td>
<td>DES_IP^N</td>
</tr>
<tr>
<td>142</td>
<td>DES_IPN</td>
</tr>
<tr>
<td>142</td>
<td>DES_KEXPANDN</td>
</tr>
<tr>
<td>142</td>
<td>DES_ROUNDN</td>
</tr>
<tr>
<td>144</td>
<td>DONE^P</td>
</tr>
<tr>
<td>146</td>
<td>EDGE&lt;8</td>
</tr>
<tr>
<td>147</td>
<td>EDGE&lt;8</td>
</tr>
<tr>
<td>220</td>
<td>F&lt;s</td>
</tr>
<tr>
<td>219</td>
<td>F&lt;s</td>
</tr>
<tr>
<td>219</td>
<td>F&lt;s</td>
</tr>
<tr>
<td>149</td>
<td>FABS&lt;s</td>
</tr>
<tr>
<td>150</td>
<td>FADD&lt;s</td>
</tr>
<tr>
<td>151</td>
<td>FALIGNNDATAg</td>
</tr>
<tr>
<td>215</td>
<td>FANDNOT&lt;1</td>
</tr>
<tr>
<td>Page</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>228</td>
<td>LDSWAPASI</td>
</tr>
<tr>
<td>249</td>
<td>LDTXAN</td>
</tr>
<tr>
<td>244</td>
<td>LDTWD^{D}</td>
</tr>
<tr>
<td>246</td>
<td>LDTWA^{D}, PASI</td>
</tr>
<tr>
<td>227</td>
<td>LDUB</td>
</tr>
<tr>
<td>228</td>
<td>LDUBPASI</td>
</tr>
<tr>
<td>228</td>
<td>LDUH</td>
</tr>
<tr>
<td>228</td>
<td>LDUHAPASI</td>
</tr>
<tr>
<td>227</td>
<td>LDX</td>
</tr>
<tr>
<td>228</td>
<td>LDXAPASI</td>
</tr>
<tr>
<td>253</td>
<td>LZCNT</td>
</tr>
<tr>
<td>255</td>
<td>MD5^{N}</td>
</tr>
<tr>
<td>256</td>
<td>MEMBAR</td>
</tr>
<tr>
<td>259</td>
<td>MONTMUL^{N}</td>
</tr>
<tr>
<td>263</td>
<td>MONTSQR^{N}</td>
</tr>
<tr>
<td>267</td>
<td>MOVcc</td>
</tr>
<tr>
<td>271</td>
<td>MOVFToi</td>
</tr>
<tr>
<td>272</td>
<td>MOVftof</td>
</tr>
<tr>
<td>270</td>
<td>MOVr</td>
</tr>
<tr>
<td>273</td>
<td>MPMUL^{N}</td>
</tr>
<tr>
<td>278</td>
<td>MULX</td>
</tr>
<tr>
<td>279</td>
<td>NOP</td>
</tr>
<tr>
<td>280</td>
<td>NORMALW</td>
</tr>
<tr>
<td>281</td>
<td>OR (ORcc)</td>
</tr>
<tr>
<td>281</td>
<td>ORN (ORNcc)</td>
</tr>
<tr>
<td>282</td>
<td>OTHERW</td>
</tr>
<tr>
<td>283</td>
<td>PAUSE</td>
</tr>
<tr>
<td>284</td>
<td>PDIST^{D}</td>
</tr>
<tr>
<td>285</td>
<td>PDISTN</td>
</tr>
<tr>
<td>286</td>
<td>POPC</td>
</tr>
<tr>
<td>288</td>
<td>PREFETCH</td>
</tr>
<tr>
<td>288</td>
<td>PREFETCHPASI</td>
</tr>
<tr>
<td>295</td>
<td>RDASI</td>
</tr>
<tr>
<td>295</td>
<td>RDASr^{PASR}</td>
</tr>
<tr>
<td>295</td>
<td>RDCCCR</td>
</tr>
<tr>
<td>295</td>
<td>RDCFRI</td>
</tr>
<tr>
<td>295</td>
<td>RDFPRS</td>
</tr>
<tr>
<td>295</td>
<td>RDGSR</td>
</tr>
</tbody>
</table>
Oracle SPARC Architecture instructions are grouped into “feature sets”. Each set comprises a collection of architectural features that were introduced at the same time. Those feature sets are listed in the following table:

<table>
<thead>
<tr>
<th>Architectural Feature Set Name</th>
<th>Sun Studio Compiler Option that enables generation of instructions in Feature Set (-xarch=)</th>
<th>SPARC Processor in which Feature Set was first implemented (year)</th>
<th>Oracle SPARC Architecture Specification in which Feature Set first appeared</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARC4</td>
<td>sparc4</td>
<td>SPARC T4 (Core S3, 2011)</td>
<td>Oracle SPARC Architecture 2011</td>
</tr>
</tbody>
</table>
# TABLE 7-4 Instruction Set - by Functional Category (1 of 7)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVcc</td>
<td>Move integer register if condition is satisfied</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>MOVR</td>
<td>Move integer register on contents of integer register</td>
<td>270</td>
<td></td>
</tr>
</tbody>
</table>

## Data Movement Operations, Between R Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOV&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point move</td>
</tr>
<tr>
<td>FMOV&lt;s</td>
<td>d</td>
<td>q&gt;cc</td>
<td>Move floating-point register if condition is satisfied</td>
</tr>
<tr>
<td>FMOV&lt;s</td>
<td>d</td>
<td>q&gt;R</td>
<td>Move f-p reg. if integer reg. contents satisfy condition</td>
</tr>
<tr>
<td>FSRC&lt;1</td>
<td>2&gt;&lt;s</td>
<td>d&gt;</td>
<td>Copy source to destination</td>
</tr>
</tbody>
</table>

## Data Movement Operations, Between F Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVfTOi</td>
<td>Move floating-point register to integer register</td>
<td>271</td>
<td>VIS 3 (2010)</td>
</tr>
<tr>
<td>MOViToF</td>
<td>Move integer register to floating-point register</td>
<td>272</td>
<td>VIS 3 (2010)</td>
</tr>
</tbody>
</table>

## Data Conversion Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>FiTO&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Convert 32-bit integer to floating-point</td>
</tr>
<tr>
<td>F&lt;s</td>
<td>d</td>
<td>q&gt;TOi</td>
<td>Convert floating point to integer</td>
</tr>
<tr>
<td>F&lt;s</td>
<td>d</td>
<td>q&gt;TOx</td>
<td>Convert floating point to 64-bit integer</td>
</tr>
<tr>
<td>F&lt;s</td>
<td>d</td>
<td>q&gt;TO&lt;s</td>
<td>d</td>
</tr>
<tr>
<td>FxTO&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Convert 64-bit integer to floating-point</td>
</tr>
</tbody>
</table>

## Logical Operations on R Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND (ANDecc)</td>
<td>Logical and (and modify condition codes)</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>OR (ORcc)</td>
<td>Inclusive-or (and modify condition codes)</td>
<td>281</td>
<td></td>
</tr>
<tr>
<td>ORN (ORNcc)</td>
<td>Inclusive-or not (and modify condition codes)</td>
<td>281</td>
<td></td>
</tr>
<tr>
<td>XNOR (XNORcc)</td>
<td>Exclusive-nor (and modify condition codes)</td>
<td>363</td>
<td></td>
</tr>
<tr>
<td>XOR (XORcc)</td>
<td>Exclusive-or (and modify condition codes)</td>
<td>363</td>
<td></td>
</tr>
</tbody>
</table>

## Logical Operations on F Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAND&lt;s</td>
<td>d&gt;</td>
<td>Logical and operation</td>
<td>215</td>
</tr>
<tr>
<td>FANDNOT&lt;1</td>
<td>2&gt;&lt;s</td>
<td>d&gt;</td>
<td>Logical and operation with one inverted source</td>
</tr>
<tr>
<td>FNAND&lt;s</td>
<td>d&gt;</td>
<td>Logical nand operation</td>
<td>215</td>
</tr>
<tr>
<td>FNOR&lt;s</td>
<td>d&gt;</td>
<td>Logical nor operation</td>
<td>215</td>
</tr>
<tr>
<td>FNOT&lt;1</td>
<td>2&gt;&lt;s</td>
<td>d&gt;</td>
<td>Copy negated source</td>
</tr>
<tr>
<td>FONE&lt;s</td>
<td>d&gt;</td>
<td>One fill</td>
<td>213</td>
</tr>
<tr>
<td>FOR&lt;s</td>
<td>d&gt;</td>
<td>Logical or operation</td>
<td>215</td>
</tr>
<tr>
<td>FORNOT&lt;1</td>
<td>2&gt;&lt;s</td>
<td>d&gt;</td>
<td>Logical or operation with one inverted source</td>
</tr>
<tr>
<td>FXNOR&lt;s</td>
<td>d&gt;</td>
<td>Logical xnor operation</td>
<td>215</td>
</tr>
<tr>
<td>FXOR&lt;s</td>
<td>d&gt;</td>
<td>Logical xor operation</td>
<td>215</td>
</tr>
<tr>
<td>FZERO&lt;s</td>
<td>d&gt;</td>
<td>Zero fill</td>
<td>213</td>
</tr>
</tbody>
</table>

## Shift Operations on R Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>Shift left logical</td>
<td>316</td>
</tr>
<tr>
<td>SLLX</td>
<td>Shift left logical, extended</td>
<td>316</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift right arithmetic</td>
<td>316</td>
</tr>
<tr>
<td>SRAX</td>
<td>Shift right arithmetic, extended</td>
<td>316</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift right logical</td>
<td>316</td>
</tr>
<tr>
<td>SRLX</td>
<td>Shift right logical, extended</td>
<td>316</td>
</tr>
</tbody>
</table>
### Special Addressing and Data Alignment Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGNADDRESS[_LITTLE]</td>
<td>Calculate address for misaligned data</td>
<td>114</td>
<td>VIS 1 (1995)</td>
</tr>
<tr>
<td>ARRAY&lt;8</td>
<td>16</td>
<td>32&gt;</td>
<td>3-D array addressing instructions</td>
</tr>
<tr>
<td>FALIGNDATAg</td>
<td>Perform data alignment for misaligned data (using GSR.align)</td>
<td>151</td>
<td>VIS 1 (1995)</td>
</tr>
</tbody>
</table>

### Control Transfers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bicc</td>
<td>Branch on integer condition codes</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>BPcc</td>
<td>Branch on integer condition codes with prediction</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>BPpr</td>
<td>Branch on contents of integer register with prediction</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>Call and link</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>CBcond</td>
<td>Compare and Branch</td>
<td>133</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>DONEP</td>
<td>Return from trap</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>FBfccD</td>
<td>Branch on floating-point condition codes</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>FBPfcc</td>
<td>Branch on floating-point condition codes with prediction</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>ILLTRAP</td>
<td>Illegal instruction</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>JMPL</td>
<td>Jump and link</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>RETRYP</td>
<td>Return from trap and retry</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>Return</td>
<td>305</td>
<td></td>
</tr>
<tr>
<td>Tcc</td>
<td>Trap on integer condition codes</td>
<td>348</td>
<td></td>
</tr>
</tbody>
</table>

### Byte Permutation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMASK</td>
<td>Set the GSR.mask field</td>
<td>122</td>
<td>VIS 2 (2001)</td>
</tr>
<tr>
<td>BSHUFFLE</td>
<td>Permute bytes as specified by GSR.mask</td>
<td>122</td>
<td>VIS 2 (2001)</td>
</tr>
<tr>
<td>CMASK&lt;8</td>
<td>16</td>
<td>32&gt;(N)</td>
<td>Create GSR.mask from SIMD comparison result</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>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEXPAND</td>
<td>Pixel expansion</td>
<td>160</td>
<td>VIS 1 (1995)</td>
</tr>
<tr>
<td>FPACK&lt;16</td>
<td>32</td>
<td>FIX&gt;</td>
<td>Pixel packing</td>
</tr>
<tr>
<td>FPMERGE</td>
<td>Pixel merge</td>
<td>206</td>
<td>VIS 1 (1995)</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>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBLOCKFD</td>
<td>Block loads</td>
<td>230</td>
<td>VIS 1 (1995)</td>
</tr>
<tr>
<td>STBLOCKF</td>
<td>Block stores</td>
<td>322</td>
<td>VIS 1 (1995)</td>
</tr>
<tr>
<td>LDDF</td>
<td>Load double floating-point</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>LDDFA(P_{ASI})</td>
<td>Load double floating-point from alternate space</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>LDF</td>
<td>Load floating-point</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>LDFA(P_{ASI})</td>
<td>Load floating-point from alternate space</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>LDQF</td>
<td>Load quad floating-point</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>LDQFA(P_{ASI})</td>
<td>Load quad floating-point from alternate space</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF</td>
<td>Short floating-point loads</td>
<td>240</td>
<td>VIS 1 (1995)</td>
</tr>
<tr>
<td>STDF</td>
<td>Store double floating-point</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>STDFA(P_{ASI})</td>
<td>Store double floating-point into alternate space</td>
<td>327</td>
<td></td>
</tr>
<tr>
<td>STF</td>
<td>Store floating-point</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>STFA(P_{ASI})</td>
<td>Store floating-point into alternate space</td>
<td>327</td>
<td></td>
</tr>
<tr>
<td>STPARTIALF</td>
<td>Partial Store instructions</td>
<td>332</td>
<td>VIS 1 (1995)</td>
</tr>
<tr>
<td>STQF</td>
<td>Store quad floating point</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>STQFA(P_{ASI})</td>
<td>Store quad floating-point into alternate space</td>
<td>327</td>
<td></td>
</tr>
<tr>
<td>STSHORTF</td>
<td>Short floating-point stores</td>
<td>335</td>
<td>VIS 1 (1995)</td>
</tr>
</tbody>
</table>
### Table 7-4: Instruction Set - by Functional Category (3 of 7)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Atomic (Load-Store) Memory Operations to/from R Registers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CASA&lt;sub&gt;PASI&lt;/sub&gt;</td>
<td>Compare and swap word in alternate space</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>CASX&lt;sub&gt;PASI&lt;/sub&gt;</td>
<td>Compare and swap doubleword in alternate space</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>LDSTUB&lt;sub&gt;LDSH,PASI&lt;/sub&gt;</td>
<td>Load-store unsigned byte</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td>LDSTUBA&lt;sub&gt;PASI&lt;/sub&gt;</td>
<td>Load-store unsigned byte in alternate space</td>
<td>243</td>
<td></td>
</tr>
<tr>
<td>SWAP&lt;sup&gt;D&lt;/sup&gt;</td>
<td>Swap integer register with memory</td>
<td>343</td>
<td></td>
</tr>
<tr>
<td>SWAPA&lt;sup&gt;D&lt;/sup&gt;,&lt;sub&gt;PASI&lt;/sub&gt;</td>
<td>Swap integer register with memory in alternate space</td>
<td>344</td>
<td></td>
</tr>
</tbody>
</table>

| **Memory Operations to/from R Registers** |
| LDSB<sub>LDSH,PASI</sub> | Load signed byte | 227 |
| LDSBA<sub>PASI</sub> | Load signed byte from alternate space | 228 |
| LDSH<sub>LDSH,PASI</sub> | Load signed halfword | 227 |
| LDSHA<sub>PASI</sub> | Load signed halfword from alternate space | 228 |
| LDSW<sub>LDSW,PASI</sub> | Load signed word | 227 |
| LDSWA<sub>PASI</sub> | Load signed word from alternate space | 228 |
| LDTXAN<sub>LDTXAN,PASI</sub> | Load integer twin extended word from alternate space | 249 |
| LDTW<sub>PASI</sub> | Load integer twin word | 244 |
| LDTWA<sub>PASI</sub> | Load integer twin word from alternate space | 246 |
| LDUB<sub>LDUB,PASI</sub> | Load unsigned byte | 227 |
| LDUBA<sub>PASI</sub> | Load unsigned byte from alternate space | 228 |
| LDUH<sub>LDUH,PASI</sub> | Load unsigned halfword | 227 |
| LDUA<sub>PASI</sub> | Load unsigned halfword from alternate space | 228 |
| LDUW<sub>LDUW,PASI</sub> | Load unsigned word | 227 |
| LDX<sub>LDX,PASI</sub> | Load extended | 227 |
| LDXA<sub>PASI</sub> | Load extended from alternate space | 228 |
| STB<sub>STB,PASI</sub> | Store byte | 319 |
| STBA<sub>PASI</sub> | Store byte into alternate space | 320 |
| STTW<sup>D</sup> | Store twin word | 337 |
| STTTWA<sub>PASI</sub> | Store twin word into alternate space | 339 |
| STHW<sub>STHW,PASI</sub> | Store halfword | 319 |
| STHA<sub>PASI</sub> | Store halfword into alternate space | 320 |
| STW<sub>STW,PASI</sub> | Store word | 319 |
| STWA<sub>PASI</sub> | Store word into alternate space | 320 |
| STX<sub>STX,PASI</sub> | Store extended | 319 |
| STXA<sub>PASI</sub> | Store extended into alternate space | 320 |

| **Memory Operations — Miscellaneous** |
| LDFSR<sup>D</sup> | Load floating-point state register (lower) | 238 |
| LDXEFSR | Load Entire floating-point state register | 251 |
| LDXFSR | Load floating-point state register | 251 |
| MEMBAR | Memory barrier | 256 |
| PREFETCH | Prefetch data | 288 |
| PREFETCHA<sub>PASI</sub> | Prefetch data from alternate space | 288 |
| STFSRD<sup>D</sup> | Store floating-point state register (lower) | 330 |
| STXFSR | Store floating-point state register | 341 |

VIS 2 (2001)<sup>+</sup>  
VIS 3 (2010), VIS 3B (2010)
### TABLE 7-4  Instruction Set - by Functional Category (4 of 7)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-Point Arithmetic Operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FABS&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point absolute value</td>
</tr>
<tr>
<td>FADD&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point add</td>
</tr>
<tr>
<td>FDIV&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point divide</td>
</tr>
<tr>
<td>FdMUULq</td>
<td>Floating-point multiply double to quad</td>
<td>185</td>
<td></td>
</tr>
<tr>
<td>FHADD&lt;s</td>
<td>d&gt;</td>
<td>Floating-point add and halve</td>
<td>161</td>
</tr>
<tr>
<td>FHSUB&lt;s</td>
<td>d&gt;</td>
<td>Floating-point subtract and halve</td>
<td>162</td>
</tr>
<tr>
<td>FMADD(s,d)</td>
<td>Floating-point multiply-add single/double (fused)</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>FMSUB(s,d)</td>
<td>Floating-point multiply-subtract single/double (fused)</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>FMUL&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point multiply</td>
</tr>
<tr>
<td>FNADD&lt;s</td>
<td>d&gt;</td>
<td>Floating-point add and negate</td>
<td>186</td>
</tr>
<tr>
<td>FNMADD&lt;s</td>
<td>d&gt;</td>
<td>Floating-point negative multiply-add single/double (fused)</td>
<td>170</td>
</tr>
<tr>
<td>FNMUL&lt;s</td>
<td>d&gt;</td>
<td>Floating-point multiply and negate</td>
<td>190</td>
</tr>
<tr>
<td>FNEG&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point negate</td>
</tr>
<tr>
<td>FNMSUB(s,d)</td>
<td>Floating-point negative multiply-subtract single/double (fused)</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>FNMsMULd</td>
<td>Floating-point multiply single to double, and negate</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>FsMULd</td>
<td>Floating-point multiply single to double</td>
<td>185</td>
<td></td>
</tr>
<tr>
<td>FSQRT&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point square root</td>
</tr>
<tr>
<td>FSUB&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point subtract</td>
</tr>
<tr>
<td>Floating-Point Comparison Operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMP&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point compare</td>
</tr>
<tr>
<td>FCMPE&lt;s</td>
<td>d</td>
<td>q&gt;</td>
<td>Floating-point compare (exception if unordered)</td>
</tr>
<tr>
<td>FLCMP{s</td>
<td>d</td>
<td>}</td>
<td>Lexicographic compare</td>
</tr>
<tr>
<td>Register-Window Control Operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALLCLEAN^P</td>
<td>Mark all register window sets as “clean”</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>INVALWP</td>
<td>Mark all register window sets as “invalid”</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>FLUSHW</td>
<td>Flush register windows</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>NORMALWP</td>
<td>“Other” register windows become “normal” register windows</td>
<td>280</td>
<td></td>
</tr>
<tr>
<td>OTHERWP</td>
<td>“Normal” register windows become “other” register windows</td>
<td>282</td>
<td></td>
</tr>
<tr>
<td>RESTORE</td>
<td>Restore caller’s window</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>RESTORED^P</td>
<td>Window has been restored</td>
<td>302</td>
<td></td>
</tr>
<tr>
<td>SAVE</td>
<td>Save caller’s window</td>
<td>307</td>
<td></td>
</tr>
<tr>
<td>SAVED^P</td>
<td>Window has been saved</td>
<td>309</td>
<td></td>
</tr>
<tr>
<td>Miscellaneous Operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td>Flush instruction memory</td>
<td>166</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>Integer Arithmetic Operations on R Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD (ADDcc)</td>
<td>Add (and modify condition codes)</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>ADDC (ADDCCc)</td>
<td>Add with carry (and modify condition codes)</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>ADDXC (ADDXcc)</td>
<td>Add with extended carry (and modify condition codes)</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>MULX</td>
<td>Multiply 64-bit integers</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>SDIV^D (SDIVcc^D)</td>
<td>32-bit signed integer divide (and modify condition codes)</td>
<td>310</td>
<td></td>
</tr>
<tr>
<td>SDIVX</td>
<td>64-bit signed integer divide</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Category and Function</td>
<td>Page</td>
<td>Added in</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------</td>
<td>------</td>
<td>----------------</td>
</tr>
<tr>
<td>SMULD(SMULccD)</td>
<td>Signed integer multiply (and modify condition codes)</td>
<td>318</td>
<td></td>
</tr>
<tr>
<td>SUB(SUBcc)</td>
<td>Subtract (and modify condition codes)</td>
<td>342</td>
<td></td>
</tr>
<tr>
<td>SUBC(SUBCcc)</td>
<td>Subtract with carry (and modify condition codes)</td>
<td>342</td>
<td></td>
</tr>
<tr>
<td>TADDc</td>
<td>Tagged add and modify condition codes (trap on overflow)</td>
<td>346</td>
<td></td>
</tr>
<tr>
<td>TADDcTVD</td>
<td>Tagged add and modify condition codes (trap on overflow)</td>
<td>347</td>
<td></td>
</tr>
<tr>
<td>TSUBcc</td>
<td>Tagged subtract and modify condition codes (trap on overflow)</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>TSUBccTVD</td>
<td>Tagged subtract and modify condition codes (trap on overflow)</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>UDIVD(UDIVccD)</td>
<td>Unsigned integer divide (and modify condition codes)</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>UDIVX</td>
<td>64-bit unsigned integer divide</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>UMLUD(UMLccD)</td>
<td>Unsigned integer multiply (and modify condition codes)</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>UMULXHI</td>
<td>64 × 64 multiply yielding upper 64 bits of product</td>
<td>356</td>
<td>VIS 3 (2010)</td>
</tr>
</tbody>
</table>

**Integer SIMD (Partitioned) Operations on F Registers**

- FMEAN16 16-bit partitioned average 172 VIS 3 (2010)
- FPADD Partitioned integer add 196 VIS 1 (1995), 3
- FPADDS Partitioned integer add with saturation 199 VIS 3 (2010)
- FPCMP Partitioned Compare signed integer values 201 VIS 1 (1995)
- FPCMPU Partitioned Compare unsigned integer values 203 VIS 3 (2010), VIS 3B (2010)
- FPSUBS Partitioned Integer Subtract with Saturation 210 VIS 3 (2010)
- FS<LL|RL|RA><16|32> 16- or 32-bit partitioned shift, left or right 216 VIS 3 (2010)

**Integer Arithmetic and Logical Operations on F Registers**

- FPADD64 Integer add, 64-bit F registers 196 VIS 3B (2010)
- FMUL8x16 8x16 partitioned product 181 VIS 1 (1995)
- FMUL8x16[AU|AL] 8x16 upper/lower α partitioned product 181 VIS 1 (1995)
- FMUL8x16[SU|UL]x16 8x16 upper/lower partitioned product 181 VIS 1 (1995)
- FMULD8x8[SU|UL]x16 8x16 upper/lower partitioned product 181 VIS 1 (1995)
- FPMADDX[HI] 64-bit Integer multiply-add (low and high 64 bit results) 205 OSA 2011
- FPSUB64 Integer Subtract, 64-bit F registers 196 VIS 3B (2010)

**Miscellaneous Operations on R Registers**

- LZCNT Leading zeroes count, on 64-bit integer register 253 VIS 3 (2010)
- POPC Population count 286
- SETHI Set high 22 bits of low word of integer register 312

**Miscellaneous Operations on F Registers**

- EDGE<8|16|32>[L]cc Edge handling instructions (and modify condition codes) 146 VIS 1 (1995)
- EDGE<8|16|32>[L]N Edge handling instructions 147 VIS 2 (2001)
- FCHKSM16 16-bit partitioned checksum 156 VIS 3 (2010)
- PDISTN Distance between eight 8-bit components with no accumulation 285 VIS 3 (2010)

**Cryptographic and Secure Hash Operations**
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES_DROUND&lt;01</td>
<td>23&gt;</td>
<td>AES Decryption, Columns 0&amp;1 (or 2&amp;3)</td>
<td>109</td>
</tr>
<tr>
<td>AES_DROUND&lt;01</td>
<td>23&gt;_LAST</td>
<td>AES Decryption, Columns 0&amp;1 (or 2&amp;3), Last Round</td>
<td>109</td>
</tr>
<tr>
<td>AES_EROUND&lt;01</td>
<td>23&gt;</td>
<td>AES Encryption, Columns 0&amp;1 (or 2&amp;3)</td>
<td>109</td>
</tr>
<tr>
<td>AES_EROUND&lt;01</td>
<td>23&gt;_LAST</td>
<td>AES Encryption, Columns 0&amp;1 (or 2&amp;3), Last Round</td>
<td>109</td>
</tr>
<tr>
<td>AES_KEXPAND0</td>
<td>AES Key Expansion, without RCON</td>
<td>112</td>
<td>Crypto</td>
</tr>
<tr>
<td>AES_KEXPAND1</td>
<td>AES Key Expansion, with RCON</td>
<td>109</td>
<td>Crypto</td>
</tr>
<tr>
<td>AES_KEXPAND2</td>
<td>AES Key Expansion, without SBOX</td>
<td>112</td>
<td>Crypto</td>
</tr>
<tr>
<td>CAMELLIA_FN</td>
<td>Camellia “F” operation</td>
<td>128</td>
<td>Crypto</td>
</tr>
<tr>
<td>CAMELLIA_FLN</td>
<td>Camellia “FL” operation</td>
<td>130</td>
<td>Crypto</td>
</tr>
<tr>
<td>CAMELLIA_FLIN</td>
<td>Camellia “FLI” operation</td>
<td>130</td>
<td>Crypto</td>
</tr>
<tr>
<td>CRC32CN</td>
<td>Cyclic Redundancy Check</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>DES_IPN</td>
<td>DES Initial Permutation</td>
<td>142</td>
<td>Crypto</td>
</tr>
<tr>
<td>DES_IIPN</td>
<td>DES Inverse Initial Permutation</td>
<td>142</td>
<td>Crypto</td>
</tr>
<tr>
<td>DES_KEXPANDN</td>
<td>DES Key Expand</td>
<td>142</td>
<td>Crypto</td>
</tr>
<tr>
<td>DES_ROUNDN</td>
<td>DES Round Operations</td>
<td>140</td>
<td>Crypto</td>
</tr>
<tr>
<td>MD5N</td>
<td>MD5 Hash Operation</td>
<td>255</td>
<td>Crypto</td>
</tr>
<tr>
<td>MPMULN</td>
<td>Multiple-Precision Multiply</td>
<td>273</td>
<td>Crypto</td>
</tr>
<tr>
<td>MONTMULN</td>
<td>Montgomery Multiplication</td>
<td>259</td>
<td>Crypto</td>
</tr>
<tr>
<td>MONTSQRN</td>
<td>Montgomery Squaring</td>
<td>263</td>
<td>Crypto</td>
</tr>
<tr>
<td>SHA1N</td>
<td>SHA1 Secure Hash operation</td>
<td>313</td>
<td>Crypto</td>
</tr>
<tr>
<td>SHA256N</td>
<td>SHA256 Secure Hash operation</td>
<td>313</td>
<td>Crypto</td>
</tr>
<tr>
<td>SHA512N</td>
<td>SHA512 Secure Hash operation</td>
<td>313</td>
<td>Crypto</td>
</tr>
</tbody>
</table>

**Control and Status Register Access**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
<th>Added in</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAUSE</td>
<td>Pause Virtual Processor</td>
<td>283</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>RDASI</td>
<td>Read ASI register</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDasrPASR</td>
<td>Read ancillary state register</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDCCR</td>
<td>Read Condition Codes register (CCR)</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDFPRS</td>
<td>Read Floating-Point Registers State register (FPRS)</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDGSR</td>
<td>Read General Status register (GSR)</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDPC</td>
<td>Read Program Counter register (PC)</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDPRP</td>
<td>Read privileged register</td>
<td>298</td>
<td></td>
</tr>
<tr>
<td>RD SOFTINTP</td>
<td>Read per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDSTICKP&lt;Hi,Hd&gt;</td>
<td>Read System Tick register</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDSTICK_CMPRP</td>
<td>Read System Tick Compare register (STICK_CMPR)</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDTICKP&lt;Hi,Hd&gt;</td>
<td>Read Tick register</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>RDYD</td>
<td>Read Y register</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>SIAM</td>
<td>Set interval arithmetic mode</td>
<td>315</td>
<td>VIS 2 (2001)</td>
</tr>
<tr>
<td>WRASI</td>
<td>Write ASI register</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>WRasrPASR</td>
<td>Write ancillary state register</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>WRCCR</td>
<td>Write Condition Codes register (CCR)</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>WRFPRS</td>
<td>Write Floating-Point Registers State register (FPRS)</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>WRGSR</td>
<td>Write General Status register (GSR)</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>WRPRP</td>
<td>Write privileged register</td>
<td>536</td>
<td></td>
</tr>
<tr>
<td>WR SOFTINTP</td>
<td>Write per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>WRSOFTINT_CLRP</td>
<td>Clear bits of per-virtual processor Soft Interrupt register(SOFTINT)</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>WRSOFTINT_SETP</td>
<td>Set bits of per-virtual processor Soft Interrupt register (SOFTINT)</td>
<td>357</td>
<td></td>
</tr>
</tbody>
</table>
TABLE 7-4  Instruction Set - by Functional Category (7 of 7)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Category and Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRSTICK'</td>
<td>Write System Tick register (STICK)</td>
<td>357</td>
</tr>
<tr>
<td>WRSTICK_CMPR'</td>
<td>Write System Tick Compare register (STICK_CMPR)</td>
<td>357</td>
</tr>
<tr>
<td>WRYD</td>
<td>Write Y register</td>
<td>357</td>
</tr>
</tbody>
</table>
In the remainder of this chapter, related instructions are grouped into subsections. Each subsection contains the following sets of information:

(1) **Instruction Table.** This section of an instruction page lists the instructions that are defined in the subsection, including the values of the field(s) that uniquely identify the instruction(s) and its assembly language syntax. In the rightmost column, Software (alphabetic) and Implementation (numeric) classifications for the instructions are provided. The meaning of the alphabetic Software Classifications is as follows:

<table>
<thead>
<tr>
<th>Software Usage Class</th>
<th>How this feature may be used</th>
<th>Attributes</th>
</tr>
</thead>
</table>
| A                    | Use freely.                 | ■ Compilers always free to use (no option to disable use).  
                         |                            | ■ Executes well across all implementations. |
| B                    | Use with care/forethought in portable software | ■ Usage is being **phased in.**  
                         |                            | ■ A compiler option exists to enable/disable references to this feature; by default, use is **enabled.** |
| C                    | Use only in platform-specific software (privileged code, DLLs, and non-portable applications) | ■ New feature; usage is being **phased in.**  
                         |                            | ■ A compiler option† exists to enable/disable references to this feature; by default, use is **disabled.**  
                         |                            | ■ An assembler option† exists to enable/disable references to this feature; by default, use is **disabled.** If use is enabled, reference to feature triggers a warning; if disabled, reference triggers an error message. |
| D                    | Use in portable software is strongly discouraged. | ■ Usage is being **phased out** and this feature may not perform as well in future implementations.  
                         |                            | ■ A compiler option† exists to enable/disable use of this feature; by default, use is **disabled.**  
                         |                            | ■ An assembler option† exists to enable/disable references to this feature; by default, use is **disabled.** If use is enabled, reference to feature triggers a warning; if disabled, reference triggers an error message. |
| N                    | Only use in platform-specific software (privileged code, hyperprivileged code, DLLs, JIT code, and [if absolutely necessary] non-portable applications) | ■ A compiler option† exists to enable/disable references to this feature; by default, use is **disabled.**  
                         |                            | ■ An assembler option† exists to enable/disable references to this feature; by default, use is **disabled.** If use is enabled, reference to feature triggers a warning; if disabled, reference triggers an error message. |

(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 Oracle SPARC Architecture implementation encounters nonzero values in these fields, its behavior is as defined in **Reserved Opcodes and Instruction Fields** on page 91.
(3) **Description.** This subsection describes the operation of the instruction, its features, restrictions, and exception-causing conditions.

(4) **Exceptions.** The exceptions that can occur as a consequence of attempting to execute the instruction(s). Exceptions due to an `IAE_*`, and interrupts are not listed because they can occur on any instruction. An instruction not implemented in hardware generates an `illegal_instruction` exception and therefore will not generate any of the other exceptions listed. Exceptions are listed in order of trap priority (see *Trap Priorities* on page 432), from highest to lowest priority.

(5) **See Also.** A list of related instructions (on selected pages).

**Note** This specification does not contain any timing information (in either cycles or elapsed time), since timing is always implementation dependent.


## 7.1 Add

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
</table>
| ADD         | 000000 | Add                     | add reg
rs1, reg_or_imm, reg
rd                | A1    |
| ADDcc       | 010000 | Add and modify cc’s     | addcc reg
rs1, reg_or_imm, reg
rd               | A1    |
| ADDC        | 001000 | Add with 32-bit Carry   | addc reg
rs1, reg_or_imm, reg
rd              | A1    |
| ADDCc       | 011000 | Add with 32-bit Carry and modify cc’s | addccc reg
rs1, reg_or_imm, reg
rd            | A1    |

### Description

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

ADDc and ADDCc ("ADD with carry") also add the CCR register’s 32-bit carry (icc.c) bit. That is, if $i = 0$, they compute "$R[rs1] + R[rs2] + \text{icc.c}$" and if $i = 1$, they compute "$R[rs1] + \text{sign}_\text{ext}(\text{simm13}) + \text{icc.c}$". In either case, the sum is written to $R[rd]$.

ADDcc and ADDCc modify the integer condition codes (CCR.icc and CCR.xcc). Overflow occurs on addition if both operands have the same sign and the sign of the sum is different from that of the operands.

#### Programming Note

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

#### SPARC V8 Compatibility Note

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

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

### Exceptions

\textit{illegal_instruction}

### See Also

Add Extended with 64-bit Carry on page 108
ADDXC

7.2 Add Extended with 64-bit Carry [VIS 3]

The ADDXC instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

The ADDXC instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDXC</td>
<td>0 0001 0001</td>
<td>Add Extended with 64-bit Carry</td>
<td>addxc (r_{s1}, r_{s2}, r_{d})</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>ADDXCCc</td>
<td>0 0001 0011</td>
<td>Add Extended with 64-bit Carry and modify cc's</td>
<td>addxc (r_{s1}, r_{s2}, r_{d})</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

Description

ADDXC and ADDXCCc (“ADD extended with carry”) both compute “\(R[r_{s1}] + R[r_{s2}] + xcc.c\)” and write the sum into \(R[r_{d}]\).

In addition, ADDXCCc modifies 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

ADDXC and ADDXCCc can be used in conjunction with MULX and UMULXHI to speed up large multiword integer multiplication computations.

Exceptions

None

See Also

Add on page 107
7.3 AES Cryptographic Operations (4-operand)

The AES instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op5</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES_EROUND01</td>
<td>0000</td>
<td>AES Encrypt columns 0&amp;1</td>
<td>aes_eround01 freg, freg, freg, freg</td>
<td>N1</td>
</tr>
<tr>
<td>AES_EROUND23</td>
<td>0001</td>
<td>AES Encrypt columns 2&amp;3</td>
<td>aes_eround23 freg, freg, freg, freg</td>
<td>N1</td>
</tr>
<tr>
<td>AES_DROUND01</td>
<td>0010</td>
<td>AES Decrypt columns 0&amp;1</td>
<td>aes_dround01 freg, freg, freg, freg</td>
<td>N1</td>
</tr>
<tr>
<td>AES_DROUND23</td>
<td>0011</td>
<td>AES Decrypt columns 2&amp;3</td>
<td>aes_dround23 freg, freg, freg, freg</td>
<td>N1</td>
</tr>
<tr>
<td>AES_EROUND01_LAST</td>
<td>0100</td>
<td>AES Encrypt columns 0&amp;1 last round</td>
<td>aes_eround01_l freg, freg, freg, freg</td>
<td>N1</td>
</tr>
<tr>
<td>AES_EROUND23_LAST</td>
<td>0101</td>
<td>AES Encrypt columns 2&amp;3 last round</td>
<td>aes_eround23_l freg, freg, freg, freg</td>
<td>N1</td>
</tr>
<tr>
<td>AES_DROUND01_LAST</td>
<td>0110</td>
<td>AES Decrypt columns 0&amp;1 last round</td>
<td>aes_dround01_l freg, freg, freg, freg</td>
<td>N1</td>
</tr>
<tr>
<td>AES_DROUND23_LAST</td>
<td>0111</td>
<td>AES Decrypt columns 2&amp;3 last round</td>
<td>aes_dround23_l freg, freg, freg, freg</td>
<td>N1</td>
</tr>
<tr>
<td>AES_KEXPAND1</td>
<td>1000</td>
<td>AES Key expansion with RCON</td>
<td>aes_kexpand1 freg, freg, imm, freg</td>
<td>N1</td>
</tr>
</tbody>
</table>

Description
The AES instructions support encryption, decryption and key expansion for the Advanced Encryption Standard. This standard is available as FIPS-197 on http://nist.gov. Encryption is performed by looping on a set of primitive functions including SubBytes, ShiftRows, MixColumns, and AddRoundKey. Decryption loops the inverse set of functions which include InvShiftRows, InvSubBytes, AddRoundKey, and InvMixColumns. Key expansion utilizes SubBytes and RotWord functions as well as the XOR operation. A number of temporary variables are used in the functional descriptions below. For example, data_sb is the result of applying the SubBytes function to the RS2 and RS3 operand.

AES_EROUND01:

data_sb[127:0] ← SubBytes( F_D[rs2][63:0] :: F_D[rs3][63:0] )
data_sr[127:0] ← ShiftRows( data_sb[127:0] )
data_mc[127:0] ← MixColumns( data_sr[127:0] )
F_D[rd][63:0] ← AddRoundKey( data_mc[63:0], F_D[rs1][63:0] )

AES_EROUND23:

data_sb[127:0] ← SubBytes( F_D[rs2][63:0] :: F_D[rs3][63:0] )
data_sr[127:0] ← ShiftRows( data_sb[127:0] )
data_mc[127:0] ← MixColumns( data_sr[127:0] )
F_D[rd][63:0] ← AddRoundKey( data_mc[127:64], F_D[rs1][63:0] )

AES_DROUND01:

data_sr[127:0] ← InvShiftRows( F_D[rs2][63:0] :: F_D[rs3][63:0] )
data_sb[127:0] ← InvSubBytes( data_sr[127:0] )
data_ark[63:0] ← AddRoundKey( data_sb[63:0] , F_D[rs1][63:0] )
F_D[rd][63:0] ← InvMixColumn( data_ark[63:0] )

CHAPTER 7 • Instructions 109
AES Crypto (4-operand)

AES_DROUND23:

\[
\begin{align*}
data_{sr}\{127:0\} & \leftarrow \text{InvShiftRows}( F_D[rs2]\{63:0\} :: F_D[rs3]\{63:0\} ); \\
data_{sb}\{127:0\} & \leftarrow \text{InvSubBytes}( data_{sr}\{127:0\}); \\
data_{ark}\{63:0\} & \leftarrow \text{AddRoundKey}( data_{sb}\{127:64\} , F_D[rs1]\{63:0\} ); \\
F_D[rd]\{63:0\} & \leftarrow \text{InvMixColumn}( data_{ark}\{63:0\} );
\end{align*}
\]

AES_ROUND01_LAST:

\[
\begin{align*}
data_{sb}\{127:0\} & \leftarrow \text{SubBytes}( F_D[rs2]\{63:0\} :: F_D[rs3]\{63:0\} ); \\
data_{sr}\{127:0\} & \leftarrow \text{ShiftRows}( data_{sb}\{127:0\} ); \\
F_D[rd]\{63:0\} & \leftarrow \text{AddRoundKey}( data_{sr}\{63:0\} , F_D[rs1]\{63:0\} );
\end{align*}
\]

AES_DROUND23_LAST:

\[
\begin{align*}
data_{sr}\{127:0\} & \leftarrow \text{InvShiftRows}( F_D[rs2]\{63:0\} :: F_D[rs3]\{63:0\} ); \\
data_{sb}\{127:0\} & \leftarrow \text{InvSubBytes}( data_{sr}\{127:0\} ); \\
F_D[rd]\{63:0\} & \leftarrow \text{AddRoundKey}( data_{sb}\{127:64\} , F_D[rs1]\{63:0\} );
\end{align*}
\]

AES_ROUND01_LAST:

\[
\begin{align*}
data_{sr}\{127:0\} & \leftarrow \text{InvShiftRows}( F_D[rs2]\{63:0\} :: F_D[rs3]\{63:0\} ); \\
data_{sb}\{127:0\} & \leftarrow \text{InvSubBytes}( data_{sr}\{127:0\} ); \\
F_D[rd]\{63:0\} & \leftarrow \text{AddRoundKey}( data_{sb}\{127:64\} , F_D[rs1]\{63:0\} );
\end{align*}
\]

AES_DROUND23_LAST:

\[
\begin{align*}
data_{sr}\{127:0\} & \leftarrow \text{InvShiftRows}( F_D[rs2]\{63:0\} :: F_D[rs3]\{63:0\} ); \\
data_{sb}\{127:0\} & \leftarrow \text{InvSubBytes}( data_{sr}\{127:0\} ); \\
F_D[rd]\{63:0\} & \leftarrow \text{AddRoundKey}( data_{sb}\{63:0\} , F_D[rs1]\{63:0\} );
\end{align*}
\]

AES_KEXPAND1:

\[
\begin{align*}
data_{rw}\{31:0\} & \leftarrow \text{RotWord}( F_D[rs2]\{31:0\} ); \\
data_{sb}\{31:0\} & \leftarrow \text{SubBytes}( data_{rw}\{31:0\} ); \\
F_D[rd]\{63:32\} & \leftarrow data_{sb}\{31:0\} \text{ xor } F_D[rs1]\{63:32\} \text{ xor } rcon\{31:0\}; \\
F_D[rd]\{31:0\} & \leftarrow data_{sb}\{31:0\} \text{ xor } F_D[rs1]\{63:32\} \text{ xor } rcon\{31:0\} \text{ xor } F_D[rs1]\{31:0\};
\end{align*}
\]

where rcon is a function of imm5, as shown below:

<table>
<thead>
<tr>
<th>imm5</th>
<th>rcon[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>000002</td>
<td>0100 000016</td>
</tr>
<tr>
<td>000012</td>
<td>0200 000016</td>
</tr>
<tr>
<td>000102</td>
<td>0400 000016</td>
</tr>
<tr>
<td>000112</td>
<td>0800 000016</td>
</tr>
<tr>
<td>001002</td>
<td>1000 000016</td>
</tr>
<tr>
<td>001012</td>
<td>2000 000016</td>
</tr>
<tr>
<td>001102</td>
<td>4000 000016</td>
</tr>
<tr>
<td>001112</td>
<td>8000 000016</td>
</tr>
<tr>
<td>010002</td>
<td>1000 000016</td>
</tr>
<tr>
<td>010012</td>
<td>3000 000016</td>
</tr>
<tr>
<td>010102</td>
<td>0000 000016</td>
</tr>
<tr>
<td>111112</td>
<td>0000 000016</td>
</tr>
</tbody>
</table>
The AES instructions are components of the overall AES algorithm. To perform an encryption or decryption, the key must first be expanded. Key expansion is done only once per session key. The expanded keys are then applied to all blocks for that session. In the following example, expanded keys are stored in F0 thru F42, plain text is stored in F52 and F54, F56 and F58 are scratch registers, F60 and F62 hold the cipher text. For each block, the following instruction sequence can be applied for an AES 128 ECB encryption:

```
fxtor %f00, %f52, %f52 //initial ARK
fxor %f02, %f54, %f54
aes_eround01 %f04, %f52, %f54, %f56  // Round 1
aes_eround23 %f06, %f52, %f54, %f58
aes_eround01 %f08, %f56, %f58, %f52  // Round 2
aes_eround23 %f10, %f56, %f58, %f54
aes_eround01 %f12, %f52, %f54, %f56  // Round 3
aes_eround23 %f14, %f52, %f54, %f58
aes_eround01 %f16, %f56, %f58, %f52  // Round 4
aes_eround23 %f18, %f56, %f58, %f54
aes_eround01 %f20, %f52, %f54, %f56  // Round 5
aes_eround23 %f22, %f52, %f54, %f58
aes_eround01 %f24, %f56, %f58, %f52  // Round 6
aes_eround23 %f26, %f56, %f58, %f54
aes_eround01 %f28, %f52, %f54, %f56  // Round 7
aes_eround23 %f30, %f52, %f54, %f58
aes_eround01 %f32, %f56, %f58, %f52  // Round 8
aes_eround23 %f34, %f56, %f58, %f54
aes_eround01 %f36, %f52, %f54, %f56  // Round 9
aes_eround23 %f38, %f52, %f54, %f58
aes_eround01 %f40, %f56, %f58, %f60  // Round 10
aes_eround23 %f42, %f56, %f58, %f62
```

If CFR.aes = 0, an attempt to execute any AES instruction causes a `compatibility_feature` exception.

Software must check that CFR.aes = 1 before executing any of these AES instructions. If CFR.aes = 0, then software should assume that an attempt to execute one of the AES instruction either
(1) will generate an `illegal_instruction` exception because it is not implemented in hardware, or
(2) will execute, but perform some other operation.

Therefore, if CFR.aes = 0, software should perform the corresponding AES operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.
The AES instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

**Description**

AES_KEXPAND0:

\[
data_{sb}[31:0] \leftarrow \text{SubBytes}(F_D[rs2][31:0]);
\]

\[
F_D[rd][63:32] \leftarrow data_{sb}[31:0] \text{ xor } F_D[rs1][63:32];
\]

\[
F_D[rd][31:0] \leftarrow data_{sb}[31:0] \text{ xor } F_D[rs1][63:32] \text{ xor } F_D[rs1][31:0];
\]

AES_KEXPAND2:

\[
F_D[rd][63:32] \leftarrow F_D[rs2][31:0] \text{ xor } F_D[rs1][63:32];
\]

\[
F_D[rd][31:0] \leftarrow F_D[rs2][31:0] \text{ xor } F_D[rs1][63:32] \text{ xor } F_D[rs1][31:0];
\]

**Programming Note**

The AES instructions are components of the overall AES algorithm. To perform an encryption or decryption, the key must first be expanded. Key expansion is done only once per session key; the expanded keys are then applied to all blocks for that session. The following is an example of key expansion for AES 128. The original keys are loaded into F0 and F2. The expanded keys are stored in F4 thru F42:

```assembly
aes_kexpand1 %f0, %f2, 0x0, %f4               !# w[4], w[5]
aes_kexpand2 %f2, %f4, %f6                   !# w[6], w[7]
aes_kexpand1 %f4, %f6, 0x1, %f8               !# w[8], w[9]
aes_kexpand2 %f6, %f8, %f10                  !# w[10], w[11]
aes_kexpand1 %f8, %f10, 0x2, %f12             !# w[12], w[13]
aes_kexpand2 %f10, %f12, %f14                 !# w[14], w[15]
aes_kexpand1 %f12, %f14, 0x3, %f16            !# w[16], w[17]
aes_kexpand2 %f14, %f16, %f18                 !# w[18], w[19]
aes_kexpand1 %f16, %f18, 0x4, %f20             !# w[20], w[21]
aes_kexpand2 %f18, %f20, %f22                 !# w[22], w[23]
aes_kexpand1 %f20, %f22, 0x5, %f24             !# w[24], w[25]
aes_kexpand2 %f22, %f24, %f26                  !# w[26], w[27]
aes_kexpand1 %f24, %f26, 0x6, %f28             !# w[28], w[29]
aes_kexpand2 %f26, %f28, %f30                  !# w[30], w[31]
aes_kexpand1 %f28, %f30, 0x7, %f32             !# w[32], w[33]
aes_kexpand2 %f30, %f32, %f34                  !# w[34], w[35]
aes_kexpand1 %f32, %f34, 0x8, %f36             !# w[36], w[37]
aes_kexpand2 %f34, %f36, %f38                  !# w[38], w[39]
aes_kexpand1 %f36, %f38, 0x9, %f40             !# w[40], w[41]
aes_kexpand2 %f38, %f40, %f42                  !# w[42], w[43]
```

If CFR.aes = 0, an attempt to execute any AES instruction causes a *compatibility_feature* exception.
**AES Crypto (3-operand)**

**Programming Note**
Software must check that CFR.aes = 1 before executing any of these AES instructions. If CFR.aes = 0, then software should assume that an attempt to execute one of the AES instruction either
1. will generate an *illegal_instruction* exception because it is not implemented in hardware, or
2. will execute, but perform some other operation.

Therefore, if CFR.aes = 0, software should perform the corresponding AES operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.

**Exceptions**
`fp_disabled`

**See Also**
AES Cryptographic Operations (4 operand) on page 109
## 7.5 Align Address [VIS1]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGNADDRESS</td>
<td>0 0001 1000</td>
<td>Calculate address for misaligned data access</td>
<td>alignaddr rs_{rs1}, rs_{rs2}, rs_{rd}</td>
<td>A1</td>
<td>UA 2005</td>
</tr>
<tr>
<td>ALIGNADDRESS_LITTLE</td>
<td>0 0001 1010</td>
<td>Calculate address for misaligned data access, little-endian</td>
<td>alignaddrl rs_{rs1}, rs_{rs2}, rs_{rd}</td>
<td>A1</td>
<td>UA 2005</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 FALIGNDATAg 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 (using GSR.align) on page 151
7.6 Mark All Register Window Sets “Clean”

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

\[
\text{CLEANWIN} \leftarrow (\text{N\_REG\_WINDOWS} - 1)
\]

**Programming Note**
ALLCLEAN is used to indicate that all register windows are “clean”; that is, do not contain data belonging to other address spaces. It is needed because the value of \(\text{N\_REG\_WINDOWS}\) is not known to privileged software.

An attempt to execute an ALLCLEAN instruction when reserved instruction bits 18:0 are nonzero causes an illegal_instruction exception.

An attempt to execute an ALLCLEAN instruction in nonprivileged mode (PSTATE.priv = 0) causes a privileged_opcode exception.

**Exceptions**
- illegal_instruction
- privileged_opcode

**See Also**
- INVALW on page 225
- NORMALW on page 280
- OTHERW on page 282
- RESTORED on page 302
- SAVED on page 309

---

### AND 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>AND</td>
<td>00 0001</td>
<td>and</td>
<td>\text{and} \text{ reg}<em>{rs1}, \text{ reg_or_imm}, \text{ reg}</em>{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>ANDcc</td>
<td>01 0001</td>
<td>and and modify cc’s</td>
<td>\text{andcc} \text{ reg}<em>{rs1}, \text{ reg_or_imm}, \text{ reg}</em>{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>ANDN</td>
<td>00 0101</td>
<td>and not</td>
<td>\text{andn} \text{ reg}<em>{rs1}, \text{ reg_or_imm}, \text{ reg}</em>{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>ANDNcc</td>
<td>01 0101</td>
<td>and not and modify cc’s</td>
<td>\text{andncc} \text{ reg}<em>{rs1}, \text{ reg_or_imm}, \text{ reg}</em>{rd}</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**
These instructions implement bitwise logical \text{and} operations. They compute “R[rs1] \text{ op } R[rs2]” if \(i = 0\), or “R[rs1] \text{ op sign\_ext (simm13)}” if \(i = 1\), and write the result into R[rd].
AND, ANDN

ANDcc and ANDNcc modify the integer condition codes (icc and xcc). They set the condition codes as follows:

- icc.v, icc.c, xcc.v, and xcc.c are set to 0
- icc.n is copied from bit 31 of the result
- xcc.n is copied from bit 63 of the result
- icc.z is set to 1 if bits 31:0 of the result are zero (otherwise to 0)
- xcc.z is set to 1 if all 64 bits of the result are zero (otherwise to 0)

ANDN and ANDNcc logically negate their second operand before applying the main (and) operation.

An attempt to execute an AND, ANDcc, ANDN or ANDNcc instruction when \( i = 0 \) and reserved instruction bits 12:5 are nonzero causes an illegal_instruction exception.

Exceptions

illegal_instruction
7.8 Three-Dimensional Array Addressing

**Description**
These instructions convert three-dimensional (3D) fixed-point addresses contained in R[rs1] to a blocked-byte address; they store the result in R[rd]. Fixed-point addresses typically are used for address interpolation for planar reformatting operations. Blocking is performed at the 64-byte level to maximize external cache block reuse, and at the 64-Kbyte level to maximize TLB entry reuse, regardless of the orientation of the address interpolation. These instructions specify an element size of 8 bits (ARRAY8), 16 bits (ARRAY16), or 32 bits (ARRAY32).

The second operand, R[rs2], specifies the power-of-2 size of the X and Y dimensions of a 3D image array. The legal values for R[rs2] and their meanings are shown in TABLE 7-5. Illegal values produce undefined results in the destination register, R[rd].

**TABLE 7-5 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.

If the texture data were laid out in the obvious fashion (the z = 0 plane, followed by the z = 1 plane, etc.), then even small changes in z would result in references to distant pages in memory. The resulting lack of locality would tend to result in TLB misses and poor performance. The three versions of the array instruction, ARRAY8, ARRAY16, and ARRAY32, differ only in the scaling of the computed memory offsets. ARRAY16 shifts its result left by one position and ARRAY32 shifts left by two in order to handle 16- and 32-bit texture data.

When using the array instructions, a “blocked-byte” data formatting structure is imposed. The N × N × M volume, where N = 2^n × 64, M = m × 32, 0 ≤ n ≤ 5, 1 ≤ m ≤ 16 should be composed of 64 × 64 × 32 smaller volumes, which in turn should be composed of 4 × 4 × 2 volumes. This data structure is optimal for 16-bit data. For 16-bit data, the 4 × 4 × 2 volume has 64 bytes of data, which is ideal for reducing cache-line misses; the 64 × 64 × 32 volume will have 256 Kbytes of data, which is good for improving the TLB hit rate. FIGURE 7-1 illustrates how the data has to be organized, where the origin
(0,0,0) is assumed to be at the lower-left front corner and the x coordinate varies faster than y than z. That is, when traversing the volume from the origin to the upper right back, you go from left to right, front to back, bottom to top.

**FIGURE 7-1**  Blocked-Byte Data Formatting Structure

The array instructions have 2 inputs:

The (x,y,z) coordinates are input via a single 64-bit integer organized in `R[rs1]` as shown in **FIGURE 7-2**.

<table>
<thead>
<tr>
<th>Z integer</th>
<th>Z fraction</th>
<th>Y integer</th>
<th>Y fraction</th>
<th>X integer</th>
<th>X fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>55</td>
<td>54</td>
<td>44</td>
<td>43</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>22</td>
<td>21</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

**FIGURE 7-2**  Three-Dimensional Array Fixed-Point Address Format

Note that z has only 9 integer bits, as opposed to 11 for x and y. Also note that since (x,y,z) are all contained in one 64-bit register, they can be incremented or decremented simultaneously with a single add or subtract instruction (ADD or SUB).

So for a $512 \times 512 \times 32$ or a $512 \times 512 \times 256$ volume, the size value is 3. Note that the x and y size of the volume must be the same. The z size of the volume is a multiple of 32, ranging between 32 and 512.

The array instructions generate an integer memory offset, that when added to the base address of the volume, gives the address of the volume element (voxel) and can be used by a load instruction. The offset is correct only if the data has been reformatted as specified above.

The integer parts of x, y, and z are converted to the following blocked-address formats as shown in **FIGURE 7-3** for ARRAY8, **FIGURE 7-4** for ARRAY16, and **FIGURE 7-5** for ARRAY32.

<table>
<thead>
<tr>
<th>UPPER</th>
<th>MIDDLE</th>
<th>LOWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>Y</td>
<td>X</td>
</tr>
<tr>
<td>+2n</td>
<td>+2n +n</td>
<td>+n</td>
</tr>
</tbody>
</table>

**FIGURE 7-3**  Three-Dimensional Array Blocked-Address Format (ARRAY8)
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 7-6</th>
<th>ARRAY8 Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result ($R[rd]$) Bits</td>
<td>Source ($R[rs1]$) Bits</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+n</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.

```assembly
add           Addr, DeltaAddr, Addr
array8        Addr, %g0, bAddr
ldda          {bAddr} #ASI_FL8_PRIMARY, data
faligndata    data, accum, accum
```

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

Unconditional branches and icc-conditional branches are described below:

- **Unconditional branches** (BA, BN) — If its annul bit is 0 (a = 0), a BN (Branch Never) instruction is treated as a NOP. If its annul bit is 1 (a = 1), the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

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

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

    | Opcode | cond | Operation | icc Test | Assembly Language Syntax | Class |
    |--------|------|-----------|----------|--------------------------|-------|
    | BA     | 1000 | Branch Always | 1        | ba{,a} label             | A1    |
    | BN     | 0000 | Branch Never | 0        | bn{,a} label             | A1    |
    | BNE    | 1001 | Branch on Not Equal | not Z | bne{,a} label | A1 |
    | BE     | 0001 | Branch on Equal | Z | be{,a} label | A1 |
    | BG     | 1010 | Branch on Greater | not (Z or (N xor V)) | bg{,a} label | A1 |
    | BLE    | 0010 | Branch on Less or Equal | Z or (N xor V) | ble{,a} label | A1 |
    | BGE    | 1011 | Branch on Greater or Equal | not (N xor V) | bge{,a} label | A1 |
    | BL     | 0011 | Branch on Less | N xor V | bl{,a} label | A1 |
    | BGU    | 1100 | Branch on Greater Unsigned | not (C or Z) | bgu{,a} label | A1 |
    | BLEU   | 0100 | Branch on Less or Equal Unsigned | C or Z | bleu{,a} label | A1 |
    | BCC    | 1101 | Branch on Carry Clear (Greater Than or Equal, Unsigned) | not C | bcc{,a} label | A1 |
    | BCS    | 0101 | Branch on Carry Set (Less Than, Unsigned) | C | bcs{,a} label | A1 |
    | BPOS   | 1110 | Branch on Positive | not N | bpos{,a} label | A1 |
    | BNEG   | 0110 | Branch on Negative | N | bneg{,a} label | A1 |
    | BVC    | 1111 | Branch on Overflow Clear | not V | bvc{,a} label | A1 |
    | BVS    | 0111 | Branch on Overflow Set | V | bvs{,a} label | A1 |

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.

Programming Note

Unconditional branches and icc-conditional branches are described below:

- **Unconditional branches** (BA, BN) — If its annul bit is 0 (a = 0), a BN (Branch Never) instruction is treated as a NOP. If its annul bit is 1 (a = 1), the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

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

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

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

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

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

Programming Note | For optimal performance, a DCTI should not be placed in the instruction word immediately following an annulled branch-always instruction. For additional information, see DCTI Couples on page 87.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the Bicc instruction will cause a transfer of control (BA or taken conditional branch), then Bicc generates a control_transfer_instruction exception instead of causing a control transfer. When a control_transfer_instruction trap occurs, PC (the address of the Bicc instruction) is stored in TPC[TL] and the value of NPC from before the Bicc was executed is stored in TNPC[TL].

Note that BN never causes a control_transfer_instruction exception.

Exceptions | control_transfer_instruction (impl. dep. #450-S20)
7.10 Byte Mask and Shuffle

**Description**

BMASK adds two integer registers, \( R[rs1] \) and \( R[rs2] \), and stores the result in the integer register \( R[rd] \). The least significant 32 bits of the result are stored in the GSR.mask field.

BSHUFFLE concatenates the two 64-bit floating-point registers \( F_D[rs1] \) (more significant half) and \( F_D[rs2] \) (less significant half) to form a 128-bit (16-byte) value. Bytes in the concatenated value are numbered from most significant to least significant, with the most significant byte being byte 0. BSHUFFLE extracts 8 of those 16 bytes and stores the result in the 64-bit floating-point register \( F_D[rd] \). Bytes in \( F_D[rd] \) are also numbered from most to least significant, with the most significant being byte 0. The following table indicates which source byte is extracted from the concatenated value to generate each byte in the destination register, \( F_D[rd] \).

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

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

**Exceptions**

`fp_disabled`

**See Also**

CMASK on page 136
### 7.11 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</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPN 0000</td>
<td>Branch Never</td>
<td>0</td>
<td>bn[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPNE 1001</td>
<td>Branch on Not Equal</td>
<td>not Z</td>
<td>bne[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPE 0001</td>
<td>Branch on Equal</td>
<td>Z</td>
<td>be[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPG 1010</td>
<td>Branch on Greater</td>
<td>not (Z or (N xor V))</td>
<td>bg[a],[pt</td>
<td>,pn]</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</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPGE 1011</td>
<td>Branch on Greater or Equal</td>
<td>not (N xor V)</td>
<td>bge[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPL 0011</td>
<td>Branch on Less</td>
<td>N xor V</td>
<td>bl[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPGU 1100</td>
<td>Branch on Greater Unsigned</td>
<td>not (C or Z)</td>
<td>bgu[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPLEU 0100</td>
<td>Branch on Less or Equal Unsigned</td>
<td>C or Z</td>
<td>bleu[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPCC 1101</td>
<td>Branch on Carry Clear</td>
<td>not C</td>
<td>bcc0[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPCS 0101</td>
<td>Branch on Carry Set</td>
<td>C</td>
<td>bcs[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPPOS 1110</td>
<td>Branch on Positive</td>
<td>not N</td>
<td>bpos[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPNEG 0110</td>
<td>Branch on Negative</td>
<td>N</td>
<td>bneg[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPVC 1111</td>
<td>Branch on Overflow Clear</td>
<td>not V</td>
<td>bvc[a],[pt</td>
<td>,pn]</td>
</tr>
<tr>
<td>BPVS 0111</td>
<td>Branch on Overflow Set</td>
<td>V</td>
<td>bvs[a],[pt</td>
<td>,pn]</td>
</tr>
</tbody>
</table>

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

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

**Programming Note** To set the annul (a) bit for BPcc instructions, append “.a” to the opcode mnemonic. For example, use bgu,a %icc,label. Braces in the preceding table signify that the “.a” is optional. To set the branch prediction bit, append to an opcode mnemonic either “.pt” for predict taken or “.pn” for predict not taken. If neither “.pt” nor “.pn” is specified, the assembler defaults to “.pt”. To select the appropriate integer condition code, include “%icc” or “%xcc” before the label.

**Description** Unconditional branches and conditional branches are described below.
BPcc

- **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 \( \text{PC} + (4 \times \text{sign}_{\text{ext}} (\text{disp19})) \) specifies an address of an instruction that is expected to be executed soon. If the Branch Never’s annul bit is 1 (\( a = 1 \)), then the following (delay) instruction is annulled (not executed). If the annul bit is 0 (\( a = 0 \)), then the following instruction is executed. Branch Always always causes a transfer of control, and in no case does Branch Never ever cause a transfer of control to take place.

BPA (Branch Always with Prediction) causes an unconditional PC-relative, delayed control transfer to the address “\( \text{PC} + (4 \times \text{sign}_{\text{ext}} (\text{disp19})) \)”. If the annul bit of the branch instruction is 1 (\( a = 1 \)), then the delay instruction is annulled (not executed). If the annul bit is 0 (\( a = 0 \)), then the delay instruction is executed.

- **Conditional branches** — Conditional BPcc instructions (except BPA and BPN) evaluate one of the two integer condition codes (icc or xcc), as selected by \( \text{cc0} \) and \( \text{cc1} \), 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 “\( \text{PC} + (4 \times \text{sign}_{\text{ext}} (\text{disp19})) \)”. If FALSE, the branch is not taken.

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

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

The predict bit (\( p \)) is used to give the hardware a hint about whether the branch is expected to be taken. A 1 in the \( p \) bit indicates that the branch is expected to be taken; a 0 indicates that the branch is expected not to be taken.

Annulment, delay instructions, prediction, and delayed control transfers are described further in Chapter 6, Instruction Set Overview.

**Programming Note** — For optimal performance, a DCTI should not be placed in the instruction word immediately following an annulled branch-always instruction. For additional information, see DCTI Couples on page 87.

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

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTASTE.tct = 1, and the BPcc instruction will cause a transfer of control (BPA or taken conditional branch), then BPcc generates a control_transfer_instruction exception instead of causing a control transfer. When a control_transfer_instruction trap occurs, PC (the address of the BPcc) is stored in TPC[TL] and the value of NPC from before the BPcc was executed is stored in TNPC[TL].

Note that BPN never causes a control_transfer_instruction exception.

**Exceptions**

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

**See Also** — Branch on Integer Register with Prediction (BPr) on page 125
7.12 Branch on Integer Register with Prediction (BPr)

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

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

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

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

An attempt to execute a BPr instruction when instruction bit 28 = 1 or rcond is a reserved value (000₂ or 100₂) causes an illegal_instruction exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the BPr instruction will cause a transfer of control (taken conditional branch), then BPr generates a control_transfer_instruction exception instead of causing a control transfer.

### 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”.

### Table: BPr Instructions

<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>—</td>
<td>000</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<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>regst1, label</td>
</tr>
<tr>
<td>BRLEZ</td>
<td>010</td>
<td>Branch on Register Less Than or Equal to Zero</td>
<td>R[rs1] ≤ 0</td>
<td>brelz [, a[, pt, pn]]</td>
<td>regst1, 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>regst1, label</td>
</tr>
<tr>
<td>—</td>
<td>100</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
<td>—</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>regst1, label</td>
</tr>
<tr>
<td>BRGZ</td>
<td>110</td>
<td>Branch on Register Greater Than Zero</td>
<td>R[rs1] &gt; 0</td>
<td>brgz [, a[, pt, pn]]</td>
<td>regst1, label</td>
</tr>
<tr>
<td>BRGEZ</td>
<td>111</td>
<td>Branch on Register Greater Than or Equal to Zero</td>
<td>R[rs1] ≥ 0</td>
<td>brgez [, a[, pt, pn]]</td>
<td>regst1, label</td>
</tr>
</tbody>
</table>

† Some very early SPARC V9 implementations (circa 1995) ignored the value of bit 28 and executed a BPr instruction even if bit 28 = 1. Since then, all implementations have treated bits 24:22 = 011₂, bit 28 = 1, and bits 31:30 = 00 ₂ as an illegal_instruction or, in the case of implementations of Oracle SPARC Architecture 2011 and later, as a CBcond instruction (see page 133).
Annulment, delay instructions, prediction, and delayed control transfers are described further in Chapter 6, *Instruction Set Overview*.

**Implementation Note**  
If this instruction is implemented by tagging each register value with an N (negative) bit and Z (zero) bit, the table below can be used to determine if \( r_{cond} \) is TRUE:

<table>
<thead>
<tr>
<th>Branch</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<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 or Z</td>
</tr>
<tr>
<td>BRGZ</td>
<td>\text{not } (N or Z)</td>
</tr>
</tbody>
</table>

**Exceptions**  
*illegal_instruction*  
*controltransfer_instruction* (impl. dep. #450-S20)

**See Also**  
Branch on Integer Condition Codes with Prediction (BPcc) on page 123
7.13 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)

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

**Exceptions**

control_transfer_instruction (impl. dep. #450-S20)

**See Also**

JMPL on page 226
CAMELLIA 4-operand Op

7.14 Camellia Operations (4 operand)

The Camellia instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op5</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAMELLIA_F</td>
<td>1100</td>
<td>Camellia F operation</td>
<td>camellia_f freg[rs1], freg[rs2], freg[rs3], freg[rd]</td>
<td>N1</td>
</tr>
</tbody>
</table>

### Description

Camellia is a block cipher that produces a 128-bit output from a 128-bit input under the control of a 128, 192, or 256-bit key. The specifications for Camellia can be found at:

http://info.isl.ntt.co.jp/crypt/eng/camellia/specifications.html

The F function is applied 18 times for a 128-bit key and 24 times for a 192 or 256-bit key. Between groups of six F functions, the FL and FLI functions are applied to the upper and lower halves of the data respectively. XOR operations are applied at the beginning and end to complete the cipher. The temporary variable data_f is used in the functional description below and represents the result of applying the F function to the rs3 data using the rs1 key. The Camellia instructions operate on 64-bit floating-point registers.

CAMELLIA_F:

\[
\text{data}_f[63:0] \leftarrow \text{camellia_F(data=F_D[rs3][63:0], key=F_D[rs1][63:0])}; \\
\text{F}_D[rd][63:0] \leftarrow \text{data}_f[63:0] \text{xor} F_D[rs2][63:0];
\]
The Camellia instructions are components of the overall Camellia algorithm. To perform an encryption or decryption, software must first expand the key. Key expansion is done only once per session key. The expanded keys are then applied to all blocks for that session. The following instruction sequence is an example of a 128-bit encrypt for one block. The expanded keys are loaded in F0 thru F50. The initial text is loaded in FD[54] and FD[52].

```plaintext
fxor       %f0,  %f54,      %f54    !# Pre-Whiten
fxor       %f2,  %f52,      %f52
.camellia_f %f4,  %f52, %f54, %f52    !# Round 1   F 1
camellia_f %f6,  %f54, %f52, %f54    !# Round 1   F 2
camellia_f %f8,  %f52, %f54, %f52    !# Round 1   F 3
camellia_f %f10, %f54, %f52, %f54    !# Round 1   F 4
camellia_f %f12, %f52, %f54, %f52    !# Round 1   F 5
camellia_f %f14, %f54, %f52, %f54    !# Round 1   F 6
camellia_f1 %f16, %f54,       %f54    !# FL
.camellia_fli %f18, %f52,      %f52    !# FLI
camellia_f %f20, %f52, %f54, %f52    !# Round 2   F 1
camellia_f %f22, %f54, %f52, %f54    !# Round 2   F 2
camellia_f %f24, %f52, %f54, %f52    !# Round 2   F 3
camellia_f %f26, %f54, %f52, %f54    !# Round 2   F 4
camellia_f %f28, %f52, %f54, %f52    !# Round 2   F 5
camellia_f %f30, %f54, %f52, %f54    !# Round 2   F 6
camellia_f1 %f32, %f54,       %f54    !# FL
.camellia_fli %f34, %f52,      %f52    !# FLI
camellia_f %f36, %f52, %f54, %f52    !# Round 3   F 1
camellia_f %f38, %f54, %f52, %f54    !# Round 3   F 2
camellia_f %f40, %f52, %f54, %f52    !# Round 3   F 3
camellia_f %f42, %f54, %f52, %f54    !# Round 3   F 4
camellia_f %f44, %f52, %f54, %f52    !# Round 3   F 5
camellia_f %f46, %f54, %f52, %f54    !# Round 3   F 6
fxor       %f48, %f52,      %f52    !# Post-Whiten
fxor       %f50, %f54,       %f54
```

If CFR.camellia = 0, an attempt to execute a CAMELLIA instruction causes a compatibility_feature exception.

Software must check that CFR.camellia = 1 before executing any of these CAMELLIA instructions. If CFR.camellia = 0, then software should assume that an attempt to execute one of the CAMELLIA instruction either (1) will generate an illegal_instruction exception because it is not implemented in hardware, or (2) will execute, but perform some other operation.

Therefore, if CFR.camellia = 0, software should perform the corresponding CAMELLIA operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.

**Exceptions**  
fp_disabled

**See Also**  
Camellia Operations (3 Operand) on page 130
Camellia 3-operand Ops

7.15 Camellia Operations (3 Operand)

The Camellia instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

The Camellia instructions operate on 64-bit floating-point registers.

- **CAMELLIA_FL**: 
  
  \[ F_{\text{rd}}[63:0] \leftarrow \text{camellia}\_\text{FL}( \text{data}=F_{\text{rs2}}[63:0], \text{key}=F_{\text{rs1}}[63:0] ); \]

- **CAMELLIA_FLI**: 
  
  \[ F_{\text{rd}}[63:0] \leftarrow \text{camellia}\_\text{FLI}( \text{data}=F_{\text{rs2}}[63:0], \text{key}=F_{\text{rs1}}[63:0] ); \]

If CFR.camellia = 0, an attempt to execute a CAMELLIA instruction causes a *compatibility_feature* exception.

**Exceptions**  
*fp_disabled*

**See Also**  
Camellia Operations (4 operand) on page 128
7.16 Compare and Swap

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

The 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 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.

Exceptions. An attempt to execute a CASXA or CASA instruction when $i = 1$ and instruction bits 12:5 are nonzero causes an `illegal_instruction` exception.
Exceptions.. 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\text{16} to 7F\text{16}, CASXA and CASA cause a privileged_action exception.

Compatibility

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

Compare and Swap (CAS) and Compare and Swap Extended (CASX) synthetic instructions are available for “big endian” memory accesses. Compare and Swap Little (CASL) and Compare and Swap Extended Little (CASXL) synthetic instructions are available for “little endian” memory accesses. See Synthetic Instructions on page 536 for the syntax of these synthetic instructions.

The compare-and-swap instructions do not affect the condition codes.

The compare-and-swap instructions can be used with any of the following ASIs, subject to the privilege mode rules described for the privileged_action exception above. Use of any other ASI with these instructions causes a DAE_invalid_asi exception.

<table>
<thead>
<tr>
<th>ASIs valid for CASA and CASXA instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>04\text{16} ASI_NUCLEUS</td>
</tr>
<tr>
<td>0C\text{16} ASI_NUCLEUS_LITTLE</td>
</tr>
<tr>
<td>10\text{16} ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>18\text{16} ASI_AS_IF_USER_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>11\text{16} ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>19\text{16} ASI_AS_IF_USER_SECONDARY_LITTLE</td>
</tr>
<tr>
<td>14\text{16} ASI_REAL</td>
</tr>
<tr>
<td>1C\text{16} ASI_REAL_LITTLE</td>
</tr>
<tr>
<td>80\text{16} ASI_PRIMARY</td>
</tr>
<tr>
<td>88\text{16} ASI_PRIMARY_LITTLE</td>
</tr>
<tr>
<td>81\text{16} ASI_SECONDARY</td>
</tr>
<tr>
<td>89\text{16} ASI_SECONDARY_LITTLE</td>
</tr>
</tbody>
</table>

Exceptions

illegal_instruction
mem_address_not_aligned
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nc_page (attempted access to noncacheable page)
DAE_nfo_page (attempted access to non-faulting-only page)

See Also
CASA on page 131
LDSTUB on page 242
LDSTUBA on page 243
SWAP on page 343
SWAPA on page 344
## CBcond (Compare and Branch)

### 7.17 Compare and Branch

<table>
<thead>
<tr>
<th>Opcode cc2 c_hi c_lo Operation</th>
<th>Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>— 1000 Reserved (illegal_instruction)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>— 0000 Reserved (illegal_instruction)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**32-bit Compare and Branch Operations**

- **CWBNE** 0 1001 Compare and Branch if Not Equal not Z cwbne\(^\dagger\) reg\(_{rs1}\) reg_or_imm5, label C1
- **CWEB** 0 0001 Compare and Branch if Equal Z cweb\(^\dagger\) reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBG** 0 1010 Compare and Branch if Greater not (Z or (N xor V)) cwbg reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBLE** 0 0010 Compare and Branch if Less or Equal Z or (N xor V) cwble reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBG** 0 1011 Compare and Branch if Greater or Equal not (N xor V) cwbg reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBL** 0 0011 Compare and Branch if Less N xor V cwbl reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBGU** 0 1100 Compare and Branch if Greater Unsigned not (C or Z) cwbgu reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBLEU** 0 0100 Compare and Branch if Less or Equal Unsigned C or Z cwbleu reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBBCC** 0 1101 Compare and Branch if Carry Clear (Greater not C Than or Equal, Unsigned) cwbcc\(^\dagger\) reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBCS** 0 0101 Compare and Branch if Carry Set (Less Than, Unsigned) C cwbc\(^\dagger\) reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBPPOS** 0 1110 Compare and Branch if Positive not N cwbpos reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBNEG** 0 0110 Compare and Branch if Negative N cwbneg reg\(_{rs1}\) reg_or_imm5, label C1
- **CWBBVC** 0 1111 Compare and Branch if Overflow Clear not V cwbbvc reg\(_{rs1}\) reg_or_imm5, label C1
- **CBVBNS** 0 0111 Compare and Branch if Overflow Set V cbvns reg\(_{rs1}\) reg_or_imm5, label C1

**64-bit Compare and Branch Operations**

- **CXBE** 1 0001 Compare and Branch if Equal Z cxbe\(^\dagger\) reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBG** 1 1010 Compare and Branch if Greater not (Z or (N xor V)) cxbg reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBLE** 1 0010 Compare and Branch if Less or Equal Z or (N xor V) cxble reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBGE** 1 1011 Compare and Branch if Greater or Equal not (N xor V) cxbge reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBL** 1 0011 Compare and Branch if Less N xor V cxbl reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBGU** 1 1100 Compare and Branch if Greater Unsigned not (C or Z) cxbgu reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBLEU** 1 0100 Compare and Branch if Less or Equal Unsigned C or Z cxbleu reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBBCC** 1 1101 Compare and Branch if Carry Clear (Greater not C Than or Equal, Unsigned) cxbbcc\(^\dagger\) reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBSCS** 1 0101 Compare and Branch if Carry Set (Less Than, Unsigned) C cxbcs\(^\dagger\) reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBPOS** 1 1110 Compare and Branch if Positive not N cxbpos reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBNEG** 1 0110 Compare and Branch if Negative N cxbneg reg\(_{rs1}\) reg_or_imm5, label C1
- **CXBVC** 1 1111 Compare and Branch if Overflow Clear not V cxbvc reg\(_{rs1}\) reg_or_imm5, label C1
- **CBVBS** 1 0111 Compare and Branch if Overflow Set V cbvbs reg\(_{rs1}\) reg_or_imm5, label C1

\(^\dagger\) synonym: c<w|x>bnz  \(\dagger\) synonym: c<w|x>bnez  \(^\dagger\) synonym: c<w|x>bgeu  \(\dagger\) synonym: c<w|x>blu

| 00 c_hi \(\dagger\) c_lo 011 cc2 d10_hi rs1 i=0 d10_lo rs2 |
|-------------------|-------------------|-------------------|-------------------|
| 00 c_hi \(\dagger\) c_lo 011 cc2 d10_hi rs1 i=1 d10_lo simm5 |

\(\dagger\) Very early SPARC V9 implementations (circa 1995) ignored bit 28 and executed a BPr instruction when bits 31:30 = 002, bit 28 = 1, and bits 24:22 = 0112. Later implementations treated that as an illegal_instruction. Starting with OSA 2011, it is a CBcond instruction.
**CBcond (Compare and Branch)**

**Description**

The compare-and-branch instruction compares two integer values, producing a TRUE or FALSE result.

If the comparison evaluates to FALSE, no control transfer (branch) occurs.

If the comparison evaluates to TRUE, the instruction causes a PC-relative, non-delayed transfer of control (branch) to address “PC + (4 × sign_ext (d10_hi : : d10_lo))”.

Since the word displacement available in this instruction is 10 bits, a CBcond instruction can target a PC-relative range of –512 to +511 words (-2048 to +2044 bytes).

The first comparison operand is always R[rs1]. If i = 0, the second comparison operand is R[rs2]; if i = 1, the second comparison operand is sign_ext (simm5).

If cc2 = 0 (the versions of these instructions with “W” in their mnemonic), the comparison is performed between the least-significant 32 bits of the two source operands, using 32-bit arithmetic semantics. If cc2 = 1 (the versions of these instructions with “X” in their mnemonic), the comparison is performed between all 64 bits of the two source operands, using 64-bit arithmetic semantics.

The CBcond instruction does not change architecturally-visible condition codes (the contents of the CCR register). The comparison (“SUBcc-like”) component of CBcond produces temporary condition codes, which are evaluated by its branch component, then discarded.

**Programming Notes**

The Compare and Branch (CBcond) instructions operate similarly to an integer comparison (SUBcc), followed by a conditional branch (Bicc), with a few differences:

1. CBcond uses only temporary condition codes (CCR is not updated).
2. CBcond implements a non-delayed control transfer (while Bicc is a dCTI).
3. CBcond has a shorter branch range (displacement) than Bicc, and
4. CBcond can perform 32- or 64-bit comparisons, while Bicc only tests 32-bit condition codes (BPcc can also test both 32- and 64-bit condition codes).

Given those caveats, then if i = 0, a CBcond instruction conceptually operates similarly to:

```
subcc rs31r, rs32r, %g0   // subcc component; CCR not updated
bcond cc2, target        // branch component; non-delayed transfer
```

And if i = 1, a CBcond instruction conceptually operates similarly to:

```
subcc rs31r, simm5, %g0  // subcc component; CCR not updated
bcond cc2, target        // branch component; non-delayed transfer
```

**Programming Notes**

- Because CBcond is a non-delayed control transfer instruction, it has no delay slot and no annul bit.
- CBcond neither needs, nor has, a static branch prediction bit.
- It is possible to construct CBcond to produce a non-delayed Branch Always instruction, using CBcond with two zero-value operands and a branch-if-equal condition. When i = 0, rs1 = rs2, CWBE and CXBE are effectively non-delayed “Branch Always” instructions.
- One can compare register contents with zero and branch based on that result by using a CBcond instruction with i = 1 and (imm5 = 0).
CBcond (Compare and Branch)

Exceptions. An attempt to execute a CBcond instruction when \( c_{lo} = 000_2 \) causes an illegal_instruction exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the CBcond instruction will cause a transfer of control (taken conditional branch-and-branch operations), any sequence of CBcond’s should be separated by NOP(s), such as:

```
CBcond
nop
CBcond
nop
CBcond
...
```

Exceptions. An attempt to execute a CBcond instruction when \( c_{lo} = 000_2 \) causes an illegal_instruction exception.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the CBcond instruction will cause a transfer of control (taken conditional branch-and-branch operations), then CBcond generates a control_transfer_instruction exception instead of causing a control transfer. When a control_transfer_instruction trap occurs, PC (the address of the CBcond instruction) is stored in TPC[TL] and the value of NPC from before the CBcond was executed is stored in TNPC[TL].

Exceptions

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

Programming Note

For optimal performance, CBcond should not be the next sequential instruction executed after any other control transfer instruction (including another CBcond).

From the above, it follows that for optimal performance in switch statements (the code for which tends to contain a sequence of multiple compare-and-branch operations), any sequence of CBcond’s should be separated by NOP(s), such as:

```
CBcond
nop
CBcond
nop
CBcond
...
```

Implementation Note

The concatenation \( c_{hi} :: c_{lo} \) encodes identical conditions to the cond field in other conditional branch instructions.
### 7.18 CMASK

The CMASK instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

#### TABLE 7-7 Source Operand Effect on GSR.mask

<table>
<thead>
<tr>
<th>R[rs2] bit</th>
<th>Value</th>
<th>CMASK8</th>
<th>CMASK16</th>
<th>CMASK32</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[rs2][0]</td>
<td>0</td>
<td>GSR.mask[3:0] ← F16</td>
<td>GSR.mask[7:0] ← EF16</td>
<td>GSR.mask[15:0] ← CDEF16</td>
</tr>
<tr>
<td>R[rs2][4]</td>
<td>0</td>
<td>GSR.mask[19:16] ← B16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>GSR.mask[19:16] ← 316</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R[rs2][5]</td>
<td>0</td>
<td>GSR.mask[23:20] ← A16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>GSR.mask[23:20] ← 216</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R[rs2][6]</td>
<td>0</td>
<td>GSR.mask[27:24] ← 916</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>GSR.mask[27:24] ← 116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R[rs2][7]</td>
<td>0</td>
<td>GSR.mask[31:28] ← 816</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>GSR.mask[31:28] ← 016</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note**

GSR.mask is generated such that if the corresponding input bit of CMASK in R[rs2] is 0, a subsequent BSHUFFLE instruction will select the corresponding byte from FD[rs2]; if it is 1, a subsequent BSHUFFLE will select the corresponding byte from FD[rs1].
An attempt to execute a CMASK instruction when instruction bits 29:25 are nonzero or 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 a CMASK instruction causes an *fp_disabled* exception.

**Programming Note**

It is envisioned that these instructions will consume the result of a prior SIMD compare instruction and generate the mask field necessary to allow the BSHUFFLE instruction to perform a conditional move. For example:

```assembly
fcmpt <command>
```

See Also

- Byte Mask and Shuffle on page 122
- Partitioned Signed Compare on page 201
- FPCMPU on page 203

Exceptions

- *illegal_instruction*
- *fp_disabled*
7.19 CRC32C Operation (3 operand)

The CR32C instructions is new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

**Description**
Accumulate a cyclic redundancy check (CRC) value using the polynomial 11EDC6F41\textsubscript{16} for two 32-bit input data blocks. The specification for this polynomial can be found at: [http://www.faqs.org/rfcs/rfc3385.html](http://www.faqs.org/rfcs/rfc3385.html)

The number of CRC32C instructions needed for a message length M is equal to \((M \mod 8)\) if evenly divisible by 8, or \((M \mod 8) + 1\) otherwise.

Software assistance is needed calculate the CRC32 value for a message whose length is not evenly divisible by 8. For cases where the remainder is 5, 6, or 7, the CRC32C instruction can be used if the remaining bytes are zeroed. For cases where the remainder is 1, 2, 3, or 4, software must process the last block, as CRC32C will not deliver a useful result.

The CRC32C algorithm calls for an inversion of the result on the final message block. This final inversion should be added after the final block is processed. See the Programming Note below for an example.

The CRC32C instruction operates on source values read from 64-bit floating-point registers \(F_D[rs1]\) and \(F_D[rs2]\), as follows:

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV</td>
<td>(F_D[rs1][31:0])</td>
</tr>
<tr>
<td>data1</td>
<td>(F_D[rs2][63:32])</td>
</tr>
<tr>
<td>data2</td>
<td>(F_D[rs2][31:0])</td>
</tr>
</tbody>
</table>

\[
\text{CRC32C: } \begin{align*}
\text{result}_1[31:0] & \leftarrow \text{CRC32c}(\text{IV}, \text{data1}) \\
\text{F}_D[\text{rd}][31:0] & \leftarrow \text{CRC32c}(\text{result}_1, \text{data2}) \\
\text{F}_D[\text{rd}][63:32] & \leftarrow 0000\ 0000_{16}
\end{align*}
\]
If CFR.crc32c = 0, an attempt to execute a CRC32C instruction causes a compatibility_feature exception.

Exceptions fp_disabled
7.20 DES CryptographicOperations(4-operand)

The DES instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op5</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_ROUND</td>
<td>1001</td>
<td>two DES round operations</td>
<td>des_round $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$</td>
<td>N1</td>
</tr>
</tbody>
</table>

### Description

The DES instructions support encryption and decryption for the Data Encryption Standard. This standard is available as FIPS-46 on "http://nist.gov". Encryption and decryption start by applying Initial Permutation function (DES_IP) to the text. Encryption then loops on the DES round (DES_ROUND) applying the expanded keys in the forward direction. Decryption loops on the DES round applying the expanded keys in reverse order. Encryption and decryption end by applying the Inverse Initial Permutation function (DES_IIP). PC-2 for both keys is included in DES_ROUND. The DES instructions operate on 64-bit floating-point registers.

#### DES_ROUND:

- Perform two rounds of the DES algorithm.
- $F_D[rs1]$ is the expanded key for the first round.
- $F_D[rs2]$ is the expanded key for the second round.
- $F_D[rs3]$ is the text for the first round.

#### Programming Note

The DES instructions are components of the overall DES algorithm. To perform an encryption or decryption, software must first expand the key(s). Key expansion is done only once per session key. Prior to processing a DES block using these instructions, the DES key must go thru PC-1. After that, the PC-1 output is used to create 16 different keys using the shifts described by the DES algorithm. These 16 keys are then applied in pairs to the DES_ROUND instructions.

Note that these 16 keys are not equivalent to “k1” thru “k16” as described in the spec until PC-2 is applied as part of the DES_ROUND instruction. The expanded keys are then applied to all blocks for that session.
For each block, the following instruction sequence can be used to encrypt or decrypt. Decryption applies the expanded keys in reverse order from encryption. For the following example, the expanded keys are stored in \( F_D[0] \) thru \( F_D[30] \) and text is in \( F_D[32] \):

\[
\begin{align*}
\text{des}_i\text{p} & \quad %f32, \quad %f32 \\
\text{des}\_\text{round} & \quad %f0, \quad %f2, \quad %f32, \quad %f32 \\
\text{des}\_\text{round} & \quad %f4, \quad %f6, \quad %f32, \quad %f32 \\
\text{des}\_\text{round} & \quad %f8, \quad %f10, \quad %f32, \quad %f32 \\
\text{des}\_\text{round} & \quad %f12, \quad %f14, \quad %f32, \quad %f32 \\
\text{des}\_\text{round} & \quad %f16, \quad %f18, \quad %f32, \quad %f32 \\
\text{des}\_\text{round} & \quad %f20, \quad %f22, \quad %f32, \quad %f32 \\
\text{des}\_\text{round} & \quad %f24, \quad %f26, \quad %f32, \quad %f32 \\
\text{des}\_\text{round} & \quad %f28, \quad %f30, \quad %f32, \quad %f32 \\
\text{des}_i\text{ip} & \quad %f32, \quad %f32
\end{align*}
\]

If CFR.des = 0, an attempt to execute a DES instruction causes a compatibility_feature exception.

**Programming Note** Software must check that CFR.des = 1 before executing any of these DES instructions. If CFR.des = 0, then software should assume that an attempt to execute one of the DES instruction either (1) will generate an illegal_instruction exception because it is not implemented in hardware, or (2) will execute, but perform some other operation. Therefore, if CFR.des = 0, software should perform the corresponding DES operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.

**Exceptions**  *fp_disabled*

**See Also** DES Cryptographic Operations (2 operand) on page 142
The DES instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### Instruction Table

<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>DES_IP</td>
<td>1 0011 0100</td>
<td>DES Initial Permutation</td>
<td>des_ip freg&lt;rs1&gt;, freg&lt;rd&gt;</td>
<td>N1</td>
</tr>
<tr>
<td>DES_IIP</td>
<td>1 0011 0101</td>
<td>DES Inverse Initial Permutation</td>
<td>des_iip freg&lt;rs1&gt;, freg&lt;rd&gt;</td>
<td>N1</td>
</tr>
<tr>
<td>DES_KEXPAND</td>
<td>1 0011 0110</td>
<td>DES Key Expand</td>
<td>des_kexpand freg&lt;rs1&gt;, imm5, freg&lt;rd&gt;</td>
<td>N1</td>
</tr>
</tbody>
</table>

### Description

The DES instructions operate on 64-bit floating-point registers.

**DES_IP**

Performs the Initial Permutation function on \(F_D[rs1]\).

\(imm5 = 000002\); other values of \(imm5\) cause an *illegal_instruction* exception.

**DES_IIP**

Performs a 32-bit swap and then the Inverse Initial Permutation function on \(F_D[rs1]\).

\(imm5 = 000002\); other values of \(imm5\) cause an *illegal_instruction* exception.

**DES_KEXPAND**

if \((imm5 = 000002)\)

then

\[\begin{align*}
\text{pc1}[55:0] & \leftarrow \text{des_pc1}( F_D[rs1][63:0] ); \\
F_D[rd][63:56] & \leftarrow 0000 0000_{16}; \\
F_D[rd][55:0] & \leftarrow ( \text{pc1}[54:28] :: \text{pc1}[55] :: \text{pc1}[26:0] :: \text{pc1}[27] ) \nonumber
\end{align*}\]

else if \((imm5 = 000012)\)

then

\[\begin{align*}
F_D[rd][63:56] & \leftarrow 0000 0000_{16}; \\
F_D[rd][55:0] & \leftarrow ( F_D[rs1][54:28] :: F_D[rs1][55] :: F_D[rs1][26:0] :: F_D[rs1][27] ) \nonumber
\end{align*}\]

else if \((imm5 = 000102)\)

then

\[\begin{align*}
F_D[rd][63:56] & \leftarrow 0000 0000_{16}; \\
F_D[rd][55:0] & \leftarrow ( F_D[rs1][53:28] :: F_D[rs1][55:54] :: F_D[rs1][25:0] :: F_D[rs1][27:26] ) \nonumber
\end{align*}\]

else if \((imm5 = 000112)\)

then

\[\begin{align*}
F_D[rd][63:56] & \leftarrow 0000 0000_{16}; \\
F_D[rd][55:0] & \leftarrow ( F_D[rs1][51:28] :: F_D[rs1][55:52] :: F_D[rs1][23:0] :: F_D[rs1][27:24] ) \nonumber
\end{align*}\]

else

*illegal_instruction* exception

endif

### Programming Note

The DES instructions are components of the overall DES algorithm. The DES_KEXPAND instruction can be used as an alternative to software key expansion. For each session, the following instruction sequence can be used to expand the key.
If CFR.des = 0, an attempt to execute a DES instruction causes a *compatibility_feature* exception.

Software must check that CFR.des = 1 before executing any of these DES instructions. If CFR.des = 0, then software should assume that an attempt to execute one of the DES instruction either

1. will generate an *illegal_instruction* exception because it is not implemented in hardware, or
2. will execute, but perform some other operation.

Therefore, if CFR.des = 0, software should perform the corresponding DES operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.

**Exceptions**  
*fp_disabled*

**See Also**  
DES Cryptographic Operations (4 operand) on page 140
7.22 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>DONEP</td>
<td>11 1110</td>
<td>Return from Trap (skip trapped instruction)</td>
<td>done</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

The DONE instruction restores the saved state from TSTATE[TL] (GL, CCR, ASI, PSTATE, and CWP), sets PC and NPC, and decrements TL. DONE sets PC ← TNPC[TL] and NPC ← TNPC[TL] + 4 (normally, the value of NPC saved at the time of the original trap and address of the instruction immediately after the one referenced by the NPC).

**Programming Notes**

The DONE and RETRY instructions are used to return from privileged trap handlers.

Unlike RETRY, DONE ignores the contents of TPC[TL].

If the saved TNPC[TL] was not altered by trap handler software, DONE causes execution to resume immediately after the instruction that originally caused the trap (as if that instruction was “done” executing).

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

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

Note that since PSTATE.tct is automatically set to 0 during entry to a trap handler, execution of a DONE instruction at the end of a trap handler will not cause a control_transfer_instruction exception unless trap handler software has explicitly set PSTATE.tct to 1. During execution of the DONE instruction, the value of PSTATE.tct is restored from TSTATE.

**Programming Notes**

If control_transfer_instruction traps are to be re-enabled (PSTATE.tct ← 1, restored from TSTATE[TL].pstate.tct) when trap handler software for the control_transfer_instruction trap returns, the trap handler must:

1. Emulate the trapped CTI, setting TPC[TL] and TNPC[TL] appropriately, remembering to compensate for annul bits) and
2. Use a DONE (not RETRY) instruction to return.

If the CTI that caused the control_transfer_instruction trap was a DONE (RETRY) instruction, the trap handler must carefully emulate the trapped DONE (RETRY) (decrementing TL may suffice) before the trap handler returns using its own DONE (RETRY) instruction.

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system.

**IMPL. DEP. #417-S10:** If (1) TSTATE[TL].pstate.am = 1 and (2) a DONE instruction is executed (which sets PSTATE.am to ‘1’ by restoring the value from TSTATE[TL].pstate.am to PSTATE.am), it is implementation dependent whether the DONE instruction masks (zeroes) the more-significant 32 bits of the values it places into PC and NPC.
DONE

**Exceptions.** In privileged mode (PSTATE.priv = 1), an attempt to execute DONE while TL = 0 causes an `illegal_instruction` exception. An attempt to execute DONE (in any mode) with instruction bits 18:0 nonzero causes an `illegal_instruction` exception.

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

**Implementation Note** In nonprivileged mode, `illegal_instruction` exception due to TL = 0 does not occur. The `privileged_opcode` exception occurs instead, regardless of the current trap level (TL).

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

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

**See Also** RETRY on page 303
7.23 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>0 0000 0000</td>
<td>Eight 8-bit edge boundary processing</td>
<td>edge8cc rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE8Lcc</td>
<td>0 0000 0010</td>
<td>Eight 8-bit edge boundary processing, little-endian</td>
<td>edge8lcc rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE16cc</td>
<td>0 0000 0100</td>
<td>Four 16-bit edge boundary processing</td>
<td>edge16cc rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE16Lcc</td>
<td>0 0000 0110</td>
<td>Four 16-bit edge boundary processing, little-endian</td>
<td>edge16lcc rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE32cc</td>
<td>0 0000 1000</td>
<td>Two 32-bit edge boundary processing</td>
<td>edge32cc rs1, rs2, rd</td>
<td>B1</td>
</tr>
<tr>
<td>EDGE32Lcc</td>
<td>0 0000 1010</td>
<td>Two 32-bit edge boundary processing, little-endian</td>
<td>edge32lcc rs1, rs2, rd</td>
<td>B1</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”.

**Description**

EDGE<8|16|32>cc, EDGE16[L]cc, and EDGE32[L]cc operate identically to EDGE<8|16|32>N, EDGE16[L]N, and EDGE32[L]N, respectively, but also set the integer condition codes (CCR.xcc and CCR.icc).

The integer condition codes are set by these instructions the same as a SUBcc instruction with the same operands (see *Subtract* on page 303).

**Exceptions**

None

**See Also**

EDGE<8|16|32>[L]N on page 147
7.24 Edge Handling Instructions (no CC)

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.

EDGE8LN, EDGE16LN, and EDGE32LN are little-endian versions of EDGE8N, EDGE16N, and EDGE32N, respectively. They produce an edge mask that is bit-reversed from their big-endian counterparts but are otherwise identical. This makes the mask consistent with the mask produced by the \textit{Partitioned Unsigned Compare} on page 203 and consumed by the Partial Store instruction (see \textit{DAE_invalid_asistore Partial Floating-Point} on page 332) for 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 \( R[rs1] \) (see TABLE 7-8) and the right edge mask is computed from the 3 least significant bits of \( R[rs2] \) (see TABLE 7-9).
2. If 32-bit address masking is disabled (PSTATE.am = 0) (that is, 64-bit addressing is in use) and the most significant 61 bits of \( R[rs1] \) are equal to the corresponding bits in \( R[rs2] \), \( R[rd] \) is set to the right edge mask and \textit{anded} with the left edge mask.
3. If 32-bit address masking is enabled (PSTATE.am = 1) (that is, 32-bit addressing is in use) and bits 31:3 of \( R[rs1] \) match bits 31:3 of \( R[rs2] \), \( R[rd] \) is set to the right edge mask and \textit{anded} with the left edge mask.
4. Otherwise, \( R[rd] \) is set to the left edge mask.

TABLE 7-8 and TABLE 7-9 list edge mask specifications.

### TABLE 7-8 Left Edge Mask Specification

<table>
<thead>
<tr>
<th>Edge Size</th>
<th>R[rs1] (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>1111 1111</td>
</tr>
<tr>
<td>8</td>
<td>001</td>
<td>0111 1111</td>
<td>1111 1110</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>0011 1111</td>
<td>1111 1100</td>
</tr>
<tr>
<td>8</td>
<td>011</td>
<td>0001 1111</td>
<td>1111 1000</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>0000 1111</td>
<td>1111 0000</td>
</tr>
<tr>
<td>8</td>
<td>101</td>
<td>0000 0111</td>
<td>1110 0000</td>
</tr>
<tr>
<td>8</td>
<td>110</td>
<td>0000 0011</td>
<td>1100 0000</td>
</tr>
</tbody>
</table>
TABLE 7-9 Right Edge Mask Specification

<table>
<thead>
<tr>
<th>Edge Size</th>
<th>R[rs2] (2:0)</th>
<th>Right Edge</th>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>000</td>
<td>1000</td>
<td>0000 0001</td>
<td>0000 0001</td>
</tr>
<tr>
<td>8</td>
<td>001</td>
<td>1100</td>
<td>0000 0011</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>1110</td>
<td>0000 0111</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>011</td>
<td>1111</td>
<td>0000 1111</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>1111 1000</td>
<td>0001 1111</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>101</td>
<td>1111 1100</td>
<td>0011 1111</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>110</td>
<td>1111 1110</td>
<td>0111 1111</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>111</td>
<td>1111 1111</td>
<td>1111 1111</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>00x</td>
<td>1100</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>01x</td>
<td>1110</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>10x</td>
<td>1111</td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0xx</td>
<td>10</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1xx</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Exceptions
None

See Also
EDGE<8,16,32>[L]cc on page 146
7.25 Floating-Point Absolute Value

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABSS</td>
<td>11 0100</td>
<td>0 0000 1001</td>
<td>Absolute Value Single</td>
<td>fabss freg2, freg3</td>
<td>A1</td>
</tr>
<tr>
<td>FABSD</td>
<td>11 0100</td>
<td>0 0000 1010</td>
<td>Absolute Value Double</td>
<td>fabsd freg2, freg3</td>
<td>A1</td>
</tr>
<tr>
<td>FABSQ</td>
<td>11 0100</td>
<td>0 0000 1011</td>
<td>Absolute Value Quad</td>
<td>fabsq freg2, freg3</td>
<td>C3</td>
</tr>
</tbody>
</table>

**Description**

FABS copies the source floating-point register(s) to the destination floating-point register(s), with the sign bit cleared (set to 0).

FABSS operates on single-precision (32-bit) floating-point registers, FABSD operates on double-precision (64-bit) floating-point register pairs, and FABSQ operates on quad-precision (128-bit) floating-point register quadruples.

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

**Note**

Oracle SPARC Architecture 2011 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.ief = 0) or if no FPU is present, an attempt to execute an FABS instruction causes an fp_disabled exception.

An attempt to execute an FABSQ instruction when rs2[1] ≠ 0 or rd[1] ≠ 0 causes an fp_exception_other (FSR.ftt = invalid_fp_register (FABSQ only)) exception.

**Exceptions**

illegal_instruction
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FABSQ only))
### 7.26 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 `FSR.rd`.

**Note** Oracle SPARC Architecture 2011 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FADDq instruction causes an `illegal_instruction` exception, allowing privileged software to emulate the instruction.

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

An attempt to execute an FADDq instruction when (`rs1[1] ≠ 0`) or (`rs2[1] ≠ 0`) or (`rd[1:0] ≠ 0`) causes an `fp_exception_other` (`FSR.ftt = invalid_fp_register`) exception.

**Note** An `fp_exception_other` with `FSR.ftt = unfinished_FPop` can occur if the operation detects unusual, implementation-specific conditions.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011*.

#### Exceptions

- `illegal_instruction`
- `fp_disabled`
- `fp_exception_other` (`FSR.ftt = invalid_fp_register` (FADDq only))
- `fp_exception_other` (`FSR.ftt = unfinished_FPop`)
- `fp_exception_ieee_754` (`OF, UF, NX, NV`)

#### See Also

FMAf on page 170
FALIGNDATAg

7.27 Align Data (using GSR.align) [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>FALIGNDATAg</td>
<td>0 0100 1000</td>
<td>Perform data alignment for misaligned data, using alignment indicated in GSR.align</td>
<td>faligndata freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
</tbody>
</table>

† The original architectural name for this instruction was “FALIGNDATA”; the “g” was later appended for clarity.

Description FALIGNDATAg concatenates two 64-bit floating-point registers F_D[rs1] and F_D[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, F_D[rd]. GSR.align specifies the number of the most significant byte to extract (and, therefore, the least significant byte extracted is numbered GSR.align+7).

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

GSR.align

F_D[rs1]::F_D[rs2]

GSR.align 101

F_D[rd]

FIGURE 7-6 FALIGNDATAg

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

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

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

Exceptions fp_disabled

See Also Align Address on page 114
### 7.28 Branch on Floating-Point Condition Codes (FBfcc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>cond</th>
<th>Operation</th>
<th>fcc Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBA&lt;sup&gt;0&lt;/sup&gt;</td>
<td>1000</td>
<td>Branch Always</td>
<td>1</td>
<td>fba[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBN&lt;sup&gt;0&lt;/sup&gt;</td>
<td>0000</td>
<td>Branch Never</td>
<td>0</td>
<td>fbn[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBU&lt;sup&gt;0&lt;/sup&gt;</td>
<td>0111</td>
<td>Branch on Unordered</td>
<td>U</td>
<td>fbu[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBG&lt;sup&gt;0&lt;/sup&gt;</td>
<td>0110</td>
<td>Branch on Greater</td>
<td>G</td>
<td>fbg[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUG&lt;sup&gt;0&lt;/sup&gt;</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&lt;sup&gt;0&lt;/sup&gt;</td>
<td>0100</td>
<td>Branch on Less</td>
<td>L</td>
<td>fbl[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBU&lt;sup&gt;0&lt;/sup&gt;</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>FBLC&lt;sup&gt;0&lt;/sup&gt;</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>FBNE&lt;sup&gt;0&lt;/sup&gt;</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&lt;sup&gt;0&lt;/sup&gt;</td>
<td>1001</td>
<td>Branch on Equal</td>
<td>E</td>
<td>fbe[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBU&lt;sup&gt;0&lt;/sup&gt;</td>
<td>1010</td>
<td>Branch on Unordered or Equal</td>
<td>E or U</td>
<td>fbuie[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBGE&lt;sup&gt;0&lt;/sup&gt;</td>
<td>1011</td>
<td>Branch on Greater or Equal</td>
<td>E or G</td>
<td>fbg[e]a label</td>
<td>A1</td>
</tr>
<tr>
<td>FBUGE&lt;sup&gt;0&lt;/sup&gt;</td>
<td>1100</td>
<td>Branch on Unordered or Greater or Equal</td>
<td>E or G or U</td>
<td>fbuge[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBE&lt;sup&gt;0&lt;/sup&gt;</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>FBOE&lt;sup&gt;0&lt;/sup&gt;</td>
<td>1110</td>
<td>Branch on Unordered or Less or Equal</td>
<td>E or L or G</td>
<td>fbole[,a] label</td>
<td>A1</td>
</tr>
<tr>
<td>FBO&lt;sup&gt;0&lt;/sup&gt;</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: fnz
‡ synonym: fbz

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

### Description
Unconditional and Fcc branches are described below:

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

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

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

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

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

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

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

If the Trap on Control Transfer feature is implemented (impl dep. #450-S20), STATE.tct = 1, and the FBfcc instruction will cause a transfer of control (FBA or taken conditional branch), then FBfcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the FBfcc instruction) is stored in TPC[TL] and the value of NPC from before the FBfcc was executed is stored in TNPC[TL]. Note that FBN never causes a *control_transfer_instruction* exception.

**Exceptions**

*fp_disabled*

*control_transfer_instruction* (impl. dep. #450-S20)
### 7.29 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>fb$a, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPN</td>
<td>0000</td>
<td>Branch Never</td>
<td>0</td>
<td>fbn, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPU</td>
<td>0111</td>
<td>Branch on Unordered</td>
<td>U</td>
<td>fbu, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPG</td>
<td>0110</td>
<td>Branch on Greater</td>
<td>G</td>
<td>fb$g, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPG</td>
<td>0101</td>
<td>Branch on Unordered or Greater</td>
<td>G</td>
<td>fb$u, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPL</td>
<td>0100</td>
<td>Branch on Less</td>
<td>L</td>
<td>fb$l, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPU</td>
<td>0011</td>
<td>Branch on Unordered or Less</td>
<td>L</td>
<td>fb$u, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPLG</td>
<td>0010</td>
<td>Branch on Less or Greater</td>
<td>L</td>
<td>fb$l, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPN</td>
<td>0001</td>
<td>Branch on Not Equal</td>
<td>L or G</td>
<td>fbn, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPE</td>
<td>1001</td>
<td>Branch on Equal</td>
<td>L or G</td>
<td>fb$e, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPU</td>
<td>1010</td>
<td>Branch on Unordered or Equal</td>
<td>L or G</td>
<td>fb$u, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPU</td>
<td>1011</td>
<td>Branch on Greater or Equal</td>
<td>G</td>
<td>fb$e, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPU</td>
<td>1100</td>
<td>Branch on Unordered or Greater or Equal</td>
<td>G</td>
<td>fb$u, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPL</td>
<td>1101</td>
<td>Branch on Less or Equal</td>
<td>E</td>
<td>fb$l, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPU</td>
<td>1110</td>
<td>Branch on Unordered or Less or Equal</td>
<td>E</td>
<td>fb$e, [a], [p]</td>
<td>A1</td>
</tr>
<tr>
<td>FBPO</td>
<td>1111</td>
<td>Branch on Ordered</td>
<td>E or L</td>
<td>fb$o, [a], [p]</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonym: fnz  ‡ synonym: fbz

### Programming Note

To set the annul (a) bit for FBPfcc instructions, append “, a” to the opcode mnemonic. For example, use “fb$l, a %fcc3, label”. In the preceding table, braces signify that the “, a” is optional. To set the branch prediction bit, append either “, pt” (for predict taken) or “, pn” (for predict not taken) to the opcode mnemonic. If neither “, pt” nor “, pn” is specified, the assembler defaults to “, pt”. To select the appropriate floating-point condition code, include “%fcc0”, “%fcc1”, “%fcc2”, or “%fcc3” before the label.

### Description

Unconditional branches and Fcc-conditional branches are described below.
FBPfcc

- **Unconditional branches (FBPA, FBPN)** — If its annul field is 0, an FBPN (Floating-Point Branch Never with Prediction) instruction acts like a NOP. If the Branch Never's annul field is 0, the following (delay) instruction is executed; if the annul (a) bit is 1, the following instruction is annulled (not executed). In no case does an FBPN cause a transfer of control to take place.

FBPA (Floating-Point Branch Always with Prediction) causes an unconditional PC-relative, delayed control transfer to the address “PC + (4 × sign_ext (disp19))”. If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul (a) bit is 0, the delay instruction is executed.

- **Fcc-conditional branches** — Conditional FBPfcc instructions (except FBPA and FBPN) evaluate one of the four floating-point condition codes (fcc0, fcc1, fcc2, fcc3) as selected by cc0 and cc1, according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address “PC + (4 × sign_ext (disp19))”. If FALSE, the branch is not taken.

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

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

The predict bit (p) gives the hardware a hint about whether the branch is expected to be taken. A 1 in the p bit indicates that the branch is expected to be taken. A 0 indicates that the branch is expected not to be taken.

Annulment, delay instructions, and delayed control transfers are described further in Chapter 6, *Instruction Set Overview*.

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

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the FBPfcc instruction will cause a transfer of control (FBPA or taken conditional branch), then FBPfcc generates a control_transfer_instruction exception instead of causing a control transfer. When a control_transfer_instruction trap occurs, PC (the address of the FBPfcc instruction) is stored in TPC[TL] and the value of NPC from before the FBPfcc was executed is stored in TNPC[TL]. Note that FBPN never causes a control_transfer_instruction exception.

*Exceptions* | fp_disabled  
| control_transfer_instruction (impl. dep. #450-S20)
7.30 Checksum

The Checksum instruction is new and is not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCHKSM16</td>
<td>0 100 0100</td>
<td>16-bit partitioned checksum</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fchksm16 freg[rs1], freg[rs2], freg[rd]</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

**Description**
This instruction performs four 16-bit additions between the corresponding integer values contained in its 64-bit source operands (\(F_D[rs1], F_D[rs2]\)). The carry out of each addition is added to the least significant bit of each intermediate sum to produce the final sum.

**Programming Note**
This is useful for networking applications requiring 16-bit TCP/UDP checksums.

![Diagram of FCHKSM16 Operation](image)

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FCHKSUM16 instruction causes an *fp_disabled* exception.

**Exceptions**
*fp_disabled*
### 7.31 Floating-Point Compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMPs</td>
<td>0101 0001</td>
<td>Compare Single</td>
<td>fcmps %fcc, freg&lt;rs1&gt;, freg&lt;rs2&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FCMPd</td>
<td>0101 0010</td>
<td>Compare Double</td>
<td>fcmpd %fcc, freg&lt;rs1&gt;, freg&lt;rs2&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FCMPq</td>
<td>0101 0011</td>
<td>Compare Quad</td>
<td>fcmpeq %fcc, freg&lt;rs1&gt;, freg&lt;rs2&gt;</td>
<td>C3</td>
</tr>
<tr>
<td>FCMPEs</td>
<td>0101 0101</td>
<td>Compare Single and Exception if Unordered</td>
<td>fcmpes %fcc, freg&lt;rs1&gt;, freg&lt;rs2&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FCMPEd</td>
<td>0101 0110</td>
<td>Compare Double and Exception if Unordered</td>
<td>fcmped %fcc, freg&lt;rs1&gt;, freg&lt;rs2&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FCMPEq</td>
<td>0101 0111</td>
<td>Compare Quad and Exception if Unordered</td>
<td>fcmped %fcc, freg&lt;rs1&gt;, freg&lt;rs2&gt;</td>
<td>C3</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>

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

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

The “?” in the preceding table means that the compared values are unordered. The unordered condition occurs when one or both of the operands to the comparison is a signalling or quiet NaN.

The “compare and cause exception if unordered” (FCMPEs, FCMPEd, and FCMPEq) instructions cause an invalid (NV) exception if either operand is a NaN.
FCMP<s|d|q>, FCMPE<s|d|q>

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

**V8 Compatibility Note**

Unlike the SPARC V8 architecture, SPARC V9 and the Oracle SPARC Architecture do not require an instruction between a floating-point compare operation and a floating-point branch (FBfcc, FBPfcc).

SPARC V8 floating-point compare instructions are required to have rd = 0. In SPARC V9 and the Oracle SPARC Architecture, bits 26 and 25 of the instruction (rd[1:0]) specify the floating-point condition code to be set. Legal SPARC V8 code will work on SPARC V9 and the Oracle SPARC Architecture because the zeroes in the R[rd] field are interpreted as fcc0 and the FBfcc instruction branches based on the value of fcc0.

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

**Note**

Oracle SPARC Architecture 2011 processors do not implement in hardware the instructions that refer to quad-precision floating-point registers. An attempt to execute FCMPq or FCMPEq generates an **illegal_instruction** exception, which causes a trap, allowing privileged software to emulate the instruction.

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

An attempt to execute an FCMPq or FCMPEq instruction when (rs1[1] ≠ 0) or (rs2[1] ≠ 0) causes an **fp_exception_other** (FSR.ftt = invalid_fp_register) exception.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011*.

**Exceptions**

- **illegal_instruction**
- **fp_disabled**
- **fp_exception_ieee_754** (NV)
- **fp_exception_other** (FSR.ftt = invalid_fp_register (FCMPq, FCMPEq only))

**See Also**

Partitioned Unsigned Compare on page 203
7.32 Floating-Point Divide

### Description
The floating-point divide instructions divide the contents of the floating-point register(s) specified by the rs1 field by the contents of the floating-point register(s) specified by the rs2 field. The instructions then write the quotient into the floating-point register(s) specified by the rd field.

Rounding is performed as specified by FSR.rd.

#### Note
Oracle SPARC Architecture 2011 processors do not implement in hardware the instructions that refer to quad-precision floating-point registers. An attempt to execute an FDIVq instruction generates an illegal_instruction exception, allowing privileged software to emulate the instruction.

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

An attempt to execute an FADDq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) causes an fp_exception_other (FSR.ftt = invalid_fp_register) exception.

#### Note
For FDIVs and FDIVd, an fp_exception_other with FSR.ftt = unfinished_FPpop can occur if the divide unit detects unusual, implementation-specific conditions.

For more details regarding floating-point exceptions, see Chapter 8, IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011.

### Exceptions
- **illegal_instruction**
- **fp_disabled**
- **fp_exception_other** (FSR.ftt = invalid_fp_register (FDIVq only))
- **fp_exception_other** (FSR.ftt = unfinished_FPpop (FDIVs, FDIV))
- **fp_exception_ieee_754** (OF, UF, DZ, NV, NX)
FEXPAND

7.33  FEXPAND

<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</td>
<td>0 1101</td>
<td>—</td>
<td>f32</td>
<td>f64</td>
<td>fexpand</td>
<td>freg</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 7-10** FEXPAND Operation

This operation is carried out as follows:

1. Left-shift each 8-bit value by 4 and zero-extend that result on the left to a 16-bit value.
2. Store the result in the destination register, $F_D[rd]$.

**Programming Note**

FEXPAND performs the inverse of the FPACK16 operation.

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

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

**Exceptions**

illegal_instruction  
fp_disabled

**See Also**

or $PSTATE.cef = 0$FPMERGE on page 206  
FPACK on page 192
7.34 Floating-point Add and Halve

FHADD<s|d>

### Description

The FHADD<s|d> instructions are used to perform an addition of two floating-point values and halve the result (divide it by 2) in a single operation. One benefit of this operation is that it cannot produce an arithmetic overflow.

### Exceptions

- `fp_disabled`
- `fp_exception_ieee_754` (UF, NX, NV)

### See Also

- FHSUB on page 162
- FNHADD on page 189
7.35 Floating-point Subtract and Halve

- **Description**: The FHSUB<s|d> instructions are used to take the difference of two floating-point values and halve the result (divide it by 2) in a single operation. One benefit of this operation is that it cannot produce an arithmetic overflow.

- **Exceptions**: `fp_disabled` and `fp_exception_ieee_754` (UF, NX, NV)

- **See Also**: FHADD on page 161, FNHADD on page 189
7.36 Convert 32-bit Integer to Floating Point

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

**Description**

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

The value of $FSR.rd$ determines how rounding is performed by FiTOs.

**Note**

Oracle SPARC Architecture 2011 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 ($FPRSxef = 0$ or $PSTATE.pef = 0$) or if no FPU is present, an attempt to execute an FiTO<s|d|q> instruction causes an *fp_disabled* exception.

An attempt to execute an FiTOq instruction when $rd[1] \neq 0$ causes an *fp_exception_other* ($FSR.ftt = invalid_fp_register$ (FiTOq)) exception.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011*.

**Exceptions**

*illegal_instruction*

*fp_disabled*

*fp_exception_other* ($FSR.ftt = invalid_fp_register$ (FiTOq))

*fp_exception_ieee_754* (NX (FiTOs only))
7.37 Floating-Point Lexicographic Compare

The FLCMP instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<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>FLCMPs</td>
<td>1010 0001</td>
<td>Single-precision lexicographic f32 f32</td>
<td>FSR.fcc</td>
<td>flcmps</td>
<td>%fccn, fregRs1, fregRs2</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>FLCMPd</td>
<td>1010 0010</td>
<td>Double-precision lexicographic compare</td>
<td>f64 f64</td>
<td>flcmpd</td>
<td>%fccn, fregRs1, fregRs2</td>
<td>C1</td>
<td></td>
</tr>
</tbody>
</table>

### Description

Lexicographic comparisons differ from standard floating-point comparisons in that −0 and +0 are treated as distinct values, with −0 comparing as less than +0. The handling of NaN operands and the encoding of the comparison results in the fcc bits is also different.

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) according to the following table:

<table>
<thead>
<tr>
<th>Relation</th>
<th>Value Written to fccn</th>
</tr>
</thead>
<tbody>
<tr>
<td>F[rs1] ≥ F[rs2], neither operand is NaN</td>
<td>0</td>
</tr>
<tr>
<td>F[rs1] &lt; F[rs2], neither operand is NaN</td>
<td>1</td>
</tr>
<tr>
<td>F[rs1] is NaN, F[rs2] is not NaN</td>
<td>2</td>
</tr>
<tr>
<td>F[rs2] is NaN, regardless of F[rs1]</td>
<td>3</td>
</tr>
</tbody>
</table>

### Programming Note

A lexicographic compare instruction is not an FPop, so it:
1) leaves all bits of FSR.cexc and FSR.aexc unchanged
2) never generates an fp_exception_other exception

### Programming Note

The encoding for the results of FP lexicographic compares (written to fccn) differ from those for regular SPARC V9 floating-point compares. This implies that a move or branch instruction dependent on a floating-point lexicographic compare might function differently from what its instruction mnemonic implies.
TABLE 7-10  Effect of Fcc Result Encoding of Floating-Point Lexicographic Compares on Dependent Conditional Moves/Branches

<table>
<thead>
<tr>
<th>Relation Specified in Branch Opcode</th>
<th>Actual Functionality When fcc Source for Branch/move was Set by FLCMPs or FLCMPd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>Always</td>
</tr>
<tr>
<td>Never</td>
<td>Never</td>
</tr>
<tr>
<td>Unordered</td>
<td>rs2 source operand of FLCMP was NaN</td>
</tr>
<tr>
<td>Greater</td>
<td>rs1 source operand of FLCMP was NaN</td>
</tr>
<tr>
<td></td>
<td>rs2 source operand of FLCMP was not NaN</td>
</tr>
<tr>
<td>Unordered or Greater</td>
<td>One or both the source operands rs1,rs2 of FLCMP was NaN</td>
</tr>
<tr>
<td>Less</td>
<td>Less</td>
</tr>
<tr>
<td>Unordered or Less</td>
<td>Less or rs2 source operand of FLCMP was NaN</td>
</tr>
<tr>
<td>Less or Greater</td>
<td>Less or (rs1 source operand of FLCMP was NaN and rs2 source operand was not NaN)</td>
</tr>
<tr>
<td>Not equal</td>
<td>Less Than, or one or both the source operands rs1,rs2 of FLCMP was NaN</td>
</tr>
<tr>
<td>Equal</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>Unordered or Equal</td>
<td>Greater Than or Equal or rs2 source of FLCMP was NaN</td>
</tr>
<tr>
<td>Greater or Equal</td>
<td>Greater Than or Equal or (rs1 source of FLCMP was NaN and rs2 source was not NaN)</td>
</tr>
<tr>
<td>Unordered or Greater or Equal</td>
<td>Greater Than or Equal or one or both the sources rs1, rs2 of FLCMP was NaN</td>
</tr>
<tr>
<td>Less or Equal</td>
<td>Less Than or Greater Than or Equal (i.e., neither source of FLCMP was NaN)</td>
</tr>
<tr>
<td>Unordered or Less or Equal</td>
<td>Less Than or Greater Than or Equal or rs2 source of FLCMP was NaN</td>
</tr>
<tr>
<td>Ordered</td>
<td>Less Than or Greater Than or Equal or (rs1 source of FLCMP was NaN) and (rs2 source was not NaN)</td>
</tr>
</tbody>
</table>

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

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

Exceptions

illegal_instruction
fp_disabled
7.38 Flush Instruction Memory

Description

Flush Instruction Memory

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

† The original assembly language syntax for a FLUSH instruction ("flush address") has been deprecated because of inconsistency with other SPARC assembly language syntax. Over time, assemblers will support the new syntax for this instruction. In the meantime, some existing assemblers may only recognize the original syntax.

\[
\begin{array}{cccccccc}
\text{op3} & \text{rs1} & \text{l=0} & \text{rs2} \\
\hline
0 & 0 & 0 & 0 & - & - \\
1 & 1 & 1 & 1 & - & - \\
\end{array}
\]

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memory

Flush Instruction Memo
FLUSH

See implementation-specific documentation for details on specific implementations of the FLUSH instruction.

On an Oracle SPARC Architecture processor:

- A FLUSH instruction causes a synchronization within the virtual processor on which the FLUSH is executed, which flushes its instruction pipeline to ensure that no instruction already fetched has subsequently been modified in memory. Any other virtual processors on the same physical processor are unaffected by a FLUSH.
- Coherency between instruction and data memories may or may not be maintained by hardware.

**IMPL. DEP. #409-S10:** The implementation of the FLUSH instruction is implementation dependent. If the implementation automatically maintains consistency between instruction and data memory,
  1. the FLUSH address is ignored and
  2. the FLUSH instruction cannot cause any data access exceptions, because its effective address operand is not translated or used by the MMU.

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

### Programming Note

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

### Programming Notes

1. Typically, FLUSH is used in self-modifying code. The use of self-modifying code is discouraged.

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

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

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

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

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

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

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

<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.</td>
</tr>
</tbody>
</table>

An attempt to execute a FLUSH instruction when instruction bits 29:25 are nonzero causes an \textit{illegal\_instruction} exception.

An attempt to execute a FLUSH instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an \textit{illegal\_instruction} exception.

Exception \textit{illegal\_instruction} \textit{DAE\_nfo\_page}
7.39 Flush Register Windows

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

**Description**

FlushW causes all active register windows except the current window to be flushed to memory at locations determined by privileged software. FlushW behaves as a NOP if there are no active windows other than the current window. At the completion of the FlushW instruction, the only active register window is the current one.

FlushW acts as a NOP if 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 88.

**Programming Note**
The FlushW instruction can be used by application software to flush register windows to memory so that it can switch memory stacks or examine register contents from previous stack frames.

FlushW acts as a NOP if CANSAVE = N_REG_WINDOWS - 2. Otherwise, there is more than one active window, so FlushW causes a spill exception. The trap vector for the spill exception is based on the contents of OTHERWIN and WSTATE. The spill trap handler is invoked with the CWP set to the window to be spilled (that is, (CWP + CANSAVE + 2) mod N_REG_WINDOWS). See Register Window Management Instructions on page 88.

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

An attempt to execute a FlushW instruction when instruction bits 29:25 or 18:0 are nonzero causes an illegal_instruction exception.

**Exceptions**

illegal_instruction
spill_n_normal
spill_n_other

31 24 02530 29 19 18

—10 0—

— — — op3 — — — — —
7.40 Floating-Point Multiply-Add and Multiply-Subtract (fused)

### Description

The fused floating-point multiply-add instructions, FMADD<|d>, perform a floating-point multiplication of the values in the floating-point registers specified by rs1 and rs2, perform a floating-point addition of the resulting product to the value in the register(s) specified by rs3, round the resulting sum, and write the result into the floating-point register(s) specified by rd.

The fused floating-point multiply-subtract instructions, FMSUB<|d>, perform a floating-point multiplication of the values in the floating-point registers specified by rs1 and rs2, perform a floating-point subtraction from the resulting product of the value in the register(s) specified by rs3, round the resulting difference, and write the result into the floating-point register(s) specified by rd.

The fused floating-point negative multiply-add instructions, FNMADD<|d>, perform a floating-point multiplication of the values in the floating-point registers specified by rs1 and rs2, perform a floating-point addition of the resulting product to the value in the register(s) specified by rs3, negate the resulting sum, round the result, and write the result into the floating-point register(s) specified by rd.

The fused floating-point negative multiply-subtract instructions, FNMSUB<|d>, perform a floating-point multiplication of the values in the floating-point registers specified by rs1 and rs2, perform a floating-point subtraction from the resulting product of the value in the register(s) specified by rs3, negate the resulting difference, round the result, and write the result into the floating-point register(s) specified by rd.

All of the above instructions are “fused” operations; no rounding is performed between the multiplication operation and the subsequent addition (or subtraction). Therefore, at most one rounding step occurs per instruction executed.

The negative fused multiply-add/subtract instructions (FNM*) treat NaN values as follows:

- A source QNaN propagates with its sign bit unchanged
A generated (default response) QNaN result has a sign bit of zero
A source SNaN that is converted to a QNaN result retains the sign bit of the source SNaN

Exceptions. If an FMAf instruction is not implemented in hardware, it generates an illegal_instruction exception, so that privileged software can emulate the instruction.

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

Overflow, underflow, and inexact exception bits within FSR.cexc and FSR.aexc are updated based on the final result of the operation and not on the intermediate result of the multiplication. The invalid operation exception bits within FSR.cexc and FSR.aexc are updated as if the multiplication and the addition/subtraction were performed using two individual instructions. An invalid operation exception is detected when any of the following conditions are true:

- A source operand (F[rs1], F[rs2], or F[rs3]) is a SNaN
- \(\infty \times 0\)
- \(\infty - \infty\)

If the instruction generates an IEEE-754 exception or exceptions for which the corresponding trap enable mask (FSR.tem) bits are set, an fp_exception_ieee_754 exception and subsequent trap is generated.

If either the multiply or the add/subtract operation detects an unfinished_FPop condition (for example, due to a subnormal operand or final result), the Multiply-Add/Subtract instruction generates an fp_exception_other exception with FSR.ftt = unfinished_FPop. An fp_exception_other exception with FSR.ftt = unfinished_FPop always takes precedence over an fp_exception_ieee_754 exception. That is, if an fp_exception_other exception occurs due to an unfinished_FPop condition, the FSR.cexc and FSR.aexc fields remain unchanged even if a floating point IEEE 754 exception occurs during the multiply operation (regardless whether traps are enabled, via FSR.tem, for the IEEE exception) and the unfinished_FPop condition occurs during the subsequent add/subtract operation.

For more details regarding floating-point exceptions, see Chapter 8, IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011.

Semantic Definitions

**FMADD:**

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

**FNMADD:**

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

**FMSUB:**

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

**FNMSUB:**

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

Exceptions

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

See Also

FMUL on page 185
FADD on page 150
FSUB on page 207
7.41 Partitioned Mean [VIS3]

The FMEAN16 instructions is new and is not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

These instructions perform four 16-bit partitioned arithmetic averages between pairs of corresponding signed integer values contained in the source operands (F_D[rs1], F_D[rs2]).

The average of each pair of 16-bit operand values is calculated as

\[
\text{trunc}\left(\frac{\text{sign_ext}(16\text{-bit operand 1}) + \text{sign_ext}(16\text{-bit operand 2}) + 1}{2}\right)
\]

where the \text{trunc} function returns the integer portion of the result of the division. FIGURE 7-9 shows the division operation as a right shift by one bit position.

This operation in effect performs integer division round-to-nearest (round up in case of a tie) for positive numbers, which is beneficial for MPEG/DVD motion compensation and motion estimation. It can be used for half-pel interpolation as well as combining motion prediction data and inverse discrete cosine transform data.

Since 17 bits are provided to hold the intermediate sum before right shifting, no overflow can occur.
If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMEAN16 instruction causes an \textit{fp\_disabled} exception.

\textbf{Exceptions} \hspace{1em} \textit{fp\_disabled}
7.42 Floating-Point Move

Description
FMOV copies the source floating-point register(s) to the destination floating-point register(s), unaltered.

FMOVs, FMOVd, and FMOVq perform 32-bit, 64-bit, and 128-bit operations, respectively.

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

Programming Note
If a 64-bit floating-point register-to-register copy is desired and simultaneous modification of FSR is not required, use of FSRC2d is strongly recommended over FMOVd. FSRC2d is at least as fast as, and on many implementations much faster than, FMOVd and FSRC1d.

Note
Oracle SPARC Architecture 2011 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.lef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMOV instruction causes an fp_disabled exception.

An attempt to execute an FMOVq instruction when rs2[1] ≠ 0 or rd[1] ≠ 0 causes an fp_exception_other (FSR.flt = invalid_fp_register) exception.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011*.

Exceptions
illegal_instruction
fp_disabled fp_exception_other (FSR.flt = invalid_fp_register (FMOVq only))

See Also
F Register Logical Operate (2 operand) on page 214
7.43 Move Floating-Point Register on Condition (FMOVcc)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf_low</th>
<th>Operation Description</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOVSicc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on 32-bit integer condition codes</td>
<td><code>fmovsicc %icc, fregRs2, fregRs</code></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><code>fmovdicc %icc, fregRs2, fregRs</code></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><code>fmovqicc %icc, fregRs2, fregRs</code></td>
<td>C3</td>
</tr>
<tr>
<td>FMOVSxcc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on 64-bit integer condition codes</td>
<td><code>fmovsxcc %xcc, fregRs2, fregRs</code></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><code>fmovdxcc %xcc, fregRs2, fregRs</code></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><code>fmovqxcc %xcc, fregRs2, fregRs</code></td>
<td>C3</td>
</tr>
<tr>
<td>FMOVSfcc</td>
<td>00 0001</td>
<td>Move Floating-Point Single, based on floating-point condition codes</td>
<td><code>fmovsfcc %fccn, fregRs2, fregRs</code></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><code>fmovdfcc %fccn, fregRs2, fregRs</code></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><code>fmovqfcc %fccn, fregRs2, fregRs</code></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 lu)</td>
</tr>
<tr>
<td>1110</td>
<td>Move if Positive</td>
<td>not N</td>
<td>pos</td>
</tr>
<tr>
<td>0110</td>
<td>Move if Negative</td>
<td>N</td>
<td>neg</td>
</tr>
<tr>
<td>1111</td>
<td>Move if Overflow Clear</td>
<td>not V</td>
<td>vc</td>
</tr>
<tr>
<td>0111</td>
<td>Move if Overflow Set</td>
<td>V</td>
<td>vs</td>
</tr>
</tbody>
</table>

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

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

Encoding of opf_cc Field (also see Table A-9 on page 465)

<table>
<thead>
<tr>
<th>opf_cc</th>
<th>Instruction</th>
<th>Condition Code to be Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>1002</td>
<td>FMOV&lt;s</td>
<td>d</td>
</tr>
<tr>
<td>1102</td>
<td>FMOV&lt;s</td>
<td>d</td>
</tr>
<tr>
<td>0002</td>
<td>FMOV&lt;s</td>
<td>d</td>
</tr>
<tr>
<td>0012</td>
<td>fcc1</td>
<td></td>
</tr>
<tr>
<td>0102</td>
<td>fcc2</td>
<td></td>
</tr>
<tr>
<td>0112</td>
<td>fcc3</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>(illegal_instruction exception)</td>
<td></td>
</tr>
<tr>
<td>1112</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

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

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

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

Note

Oracle SPARC Architecture 2011 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 (FPRSfef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FMOVQicc, FMOVQxcc, or FMOVQfcc instruction causes an fp_disabled exception.

An attempt to execute an FMOVQicc, FMOVQxcc, or FMOVQfcc instruction when rs2[1] ≠ 0 or rd[1] ≠ 0 causes an fp_exception_other (FSR.ftt = invalid_fp_register) exception.
Branches cause the performance of most implementations to degrade significantly. Frequently, the MOVcc and FMOVcc instructions can be used to avoid branches. For example, the following C language segment:

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

```assembly
! assume A is in %f0; B is in %f2; %xx points to ! constant area
ldd [%xx+C_1.03],%f4 ! X = 1.03
fcmpd %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

```assembly
! assume A is in %f0; B is in %f2; %xx points to ! constant area
ldd [%xx+C_1.03],%f4 ! X = 1.03
fsubd %f4,%f4,%f6 ! X' = 0.0
fcmpd %fcc3,%f0,%f2 ! A > B
fmovdle %fcc3,%f6,%f4 ! X = 0.0
```
This code also takes four instructions but requires no branches and may boost performance significantly. Use MOVcc and FMOVcc instead of branches wherever these instructions would improve performance.

**Exceptions**
- `illegal_instruction`
- `fp_disabled`
- `fp_exception_other` (FSR.ftt = invalid_fp_register (FMOVQ instructions))
### Move Floating-Point Register on Integer Register Condition (FMOVR)

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

#### Assembly Language Syntax

- `fmovr[s,d,q]z reg[rs1], freg[rs2], freg[rd]` (synonym: `fmovr[s,d,q]e reg[rs1], freg[rs2], freg[rd]`)
- `fmovr[s,d,q]le reg[rs1], freg[rs2], freg[rd]`
- `fmovr[s,d,q]lz reg[rs1], freg[rs2], freg[rd]`
- `fmovr[s,d,q]nz reg[rs1], freg[rs2], freg[rd]` (synonym: `fmovr[s,d,q]ne reg[rs1], freg[rs2], freg[rd]`)
- `fmovr[s,d,q]gz reg[rs1], freg[rs2], freg[rd]`
- `fmovr[s,d,q]gez reg[rs1], freg[rs2], freg[rd]`
**FMOVR**

**Description**
If the contents of integer register \( R[rs1] \) satisfy the condition specified in the \( r\text{cond} \) 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 \( FSR.c\text{exc} \) and \( FSR.f\text{tt} \). They do not round, do not modify \( FSR.a\text{exc} \), and do not treat floating-point NaN values differently from other floating-point values.

**Note** Oracle SPARC Architecture 2011 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 \( r\text{cond} = 000_2 \) or \( 100_2 \) causes an illegal_instruction exception.

If the FPU is not enabled (\( FPRS.f\text{ef} = 0 \) or \( P\text{STATE}.p\text{ef} = 0 \)) or if no FPU is present, an attempt to execute an FMOVR instruction causes an fp_disabled exception.

An attempt to execute an FMOVRq instruction when \( rs2[1] \neq 0 \) or \( rd[1] \neq 0 \) causes an fp_exception_other (\( FSR.f\text{tt} = \text{invalid_fp_register} \)) exception.

**Implementation Note** If this instruction is implemented by tagging each register value with an \( N \) (negative) and a \( Z \) (zero) condition bit, use the following table to determine whether \( r\text{cond} \) is TRUE:

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

**Exceptions**
- illegal_instruction
- fp_disabled
- fp_exception_other (\( FSR.f\text{tt} = \text{invalid_fp_register} \) (FMOVRq instructions))
7.45 Partitioned Multiply Instructions

### Description

The following sections describe the versions of partitioned multiplies.

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

### Exceptions

**fp_disabled**

#### 7.45.1 FMUL8x16 Instruction

**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.
7.45.2 FMUL8x16AU Instruction

FMUL8x16AU is the same as FMUL8x16, except that one 16-bit fixed-point value is used as the multiplier for all four multiplies. This multiplier is the most significant (“upper”) 16 bits of the 32-bit register $F_S[rs2]$ (typically an $\alpha$ pixel component value). Figure 7-11 illustrates the operation.

7.45.3 FMUL8x16AL Instruction

FMUL8x16AL is the same as FMUL8x16AU, except that the least significant (“lower”) 16 bits of the 32-bit register $F_S[rs2]$ register are used as a multiplier. Figure 7-12 illustrates the operation.
7.45.4 FMUL8SUx16 Instruction

FMUL8SUx16 multiplies the most significant ("upper") 8 bits of each 16-bit signed value in the 64-bit floating-point register \( F_D[rs1] \) by the corresponding signed, 16-bit, fixed-point, signed integer in the 64-bit floating-point register \( F_D[rs2] \). It rounds the 24-bit product toward the nearest representable value and then stores the most significant 16 bits of the result into the corresponding 16-bit field of the 64-bit floating-point destination register \( F_D[rd] \). If the product is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE 7-13 illustrates the operation.

![FIGURE 7-13 FMUL8SUx16 Operation](image)

7.45.5 FMUL8ULx16 Instruction

FMUL8ULx16 multiplies the unsigned least significant ("lower") 8 bits of each 16-bit value in the 64-bit floating-point register \( F_D[rs1] \) by the corresponding fixed-point signed 16-bit integer in the 64-bit floating-point register \( F_D[rs2] \). Each 24-bit product is sign-extended to 32 bits. The most significant ("upper") 16 bits of the sign-extended value are rounded to nearest and then stored in the corresponding 16-bit field of the 64-bit floating-point destination register \( F_D[rd] \). If the result is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE 7-14 illustrates the operation; CODE EXAMPLE 7-1 exemplifies the operation.

![FIGURE 7-14 FMUL8ULx16 Operation](image)

**CODE EXAMPLE 7-1** 16-bit × 16-bit 16-bit Multiply

```assembly
fmul8sux16  %f0, %f2, %f4
fmul8ulx16  %f0, %f2, %f6
fpadd16     %f4, %f6, %f8
```
7.45.6 FMULD8SUx16 Instruction

FMULD8SUx16 multiplies the most significant ("upper") 8 bits of each 16-bit signed value in F[rs1] by the corresponding signed 16-bit fixed-point value in F[rs2]. Each 24-bit product is shifted left by 8 bits to generate a 32-bit result, which is then stored in the 64-bit floating-point register specified by rd. FIGURE 7-15 illustrates the operation.

![FIGURE 7-15 FMULD8SUx16 Operation](image)

7.45.7 FMULD8ULx16 Instruction

FMULD8ULx16 multiplies the unsigned least significant ("lower") 8 bits of each 16-bit value in F[rs1] by the corresponding 16-bit fixed-point signed integer in F[rs2]. Each 24-bit product is sign-extended to 32 bits and stored in the corresponding half of the 64-bit floating-point register specified by rd. FIGURE 7-16 illustrates the operation; CODE EXAMPLE 7-2 exemplifies the operation.

![FIGURE 7-16 FMULD8ULx16 Operation](image)

**CODE EXAMPLE 7-2** 16-bit by 16-bit 32-bit Multiply

```assembly
fmuld8sux16 %f0, %f2, %f4
fmuld8ulx16 %f0, %f2, %f6
fpadd32 %f4, %f6, %f8
```
CHAPTER 7 • Instructions

7.46 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`.

Oracle SPARC Architecture 2011 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.peek = 0` or `PSTATE.peek = 0`) or if no FPU is present, an attempt to execute any FMUL instruction causes an `fp_disabled` exception.

An attempt to execute an FMULq instruction when `rs1{1} ≠ 0` or `rs2{1} ≠ 0` or `rd{1:0} ≠ 0` causes an `fp_exception_other` (`FSR.flt = invalid_fp_register`) exception.

An attempt to execute an FdMULq instruction when `rd{1} ≠ 0` causes an `fp_exception_other` (`FSR.flt = invalid_fp_register`) exception.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011*.

### Exceptions

- `illegal_instruction`
- `fp_disabled`
- `fp_exception_other` (`FSR.flt = invalid_fp_register` (FMULq and FdMULq only))
- `fp_exception_other` (`FSR.flt = unfinished_FPop`)
- `fp_exception_ieee_754` (any: NV; FMUL<s|d|q> only: OF, UF, NX)

### See Also

FMAf on page 170
7.47 Floating-Point Negative Add

<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>FNADDs</td>
<td>0</td>
<td>Negative Add, Single</td>
<td>fnadds freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C1</td>
</tr>
<tr>
<td>FNADDd</td>
<td>0</td>
<td>Negative Add, Double</td>
<td>fnaddd freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C1</td>
</tr>
</tbody>
</table>

**Description**

The floating-point negative add instructions, FNADD<s|d>, add the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, negate the sum, and write the result into the floating-point register(s) specified by rd.

No rounding is performed between the addition operation and the subsequent negation; at most one rounding step occurs.

The FNADD instructions treat NaN values as follows:
- A source QNaN propagates with its sign bit unchanged
- A generated (default response) QNaN result has a sign bit of zero
- A source SNaN that is converted to a QNaN result retains the sign bit of the source SNaN

**Exceptions.** If an FNADD instruction is not implemented in hardware, it generates an illegal_instruction exception, so that privileged software can emulate the instruction.

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

Overflow, underflow, and inexact exception bits within FSR.cexc and FSR.aexc are updated based on the final result of the operation and not on the intermediate result of the addition. The invalid operation exception bits within FSR.cexc and FSR.aexc are updated as if the addition and negation were performed using two individual instructions. An invalid operation exception is detected when any of the following conditions are true:
- A source operand (F[rs1] or F[rs2]) is a NaN
- ∞ − ∞

If the instruction generates an IEEE-754 exception or exceptions for which the corresponding trap enable mask (FSR.tem) bits are set, an fp_exception_ieee_754 exception and subsequent trap is generated.

If the addition operation detects an unfinished_FPop condition (for example, due to a subnormal operand or intermediate result), the Negative Add instruction generates an fp_exception_other exception with FSR.ftt = unfinished_FPop. The unfinished_FPop trap takes priority over any potential IEEE-754 exception (regardless whether traps are enabled, via FSR.tem, for the IEEE exception) and the FSR.cexc and FSR.aexc fields remain unchanged.

For more details regarding floating-point exceptions, see Chapter 8, IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011.
** FNADD **

**Semantic Definitions**

FNADD:

1. `tmp ← F[rs1] + F[rs2]`
2. `tmp ← −tmp`
3. `F[rd] ← round(tmp)`

**Exceptions**

- `fp_disabled`
- `fp_exception_ieee_754` (OF, UF, NX, NV)
- `fp_exception_other` (FSR.flt = unfinished_FPop)

**See Also**

- FMAf on page 170
- FADD on page 150
- FNEG on page 188
- FNMR on page 170
7.48 Floating-Point Negate

### Description
FNEG copies the source floating-point register(s) to the destination floating-point register(s), with the sign bit complemented.

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

Note: Oracle SPARC Architecture 2011 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FNEGq instruction causes an illegal_instruction exception, allowing privileged software to emulate the instruction.

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

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

An attempt to execute an FNEGq instruction when rs2[1] ≠ 0 or rd[1] ≠ 0 causes an fp_exception_other (FSR.ftt = invalid_fp_register) exception.

### Exceptions
- **illegal_instruction**
- **fp_disabled**
- **fp_exception_other** (FSR.ftt = invalid_fp_register (FNEGq only))

### Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNEGs</td>
<td>11 100</td>
<td>0 0000 0101</td>
<td>Negate Single</td>
<td>fnegs freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FNEGd</td>
<td>11 100</td>
<td>0 0000 0110</td>
<td>Negate Double</td>
<td>fnegd freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FNEGq</td>
<td>11 100</td>
<td>0 0000 0111</td>
<td>Negate Quad</td>
<td>fnegq freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C3</td>
</tr>
</tbody>
</table>

Note: Oracle SPARC Architecture 2011 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FNEGq instruction causes an illegal_instruction exception, allowing privileged software to emulate the instruction.
### 7.49 Floating-point Negative Add and Halve

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNHADDs</td>
<td>11 0100</td>
<td>0 0111 0001</td>
<td>Single-precision Add, Halve, and Negate</td>
<td>fnhadds freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C2</td>
</tr>
<tr>
<td>FNHADDd</td>
<td>11 0100</td>
<td>0 0111 0010</td>
<td>Double-precision Add, Halve, and Negate</td>
<td>fnhaddd freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C2</td>
</tr>
</tbody>
</table>

**Description**

The FNHADD<s|d> instructions are used to perform an addition of two floating-point values, halve the result (divide it by 2), and then negate it, all in a single operation. One benefit of this operation is that it cannot produce an arithmetic overflow.

**Exceptions**

- fp_disabled
- fp_exception_ieee_754 (UF, NX, NV)

**See Also**

- FHADD on page 161
- FHSUB on page 162
7.50 Floating-Point Negative Multiply

Description

The floating-point negative multiply instructions, FNMUL<s|d>, multiply the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, negate the product, and write the result into the floating-point register(s) specified by rd.

The floating-point negative multiply single to double instruction, FNsMULd, multiplies the single-precision floating point registers F_S[rs1] and F_S[rs2], generates a double-precision product, negates the product, and writes the result into the double-precision floating-point register F_D[rd].

No rounding is performed between the multiplication operation and the subsequent negation; at most one rounding step occurs.

The FNMUL instructions treat NaN values as follows:
- A source QNaN propagates with its sign bit unchanged
- A generated (default response) QNaN result has a sign bit of zero
- A source SNaN that is converted to a QNaN result retains the sign bit of the source SNaN

Exceptions. If an FNMUL instruction is not implemented in hardware, it generates an illegal_instruction exception, so that privileged software can emulate the instruction.

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

Overflow, underflow, and inexact exception bits within FSR.cexc and FSR.aexc are updated based on the final result of the operation and not on the intermediate result of the multiplication. The invalid operation exception bits within FSR.cexc and FSR.aexc are updated as if the multiplication and negation were performed using two individual instructions. An invalid operation exception is detected when any of the following conditions are true:
- A source operand (F[rs1] or F[rs2]) is a SNaN
- ∞ x 0

If the instruction generates an IEEE-754 exception or exceptions for which the corresponding trap enable mask (FSR.tem) bits are set, an fp_exception_ieee_754 exception and subsequent trap is generated.

If the multiplication operation detects an unfinished_FPop condition (for example, due to a subnormal operand or intermediate result), the Negative Multiply instruction generates an fp_exception_other exception with FSR.ftt = unfinished_FPop. The unfinished_FPop trap takes priority over any potential IEEE-754 exception (regardless whether traps are enabled, via FSR.tem, for the IEEE exception) and the FSR.cexc and FSR.aexc fields remain unchanged.
For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011*.

**Semantic Definitions:**

FNMUL:

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

**Exceptions**

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

**See Also**

- FMAf on page 170
- FMUL on page 185
- FNADD on page 170
- FNEG on page 188
7.51 FPACK

Description
The FPACK instructions convert multiple values in a source register to a lower-precision fixed or pixel format and stores the resulting values in the destination register. Input values are clipped to the dynamic range of the output format. Packing applies a scale factor from GSR.scale to allow flexible positioning of the binary point. See the subsections on following pages for more detailed descriptions of the operations of these instructions.

An attempt to execute an FPACK16 or FPACKFIX instruction when rs1 ≠ 0 causes an illegal_instruction exception.

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

Exceptions
illegal_instruction  fp_disabled

See Also
FEXPAND on page 160
or PSTATE.pef = 0FPMERGE on page 206
7.51.1 FPACK16

FPACK16 takes four 16-bit fixed values from the 64-bit floating-point register F\textsubscript{D}[rs2], scales, truncates, and clips them into four 8-bit unsigned integers, and stores the results in the 32-bit destination register, F\textsubscript{S}[rd]. Figure 7-17 illustrates the FPACK16 operation.

![FPACK16 Operation Diagram](image)

**FIGURE 7-17 FPACK16 Operation**

**Note** FPACK16 ignores the most significant bit of GSR.scale (GSR.scale\{4\}).

This operation is carried out as follows:

1. Left-shift the value from F\textsubscript{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\textsubscript{S}[rd].

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

```plaintext
tmp ← source_operand{15:0} << GSR.scale;
// Pick off the bits from bit position 15+GSR.scale to bit position 7 from the shifted result
trunc_signed_value ← tmp{(15+GSR.scale):7};
If (trunc_signed_value < 0)
unsigned_8bit_result ← 0;
else if (trunc_signed_value > 255)
unsigned_8bit_result ← 255;
else
unsigned_8bit_result ← trunc_signed_value{14:7};
```
FPACK32 takes two 32-bit fixed values from the second source operand (64-bit floating-point register \( F_D[r2] \)) and scales, truncates, and clips them into two 8-bit unsigned integers. The two 8-bit integers are merged at the corresponding least significant byte positions of each 32-bit word in the 64-bit floating-point register \( F_D[r1] \), left-shifted by 8 bits. The 64-bit result is stored in \( F_D[rd] \). Thus, successive FPACK32 instructions can assemble two pixels by using three or four pairs of 32-bit fixed values. FIGURE 7-18 illustrates the FPACK32 operation.

This operation, illustrated in FIGURE 7-18, is carried out as follows:

1. Left-shift each 32-bit value in \( F_D[r2] \) by the number of bits specified in \( GSR.scale \), while maintaining clipping information.
2. For each 32-bit value, truncate and clip to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 23 and 22 for each 32-bit word). Truncation is performed to convert the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative (that is, the most significant bit is 1), then 0 is returned as the clipped value. If the value is greater than 255, then 255 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.
3. Left-shift each 32-bit value from \( F_D[r1] \) by 8 bits.
4. Merge the two clipped 8-bit unsigned values into the corresponding least significant byte positions in the left-shifted \( F_D[r2] \) value.
5. Store the result in the 64-bit destination register \( F_D[rd] \).

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

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

7.51.3 FPACKFIX

FPACKFIX takes two 32-bit fixed values from the 64-bit floating-point register \( F_D[rs2] \), scales, truncates, and clips them into two 16-bit unsigned integers, and then stores the result in the 32-bit destination register \( F_S[rd] \). FIGURE 7-19 illustrates the FPACKFIX operation.

FIGURE 7-19 FPACKFIX Operation

This operation is carried out as follows:

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

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

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

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

```pseudo
tmp ← source_operand{31:0} << GSR.scale;
// Pick off the bits from bit position 31+GSR.scale to
// bit position 16 from the shifted result
trunc_signed_value ← tmp{(31+GSR.scale):16};
if (trunc_signed_value < -32768) then signed_16bit_result ← -32768;
else if (trunc_signed_value > 32767) then signed_16bit_result ← 32767;
else signed_16bit_result ← trunc_signed_value{31:16};
```
### 7.52 Partitioned Add

The FPADD64 instruction is new and not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPADD16</td>
<td>0 0101 0000</td>
<td>Four 16-bit adds</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpadd16 rs1, rs2, rd</td>
<td>A1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPADD32</td>
<td>0 0101 0010</td>
<td>Two 32-bit adds</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpadd32 rs1, rs2, rd</td>
<td>A1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPADD64</td>
<td>0 0100 0010</td>
<td>64-bit integer add</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpadd64 rs1, rs2, rd</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FPADD16s</td>
<td>0 0101 0001</td>
<td>Two 16-bit adds</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>fpadd16s rs1, rs2, rd</td>
<td>A1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPADD32s</td>
<td>0 0101 0011</td>
<td>One 32-bit add</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>fpadd32s rs1, rs2, rd</td>
<td>A1</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

#### Description

The 64-bit versions of these instructions perform partitioned integer addition of four 16-bit (FPADD16), two 32-bit (FPADD32) values, or one 64-bit value (FPADD64) value in \( F_D[rs1] \) to corresponding value(s) in \( F_D[rs2] \). The result values are written to the destination register, \( F_D[rd] \).

The 32-bit versions of these instructions perform two 16-bit partitioned additions (FPADD16s) or one 32-bit (FPADD32s) partitioned addition, writing the result(s) to \( F_S[rd] \).

Any carry out from each addition is discarded and a 2's-complement arithmetic result is produced.
FIGURE 7-21 FPADD32 Operation

FIGURE 7-22 FPADD64 Operation

FIGURE 7-23 FPADD16s Operation

FIGURE 7-24 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`

**See Also**  
`Partitioned Add with Saturation` on page 199  
`or PSTATE.pef = 0`  
`Partitioned Subtract` on page 207
7.53 Partitioned Add with Saturation

The FPADDS instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPADDS16N</td>
<td>00101 1000</td>
<td>Four 16-bit adds with saturation</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpadds16 freg$<em>{rs1}$, freg$</em>{rs2}$, freg$_{rd}$</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FPADDS32N</td>
<td>00101 1010</td>
<td>Two 32-bit adds with saturation</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpadds32 freg$<em>{rs1}$, freg$</em>{rs2}$, freg$_{rd}$</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FPADDS16sN</td>
<td>00101 1001</td>
<td>Two 16-bit adds with saturation</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>fpadds16s freg$<em>{rs1}$, freg$</em>{rs2}$, freg$_{rd}$</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FPADDS32sN</td>
<td>00101 1011</td>
<td>One 32-bit add with saturation</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>fpadds32s freg$<em>{rs1}$, freg$</em>{rs2}$, freg$_{rd}$</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

**Description**

The 64-bit versions of these instructions perform partitioned integer addition with saturation of four 16-bit (FPADDS16), or two 32-bit (FPADDS32) values in $F_D[rs1]$ to corresponding values in $F_D[rs2]$. The result values are written to the destination register, $F_D[rd]$.

The 32-bit versions of these instructions perform partitioned integer addition of two 16-bit values (FPADDS16s) or one 32-bit value (FPADDS32s) in $F_S[rs1]$ to corresponding value(s) in $F_S[rs2]$. The result value(s) are written to $F_S[rd]$.

These instructions clip (saturate) overflow results, as indicated in Table 7-11.

**TABLE 7-11**  Clipping Values for FPADDS instructions

<table>
<thead>
<tr>
<th>Element (Partition) Size</th>
<th>Signed Add with Saturation</th>
<th>Unsigned Add with Saturation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Negative Overflow Clipped in Result to</td>
<td>Positive Overflow Clipped in Result to</td>
</tr>
<tr>
<td>16 bits</td>
<td>$-2^{15}$ (−32,768)</td>
<td>$2^{15}$−1 (32,767)</td>
</tr>
<tr>
<td>32 bits</td>
<td>$-2^{31}$</td>
<td>$2^{31}$−1</td>
</tr>
</tbody>
</table>

**FIGURE 7-25** FPADDS16 Operation
If the FPU is not enabled (FPRS.ife = 0 or PSTATE.ife = 0) or if no FPU is present, an attempt to execute an FPADD instruction causes an \textit{fp\_disabled} exception.

\textbf{Exceptions} \hspace{1cm} \textit{fp\_disabled}

\textbf{See Also} \hspace{1cm} \textit{Partitioned Add} on page 196
\hspace{1cm} \textit{Partitioned Subtract with Saturation} on page 210
### 7.54  Partitioned Signed Compare

<table>
<thead>
<tr>
<th>instruction</th>
<th>opf</th>
<th>Operation</th>
<th>s1</th>
<th>s2</th>
<th>d</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPIE8</td>
<td>0010 0010</td>
<td>Eight 8-bit compares; set bit in R[rd] if src1 ≠ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpne8 (or fcmpune8 ‡)</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FPCMPIEQ8</td>
<td>0010 1010</td>
<td>Eight 8-bit compares; set bit in R[rd] if src1 = src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmppeq8 (or fcmpeeq8 ‡)</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FPCMPIEQ16</td>
<td>0010 1010</td>
<td>Four 16-bit compares; set bit in R[rd] if src1 = src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpeq16 (or fcmpeq16 ‡)</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPCMPIE16</td>
<td>0010 0010</td>
<td>Four 16-bit compares; set bit in R[rd] if src1 ≠ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpne16 (or fcmpeq16 ‡)</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPCMPLE16</td>
<td>0010 0000</td>
<td>Four 16-bit compares; set bit in R[rd] if src1 ≤ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmple16 (or fcmple16 ‡)</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPCMPGIE16</td>
<td>0010 1000</td>
<td>Four 16-bit compares; set bit in R[rd] if src1 &gt; src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpgt16 (or fcmple16 ‡)</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPCMPIEQ32</td>
<td>0010 1110</td>
<td>Two 32-bit compares; set bit in R[rd] if src1 = src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpne32 (or fcmpee32 ‡)</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPCMPIE32</td>
<td>0010 0110</td>
<td>Two 32-bit compares; set bit in R[rd] if src1 ≠ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpne32 (or fcmpee32 ‡)</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPCMPLE32</td>
<td>0010 0100</td>
<td>Two 32-bit compares; set bit in R[rd] if src1 ≤ src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmple32 (or fcmple32 ‡)</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPCMPGIE32</td>
<td>0010 1100</td>
<td>Two 32-bit compares; set bit in R[rd] if src1 &gt; src2</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fcmpgt32 (or fcmple32 ‡)</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

† the older, deprecated mnemonic for this instruction (still recognized by the assembler)
‡ a synonymous mnemonic, since signed and unsigned comparisons for equality (or inequality) are identical

#### Description
Four 16-bit, or two 32-bit signed integer values in F_D[rs1] and F_D[rs2] are compared. The 4, or 2 bits (respectively) of comparison results are stored in the least significant bits of integer register R[rd]. The least significant 16-bit, or 32-bit comparison result corresponds to bit 0 of R[rd].

**Note** Bits not written with comparison results in destination integer R[rd] are set to zero.

For FPCMPGIE16,32, each bit in the result is set to 1 if the corresponding signed integer value in F_D[rs1] is greater than the signed value in F_D[rs2].

**Programming Note** Less-than comparisons are made by swapping the source operands of FPCMPGIE.

For FPCMPLE16,32, each bit in the result is set to 1 if the corresponding signed integer value in F_D[rs1] is less than or equal to the signed value in F_D[rs2].

**Programming Note** Greater-than-or-equal comparisons are made by swapping the source operands of FPCMPLE.

<table>
<thead>
<tr>
<th>rd</th>
<th>31 30 29</th>
<th>11 10 11</th>
<th>rs1</th>
<th>19 18</th>
<th>14 13</th>
<th>opf</th>
<th>5 4</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30 29</td>
<td>25 24</td>
<td>18 17</td>
<td>14 13</td>
<td>5 4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For FPCMPEQ16 and FPCMPEQ32, each bit in the result is set to 1 if the corresponding signed integer value in \( F_D[rs1] \) is equal to the signed value in \( F_D[rs2] \).

For FPCMPNE16 and FPCMPNE32, each bit in the result is set to 1 if the corresponding signed integer value in \( F_D[rs1] \) is not equal to the signed value in \( F_D[rs2] \).

FIGURE 7-29 and FIGURE 7-30 illustrate 16-bit, and 32-bit pixel comparison operations, respectively.

**FIGURE 7-29** Four 16-bit Signed Integer Partitioned Comparison Operations

**FIGURE 7-30** Two 32-bit Signed Integer Partitioned Comparison Operation

In all comparisons, if a compare condition is not true, the corresponding bit in the result is set to 0.

**Programming Note** The results of a Partitioned signed compare operation can be used directly by integer operations (for example, partial stores) and by the CMASK instruction.

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

**Exception** \( \text{fp_disabled} \)

**See Also** CMASK on page 136
Floating-Point Compare on page 157
FPCMPU on page 203
STPARTIALF on page 332
### 7.55 Partitioned Unsigned Compare

The FPCMPU instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### FPCMPUEQ8

Eight 8-bit compare; set bit in \( R[rd] \) if \( src1 = src2 \)

```
64 64 0 25 0 10
```

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPUEQ8</td>
<td>1 0010 1010</td>
<td>Eight 8-bit compares [VIS 3B] set bit in ( R[rd] ) if ( src1 = src2 )</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fpcmpueq8 ( freg_{rs1}, freg_{rs2}, reg_{rd} )</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

### FPCMPUNE8

Eight 8-bit compare; set bit in \( R[rd] \) if \( src1 \neq src2 \)

```
64 64 0 25 0 10
```

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPUNE8</td>
<td>1 0010 0010</td>
<td>Eight 8-bit compares [VIS 3B] set bit in ( R[rd] ) if ( src1 \neq src2 )</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fpcmpune8 ( freg_{rs1}, freg_{rs2}, reg_{rd} )</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

### FPCMPULE8

Eight 8-bit compare; set bit in \( R[rd] \) if \( src1 \leq src2 \)

```
64 64 0 25 0 10
```

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPULE8</td>
<td>1 0010 0000</td>
<td>Eight 8-bit compares [VIS 3B] set bit in ( R[rd] ) if ( src1 \leq src2 )</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fpcmpule8 ( freg_{rs1}, freg_{rs2}, reg_{rd} )</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

### FPCMPUGT8

Eight 8-bit compare; set bit in \( R[rd] \) if \( src1 > src2 \)

```
64 64 0 25 0 10
```

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPUGT8</td>
<td>1 0010 1000</td>
<td>Eight 8-bit compares [VIS 3B] set bit in ( R[rd] ) if ( src1 &gt; src2 )</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fpcmpugt8 ( freg_{rs1}, freg_{rs2}, reg_{rd} )</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

### FPCMPUEQ16

Four 16-bit compares; set bit in \( R[rd] \) if \( src1 = src2 \)

```
64 64 0 25 0 10
```

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPUEQ16</td>
<td>0 0010 1010</td>
<td>Four 16-bit compares [VIS 3B] set bit in ( R[rd] ) if ( src1 = src2 )</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fpcmpueq16 ( freg_{rs1}, freg_{rs2}, reg_{rd} )</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

### FPCMPUNE16

Four 16-bit compares; set bit in \( R[rd] \) if \( src1 \neq src2 \)

```
64 64 0 25 0 10
```

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPUNE16</td>
<td>0 0010 0010</td>
<td>Four 16-bit compares [VIS 3B] set bit in ( R[rd] ) if ( src1 \neq src2 )</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fpcmpune16 ( freg_{rs1}, freg_{rs2}, reg_{rd} )</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

### FPCMPUEQ32

Two 32-bit compares; set bit in \( R[rd] \) if \( src1 = src2 \)

```
64 64 0 25 0 10
```

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPUEQ32</td>
<td>0 0010 1110</td>
<td>Two 32-bit compares [VIS 3B] set bit in ( R[rd] ) if ( src1 = src2 )</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fpcmpueq32 ( freg_{rs1}, freg_{rs2}, reg_{rd} )</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

### FPCMPUNE32

Two 32-bit compares; set bit in \( R[rd] \) if \( src1 \neq src2 \)

```
64 64 0 25 0 10
```

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCMPUNE32</td>
<td>0 0010 0110</td>
<td>Two 32-bit compares [VIS 3B] set bit in ( R[rd] ) if ( src1 \neq src2 )</td>
<td>f64</td>
<td>f64</td>
<td>i64</td>
<td>fpcmpune32 ( freg_{rs1}, freg_{rs2}, reg_{rd} )</td>
<td>B1</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

‡ a synonymous mnemonic, since signed and unsigned comparisons for equality (or inequality) are identical
FIGURE 7-31 illustrates the 8-bit unsigned Partitioned comparison operations.

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

Exceptions

`fp_disabled`

See Also

CMASK on page 136
STPARTIALF on page 332
7.56 Integer Multiply-Add

The integer multiply-add instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### Description

The Integer Multiply-Add instruction performs a fused multiply and add operation on unsigned 64-bit integer values residing in floating-point registers.

FPMADDX performs an unsigned integer multiplication of the 64-bit floating-point registers $F_D[rs1]$ and $F_D[rs2]$, adds the resulting product to the unsigned integer value from $F_D[rs3]$, then writes the least significant 8 bytes of the result into $F_D[rd]$.

FPMADDXHI performs an unsigned integer multiplication of the 64-bit floating-point registers $F_D[rs1]$ and $F_D[rs2]$, adds the resulting product to the unsigned integer value from $F_D[rs3]$, then writes the most significant 8 bytes of the result into $F_D[rd]$.

FPMADDX and FPMADDXHI do not modify any portion of FSR.

### Exceptions

- $fp\_disabled$
FPMERGE

7.57 \text{\texttt{VIS}} 4 \text{ or \texttt{PSTATE.pef = 0}} \textbf{FPMERGE} \text{ \texttt{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>FPMERGE</td>
<td>0100</td>
<td>Two 32-bit merges</td>
<td>f32</td>
<td>f32</td>
<td>f64</td>
<td>\texttt{fpmerge _frs1, _frs2, _fdrd}</td>
<td>B1</td>
</tr>
</tbody>
</table>

\textit{Description}  
FPMERGE interleaves eight 8-bit unsigned values in \( F_S[r_{s1}] \) and \( F_S[r_{s2}] \) to produce a 64-bit value in the destination register \( F_D[r_{d}] \). This instruction converts from packed to planar representation when it is applied twice in succession; for example, \( R1G1B1A1, R3G3B3A3 \rightarrow R1R3G1G3A1A3 \rightarrow R1R2R3R4G1G2G3G4 \).

FPMERGE also converts from planar to packed when it is applied twice in succession; for example, \( R1R2R3R4, B1B2B3B4 \rightarrow R1B1B2B3B4 \rightarrow R1G1B1A1R2G2B2A2 \).

\textbf{FIGURE 7-32} illustrates the operation.

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
10 & rd & 110110 & rs1 & opf & rs2 \\
\hline
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\textbf{FIGURE 7-32} FPMERGE Operation
\end{center}

\begin{verbatim}
%d0 !R1 G1 B1 A1 R2 G2 B2 A2
%d2 !R3 G3 B3 A3 R4 G4 B4 A4 \} packed representation
fpmerge %f0, %f2, %d4 !R1 R3 G1 G3 B1 B3 A1 A3
fpmerge %f1, %f3, %d6 !R2 R4 G2 G4 B2 B4 A2 A4 \} intermediate
fpmerge %f4, %f6, %d0 !R1 R2 R3 R4 G1 G2 G3 G4
fpmerge %f5, %f7, %d2 !B1 B2 B3 B4 A1 A2 A3 A4 \} planar representation
fpmerge %f0, %f2, %d4 !R1 B1 R2 B2 R3 B3 R4 B4
fpmerge %f1, %f3, %d6 !G1 A1 G2 A2 G3 A3 G4 A4 \} intermediate
fpmerge %f4, %f6, %d0 !R1 G1 B1 A1 R2 G2 B2 A2
fpmerge %f5, %f7, %d2 !R3 G3 B3 A3 R4 G4 B4 A4 \} packed representation
\end{verbatim}

\textbf{CODE EXAMPLE 7-3}  
FPMERGE example

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

\textit{Exceptions}  
\texttt{fp\_disabled}

\textit{See Also}  
FPACK on page 192
FEXPAND on page 160
### 7.58 Partitioned Subtract

The **FPSUB64** instruction is new and not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPSUB16</td>
<td>0010101</td>
<td>Four 16-bit subtracts</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpsub16 freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPSUB32</td>
<td>0010101</td>
<td>Two 32-bit subtracts</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpsub32 freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPSUB64</td>
<td>00100010</td>
<td>64-bit integer subtract</td>
<td>f64</td>
<td>f64</td>
<td>f64</td>
<td>fpsub64 freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FPSUB16s</td>
<td>0010101</td>
<td>Two 16-bit subtracts</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>fpsub16s freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
<td>VIS 1</td>
</tr>
<tr>
<td>FPSUB32s</td>
<td>0010101</td>
<td>One 32-bit subtract</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>fpsub32s freg&lt;rs1&gt;, freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
<td>VIS 1</td>
</tr>
</tbody>
</table>

**Description**

The 64-bit versions of these instructions perform partitioned integer subtraction of four 16-bit (FPSUB16), two 32-bit (FPSUB32) values, or one 64-bit value (FPSUB64) value in $F_D[rs2]$ from corresponding value(s) in $F_D[rs1]$. The result value(s) are written to the destination register, $F_D[rd]$.

The 32-bit versions of these instructions perform partitioned integer subtraction of two 16-bit values (FPSUB16s) or one 32-bit value (FPSUB32s) in $F_S[rs2]$ from the corresponding value(s) in $F_S[rs1]$. The result(s) are written to $F_S[rd]$.

Any borrow out from each subtraction is discarded and a 2’s-complement arithmetic result is produced.

**FIGURE 7-33** FPSUB16 Operation
FPSUB

FIGURE 7-34 FPSUB32 Operation

FIGURE 7-35 FPSUB64 Operation

FIGURE 7-36 FPSUB16s Operation

FIGURE 7-37 FPSUB32s Operation
FPSUB

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

**Exceptions** \text{fp_disabled}

**See Also**
- \text{Partitioned Add on page 196}
- \text{Partitioned Subtract with Saturation on page 210}
Partitioned Subtract with Saturation

The FPSUBS instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### Description

The 64-bit versions of these instructions perform partitioned integer subtraction with saturation of four 16-bit (FPSUBS16), or two 32-bit (FPSUBS32) values in $F_D[rs2]$ from the corresponding values in $F_D[rs1]$. The result values are written to the destination register, $F_D[rd]$. The 32-bit versions of these instructions (FPSUBS16s and FPSUBS32s) perform partitioned integer subtraction of two 16-bit or one 32-bit value in $F_S[rs2]$ from corresponding values in $F_S[rs1]$. The result values are written to $F_S[rd]$. These instructions clip (saturate) overflow results, as indicated in TABLE 7-12.

### TABLE 7-12 Clipping Values for FPSUBS instructions

<table>
<thead>
<tr>
<th>Element (Partition) Size</th>
<th>Signed Subtract with Saturation</th>
<th>Unsigned Subtract with Saturation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Negative Overflow Clipped in Result to</td>
<td>Positive Overflow Clipped in Result to</td>
</tr>
<tr>
<td>16 bits</td>
<td>$-2^{15}$ (−32,768)</td>
<td>$2^{15}$=1 (32,767)</td>
</tr>
<tr>
<td>32 bits</td>
<td>$-2^{31}$</td>
<td>$2^{31}$=1</td>
</tr>
</tbody>
</table>
**FIGURE 7-38** FPSUBS16 Operation

**FIGURE 7-39** FPSUBS32 Operation

**FIGURE 7-40** FPSUBS16s Operation
If the FPU is not enabled (FPRS.ief = 0 or PSTATE.ife = 0) or if no FPU is present, an attempt to execute an FPSUBS instruction causes an `fp_disabled` exception.

**Exceptions**  
`fp_disabled`

**See Also**  
Partitioned Add with Saturation on page 199  
or `PSTATE.ife = 0` Partitioned Subtract on page 207
F Register Logical Operate (1 operand) [VIS1]

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

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

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

If the FPU is not enabled (FPRS.\( \text{p} \text{ef} \) = 0 or PSTATE.\( \text{p} \text{ef} \) = 0) or if no FPU is present, an attempt to execute an FZERO or FONE instruction causes an \( \text{fp} \_\text{disabled} \) exception.

### Exceptions
- illegal_instruction
- \( \text{fp} \_\text{disabled} \)

### See Also
- F Register 2-operand Logical Operations on page 214
- F Register 3-operand Logical Operations on page 215
F Register 2-operand Logical Ops

### 7.61 F Register Logical Operate (2 operand) [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>FSRC1d</td>
<td>0 111 0100</td>
<td>Copy F_D[rs1] to F_D[rd]</td>
<td>fsrc1d†, frsrs1, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FSRC1s</td>
<td>0 111 0101</td>
<td>Copy F_S[rs1] to F_S[rd], 32-bit</td>
<td>fsrc1s, frsrs1, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FSRC2d</td>
<td>0 111 1000</td>
<td>Copy F_D[rs2] to F_D[rd]</td>
<td>fsrc2d†, frsrs2, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FSRC2s</td>
<td>0 111 1001</td>
<td>Copy F_S[rs2] to F_S[rd], 32-bit</td>
<td>fsrc2s, frsrs2, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FNOT1d</td>
<td>0 1110 1010</td>
<td>Negate (1's complement) F_D[rs1]</td>
<td>fnot1d†, frsrs1, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FNOT1s</td>
<td>0 1110 1011</td>
<td>Negate (1's complement) F_S[rs1], 32-bit</td>
<td>fnot1s, frsrs1, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FNOT2d</td>
<td>0 1110 0110</td>
<td>Negate (1's complement) F_D[rs2]</td>
<td>fnot2d†, frsrs2, frsrd</td>
<td>A1</td>
</tr>
<tr>
<td>FNOT2s</td>
<td>0 1110 0111</td>
<td>Negate (1's complement) F_S[rs2], 32-bit</td>
<td>fnot2s, frsrs2, frsrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

† this assembly-language instruction mnemonic is also accepted without a trailing "d", for backward compatibility

**Description**

The standard 64-bit versions of these instructions perform one of four 64-bit logical operations on the data from the 64-bit floating-point source register F_D[rs1] (or F_D[rs2]) and store the result in the 64-bit floating-point destination register F_D[rd].

The 32-bit (single-precision) versions of these instructions perform 32-bit logical operations on F_S[rs1] (or F_S[rs2]) and store the result in F_S[rd].

An attempt to execute an FSRC1 or FNOT1 instruction when instruction bits 4:0 are nonzero causes an illegal_instruction exception. An attempt to execute an FSRC2(s) or FNOT2(s) instruction when instruction bits 18:14 are nonzero causes an illegal_instruction exception.

If the FPU is not enabled (FPRS.pef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FSRC1<s|d>, FNOT1<s|d>, FSRC1<s|d>, or FNOT1<s|d> instruction causes an fp_disabled exception.

**Programming Note**

FSRC1s (FSRC1d) and FSRC2s (FSRC2d) function similarly to FMOVs (FMOVd), except that the FSRC* instructions do not modify the FSR register while FMOVs (FMOVd) update some fields of FSR (see Floating-Point Move on page 174).

If a 64-bit floating-point register-to-register copy is desired and simultaneous modification of FSR is not required, use of FSRC2d is strongly recommended over FMOVd. FSRC2d is at least as fast as, and on many implementations much faster than, FMOVd and FSRC1d.

**Exceptions**

illegal_instruction

fp_disabled

**See Also**

Floating-Point Move on page 174

F Register 1-operand Logical Operations on page 213

F Register 3-operand Logical Operations on page 215
F Register 3-operand Logical Ops

7.62  F Register Logical Operate (3 operand) [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>FORd</td>
<td>0 111 1100</td>
<td>Logical or</td>
<td>fordt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FORs</td>
<td>0 111 1101</td>
<td>Logical or, 32-bit</td>
<td>fors $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F NORd</td>
<td>0 110 0010</td>
<td>Logical nor</td>
<td>fnor dt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F NORs</td>
<td>0 110 0011</td>
<td>Logical nor, 32-bit</td>
<td>fnors $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F ANDd</td>
<td>0 111 0000</td>
<td>Logical and</td>
<td>fanddt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F ANDs</td>
<td>0 111 0001</td>
<td>Logical and, 32-bit</td>
<td>fands $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F NANDd</td>
<td>0 110 1110</td>
<td>Logical nand</td>
<td>fnanddt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F NANDs</td>
<td>0 110 1111</td>
<td>Logical nand, 32-bit</td>
<td>fnands $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FXORd</td>
<td>0 110 1100</td>
<td>Logical xor</td>
<td>fxordt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FXORs</td>
<td>0 110 1101</td>
<td>Logical xor, 32-bit</td>
<td>fxors $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FXNORd</td>
<td>0 111 0010</td>
<td>Logical xnor</td>
<td>fxnordt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FXNORs</td>
<td>0 111 0011</td>
<td>Logical xnor, 32-bit</td>
<td>fxnors $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FOR NOT1d</td>
<td>0 111 1010</td>
<td>(not $F_D[rs1]$) or $F_D[rs2]$</td>
<td>fornot1 dt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FOR NOT1s</td>
<td>0 111 1011</td>
<td>(not $F_S[rs1]$) or $F_S[rs2]$, 32-bit</td>
<td>fornot1s $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FOR NOT2d</td>
<td>0 111 0110</td>
<td>$F_D[rs1]$ or (not $F_D[rs2]$)</td>
<td>fornot2 dt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>FOR NOT2s</td>
<td>0 111 0111</td>
<td>$F_S[rs1]$ or (not $F_S[rs2]$), 32-bit</td>
<td>fornot2s $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F AND NOT1d</td>
<td>0 110 1000</td>
<td>(not $F_D[rs1]$) and $F_D[rs2]$</td>
<td>fandnot1 dt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F AND NOT1s</td>
<td>0 110 1001</td>
<td>(not $F_S[rs1]$) and $F_S[rs2]$, 32-bit</td>
<td>fandnot1s $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F AND NOT2d</td>
<td>0 110 0100</td>
<td>$F_D[rs1]$ and (not $F_D[rs2]$)</td>
<td>fandnot2 dt $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
<tr>
<td>F AND NOT2s</td>
<td>0 110 0101</td>
<td>$F_S[rs1]$ and (not $F_S[rs2]$), 32-bit</td>
<td>fandnot2s $f_{rs1}$, $f_{rs2}$, $f_{rd}$</td>
<td>A1</td>
</tr>
</tbody>
</table>

† this assembly-language instruction mnemonic is also accepted without a trailing "d", for backward compatibility

### Description

The standard 64-bit versions of these instructions perform one of ten 64-bit logical operations between the 64-bit floating-point registers $F_D[rs1]$ and $F_D[rs2]$. The result is stored in the 64-bit floating-point destination register $F_D[rd]$.

The 32-bit (single-precision) versions of these instructions perform 32-bit logical operations between $F_S[rs1]$ and $F_S[rs2]$, storing the result in $F_S[rd]$.

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

### Exceptions

*fp_disabled*

### See Also

F Register 1-operand Logical Operations on page 213
F Register 2-operand Logical Operations on page 214
Partitioned Shift

The Partitioned Shift instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### Instruction Details

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opf</th>
<th>Operation</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSLL16</td>
<td>0010 0001</td>
<td>16-bit partitioned shift left</td>
<td>f64</td>
<td>f64</td>
<td></td>
<td><code>fregrs1, fregrs2, fregrd</code></td>
<td>N1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FSRL16</td>
<td>0010 0011</td>
<td>16-bit partitioned shift right</td>
<td>f64</td>
<td>f64</td>
<td></td>
<td><code>fregrs1, fregrs2, fregrd</code></td>
<td>N1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FSLL32</td>
<td>0010 0101</td>
<td>32-bit partitioned shift left</td>
<td>f64</td>
<td>f64</td>
<td></td>
<td><code>fregrs1, fregrs2, fregrd</code></td>
<td>N1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FSRL32</td>
<td>0010 0111</td>
<td>32-bit partitioned shift right</td>
<td>f64</td>
<td>f64</td>
<td></td>
<td><code>fregrs1, fregrs2, fregrd</code></td>
<td>N1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FSLAS16</td>
<td>0010 1001</td>
<td>16-bit partitioned shift left</td>
<td>f64</td>
<td>f64</td>
<td></td>
<td><code>fregrs1, fregrs2, fregrd</code></td>
<td>N1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FSRA16</td>
<td>0010 1011</td>
<td>16-bit partitioned shift right</td>
<td>f64</td>
<td>f64</td>
<td></td>
<td><code>fregrs1, fregrs2, fregrd</code></td>
<td>N1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FSLAS32</td>
<td>0010 1101</td>
<td>32-bit partitioned shift left</td>
<td>f64</td>
<td>f64</td>
<td></td>
<td><code>fregrs1, fregrs2, fregrd</code></td>
<td>N1</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>FSRA32</td>
<td>0010 1111</td>
<td>32-bit partitioned shift right</td>
<td>f64</td>
<td>f64</td>
<td></td>
<td><code>fregrs1, fregrs2, fregrd</code></td>
<td>N1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

**Description**

These operations shift data in each partition in $F_D[rs1]$ register according to the least significant bits of each corresponding partition in $F_D[rs2]$. 32-bit shift instructions use 5 bits in each of the two partitions in $F_D[rs2]$ to specify the shift amount. 16-bit shift instructions use 4 bits in each of the 4 partitions in $F_D[rs2]$ to specify the shift amount.

![Figure 7-42: FSLL16 Operation](image-url)
FIGURE 7-43 FSLL32 Operation

Shift right arithmetic and shift left arithmetic with saturation treat their first operand as a signed value. For FSLAS[16,32], if a value opposite to the original sign is shifted through the sign position, the overflow result is clipped to the appropriate (same as the original sign) maximum positive or negative number representable by 16 (or 32) bits. Thus, the saturation values are $2^{15} - 1$ and $-2^{15}$ for 16-bit partitioned saturating shifts and $2^{31} - 1$ and $-2^{31}$ for 32-bit partitioned saturating shifts.

If the FPU is not enabled (FPRS.peq = 0 or PSTATE.peq = 0) or if no FPU is present, an attempt to execute any partitioned shift instruction causes an fp_disabled exception.

Exceptions  
fp_disabled
7.64 Floating-Point Square Root

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSQR Ts</td>
<td>11 0100</td>
<td>0 0010 1001</td>
<td>Square Root Single</td>
<td>fsqrts fregrs2, fregrd</td>
<td>A1</td>
</tr>
<tr>
<td>FSQR Td</td>
<td>11 0100</td>
<td>0 0010 1010</td>
<td>Square Root Double</td>
<td>fsqrtd fregrs2, fregrd</td>
<td>A1</td>
</tr>
<tr>
<td>FSQR Tq</td>
<td>11 0100</td>
<td>0 0010 1011</td>
<td>Square Root Quad</td>
<td>fsqrtq fregrs2, fregrd</td>
<td>C3</td>
</tr>
</tbody>
</table>

**Description**

These SPARC V9 instructions generate the square root of the floating-point operand in the floating-point register(s) specified by the rs2 field and place the result in the destination floating-point register(s) specified by the rd field. Rounding is performed as specified by FSR.rd.

**Note** Oracle SPARC Architecture 2011 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FSQRTq instruction causes an illegal_instruction exception, allowing privileged software to emulate the instruction.

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

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

An attempt to execute an FSQRTq instruction when rs2[1] ≠ 0 or rd[1] ≠ 0 causes an fp_exception_other (FSR.ftt = invalid_fp_register) exception.

An fp_exception_other (with FSR.ftt = unfinished_FPop) can occur if the operand to the square root is positive and subnormal. See FSR_floating-point_trap_type (ftt) on page 55 for additional details.

For more details regarding floating-point exceptions, see Chapter 8, IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011.

**Exceptions**

illegal_instruction
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FSQRTq only))
fp_exception_other (FSR.ftt = unfinished_FPop)
fp_exception_ieee_754 (IEEE_754_exception (NV, NX))
7.65 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>01000 0001</td>
<td>Convert Single to 64-bit Integer</td>
<td>—</td>
<td>f32</td>
<td>f64</td>
<td>fstox freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FdTOx</td>
<td>01000 0010</td>
<td>Convert Double to 64-bit Integer</td>
<td>—</td>
<td>f64</td>
<td>f64</td>
<td>fdtox freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FqTOx</td>
<td>01000 0011</td>
<td>Convert Quad to 64-bit Integer</td>
<td>—</td>
<td>f128</td>
<td>f64</td>
<td>fqtox freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>C3</td>
</tr>
<tr>
<td>FsTOi</td>
<td>01101 0001</td>
<td>Convert Single to 32-bit Integer</td>
<td>—</td>
<td>f32</td>
<td>f32</td>
<td>fstoi freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FdTOi</td>
<td>01101 0010</td>
<td>Convert Double to 32-bit Integer</td>
<td>—</td>
<td>f64</td>
<td>f32</td>
<td>fdtoi freg&lt;rs2&gt;, freg&lt;rd&gt;</td>
<td>A1</td>
</tr>
<tr>
<td>FqTOi</td>
<td>01101 0011</td>
<td>Convert Quad to 32-bit Integer</td>
<td>—</td>
<td>f128</td>
<td>f32</td>
<td>fqtoi freg&lt;rs2&gt;, freg&lt;rd&gt;</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**

Oracle SPARC Architecture 2011 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FqTOx or FqTOi instruction causes an `illegal_instruction` exception, allowing privileged software to emulate the instruction.

An attempt to execute an F<s|d|q>TO<i|x> instruction when instruction bits 18:14 are nonzero causes an `illegal_instruction` exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an F<s|d|q>TO<i|x> instruction causes an `fp_disabled` exception.

An attempt to execute an FqTOi or FqTOx instruction when rs2[1] ≠ 0 causes an `fp_exception_other` (FSR.ftt = invalid fp_register) exception.

If the floating-point operand’s value is too large to be converted to an integer of the specified size or is a NaN or infinity, then an `fp_exception_ieee_754” invalid” exception occurs. The value written into the floating-point register(s) specified by rd in these cases is as defined in Integer Overflow Definition on page 367.

For more details regarding floating-point exceptions, see Chapter 8.

**Exceptions**

- `illegal_instruction`
- `fp_disabled`
- `fp_exception_other` (FSR.ftt = invalid fp_register (FqTOx and FqTOi only))
- `fp_exception_other` (FSR.ftt = unfinished FPop)
- `fp_exception_ieee_754` (NV, NX)
7.66 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\(\text{rd}\) determines how rounding is performed by these instructions.

**Note** Oracle SPARC Architecture 2011 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FsTOq, FdTOq, FqTOs, or FqTOd instruction causes an `illegal_instruction` exception, allowing privileged software to emulate the instruction.

An attempt to execute an F<s|d|q>TO<s|d|q> instruction when instruction bits 18:14 are nonzero causes an `illegal_instruction` exception.

If the FPU is not enabled (FPRS.\text{lef} = 0 or PSTATE.\text{pelf} = 0) or if no FPU is present, an attempt to execute an F<s|d|q>TO<s|d|q> instruction causes an `fp_disabled` exception.

An attempt to execute an FsTOq or FdTOq instruction when \(\text{rd}[1] \neq 0\) causes an `fp_exception_other` (FSR.\text{ftt} = invalid_fp_register) exception. An attempt to execute an FqTOs or FqTOd instruction when \(\text{rs2}[1] \neq 0\) causes an `fp_exception_other` (FSR.\text{ftt} = invalid_fp_register) exception.

FqTOd, FqTOs, and FdTOs (the “narrowing” conversion instructions) can cause `fp_exception_ieee_754` OF, UF, and NX exceptions. FdTOq, FsTOq, and FsTOd (the “widening” conversion instructions) cannot.

Any of these six instructions can trigger an `fp_exception_ieee_754` NV exception if the source operand is a signalling NaN.

**Note** For FdTOs, and FsTOd, FdTOq, and FqTOd, an `fp_exception_other` with FSR.\text{ftt} = unfinished_FPops can occur if implementation-dependent conditions are detected during the conversion operation.

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011*.

### Exceptions

- `illegal_instruction`
- `fp_disabled`
- `fp_exception_other` (FSR.\text{ftt} = invalid_fp_register (FsTOq, FqTOs, FdTOq, and FqTOd only))
- `fp_exception_other` (FSR.\text{ftt} = unfinished_FPops)
F<s|d|q>TO<s|d|q>

*fp_exception_ieee_754* (NV)
*fp_exception_ieee_754* (OE, UF, NX (FqTOd, FqTOs, and FdTOs))
7.67 Floating-Point Subtract

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>opf</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSUBs</td>
<td>11</td>
<td>100</td>
<td>0 0100 0101</td>
<td>fsub s</td>
<td>A1</td>
</tr>
<tr>
<td>FSUBd</td>
<td>11</td>
<td>100</td>
<td>0 0100 0110</td>
<td>fsub d</td>
<td>A1</td>
</tr>
<tr>
<td>FSUBq</td>
<td>11</td>
<td>100</td>
<td>0 0100 0111</td>
<td>fsub q</td>
<td>C3</td>
</tr>
</tbody>
</table>

**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**

Oracle SPARC Architecture 2011 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FSUBq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

An attempt to execute an FSUBq instruction when \( (rs1[1] \neq 0) \) or \( (rs2[1] \neq 0) \) or \( (rd[1:0] \neq 0) \) causes an *fp_exception_other* (FSR.\( ftt = \text{invalid} \_ \text{fp} \_ \text{register} \)) exception.

**Note**

An *fp_exception_other* with FSR.\( ftt = \text{unfinished} \_ \text{FPop} \) can occur if the operation detects unusual, implementation-specific conditions (for FSUBs or FSUBd).

For more details regarding floating-point exceptions, see Chapter 8, *IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011*.

**Exceptions**

- *illegal_instruction*
- *fp_disabled*
- *fp_exception_other* (FSR.\( ftt = \text{invalid} \_ \text{fp} \_ \text{register} \) (FSUBq only))
- *fp_exception_other* (FSR.\( ftt = \text{unfinished} \_ \text{FPop} \))
- *fp_exception_ieee_754* (OF, UF, NX, NV)

**See Also**

FMAf on page 170
7.68 Convert 64-bit Integer to Floating Point

FxTO<s|d|q>

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

Description

FxTOs, FxTOd, and FxTOq convert the 64-bit signed integer operand in the floating-point register \( F_D[rs2] \) into a floating-point number in the destination format.

All write their result into the floating-point register(s) specified by \( rd \).

The value of \( FSR.\text{rd} \) determines how rounding is performed by FxTOs and FxTOd.

Note | Oracle SPARC Architecture 2011 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FxTOq instruction causes an \textit{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 \textit{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 \textit{fp\_disabled} exception.

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

For more details regarding floating-point exceptions, see Chapter 8, \textit{IEEE Std 754-1985 Requirements for Oracle SPARC Architecture 2011}.

Exceptions

\textit{illegal\_instruction}
\textit{fp\_disabled}
\textit{fp\_exception\_other} (\( FSR.\text{ftt} = \text{invalid\_fp\_register} \) (FxTOq))
\textit{fp\_exception\_ieee\_754} (NX (FxTOs and FxTOd only))
7.69 Illegal Instruction Trap

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op</th>
<th>op2</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILLTRAP</td>
<td>00</td>
<td>000</td>
<td>illegal_instruction trap</td>
<td>illtrap const22</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

The ILLTRAP instruction causes an *illegal_instruction* exception. The *const22* value in the instruction is ignored by the virtual processor; specifically, this field is *not* reserved by the architecture for any future use.

**V9 Compatibility Note**

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

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

**Exceptions**

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

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

- \[ \text{CANSAVE} \leftarrow (\text{N\_REG\_WINDOWS} - 2) \]
- \[ \text{CANRESTORE} \leftarrow 0 \]
- \[ \text{OTHERWIN} \leftarrow 0 \]

An attempt to execute an INVALW instruction when instruction bits 18:0 are nonzero causes an \textit{illegal\_instruction} exception.

An attempt to execute an INVALW instruction in nonprivileged mode \((\text{PSTATEpriv} = 0)\) causes a \textit{privileged\_opcode} exception.

### Exceptions
- \textit{illegal\_instruction}
- \textit{privileged\_opcode}

### See Also
- ALLCLEAN on page 115
- NORMALW on page 280
- OTHERW on page 282
- RESTORED on page 302
- SAVED on page 309
7.71 Jump and Link

**Description**

The JMPL instruction causes a register-indirect delayed control transfer to the address given by 
“\( R[rs1] + R[rs2] \)” if \( i = 0 \), or “\( R[rs1] + \text{sign\_ext}\,(\text{simm13}) \)” 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.

If a JMPL instruction with \( rd = 15 \) functions as a register-indirect call using the standard link register.

JMPL with \( rd = 0 \) can be used to return from a subroutine. The typical return address is “\( r[31] + 8 \)” if a nonleaf routine (one that uses the SAVE instruction) is entered by a CALL instruction, or “\( R[15] + 8 \)” if a leaf routine (one that does not use the SAVE instruction) is entered by a CALL instruction or by a JMPL instruction with \( rd = 15 \).

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then JMPL generates a control_transfer_instruction exception instead of causing a control transfer. When a control_transfer_instruction trap occurs, PC (the address of the JMPL instruction) is stored in TPC[TL] and the value of NPC from before the JMPL was executed is stored in TNPC[TL].

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system or being written into \( R[rd] \) (or, if a control_transfer_instruction trap occurs, into TPC[TL]). (closed impl. dep. #125-V9-Cs10)

**Exceptions**

illegal_instruction
mem_address_not_aligned
control_transfer_instruction (impl. dep. #450-S20)

**See Also**

CALL on page 127
Bicc on page 120
BPcc on page 125
7.72 Load 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>LDSB</td>
<td>00 1001</td>
<td>Load Signed Byte</td>
<td>ldsb [address], re_rd</td>
<td>A1</td>
</tr>
<tr>
<td>LDSH</td>
<td>00 1010</td>
<td>Load Signed Halfword</td>
<td>ldsh [address], re_rd</td>
<td>A1</td>
</tr>
<tr>
<td>LDSW</td>
<td>00 1000</td>
<td>Load Signed Word</td>
<td>ldsw [address], re_rd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUB</td>
<td>00 0001</td>
<td>Load Unsigned Byte</td>
<td>ldub [address], re_rd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUH</td>
<td>00 0010</td>
<td>Load Unsigned Halfword</td>
<td>lduh [address], re_rd</td>
<td>A1</td>
</tr>
<tr>
<td>LDUW</td>
<td>00 0000</td>
<td>Load Unsigned Word</td>
<td>lduw\† [address], re_rd</td>
<td>A1</td>
</tr>
<tr>
<td>LDX</td>
<td>00 1011</td>
<td>Load Extended Word</td>
<td>ld [address], re_rd</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonym: ld

Description: The load integer instructions copy a byte, a halfword, a word, or an extended word from memory. All copy the fetched value into \(R_{rd}\). A fetched byte, halfword, or word is right-justified in the destination register \(R_{rd}\); it is either sign-extended or zero-filled on the left, depending on whether the opcode specifies a signed or unsigned operation, respectively.

Load integer instructions access memory using the implicit ASI (see page 81). The effective address is \(R_{rs1} + R_{rs2}\) if \(i = 0\), or \(R_{rs1} + \text{sign}\_\text{ext}(\text{simm13})\) if \(i = 1\).

A successful load (notably, load extended) instruction operates atomically.

An attempt to execute a load integer instruction when \(i = 0\) and instruction bits 12-5 are nonzero causes an illegal\_instruction exception.

Any of the following conditions cause a mem\_address\_not\_aligned exception:
- an attempt to execute LDUH or LDSH when the effective address is not halfword-aligned
- an attempt to execute LDUW or LDSW when the effective address is not word-aligned
- an attempt to execute LDX when the effective address is not doubleword-aligned

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 244 for details.

Exceptions:
- illegal\_instruction
- mem\_address\_not\_aligned (all except LDSB, LDUB)
- VA\_watchpoint
- DAE\_privilege\_violation
- DAE\_nfo\_page
7.73 Load Integer from Alternate Space

**Description**

The load integer from alternate space instructions copy a byte, a halfword, a word, or an extended word from memory. All copy the fetched value into $R[rd]$. A fetched byte, halfword, or word is right-justified in the destination register $R[rd]$; it is either sign-extended or zero-filled on the left, depending on whether the opcode specifies a signed or unsigned operation, respectively.

The load integer from alternate space instructions contain the address space identifier (ASI) to be used for the load in the `imm_asi` field if $i = 0$, or in the ASI register if $i = 1$. The 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 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 246 for details.

Any of the following conditions cause a `mem_address_not_aligned` exception:

- an attempt to execute LDUHA or LDSHA when the effective address is not halfword-aligned
- an attempt to execute LDUWA or LDSWA when the effective address is not word-aligned
- an attempt to execute LDXA when the effective 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.
LDA

LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA can be used with any of the following ASIs, subject to the privilege mode rules described for the privileged_action exception above. Use of any other ASI with these instructions causes a DAE_invalid_asi exception.

### ASIs valid for LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA

<table>
<thead>
<tr>
<th>ASI Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0416</td>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td>1016</td>
<td>ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>1116</td>
<td>ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>1416</td>
<td>ASI_REAL</td>
</tr>
<tr>
<td>1516</td>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>8016</td>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td>8116</td>
<td>ASI_SECONDARY</td>
</tr>
<tr>
<td>8216</td>
<td>ASI_PRIMARY_NO_FAULT</td>
</tr>
<tr>
<td>8316</td>
<td>ASI_SECONDARY_NO_FAULT</td>
</tr>
</tbody>
</table>

LDXA can be used with any ASI (including, but not limited to, the above two lists), unless it either (a) violates the privilege mode rules described for the privileged_action exception above or (b) is used with any of the following ASIs, which causes a DAE_invalid_asi exception.

### ASIs invalid for LDXA (cause DAE_invalid_asi exception)

<table>
<thead>
<tr>
<th>ASI Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2216</td>
<td>ASI_TWIX_AIUP</td>
</tr>
<tr>
<td>2316</td>
<td>ASI_TWIX_AIUS</td>
</tr>
<tr>
<td>2616</td>
<td>ASI_TWIX_REAL</td>
</tr>
<tr>
<td>2716</td>
<td>ASI_TWIX_N</td>
</tr>
<tr>
<td>C016</td>
<td>ASI_PST8_PRIMARY*</td>
</tr>
<tr>
<td>C116</td>
<td>ASI_PST8_SECONDARY*</td>
</tr>
<tr>
<td>C216</td>
<td>ASI_PST16_PRIMARY*</td>
</tr>
<tr>
<td>C316</td>
<td>ASI_PST16_SECONDARY*</td>
</tr>
<tr>
<td>C416</td>
<td>ASI_PST32_PRIMARY*</td>
</tr>
<tr>
<td>C516</td>
<td>ASI_PST32_SECONDARY*</td>
</tr>
<tr>
<td>D016</td>
<td>ASI_FL8_PRIMARY</td>
</tr>
<tr>
<td>D116</td>
<td>ASI_FL8_SECONDARY</td>
</tr>
<tr>
<td>D216</td>
<td>ASI_FL16_PRIMARY</td>
</tr>
<tr>
<td>D316</td>
<td>ASI_FL16_SECONDARY</td>
</tr>
<tr>
<td>E016</td>
<td>ASI_BLOCK_COMMIT_PRIMARY</td>
</tr>
<tr>
<td>E216</td>
<td>ASI_TWIX_P</td>
</tr>
<tr>
<td>E316</td>
<td>ASI_TWIX_S</td>
</tr>
<tr>
<td>F016</td>
<td>ASI_BLOCK_COMMIT_SECONDARY</td>
</tr>
<tr>
<td>F116</td>
<td>ASI_BLOCK_SECONDARY</td>
</tr>
</tbody>
</table>

### Exceptions

- mem_address_not_aligned (all except LDSBA and LDUBA)
- privileged_action
- VA_watchpoint
- DAE_invalid_asi
- DAE_privilege_violation
- DAE_nfo_page
- DAE_side_effect_page

### See Also
- LD on page 227
- STA on page 320
7.74 Block Load [VIS 1]

The LDBLOCKF instructions are deprecated and should not be used in new software. A sequence of LDDF instructions should be used instead.

The LDBLOCKF instruction is intended to be a processor-specific instruction, which may or may not be implemented in future Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

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

Description

A block load (LDBLOCKF) instruction uses one of several special block-transfer ASIs. Block transfer ASIs allow block loads to be performed accessing the same address space as normal loads. Little-endian ASIs (those with an ‘L’ suffix) access data in little-endian format; otherwise, the access is assumed to be big-endian. Byte swapping is performed separately for each of the eight 64-bit (double-precision) F registers used by the instruction.

A block load instruction loads 64 bytes of data from a 64-byte aligned memory area into the eight double-precision floating-point registers specified by rd. The lowest-addressed eight bytes in memory are loaded into the lowest-numbered 64-bit (double-precision) destination F register.

A block load only guarantees atomicity for each 64-bit (8-byte) portion of the 64 bytes it accesses.

Programming Note

The block load instruction, LDBLOCKFD, and its companion, STBLOCKFD, were originally defined to provide a fast mechanism for block-copy operations. However, in modern implementations they are rarely much faster than a sequence of regular loads and stores, so are now deprecated.
The following aspects of the behavior of block load (LDBLOCKF) instructions are implementation dependent:

- What memory ordering model is used by LDBLOCKFD (LDBLOCKFD is not required to follow TSO memory ordering)
- Whether LDBLOCKFD 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 LDBLOCKFD appears to execute out of order, or follow LoadLoad ordering (with respect to older loads, younger loads, and other LDBLOCKFs)
- Whether LDBLOCKFD follows register-dependency interlocks, as do ordinary load instructions
- Whether VA_watchpoint exceptions are recognized on accesses to all 64 bytes of a LDBLOCKFD (the recommended behavior), or only on the first eight bytes
- Whether the MMU ignores the side-effect bit (TTE.e) for LDBLOCKFD accesses

For further restrictions on the behavior of the block load instruction, see implementation-specific processor documentation.

An illegal_instruction exception occurs if LDBLOCKF’s floating-point destination registers are not aligned on an eight-double-precision register boundary.

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

If the least significant 6 bits of the effective memory address in an LDBLOCKFD instruction are nonzero, a mem_address_not_aligned exception occurs.

In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0 (ASIs 16, 17, 1E, and 1F), LDBLOCKFD causes a privileged_action exception.

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

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

**Implementation**

LDBLOCKF shares an opcode with LDDFA and LDSHORTF; they are distinguished by the ASI used.

**Note**

**Exceptions**

- illegal_instruction
- fp_disabled
- mem_address_not_aligned
- privileged_action
- VA_watchpoint (impl. dep. #410-S10)
- DAE_privilege_violation
- DAE_nc_page

**See Also**

LDDF on page 233
STBLOCKFD on page 322
### 7.75 Load Floating-Point Register

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

† Encoded floating-point register value, as described on page 51.

### Description

The load single floating-point instruction (LDF) copies a word from memory into a 32-bit floating-point destination register F<sub>S</sub>[rd].

The load doubleword floating-point instruction (LDDF) copies a word-aligned doubleword from memory into a 64-bit floating-point destination register, F<sub>D</sub>[rd]. The unit of atomicity for LDDF is 4 bytes (one word).

The load quad floating-point instruction (LDQF) copies a word-aligned quadword from memory into a 128-bit floating-point destination register, F<sub>Q</sub>[rd]. The unit of atomicity for LDQF is 4 bytes (one word).

These load floating-point instructions access memory using the implicit ASI (see page 81).

If i = 0, the effective address for these instructions is “R[rs1] + R[rs2]” and if i = 1, the effective address is “R[rs1] + sign_ext(simm13)”.

### Exceptions

An attempt to execute an LDF, LDDF, or LDQF instruction when i = 0 and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

If the FPU is not enabled (FPRS.<sub>f</sub>e = 0 or PSTATE.<sub>p</sub>e = 0) or if no FPU is present, an attempt to execute an LDF, LDDF, or LDQF instruction causes an fp_disabled exception.

Any of the following conditions cause an exception:

- an attempt to execute LDF, LDDF, or LDQF when the effective address is not word-aligned causes a mem_address_not_aligned exception
- an attempt to execute LDDF when the effective address is word-aligned but not doubleword-aligned causes an LDDF_mem_address_not_aligned exception. In this case, trap handler software must emulate the LDDF instruction and return ((impl. dep. #109-V9-Cs10(a))).
- an attempt to execute LDQF when the effective address is word-aligned but not quadword-aligned causes an LDQF_mem_address_not_aligned exception. In this case, trap handler software must emulate the LDQF instruction and return (impl. dep. #111-V9-Cs10(a)).

### Programming Note

Some compilers issued sequences of single-precision loads for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9 processors, since emulation of misaligned loads is expected to be fast, compilers should issue sets of single-precision loads only when they can determine that doubleword or quadword operands are not properly aligned.
LDF / LDDF / LDQF

An attempt to execute an LDQF instruction when \( rd[1] \neq 0 \) causes an `fp_exception_other` (FSR.ftt = invalid_fp_register) exception.

**Implementation**

Since Oracle SPARC Architecture 2011 processors do not implement in hardware instructions (including LDQF) that refer to quad-precision floating-point registers, the `LDQF_mem_address_not_aligned` and `fp_exception_other` (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an `illegal_instruction` exception and subsequent trap.

**Destination Register(s) when Exception Occurs.** If a load floating-point instruction generates an exception that causes a precise trap, the destination floating-point register(s) remain unchanged.

**IMPL. DEP. #44-V8-Cs10(a)(1):** If a load floating-point instruction generates an exception that causes a non-precise trap, the contents of the destination floating-point register(s) remain unchanged or are undefined.

**Exceptions**

`illegal_instruction`
`fp_disabled`
`LDDF_mem_address_not_aligned`
`LDQF_mem_address_not_aligned` (not used in Oracle SPARC Architecture 2011)
`mem_address_not_aligned`
`fp_exception_other` (FSR.ftt = invalid_fp_register (LDQF only))
`VA_watchpoint`
`DAE_privilege_violation`
`DAE_nfo_page`

**See Also**

- Load Floating-Point from Alternate Space on page 235
- Load Floating-Point State Register (Lower) on page 238
- Store Floating-Point on page 325
7.76 Load Floating-Point from Alternate Space

**Description**

The load single floating-point from alternate space instruction (LDFA) copies a word from memory into 32-bit floating-point destination register \( F_S[rd] \).

The load double floating-point from alternate space instruction (LDDFA) copies a word-aligned doubleword from memory into a 64-bit floating-point destination register, \( F_D[rd] \). The unit of atomicity for LDDFA is 4 bytes (one word).

The load quad floating-point from alternate space instruction (LDQFA) copies a word-aligned quadword from memory into a 128-bit floating-point destination register, \( F_Q[rd] \). The unit of atomicity for LDQFA is 4 bytes (one word).

If \( i = 0 \), these instructions contain the address space identifier (ASI) to be used for the load in the \( \text{imm}_\text{asi} \) field and the effective address for the instruction is \( \text{"R}[rs1] + \text{R}[rs2]" \). If \( i = 1 \), the ASI to be used is contained in the ASI register and the effective address for the instruction is \( \text{"R}[rs1] + \text{sign_ext}(\text{simm13})" \).

**Exceptions.** 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 LDFA, LDDFA, or LDQFA instruction causes an \text{fp_disabled} exception.

**V9 Compatibility** | LDFA, LDDFA, and LDQFA cause a \text{privileged_action} exception if PSTATE.\text{priv} = 0 and bit 7 of the ASI is 0.

Any of the following conditions cause an exception:

- An attempt to execute LDFA, LDDFA, or LDQFA when the effective address is not word-aligned causes a \text{mem_address_not_aligned} exception.
- An attempt to execute LDDFA when the effective address is word-aligned but not doubleword-aligned causes an \text{LDDF_mem_address_not_aligned} exception. In this case, trap handler software must emulate the LDDFA instruction and return ((impl. dep. \#109-V9-Cs10(a))).
- An attempt to execute LDQFA when the effective address is word-aligned but not quadword-aligned causes an \text{LDQF_mem_address_not_aligned} exception. In this case, trap handler software must emulate the LDQFA instruction and return (impl. dep. \#111-V9-Cs10(a)).

An attempt to execute an LDQFA instruction when \( \text{rd}[1] \neq 0 \) causes an \text{fp_exception_other} (with FSR.\text{ftt} = \text{invalid_fp_register}) exception.
LDFA / LDDFA / LDQFA

Implementation Note
Since Oracle SPARC Architecture 2011 processors do not implement in hardware instructions (including LDQFA) that refer to quad-precision floating-point registers, the `LDQF_mem_address_not_aligned` and `fp_exception_other` (with `FSR.ftt = invalid_fp_register`) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an `illegal_instruction` exception and subsequent trap.

Programming Note
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.

LDFA and LDQFA can be used with any of the following ASIs, subject to the privilege mode rules described for the `privileged_action` exception above. Use of any other ASI with these instructions causes a `DAE_invalid_asi` exception.

<table>
<thead>
<tr>
<th>ASIs valid for LDFA and LDQFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>04_{16} ASI_NUCLEUS</td>
</tr>
<tr>
<td>10_{16} ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>11_{16} ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>14_{16} ASI_REAL</td>
</tr>
<tr>
<td>15_{16} ASI_REAL_IO</td>
</tr>
<tr>
<td>80_{16} ASI_PRIMARY</td>
</tr>
<tr>
<td>81_{16} ASI_SECONDARY</td>
</tr>
<tr>
<td>82_{16} ASI_PRIMARY_NO_FAULT</td>
</tr>
<tr>
<td>83_{16} ASI_SECONDARY_NO_FAULT</td>
</tr>
</tbody>
</table>

LDDFA can be used with any of the following ASIs, subject to the privilege mode rules described for the `privileged_action` exception above. Use of any other ASI with the LDDFA instruction causes a `DAE_invalid_asi` exception.

<table>
<thead>
<tr>
<th>ASIs valid for LDDFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>04_{16} ASI_NUCLEUS</td>
</tr>
<tr>
<td>10_{16} ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>11_{16} ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>14_{16} ASI_REAL</td>
</tr>
<tr>
<td>15_{16} ASI_REAL_IO</td>
</tr>
<tr>
<td>80_{16} ASI_PRIMARY</td>
</tr>
<tr>
<td>81_{16} ASI_SECONDARY</td>
</tr>
<tr>
<td>82_{16} ASI_PRIMARY_NO_FAULT</td>
</tr>
<tr>
<td>83_{16} ASI_SECONDARY_NO_FAULT</td>
</tr>
</tbody>
</table>

Behavior with Block-Store-with-Commit ASIs. ASIs `E0_{16}` and `E1_{16}` are only defined for use in Block Store with Commit operations (see page 322). Neither ASI `E0_{16}` nor `E1_{16}` should be used with LDDFA; however, if it is used, the LDDFA behaves as follows:
1. If an LDDFA opcode is used with an ASI of E0\textsubscript{16} or E1\textsubscript{16} and a destination register number \(\text{rd}\) is specified which is not a multiple of 8 (“misaligned” \(\text{rd}\)), an Oracle SPARC Architecture 2011 virtual processor generates an \textit{illegal_instruction} exception (impl. dep. #255-U3-Cs10).

2. **IMPL. DEP. #256-U3**: If an LDDFA opcode is used with an ASI of E0\textsubscript{16} or E1\textsubscript{16} and a memory address is specified with less than 64-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the exception generated is \textit{DAE_invalid_asi}, \textit{mem_address_not_aligned}, or \textit{LDDF_mem_address_not_aligned}.

3. If both \(\text{rd}\) and the memory address are correctly aligned, a \textit{DAE_invalid_asi} exception occurs.

**Behavior with Partial Store ASIs.** ASIs C0\textsubscript{16}–C5\textsubscript{16} and C8\textsubscript{16}–CD\textsubscript{16} are only defined for use in Partial Store operations (see page 332). None of them should be used with LDDFA; however, if any of those ASIs is used with LDDFA, the LDDFA behaves as follows:

1. **IMPL. DEP. #257-U3**: If an LDDFA opcode is used with an ASI of C0\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 generated exception is a \textit{DAE_invalid_asi}, \textit{mem_address_not_aligned}, or \textit{LDDF_mem_address_not_aligned} exception.

2. If the memory address is correctly aligned, the virtual processor generates a \textit{DAE_invalid_asi}.

**Destination Register(s) when Exception Occurs.** If a load floating-point alternate instruction generates an exception that causes a precise trap, the destination floating-point register(s) remain unchanged.

**IMPL. DEP. #44-V8-Cs10(b)**: If a load floating-point alternate instruction generates an exception that causes a non-precise trap, it is implementation dependent whether the contents of the destination floating-point register(s) are undefined or are guaranteed to remain unchanged.

**Implementation Note**

LDDFA shares an opcode with the LDBLOCKFD and LDSHORTF instructions; they are distinguished by the ASI used.

**Exceptions**

- \textit{illegal_instruction}
- \textit{fp_disabled}
- \textit{LDDF_mem_address_not_aligned}
- \textit{LDQF_mem_address_not_aligned} (not generated in Oracle SPARC Architecture 2011)
- \textit{mem_address_not_aligned}
- \textit{fp_exception_other} (FSR.flt = invalid_fp_register (LDQFA only))
- \textit{privileged_action}
- \textit{VA_watchpoint}
- \textit{DAE_invalid_asi}
- \textit{DAE_privilege_violation}
- \textit{DAE_nfo_page}
- \textit{DAE_side_effect_page}

**See Also**

- Load Floating-Point Register on page 233
- Block Load on page 230
- Store Short Floating-Point on page 335
- Store Floating-Point into Alternate Space on page 327
7.77 Load Floating-Point State Register (Lower)

The LDFSR instruction is deprecated and should not be used in new software. The LDXFSR instruction should be used instead.

<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 0</td>
<td>Load Floating-Point State Register (Lower)</td>
<td>ld [address], %fsr</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 0001 1-31</td>
<td>(see page 251)</td>
<td></td>
<td></td>
<td></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 more-significant 32 bits of FSR are unaffected by LDFSR. LDFSR does not alter the ver, fft, qne, reserved, or unimplemented (for example, ns) fields of FSR (see page 42).

**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 81).

An attempt to execute an LDFSR instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an `illegal_instruction` exception.

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

LDFSR causes a `mem_address_not_aligned` exception if the effective memory address is not word-aligned.

**V8 Compatibility Note**

The SPARC V9 architecture supports two different instructions to load the FSR: the (deprecated) SPARC V8 LDFSR instruction is defined to load only the less-significant 32 bits of the FSR, whereas LDXFSR allows SPARC V9 programs to load all 64 bits of the FSR.

**Implementation Note**

LDFSR shares an opcode with the LDXFSR and LDXEFSR instructions (and possibly with other implementation-dependent instructions); they are differentiated by the instruction \( rd \) field. An attempt to execute the \( op = 112, op3 = 10,00012 \) opcode with an invalid \( rd \) value causes an `illegal_instruction` exception.

**Exceptions**

- `illegal_instruction`
- `fp_disabled`
- `mem_address_not_aligned`
- `VA_watchpoint`
LDFSR (Deprecated)

DAE_privilege_violation
DAE_nfo_page

See Also
Load Floating-Point Register on page 233
Load Floating-Point State Register on page 251
Store Floating-Point on page 325

DAE_invalid_asi e
### 7.78 Load 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>LDSHORTF D016</td>
<td>8-bit load from primary address space</td>
<td>ldda [regaddr] #ASI_FL8_P, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D116</td>
<td>8-bit load from secondary address space</td>
<td>ldda [regaddr] #ASI_FL8_S, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D816</td>
<td>8-bit load from primary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL8_PL, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D916</td>
<td>8-bit load from secondary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL8_SL, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D216</td>
<td>16-bit load from primary address space</td>
<td>ldda [regaddr] #ASI_FL16_P, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF D316</td>
<td>16-bit load from secondary address space</td>
<td>ldda [regaddr] #ASI_FL16_S, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF DA16</td>
<td>16-bit load from primary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL16_PL, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>LDSHORTF DB16</td>
<td>16-bit load from secondary address space, little-endian</td>
<td>ldda [regaddr] #ASI_FL16_SL, freg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>B1</td>
<td></td>
</tr>
</tbody>
</table>

### Description
Short floating-point load instructions allow an 8- or 16-bit value to be loaded from memory into a 64-bit floating-point register.

An 8-bit load places the loaded value in the least significant byte of F<sub>D</sub>[rd] and zeroes in the most-significant seven bytes of F<sub>D</sub>[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<sub>D</sub>[rd] and zeroes in the more-significant six bytes of F<sub>D</sub>[rd].

Little-endian ASIs transfer data in little-endian format from memory; otherwise, memory is assumed to be in big-endian byte order.

#### Programming Note
LDSHORTF is typically used with the FALIGNDATAg instruction (see Align Data (using GSR.align) on page 151) to assemble or store 64 bits from noncontiguous components.

#### Implementation Note
LDSHORTF shares an opcode with the LDBLOCKF<sub>D</sub> and LDDFA instructions; they are distinguished by the ASI used.

### Exceptions
If the FPU is not enabled (FPRS.<sub>le</sub> = 0 or PSTATE.<sub>p</sub> = 0) or if no FPU is present, an attempt to execute an LDSHORTF instruction causes an fp_disabled exception.

An attempt to execute a 16-bit (halfword) LDSHORTF instruction when the effective address is not halfword-aligned causes a mem_address_not_aligned exception.
Exceptions

- fp_disabled
- mem_address_not_aligned
- VA_watchpoint
- DAE_privilege_violation
- DAE_nfo_page

See Also

STSHORTF on page 335
Align Data (using GSR.align) on page 151
7.79 Load-Store Unsigned Byte

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 81). The effective address for this instruction is “\( R[rs1] + R[rs2] \)” if \( i = 0 \), or “\( R[rs1] + \text{sign_ext}(\text{simm13}) \)” if \( i = 1 \).

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

An attempt to execute an LDSTUB instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

### Exceptions

- illegal_instruction
- VA_watchpoint
- DAE_privilege_violation
- DAE_nc_page
- DAE_nfo_page

### See Also

- CASA on page 131
- LDSTUBA on page 243
- SWAP on page 343
7.80 Load-Store Unsigned Byte to Alternate Space

**Description**

The load-store unsigned byte into alternate space instruction copies a byte from memory into R[rd], then rewrites the addressed byte in memory to all 1's. The fetched byte is right-justified in the destination register R[rd] and zero-filled on the left.

The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing LDSTUB, LDSTUBA, CASA, CASXA, SWAP, or SWAPA instructions addressing all or parts of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

If \( i = 0 \), LDSTUBA contains the address space identifier (ASI) to be used for the load in the \( \text{imm}_{\text{asi}} \) field. If \( i = 1 \), the ASI is found in the ASI register. In nonprivileged mode (PSTATE.\( \text{priv} = 0 \)), if bit 7 of the ASI is 0, this instruction causes a privileged_action exception. In privileged mode (PSTATE.\( \text{priv} = 1 \)), if the ASI is in the range 30 to 7F, this instruction causes a privileged_action exception.

LDSTUBA can be used with any of the following ASIs, subject to the privilege mode rules described for the privileged_action exception above. Use of any other ASI with this instruction causes a DAE_invalidasi exception.

### ASIs valid for LDSTUBA

- \( 04_{16} \text{ASI_NUCLEUS} \)
- \( 10_{16} \text{ASI_AS_IF_USER_PRIMARY} \)
- \( 11_{16} \text{ASI_AS_IF_USER_SECONDARY} \)
- \( 14_{16} \text{ASI_REAL} \)
- \( 80_{16} \text{ASI_PRIMARY} \)
- \( 81_{16} \text{ASI_SECONDARY} \)
- \( 0C_{16} \text{ASI_NUCLEUS_LITTLE} \)
- \( 18_{16} \text{ASI_AS_IF_USER_PRIMARY_LITTLE} \)
- \( 19_{16} \text{ASI_AS_IF_USER_SECONDARY_LITTLE} \)
- \( 1C_{16} \text{ASI_REAL_LITTLE} \)
- \( 88_{16} \text{ASI_PRIMARY_LITTLE} \)
- \( 89_{16} \text{ASI_SECONDARY_LITTLE} \)

### Exceptions

- privileged_action
- VA_watchpoint
- DAE_invalidasi
- DAE_privilege_violation
- DAE_nc_page
- DAE_nfo_page

### See Also

- CASA on page 131
- LDSTUB on page 242
- SWAP on page 343
- SWAPA on page 344
### 7.81 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.

<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>LDTWD</td>
<td>00 0011</td>
<td>Load Integer Twin Word</td>
<td>ldtw [address], regrd</td>
<td>D2</td>
</tr>
</tbody>
</table>

† The original assembly language syntax for this instruction used an “ldd” instruction mnemonic, which is now deprecated. Over time, assemblers will support the new “ldtw” mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original “ldd” mnemonic.

**Description**

The load integer twin word instruction (LDTW) copies two words (with doubleword alignment) from memory into a pair of R registers. The word at the effective memory address is copied into the least significant 32 bits of the even-numbered R register. The word at the effective memory address + 4 is copied into the least significant 32 bits of the following odd-numbered R register. The most significant 32 bits of both the even-numbered and odd-numbered R registers are zero-filled.

**Note**

Execution of an LDTW instruction with rd = 0 modifies only R[1].

Load integer twin word instructions access memory using the implicit ASI (see page 81). If i = 0, the effective address for these instructions is “R[rs1] + R[rs2]” and if i = 1, the effective address is “R[rs1] + sign_ext(simm13)”.

With respect to little endian memory, an LDTW instruction behaves as if it comprises two 32-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

**IMPL. DEP. #107-V9a:** It is implementation dependent whether LDTW is implemented in hardware. If not, an attempt to execute an LDTW instruction will cause an `unimplemented_LDTW` exception.

**Programming Note**

LDTW is provided for compatibility with existing SPARC V8 software. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties.

**SPARC V9 Compatibility Note**

LDTW was (inaccurately) named LDD in the SPARC V8 and SPARC V9 specifications. It does not load a doubleword; it loads two words (into two registers), and has been renamed accordingly.

The least significant bit of the rd field in an LDTW instruction is unused and should always be set to 0 by software. An attempt to execute an LDTW instruction that refers to a misaligned (odd-numbered) destination register causes an `illegal_instruction` exception.

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

If the effective address is not doubleword-aligned, an attempt to execute an LDTW instruction causes a `mem_address_not_aligned` exception.
LDTW (Deprecated)

A successful LDTW instruction operates atomically.

**Programming Notes**

LDTW 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 LDTW.

If LDTW is emulated in software, an LDX instruction should be used for the memory access in the emulation code to preserve atomicity. Emulation software should examine TSTATE[TL].pstate.cle (and, if appropriate, TTE.ie) to determine the endianness of the emulated memory access.

Note that the value of TTE.ie is not saved during a trap. Therefore, if it is examined in the emulation trap handler, that should be done as quickly as possible, to minimize the window of time during which the value of TTE.ie could possibly be changed from the value it had at the time of the attempted execution of LDTW.

**Exceptions**

unimplemented_LDTW (not used in Oracle SPARC Architecture 2011)
illegal_instruction
mem_address_not_aligned
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

**See Also**

LDW/LDX on page 227
STTW on page 337
7.82 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.

<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>LDTWA^D, PASI</td>
<td>01 0011</td>
<td>Load Integer Twin Word from Alternate Space</td>
<td>ldtwa [regaddr] imm_asi, reg^rd</td>
<td>D2, Y3‡</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ldtwa [reg_plus_imm] %asi, reg^rd</td>
<td></td>
</tr>
</tbody>
</table>

^ The original assembly language syntax for this instruction used an “ldda” instruction mnemonic, which is now deprecated. Over time, assemblers will support the new “ldtwa” mnemonic for this instruction. In the meantime, some assemblers may only recognize the original “ldda” mnemonic.

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

Description

The load integer twin word from alternate space instruction (LDTWA) copies two 32-bit words from memory (with doubleword memory alignment) into a pair of R registers. The word at the effective memory address is copied into the least significant 32 bits of the even-numbered R register. The word at the effective memory address + 4 is copied into the least significant 32 bits of the following odd-numbered R register. The most significant 32 bits of both the even-numbered and odd-numbered R registers are zero-filled.

Note

Execution of an LDTWA instruction with rd = 0 modifies only R[1].

If i = 0, the LDTWA instruction contains the address space identifier (ASI) to be used for the load in its imm_asi field and the effective address for the instruction is “R[rs1] + R[rs2]”. If i = 1, the ASI to be used is contained in the ASI register and the effective address for the instruction is “R[rs1] + sign_ext (simm13)”. With respect to little endian memory, an LDTWA instruction behaves as if it is composed of two 32-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

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 Notes

LDTWA 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 LDTWA.

If LDTWA is emulated in software, an LDXA instruction should be used for the memory access in the emulation code to preserve atomicity. Emulation software should examine TSTATE[TL].pstate.cl (and, if appropriate, TTE.ie) to determine the endianness of the emulated memory access.
LDTWA (Deprecated)

Note that the value of TTE.i.e is not saved during a trap. Therefore, if it is examined in the emulation trap handler, that should be done as quickly as possible, to minimize the window of time during which the value of TTE.i.e could possibly be changed from the value it had at the time of the attempted execution of LDTWA.

The least significant bit of the rd field in an LDTWA instruction is unused and should always be set to 0 by software. An attempt to execute an LDTWA instruction that refers to a misaligned (odd-numbered) destination register causes an illegal_instruction exception.

If the effective address is not doubleword-aligned, an attempt to execute an LDTWA instruction causes a mem_address_not_aligned exception.

A successful LDTWA instruction operates atomically.

LDTWA causes a mem_address_not_aligned exception if the address is not doubleword-aligned.

In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0, these instructions cause a privileged_action exception. In privileged mode (PSTATE.priv = 1), if the ASI is in the range 3016 to 7F16, these instructions cause a privileged_action exception.

LDTWA can be used with any of the following ASIs, subject to the privilege mode rules described for the privileged_action exception above. Use of any other ASI with this instruction causes a DAE_invalid_asi exception (impl. dep. #300-U4-Cs10).

<table>
<thead>
<tr>
<th>ASIs valid for LDTWA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0416 ASI_NUCLEUS</td>
</tr>
<tr>
<td>1016 ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>1116 ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>1416 ASI_REAL</td>
</tr>
<tr>
<td>1516 ASI_REAL_IO</td>
</tr>
<tr>
<td>2216‡ (ASI_TWINX_AIUP)</td>
</tr>
<tr>
<td>2316‡ (ASI_TWINX_AIUS)</td>
</tr>
<tr>
<td>2616‡ (ASI_TWINX_REAL)</td>
</tr>
<tr>
<td>2716‡ (ASI_TWINX_N)</td>
</tr>
<tr>
<td>8016 ASI_PRIMARY</td>
</tr>
<tr>
<td>8116 ASI_SECONDARY</td>
</tr>
<tr>
<td>8216 ASI_PRIMARY_NO_FAULT</td>
</tr>
<tr>
<td>8316 ASI_SECONDARY_NO_FAULT</td>
</tr>
<tr>
<td>E216‡ (ASI_TWINX_P)</td>
</tr>
<tr>
<td>E316‡ (ASI_TWINX_S)</td>
</tr>
</tbody>
</table>

‡ 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 249.

If this ASI is used with the opcode for LDTWA and i = 1, a DAE_invalid_asi exception occurs.

Programming Note Nontranslating ASIs (see page 397) should only be accessed using LDXA (not LDTWA) instructions. If an LDTWA referencing a nontranslating ASI is executed, per the above table, it generates a DAE_invalid_asi exception (impl. dep. #300-U4-Cs10).
LDTWA (Deprecated)

**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 249.

**Exceptions**
- **unimplemented_LDTW** (not used in Oracle SPARC Architecture 2011)
- illegal_instruction
- mem_address_not_aligned
- privileged_action
- VA_watchpoint
- DAE_invalid_asi
- DAE_privilegeViolation
- DAE_nfo_page
- DAE_side_effect_page

**See Also**
- LDWA/LDXA on page 228
- LDTXA on page 249
- STIWA on page 339
7.83 Load Integer Twin Extended Word from Alternate Space [VIS 2+]

The LDTXA instructions are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

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

† The original assembly language syntax for these instructions used the “ldda” instruction mnemonic. That syntax is now deprecated. Over time, assemblers will support the new “ldtxa” mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original “ldda” mnemonic.

Description

ASIs 2616, E216, E316, EA16, and EB16 are used with the LDTXA instruction to atomically read a 128-bit data item into a pair of 64-bit R registers (a “twin extended word”). The data are placed in an even/odd pair of 64-bit registers. The lowest-address 64 bits are placed in the even-numbered register; the highest-address 64 bits are placed in the odd-numbered register.

Note | Execution of an LDTXA instruction with rd = 0 modifies only R[1].
ASIs E216, E316, EA16, and EB16 perform an access using a virtual address, while ASIs 2616 and 2E16 use a real address.

An LDTXA instruction that performs a little-endian access behaves as if it comprises two 64-bit loads (performed atomically), each of which is byte-swapped independently before being written into its respective destination register.

Exceptions. An attempt to execute an LDTXA instruction with an odd-numbered destination register (rd[0] = 1) causes an illegal_instruction exception.

An attempt to execute an LDTXA instruction with an effective memory address that is not aligned on a 16-byte boundary causes a mem_address_not_aligned exception.

IMPL. DEP. #413-S10: It is implementation dependent whether VA_watchpoint exceptions are recognized on accesses to all 16 bytes of a LDTXA instruction (the recommended behavior) or only on accesses to the first 8 bytes.

An attempted access by an LDTXA instruction to noncacheable memory causes an a DAE_nc_page exception (impl. dep. #306-U4-Cs10).

- Programming Note: A key use for this instruction is to read a full TTE entry (128 bits, tag and data) in a TSB directly, without using software interlocks. The “real address” variants can perform the access using a real address, bypassing the VA-to-RA translation.

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.

- Implementation Note: LDTXA shares an opcode with the “i = 0” variant of the (deprecated) LDTWA instruction; they are differentiated by the combination of the i instruction field and the ASI used in the instruction. See Load Integer Twin Word from Alternate Space on page 246.

Exceptions

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

See Also

- LDTWA on page 246
7.84 Load Floating-Point State Register

**Description**
A load floating-point state register instruction (LDXFSR or LDXEFSR) 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, ft, qne, reserved, or unimplemented (for example, ns) fields of FSR (see page 42).

An LDXEFSR instruction loads from memory the “entire” FSR, including FSR.ft. However, it does not alter the ver, qne, reserved, or unimplemented (for example, ns) fields of FSR; writes to those fields are ignored.

**Programming Note**
For future compatibility, software should only issue an LDXFSR or LDXEFSR instruction with a zero value (or a value previously read from the same field) written into any reserved field of FSR.

LDXFSR and LDXEFSR access memory using the implicit ASI (see page 81).

If \( i = 0 \), the effective address for these instructions is \( \text{"R}[rs1] + \text{R}[rs2]" \) and if \( i = 1 \), the effective address is \( \text{"R}[rs1] + \text{sign_ext}(\text{simm13})" \).

**Exceptions.** An attempt to execute an instruction encoded as \( \text{op} = 2 \) and \( \text{op3} = 21_{16} \) when any of the following conditions exist causes an illegal_instruction exception:

- \( i = 0 \) and instruction bits 12:5 are nonzero
- \( \text{rd} = 2 \) or \( \text{rd} \neq 3 \)

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

If the effective address is not doubleword-aligned, an attempt to execute an LDXFSR or LDXEFSR instruction causes a mem_address_not_aligned exception.

**Destination Register(s) when Exception Occurs.** If a load floating-point state register instruction generates an exception that causes a precise trap, the destination register (FSR) remains unchanged.

**IMPL. DEP. #44-V8-Cs10(a)(2):** If an LDXFSR or LDXEFSR instruction generates an exception that causes a non-precise trap, it is implementation dependent whether the contents of the destination register (FSR) is undefined or is guaranteed to remain unchanged.
**LDXEFSR / LDXFSR**

**Implementation Note**
LDXFSR and LDXEFSR share an opcode with the (deprecated) LDFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = \(11_2\), op3 = 10 0001_2 opcode with an invalid rd value causes an illegal_instruction exception.

**Exceptions**
- illegal_instruction
- fp_disabled
- mem_address_not_aligned
- VA_watchpoint
- DAE_privilege_violation
- DAE_nfo_page

**See Also**
- Load Floating-Point Register on page 233
- Load Floating-Point State Register (Lower) on page 238
- Store Floating-Point State Register on page 341
7.85 Leading Zeroes Count [VIS 3]

The LZCNT instruction is new and is not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZCNT</td>
<td>0001 0111</td>
<td>Leading zeroes count (from left end) —</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>$\text{lzcnt } r_2 , r_3$ (or $\text{lzd}$)</td>
<td>C1</td>
<td>OSA 2011</td>
</tr>
</tbody>
</table>

† The original assembly language syntax for this instruction used an "lzd" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "lzcnt" mnemonic for this instruction. In the meantime, some assemblers may only recognize the original "lzd" mnemonic.

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

**Description**
The number of leading zeros (0 to 64) in $R[rs2]$ is counted and written as the value $n$ in the least significant 7 bits of the destination register, $R[rd]$. The most significant 57 bits of $R[rd]$ bits are written with zeros.

![Diagram of leading zeroes count]

FIGURE 7-44 Leading Zeroes Count

LZCNT can also be used for "leading 1" detection, by taking the 1's complement of the source operand before executing the LZCNT instruction.

**Programming Note**

See the Programming Note on page 286, regarding how to a "Trailing Zeroes Count" operation can be synthesized using POPC.

**Historical Note**
LZCNT’s original name was “LZD” for “Leading Zero Detect”. That was a misnomer, so it was renamed LZCNT. Software tools will continue recognizing the original assembly-language mnemonic.

**Exceptions**
illegal_instruction
See Also  
Population Count on page 286
The Hash instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### Description

The MD5 instruction implements the MD5 algorithm, which is available as RFC-1321 at [http://nist.gov](http://nist.gov).

The hash instructions operate on 64-bit floating-point registers and process an entire 512-bit block. The locations of the Initialization Vector (IV), Data, and Result in the floating-point register file are described below. To compute the hash over multiple blocks, the result from the previous hash instruction is used as the IV for the next block. Software must appropriately pad the final block as specified by the given algorithm.

MD5:

- **IV[127:0]** = (F_D[0] :: F_D[2])
- **Result[127:0]** = (F_D[0] :: F_D[2])

If \( rd \neq 0 \), \( rs1 \neq 0 \), or \( rs2 \neq 0 \), an attempt to execute an MD5 instruction causes an `illegal_instruction` exception.

If CFR.md5 = 0, an attempt to execute an MD5 instruction causes a `compatibility_feature` exception.

### Programming Note

Software must check that CFR.md5 = 1 before executing the MD5 instruction. If CFR.md5 = 0, then software should assume that an attempt to execute the MD5 instruction will either

1. generate an `illegal_instruction` exception because it is not implemented in hardware, or
2. execute, but perform some other operation.

Therefore, if CFR.md5 = 0, software should perform the MD5 operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.

### Exceptions

- `fp_disabled`
7.87 Memory Barrier

<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>MEMBAR</td>
<td>10 1000</td>
<td>Memory Barrier</td>
<td>membar membar_mask</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

The memory barrier instruction, MEMBAR, has two complementary functions: to express order constraints between memory references and to provide explicit control of memory-reference completion. The `membar_mask` field in the suggested assembly language is the concatenation of the `cmask` and `mmask` instruction fields.

MEMBAR introduces an order constraint between classes of memory references appearing before the MEMBAR and memory references following it in a program. The particular classes of memory references are specified by the `mmask` field. Memory references are classified as loads (including instructions `LDSTUB[A]`, `SWAP[A]`, `CASA`, and `CASX[A]`) and stores (including instructions `LDSTUB[A]`, `SWAP[A]`, `CASA`, `CASX[A]`, and `FLUSH`). The `mmask` field specifies the classes of memory references subject to ordering, as described below. MEMBAR applies to all memory operations in all address spaces referenced by the issuing virtual processor, but it has no effect on memory references by other virtual processors. When the `cmask` field is nonzero, completion as well as order constraints are imposed, and the order imposed can be more stringent than that specifiable by the `mmask` field alone.

A load has been performed when the value loaded has been transmitted from memory and cannot be modified by another virtual processor. A store has been performed when the value stored has become visible, that is, when the previous value can no longer be read by any virtual processor. In specifying the effect of MEMBAR, instructions are considered to be executed as if they were processed in a strictly sequential fashion, with each instruction completed before the next has begun.

The `mmask` field is encoded in bits 3 through 0 of the instruction. **TABLE 7-13** specifies the order constraint that each bit of `mmask` (selected when set to 1) imposes on memory references appearing before and after the MEMBAR. From zero to four mask bits may be selected in the `mmask` field.

<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Assembly Language Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mmask[3]</code></td>
<td>#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.</td>
</tr>
<tr>
<td><code>mmask[2]</code></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><code>mmask[1]</code></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><code>mmask[0]</code></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 7-14, specify additional constraints on the order of memory references and the processing of instructions. If cmask is zero, then MEMBAR enforces the partial ordering specified by the mmask field; if cmask is nonzero, then completion and partial order constraints are applied.

**TABLE 7-14 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 must 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>#LookasideD</td>
<td>(Deprecated - software should not use) A store appearing prior to the MEMBAR must complete before any load following the MEMBAR referencing the same address can be initiated. MEMBAR #Lookaside is deprecated and is supported only for legacy code; it should not be used in new software. A slightly more restrictive MEMBAR operation (such as MEMBAR #StoreLoad) should be used, instead. <strong>Implementation Note:</strong> Since #Lookaside is deprecated, implementations are not expected to perform address matching, but instead provide #Lookaside functionality using a more restrictive MEMBAR operation (such as #StoreLoad).</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 . For additional information about the memory models themselves, see Chapter 9, Memory.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

**V9 Compatibility Note** MEMBAR with mmask = 8\textsubscript{16} and cmask = 0\textsubscript{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 #15; MEMBAR is distinguished by rs1 = 15, rd = 0, i = 1, and bit 12 = 0.

### 7.87.1 Memory Synchronization

The Oracle SPARC 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 Oracle SPARC Architecture implementation may define the operation of each MEMBAR variant in any manner that provides the required semantics.
7.87.2 Synchronization of the Virtual Processor

Synchronization of a virtual processor forces all outstanding instructions to be completed and any associated hardware errors to be detected and reported before any instruction after the synchronizing instruction is issued.

Synchronization can be explicitly caused by executing a synchronizing MEMBAR instruction (MEMBAR #Sync) or by executing an LDXA/STXA/LDDFA/STDFA instruction with an ASI that forces synchronization.

Completion of a MEMBAR #Sync instruction does not guarantee that data previously stored has been written all the way out to external memory. Software cannot rely on that behavior. There is no mechanism in the Oracle SPARC Architecture that allows software to wait for all previous stores to be written to external memory.

7.87.3 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 391.
7.88 MONTMUL [Crypto]

This instruction is new and is not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

Description

MONTMUL performs the Montgomery Multiplication shown below with length equal to imm5. The starting locations of the operands are constant in the integer register file (IRF), regardless of size. The starting location of the result is also constant. Below are the locations for N=31. For smaller MONTMULs, the remaining operand locations are not used and the remaining result locations will be unchanged.

FSR.fcc3 is set upon completion of MONTMUL to reflect whether or not a hardware error occurred during execution (see the Programming Note below).

The following pseudo-code specifies the operation of the MONTMUL instruction:

```
MontMul (l_uint A, l_uint B, l_uint N, l_uint *M, l_uint Nprime,
        char Length,l_uint *X) {
    // compute Montgomery Multiplication as described below
    // use M as temporary variable
    // return result in X
    // A,B,N,M,X 64 × (Length+1) bit long
    // Nprime is 64 bits long
    ACCUM ← 0
    for I←0 to Length begin // Length is one less than the number of words
        for j←0 to I-1 begin //skipped on first I iteration
            ACCUM ← ACCUM + (A[j] × B[I-j])
            ACCUM ← ACCUM + (M[j] × N[I-j])
        end
        ACCUM ← ACCUM + (A[I] × B[0])
        M[I] ← ACCUM × Nprime // 64 LSB of accum, store 64 LSB of product
        ACCUM ← ACCUM + (M[I] × N[0])
        ACCUM ← ACCUM >> 64
    end
    for I ← (Length+1) to ((2×Length)+1) begin // skip last I iteration
        for j ← (I-Length) to Length begin
            ACCUM ← ACCUM + (A[j] × B[I-j])
            ACCUM ← ACCUM + (M[j] × N[I-j])
        end
        X[I-Length-1] ← ACCUM // 64 LSB of accum
        ACCUM ← ACCUM >> 64
    end
    ModReduction(ACCUM, X, N, Length) // Reduce the final result
}
```
ModReduction (bit ACCUM, l_uint *X, l_uint N, char Length) {
    // compute (ACCUM | X) mod N
    // return result in X
    // ACCUM 1 bit long
    // N,X 64 x (Length+1) bit long

    if (ACCUM != 0) begin
        for I ← 0 to Length begin // Length is one less than the number of words
            X[I] ← X[I] - N[I]        // Subtraction with borrow
        end
    end else begin
        I ← Length
        while ((I≥0) and (X[I] = N[I])) begin
            I ← I-1
        end
        if ((I<0) or (X[I]>N[I])) begin
            for I ← 0 to Length begin
                X[I] ← X[I] - N[I]  // Subtraction with borrow
            end
        end
    }
}

MONTMUL

<table>
<thead>
<tr>
<th>Operand</th>
<th>Window (CWP value)</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nprime</td>
<td>—</td>
<td>F_D[0]</td>
</tr>
</tbody>
</table>
Exceptions. If \( rd \neq 0 \) or \( rs1 \neq 0 \), an attempt to execute a MONTMUL instruction causes an \texttt{illegal_instruction} exception.

If \( CFR.montmul = 0 \), an attempt to execute an MONTMUL instruction causes a \texttt{compatibility_feature} exception.

Software must check that \( CFR.montmul = 1 \) before executing the MONTMUL instruction. If \( CFR.montmul = 0 \), then software should assume that an attempt to execute the MONTMUL instruction either

1. will generate an \texttt{illegal_instruction} exception because it is not implemented in hardware, or
2. will execute but perform some other operation.

Therefore, if \( CFR.montmul = 0 \), software should perform the MONTMUL operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.

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

The MONTMUL instruction causes a \texttt{fill_n_normal} or \texttt{fill_n_other} exception if CANRESTORE is not equal to NWINDOWS-2. The fill trap handler is called with CWP set to point to the window to be filled, that is, old CWP-CANRESTORE-1. 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 411. The fill trap handler performs a RESTORED and a RETRY as part of filling the window. When the virtual processor reexecutes the MONTMUL (due to the handler ending in RETRY), another fill trap will result if more than one window needed to be filled.

### MONTMUL Operand Locations

<table>
<thead>
<tr>
<th>Operand</th>
<th>Window (CWP value)</th>
<th>Register(s)</th>
</tr>
</thead>
</table>
Exceptions

- fp_disabled
- fill_n_normal (n = 0-7)
- fill_n_other (n = 0-7)

See Also

- MONTSQR on page 263
- MPMUL on page 273
7.89 MONTSQR [Crypto]

This instruction is new and is not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

**Description**

MONTSQR performs the Montgomery Squaring shown below with length equal to imm5. The starting locations of the operands are constant in the integer register file (IRF) regardless of size. The starting location of the result is also constant. Below are the locations for N=31. For smaller MONTSQRs, the remaining operand locations are not used and the remaining result locations will be unchanged.

FSR.fcc3 is set upon completion of MONTSQR to reflect whether or not a hardware error occurred during execution (see the Programming Note below).

```
MontSqr (l_uint A, l_uint N, l_uint *M, l_uint Nprime, char Length, l_uint *X) {
    // compute Montgomery Squaring as described below
    // use M as temporary variable
    // return result in X
    // A,N,M,X 64×(Length+1) bit long
    // Nprime is 64 bits long
    ACCUM ← 0
    for I ← 0 to Length begin
        for j ← 0 to ((I-1) >> 1) begin // words skipped on first I iteration
            ACCUM ← ACCUM + (2 × A[j] × A[I-j])
        end
        if I is even begin
            ACCUM ← ACCUM + A[I/2]^2
        end
        for j ← 0 to I-1 begin
            ACCUM ← ACCUM + M[j] × N[I-j]
        end
        M[I] ← ACCUM × Nprime // 64 LSB of accum, store 64 LSB of product
        ACCUM ← ACCUM + (M[I] × N[0])
        ACCUM ← (ACCUM >> 64)
    end
    for I ← (Length+1) to ((2×Length)+1) begin
        for j ← (I-Length) to ((I-1) >> 1) begin // skip last I iteration
            ACCUM ← ACCUM + (2 × A[j] × A[I-j])
        end
        if I is even begin
            ACCUM ← ACCUM + A[I/2]^2
        end
    end
}
```

CHAPTER 7 • Instructions 263
end
X[I-Length-1] ← ACCUM  // 64 LSB of accum
ACCUM ← (ACCUM >> 64)
end

ModReduction(ACCUM, X, N, Length)  // Reduce the final result
}

ModReduction(bit ACCUM, l_uint *X, l_uint N, char Length) {
  // compute (ACCUM|X) mod N
  // return result in X
  // ACCUM 1 bit long
  // N,X 64 × (Length+1) bit long

  if (ACCUM ≠ 0) begin
    for I ← 0 to Length begin  // Length is one less than the number of words
      X[I] ← X[I] - N[I]     // Subtraction with borrow
    end
  end

  else begin
    I ← Length
    while (((I ≥ 0) and (X[I] = N[I]))) begin
      I ← I-1
    end

    if ((I < 0) or (X[I] > N[I])) begin
      for I ← 0 to Length begin
        X[I] ← X[I] - N[I]     // Subtraction with borrow
      end
    end
  end
}

<table>
<thead>
<tr>
<th>Operand Location</th>
<th>Window (CWP value)</th>
<th>Register(s)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source operands</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nprime</td>
<td>—</td>
<td>F_D[60]</td>
<td></td>
</tr>
</tbody>
</table>
Exceptions. If $rd \neq 0$ or $rs1 \neq 0$, an attempt to execute a MONTSQR instruction causes an illegal_instruction exception.

If CFR.montsqr = 0, an attempt to execute an MONTSQR instruction causes a compatibility_feature exception.

Programming Note | The MONTSQR instruction uses seven windows of the integer register file (IRF).

Due to the special nature of the MONTSQR instruction, the processor cannot protect the programmer from hardware errors in integer and floating-point register file locations during execution of MONTSQR. The MONTSQR instruction sets $FSR.fcc3$ to 002 if no hardware error occurred, or to 112 (unordered) if an error did occur. It is the responsibility of the programmer to check $FSR.fcc3$ when the instruction completes and to retry the instruction if it encountered a hardware error. The programmer should provide an error counter to allow for a limited number of attempts to retry the operation. The instruction may be retried after first reloading all the input data from a backing storage location; thus, the programmer must take care to preserve a clean copy of the input data.

Exceptions. If rd $\neq 0$ or rs1 $\neq 0$, an attempt to execute a MONTSQR instruction causes an illegal_instruction exception.

If CFR.montsqr = 0, an attempt to execute an MONTSQR instruction causes a compatibility_feature exception.

Programming Note | Software must check that CFR.montsqr = 1 before executing the MONTSQR instruction. If CFR.montsqr = 0, then software should assume that an attempt to execute the MONTSQR instruction either (1) will generate an illegal_instruction exception because it is not implemented in hardware, or (2) will execute but perform some other operation.

Therefore, if CFR.montsqr = 0, software should perform the MONTSQR operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.

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

The MONTSQR instruction causes a fill_n_normal or fill_n_other exception if CANRESTORE is not equal to N_WINDOWS-2. The fill trap handler is called with CWP set to point to the window to be filled, that is, old CWP-CANRESTORE-1. 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 411. The fill trap handler performs a RESTORED and a RETRY as part of filling the window. When the virtual processor reexecutes the MONTSQR (due to the handler ending in RETRY), another fill trap will result if more than one window needed to be filled.
Exceptions

- *fp_disabled*
- *fill_n_normal* (n = 0-7)
- *fill_n_other* (n = 0-7)

See Also

- MONTMUL on page 259
- MPMUL on page 273
## 7.90 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</td>
<td>1100</td>
<td>Move Always</td>
<td>1</td>
<td>mova i_or_x_cc, reg_or_imm11, reg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>MOVN</td>
<td>10</td>
<td>1100</td>
<td>Move Never</td>
<td>0</td>
<td>movn i_or_x_cc, reg_or_imm11, reg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>MOVNE</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVE</td>
<td>10</td>
<td>1100</td>
<td>Move if Equal</td>
<td>Z</td>
<td>movet^{‡} i_or_x_cc, reg_or_imm11, reg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>MOVG</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVLE</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVGE</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVL</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVGU</td>
<td>10</td>
<td>1100</td>
<td>Move if Greater, Unsigned</td>
<td>not (C or Z)</td>
<td>movgu i_or_x_cc, reg_or_imm11, reg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>MOVLEU</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVCC</td>
<td>10</td>
<td>1100</td>
<td>Move if Carry Clear (Greater or Equal, Unsigned)</td>
<td>not C</td>
<td>movc^{◊} i_or_x_cc, reg_or_imm11, reg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>MOVCS</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVPOS</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVNEG</td>
<td>10</td>
<td>1100</td>
<td>Move if Negative</td>
<td>N</td>
<td>movneg i_or_x_cc, reg_or_imm11, reg_{rd}</td>
<td>A1</td>
</tr>
<tr>
<td>MOVVC</td>
<td>10</td>
<td>1100</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>
</tr>
<tr>
<td>MOVVS</td>
<td>10</td>
<td>1100</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>
</tr>
</tbody>
</table>

^{†} synonym: movnz  \quad ^{‡} synonym: movz  \quad ^{◊} synonym: movgeu  \quad ^{▼} synonym: movlu

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

### For Floating-Point Condition Codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>cond</th>
<th>Operation</th>
<th>fcc Test</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVFA</td>
<td>10</td>
<td>1100</td>
<td>Move Always</td>
<td>1</td>
<td>mova %fccn, 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 %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFU</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered</td>
<td>U</td>
<td>movu %fccn, 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 %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFUG</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Greater</td>
<td>G or U</td>
<td>movug %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFL</td>
<td>10</td>
<td>1100</td>
<td>Move if Less</td>
<td>L</td>
<td>movl %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFUL</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Less</td>
<td>L or U</td>
<td>movul %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFLG</td>
<td>10</td>
<td>1100</td>
<td>Move if Less or Greater</td>
<td>L or G</td>
<td>movlg %fccn, 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† %fccn, 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† %fccn, 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† %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFUE</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Equal</td>
<td>E or U</td>
<td>movue %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFGE</td>
<td>10</td>
<td>1100</td>
<td>Move if Greater or Equal</td>
<td>E or G</td>
<td>movge %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFUGE</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Greater or Equal</td>
<td>E or G or U</td>
<td>movuge %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFL</td>
<td>10</td>
<td>1100</td>
<td>Move if Less or Equal</td>
<td>E or L</td>
<td>movle %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFULE</td>
<td>10</td>
<td>1100</td>
<td>Move if Unordered or Less or Equal</td>
<td>E or L or U</td>
<td>movule %fccn, reg_or_imm11, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>MOVFO</td>
<td>10</td>
<td>1100</td>
<td>Move if Ordered</td>
<td>E or L or G</td>
<td>movo %fccn, 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.

---

### Condition Code

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

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

These instructions copy an integer register to another integer register if the condition is \( \text{TRUE} \). The condition code that is used to determine whether the move will occur can be either integer condition code (icc or xcc) or any floating-point condition code (fcc0, fcc1, fcc2, or fcc3).

These instructions do not modify any condition codes.

**Programming Note**
Branches cause the performance of many implementations to degrade significantly. Frequently, the MOVcc and FMOVcc instructions can be used to avoid branches. For example, the C language if-then-else statement

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

can be coded as

\[
\begin{align*}
\text{cmp} & \quad %i0,%i2 \\
\text{bg},a & \quad %\text{xcc},\text{label} \\
\text{or} & \quad %g0,1,%i3! X = 1 \\
\text{or} & \quad %g0,0,%i3! X = 0 \\
\text{label} : & ...
\end{align*}
\]

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

\[
\begin{align*}
\text{cmp} & \quad %i0,%i2 \\
\text{or} & \quad %g0,1,%i3! \text{ assume } X = 1 \\
\text{movle} & \quad %\text{xcc},0,%i3! \text{ overwrite with } X = 0
\end{align*}
\]

This approach takes only three instructions and no branches and may boost performance significantly. Use MOVcc and FMOVcc instead of branches wherever these instructions would increase performance.

An attempt to execute a MOVcc instruction when either instruction bits 10:5 are nonzero or \( (cc2 :: cc1 :: cc0) = 101_2 \) or \( 111_2 \) causes an *illegal_instruction* exception.

If \( cc2 = 0 \) (that is, a floating-point condition code is being referenced in the MOVcc instructions) and either the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a MOVcc instruction causes an *fp_disabled* exception.
7.91  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>
<tbody>
<tr>
<td></td>
<td>10 1111</td>
<td>000</td>
<td>Reserved (illegal_instruction)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MOVZ</td>
<td>10 1111</td>
<td>001</td>
<td>Move if Register Zero</td>
<td>( R[rs1] = 0 )</td>
<td>movrz(^{†}) ( reg_{rs1}, reg_{or_imm10}, reg_{rd} )</td>
<td>A1</td>
</tr>
<tr>
<td>MOVRLEZ</td>
<td>10 1111</td>
<td>010</td>
<td>Move if Register Less Than or Equal to Zero</td>
<td>( R[rs1] \leq 0 )</td>
<td>movrlez ( reg_{rs1}, reg_{or_imm10}, reg_{rd} )</td>
<td>A1</td>
</tr>
<tr>
<td>MOVRNZ</td>
<td>10 1111</td>
<td>011</td>
<td>Move if Register Not Zero</td>
<td>( R[rs1] \neq 0 )</td>
<td>movrnz(^{‡}) ( reg_{rs1}, reg_{or_imm10}, reg_{rd} )</td>
<td>A1</td>
</tr>
<tr>
<td>MOVRGZ</td>
<td>10 1111</td>
<td>110</td>
<td>Move if Register Greater Than Zero</td>
<td>( R[rs1] &gt; 0 )</td>
<td>movrgz ( reg_{rs1}, reg_{or_imm10}, reg_{rd} )</td>
<td>A1</td>
</tr>
<tr>
<td>MOVRGEZ</td>
<td>10 1111</td>
<td>111</td>
<td>Move if Register Greater Than or Equal to Zero</td>
<td>( R[rs1] \geq 0 )</td>
<td>movrgez ( reg_{rs1}, reg_{or_imm10}, reg_{rd} )</td>
<td>A1</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.

**Programming Note**  
The MOVr instructions are “64-bit-only” instructions; there is no version of these instructions that operates on just the less-significant 32 bits of their source operands.

**Implementation Note**  
If this instruction is implemented by tagging each register value with an \( n \) (negative) and a \( z \) (zero) bit, use the table below to determine if \( rcond \) is TRUE.

<table>
<thead>
<tr>
<th>Move</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVNZ</td>
<td>( \text{not } Z )</td>
</tr>
<tr>
<td>MOVZ</td>
<td>( Z )</td>
</tr>
<tr>
<td>MOVRNZ</td>
<td>( \text{not } N )</td>
</tr>
<tr>
<td>MOVRLZ</td>
<td>( N )</td>
</tr>
<tr>
<td>MOVRLEZ</td>
<td>( N \text{ or } Z )</td>
</tr>
<tr>
<td>MOVRGZ</td>
<td>( N \text{ nor } Z )</td>
</tr>
</tbody>
</table>

An attempt to execute a MOVr instruction when either instruction bits 9:5 are nonzero or \( rcond = 000_2 \) or \( 100_2 \) causes an illegal_instruction exception.

**Exceptions**  
illegal_instruction
7.92 Move Floating-Point Register to Integer Register

**Description**

The MOVsTosw instruction copies 32 bits from floating-point register \( F_s[rs2] \) to general-purpose register \( R[rd][31:0] \) (with no conversion). It places a copy of the sign bit in each bit of \( R[rd][63:32] \); that is, it sign-extends the destination result from 32 to 64 bits.

The MOVsTOuw instruction copies 32 bits from floating-point register \( F_s[rs2] \) to general-purpose register \( R[rd][31:0] \) (with no conversion). It sets \( R[rd][63:32] \) to zero; that is, it does not sign-extend the result.

The MOVdTOx instruction copies 64 bits from general-purpose register \( F_d[rs2] \) to general-purpose register \( R[rd] \). No conversion is performed.

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

If the FPU is not enabled (\( FPRESS.ef = 0 \) or \( PSTATE.pef = 0 \)) or if no FPU is present, an attempt to execute a MOVfTOi instruction causes an *fp_disabled* exception.

**Exceptions**

*illegal_instruction*

*fp_disabled*

**See Also**

MOVfTOf on page 272
7.93 Move Integer Register to Floating-Point Register

**Description**

The MOVwTOs instruction copies 32 bits from general-purpose register \( R[rs1][31:0] \) to floating-point register \( F_S[rd] \). No conversion is performed on the copied bits.

The MOVxTOd instruction copies 64 bits from general-purpose register \( R[rs1] \) to floating-point register \( F_D[rd] \). No conversion is performed on the copied bits.

An attempt to execute a MOViTOf 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 a MOViTOf instruction causes an *fp_disabled* exception.

**Exceptions.**

- *illegal_instruction*
- *fp_disabled*

**See Also**

MOVfTOi on page 271
MPMUL

7.94 MPMUL [Crypto]

This instruction is new and is not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<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>MPMUL&lt;sup&gt;N&lt;/sup&gt;</td>
<td>1 0100 1000</td>
<td>Multiple Precision Multiply</td>
<td>mpmul imm5</td>
<td>N1</td>
</tr>
</tbody>
</table>

**Description**

MPMUL multiplies two values, each of width \((N+1)\times64\) bits, where \(N\) is specified in imm5 field of the instruction. The starting locations of the multiplier and multiplicand are constant relative to CWP in the integer register file (IRF), regardless of size. The starting location of the product is also constant. Below are the locations for \(N = 31\). For smaller MPMULs \((N < 31)\), the remaining multiplier and multiplicand locations are unused and the remaining product locations are unchanged.

\[
\text{MPMUL} : \text{product}[i], \text{multiplier}[i], \text{and multiplicand}[i] \text{ are arrays of 64-bit doublewords.}
\]

\[
\text{product}[2N+1]:0 \leftarrow \text{multiplier}[N]:0 \times \text{multiplicand}[N]:0
\]

Let \(i = \text{CWP}\) when MPMUL is executed.

Note: doubleword 0 is the least significant doubleword.

The operands are located in registers as follows:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Window (CWP value)</th>
<th>Register(s)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Source operands</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Destination operands</strong></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
The following code shows an example usage of MPMUL with length equal to 31.

```assembly
setx multiplier, %g1, %g4
setx multiplicand, %g1, %g5
load_multiplier:
  ldd [%g4 + 0x000], %f22  !# CWP = i-6
  ldd [%g4 + 0x008], %f20
  ldd [%g4 + 0x010], %f18
  ldd [%g4 + 0x018], %f16
  ldd [%g4 + 0x020], %f14
  ldd [%g4 + 0x028], %f12
  ldd [%g4 + 0x030], %f10
  ldd [%g4 + 0x038], %f8
  ldd [%g4 + 0x040], %f6
  ldd [%g4 + 0x048], %f4
  ldx [%g4 + 0x050], %i5
  ldx [%g4 + 0x058], %i4
  ldx [%g4 + 0x060], %i3
  ldx [%g4 + 0x068], %i2
  ldx [%g4 + 0x070], %i1
  ldx [%g4 + 0x078], %i0
  ldx [%g4 + 0x080], %i17
  ldx [%g4 + 0x088], %i16
  ldx [%g4 + 0x090], %i15
  ldx [%g4 + 0x098], %i14
  ldx [%g4 + 0x0a0], %i13
  ldx [%g4 + 0x0a8], %i12
  ldx [%g4 + 0x0b0], %i11
  ldx [%g4 + 0x0b8], %i10
  ldd [%g4 + 0x0c0], %f2
  ldd [%g4 + 0x0c8], %f0
  ldx [%g4 + 0x0d0], %o5
  ldx [%g4 + 0x0d8], %o4
  ldx [%g4 + 0x0e0], %o3
  ldx [%g4 + 0x0e8], %o2
  ldx [%g4 + 0x0f0], %o1
  ldx [%g4 + 0x0f8], %o0
save  !# CWP = i-5
```

The MPMUL instruction uses seven windows of the integer register file (IRF). Before using the code sequence below, the contents of the IRF must be saved. The code shown below assumes the following window register values prior to execution:

- CANSAVE=NWINDOWS-2, CANRESTORE=0, OTHERWIN=0.

**Programming Note**

The following code shows an example usage of MPMUL with length equal to 31.
load_multiplicand:
   ldd [%g5 + 0x000], %f58
   ldd [%g5 + 0x008], %f56
   ldd [%g5 + 0x010], %f54
   ldd [%g5 + 0x018], %f52
   ldd [%g5 + 0x020], %f50
   ldd [%g5 + 0x028], %f48
   ldd [%g5 + 0x030], %f46
   ldd [%g5 + 0x038], %f44
   ldd [%g5 + 0x040], %f42
   ldd [%g5 + 0x048], %f40
   ldd [%g5 + 0x050], %f38
   ldd [%g5 + 0x058], %f36
   ldd [%g5 + 0x060], %f34
   ldd [%g5 + 0x068], %f32
   ldd [%g5 + 0x070], %f30
   ldd [%g5 + 0x078], %f28
   ldd [%g5 + 0x080], %f26
   ldd [%g5 + 0x088], %f24
   ldx [%g5 + 0x090], %o5
   ldx [%g5 + 0x098], %o4
   ldx [%g5 + 0x0a0], %o3
   ldx [%g5 + 0x0a8], %o2
   ldx [%g5 + 0x0b0], %o1
   ldx [%g5 + 0x0b8], %o0
   ldx [%g5 + 0x0c0], %l7
   ldx [%g5 + 0x0c8], %l6
   ldx [%g5 + 0x0d0], %l5
   ldx [%g5 + 0x0d8], %l4
   ldx [%g5 + 0xe0], %l3
   ldx [%g5 + 0xe8], %l2
   ldx [%g5 + 0xf0], %l1
   ldx [%g5 + 0xf8], %l0
   save  !# CWP = i-4
   save  !# CWP = i-3
   save  !# CWP = i-2
   save  !# CWP = i-1
   save  !# CWP = i

run_mpmul:
   mpmul 0x1f  !# CWP = i

store_result:
   setx vt_result, %g1, %g4
   stx %l7, [%g4 + 0x000]  !# CWP = i
   stx %l6, [%g4 + 0x008]
   stx %l5, [%g4 + 0x010]
   stx %l4, [%g4 + 0x018]
   stx %l3, [%g4 + 0x020]
   stx %l2, [%g4 + 0x028]
   stx %l1, [%g4 + 0x030]
   stx %l0, [%g4 + 0x038]
   restore  !# CWP = i-1

   stx %o5, [%g4 + 0x040]
stx %o4, [%g4 + 0x048]
stx %o3, [%g4 + 0x050]
stx %o2, [%g4 + 0x058]
stx %o1, [%g4 + 0x060]
stx %o0, [%g4 + 0x068]

stx %l7, [%g4 + 0x070]
stx %l6, [%g4 + 0x078]
stx %l5, [%g4 + 0x080]
stx %l4, [%g4 + 0x088]
stx %l3, [%g4 + 0x090]
stx %l2, [%g4 + 0x098]

restore !# CWP = i-2

stx %o5, [%g4 + 0x0b0]
stx %o4, [%g4 + 0x0b8]
stx %o3, [%g4 + 0x0c0]
stx %o2, [%g4 + 0x0c8]
stx %o1, [%g4 + 0x0d0]

stx %l7, [%g4 + 0x0e0]
stx %l6, [%g4 + 0x0e8]

restore !# CWP = i-3

stx %o5, [%g4 + 0x120]

stx %o4, [%g4 + 0x128]

stx %o3, [%g4 + 0x130]
stx %o2, [%g4 + 0x138]
stx %o1, [%g4 + 0x140]

stx %l7, [%g4 + 0x150]

stx %l6, [%g4 + 0x158]

stx %l5, [%g4 + 0x160]

stx %l4, [%g4 + 0x168]

stx %l3, [%g4 + 0x170]

stx %l2, [%g4 + 0x178]

stx %l1, [%g4 + 0x180]

stx %l0, [%g4 + 0x188]

restore !# CWP = i-4

stx %o5, [%g4 + 0x190]
stx %o4, [%g4 + 0x198]
stx %o3, [%g4 + 0x1a0]
stx %o2, [%g4 + 0x1a8]
stx %o1, [%g4 + 0x1b0]

stx %o0, [%g4 + 0x1b8]
Exceptions. If rd ≠ 0 or rs1 ≠ 0, an attempt to execute an MPMUL instruction causes an illegal_instruction exception.

If CFR.mpmul = 0, an attempt to execute an MPMUL instruction causes a compatibility_feature exception.

Programming Note

Software must check that CFR.mpmul = 1 before executing the MPMUL instruction. If CFR.mpmul = 0, then software should assume that an attempt to execute the MPMUL instruction either
(1) will generate an illegal_instruction exception because it is not implemented in hardware, or
(2) will execute but perform some other operation.

Therefore, if CFR.mpmul = 0, software should perform the MPMUL operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.

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

The MPMUL instruction causes a fill_n_normal or fill_n_other exception if CANRESTORE is not equal to NWINDOWS-2. The fill trap handler is called with CWP set to point to the window to be filled, that is, old CWP-CANRESTORE-1. 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 432. The fill trap handler performs a RESTORED and a RETRY as part of filling the window. When the virtual processor reexecutes MPMUL (due to the handler ending in RETRY), another fill trap results if more than one window needed to be filled.

Exceptions

fp_disabled
fill_n_normal (n = 0-7)
fill_n_other (n = 0-7)

See Also

MONTMUL on page 259
MONTSQR on page 263
7.95 Multiply and Divide (64-bit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULX</td>
<td>00 1001</td>
<td>Multiply (signed or unsigned)</td>
<td>mulx rsrs1, reg_or_imm, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>SDIVX</td>
<td>10 1101</td>
<td>Signed Divide</td>
<td>sdivx rsrs1, reg_or_imm, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>UDIVX</td>
<td>00 1101</td>
<td>Unsigned Divide</td>
<td>udivx rsrs1, reg_or_imm, regrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

Description

MULX computes “R[rs1] × R[rs2]” if i = 0 or “R[rs1] × sign_ext (simm13)” if i = 1, and writes the 64-bit product into R[rd]. MULX can be used to calculate the 64-bit product for signed or unsigned operands (the product is the same).

SDIVX and UDIVX compute “R[rs1] ÷ R[rs2]” if i = 0 or “R[rs1] ÷ sign_ext (simm13)” if i = 1, and write the 64-bit result into R[rd]. SDIVX operates on the operands as signed integers and produces a corresponding signed result. UDIVX operates on the operands as unsigned integers and produces a corresponding unsigned result.

For SDIVX, if the largest negative number is divided by –1, the result should be the largest negative number. That is:

\[8000 0000 0000 0000_{16} ÷ FFFF FFFF FFFF_{16} = 8000 0000 0000 0000_{16}.\]

These instructions do not modify any condition codes.

An attempt to execute a MULX, SDIVX, or UDIVX instruction when i = 0 and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

Exceptions

illegal_instruction
division_by_zero
7.96 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 \( \text{imm}22 = 0 \) and \( rd = 0 \).

**Programming Note**

There are many other opcodes that may execute as NOPs; however, this dedicated NOP instruction is the only one guaranteed to be implemented efficiently across all implementations.

**Exceptions**

None
NORMALW

7.97 NORMALW

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

| 10 | fcn = 0 0100 | 11 0001 | — |
| 31 30 29 25 24 19 18 0 |

Description
NORMALWP is a privileged instruction that copies the value of the OTHERWIN register to the CANRESTORE register, then sets the OTHERWIN register to zero.

Programming Notes
The NORMALW instruction is used when changing address spaces. NORMALW indicates the current "other" windows are now "normal" windows and should use the spill_n_normal and fill_n_normal traps when they generate a trap due to window spill or fill exceptions. The window state may become inconsistent if NORMALW is used when CANRESTORE is nonzero.

An attempt to execute a NORMALW instruction when instruction bits 18:0 are nonzero causes an illegal_instruction exception.

An attempt to execute an NORMALW instruction in nonprivileged mode (PSTATE.priv = 0) causes a privileged_opcode exception.

Exceptions
illegal_instruction
privileged_opcode

See Also
ALLCLEAN on page 115
INVALW on page 225
OTHERW on page 282
RESTORED on page 302
SAVED on page 309
### OR Logical Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>000010</td>
<td>Inclusive or</td>
<td>or rs1, reg_or_imm, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>ORcc</td>
<td>010010</td>
<td>Inclusive or and modify cc’s</td>
<td>orcc rs1, reg_or_imm, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>ORN</td>
<td>000110</td>
<td>Inclusive or not</td>
<td>orn rs1, reg_or_imm, regrd</td>
<td>A1</td>
</tr>
<tr>
<td>ORNcc</td>
<td>010110</td>
<td>Inclusive or not and modify cc’s</td>
<td>ornc rs1, reg_or_imm, regrd</td>
<td>A1</td>
</tr>
</tbody>
</table>

#### Description

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

ORcc and ORNcc modify the integer condition codes (icc and xcc). They set the condition codes as follows:

- icc,v, icc,c, xcc,v, and xcc.c are set to 0
- icc.n is copied from bit 31 of the result
- xcc.n is copied from bit 63 of the result
- icc.z is set to 1 if bits 31:0 of the result are zero (otherwise to 0)
- xcc.z is set to 1 if all 64 bits of the result are zero (otherwise to 0)

ORN and ORNcc logically negate their second operand before applying the main (or) operation.

An attempt to execute an OR[N][cc] instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

#### Exceptions

illegal_instruction
### OTHERW

#### Description

OTHERW is a privileged instruction that copies the value of the CANRESTORE register to the OTHERWIN register, then sets the CANRESTORE register to zero.

#### Programming Notes

The OTHERW instruction is used when changing address spaces. OTHERW indicates the current “normal” register windows are now “other” register windows and should use the `spill_n_other` and `fill_n_other` traps when they generate a trap due to window spill or fill exceptions. The window state may become inconsistent if OTHERW is used when OTHERWIN is nonzero.

An attempt to execute an OTHERW instruction when instruction bits 18:0 are nonzero causes an `illegal_instruction` exception.

An attempt to execute an OTHERW instruction in nonprivileged mode (PSTATE.priv = 0) causes a `privileged_opcode` exception.

#### Exceptions

- `illegal_instruction`
- `privileged_opcode`

#### See Also

- ALLCLEAN on page 115
- INVALW on page 225
- NORMALW on page 280
- RESTORED on page 302
- SAVED on page 309
7.100  Pause

Description  Execution of the PAUSE instruction voluntarily pauses (temporarily suspends execution on) a virtual processor for up to the specified number of processor cycles. If \( i = 0 \), the virtual processor is requested to pause for \( \text{R[rs2]} \) processor cycles; if \( i = 1 \), the virtual processor is requested to pause for \( \text{sign ext (simm13)} \) processor cycles. If certain asynchronous events occur, the virtual processor may terminate the pause sooner than the requested length of time.

The PAUSE instruction operates by writing the specified value to the Pause Count (PAUSE) register. For details, see Pause Count (PAUSE) Register (ASR 27) on page 60.

Programming Note  PAUSE is intended to be used as part of a progressive (exponential) backoff algorithm. A pause may not affect a given implementation’s behavior if its duration is below some implementation-specific threshold. However, a pause of sufficient duration (larger than the implementation-dependent threshold) causes the virtual processor to suspend execution and retirement of instructions.

Implementation Note  The PAUSE instruction is implemented as a degenerate case of a WRasr to ASR 27 (1B16), the Pause Count register (PAUSE). A full WRasr to the PAUSE register (WRPAUSE) may also be used to write a value to PAUSE, allowing the value to be constructed as the xor of two operands.

Exceptions  (none)

See Also  PAUSE register on page 59
WRasr on page 357
7.101 Pixel Component Distance (with Accumulation) [VIS 1]

The PDIST instruction is deprecated and should not be used in new software. The PDISTN instructions should be used, instead.

### Description

Eight unsigned 8-bit values are contained in the 64-bit floating-point source registers \( F_D[rs1] \) and \( F_D[rs2] \). The corresponding 8-bit values in the source registers are subtracted (that is, each byte in \( F_D[rs2] \) is subtracted from the corresponding byte in \( F_D[rs1] \)). The sum of the absolute value of each difference is added to the integer in \( F_D[rd] \) and the resulting integer sum is stored in the destination register, \( F_D[rd] \).

### Programming Notes

PDIST is a “destructive” instruction, in that \( F_D[rd] \) serves as both a source and a destination register (read-modify-write).

Typically, PDIST is used for motion estimation in video compression algorithms.

The (deprecated) PDIST instruction does not perform well on every Oracle SPARC Architecture implementation and its performance is expected to degrade further on future implementations. Use of PDISTN plus a subsequent ADD is recommended as a higher-performance replacement for PDIST.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a PDIST instruction causes an \( fp\_disabled \) exception.

### Exceptions

- \( fp\_disabled \)

### See Also

PDISTN on page 285
7.102 Pixel Component Distance (No Accumulation) [VIS3]

**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 $F_D[[rs1]$ and $F_D[rs2]$ are subtracted (that is, $F_D[rs1][n+7:n] - F_D[rs2][n+7:n]$). The integer sum of the absolute values of each of the eight differences is stored in the destination register, $R[rd]$.

**Note**
PDISTN is intended as a functional replacement for PDIST. As explained on the PDIST instruction page, new software should avoid using PDIST and use PDISTN, instead.

**Note**
There are two differences between PDISTN and PDIST.
1. PDISTN is a two-source-operand instruction, unlike PDIST, which is a three-source-operand instruction that also performs accumulation.
2. PDISTN’s destination register is an integer register, while PDIST’s destination register is a floating-point register.

**Programming Notes**
Typically, PDISTN is used for motion estimation in video compression algorithms and is followed by an ADD[cc] instruction, which performs accumulation.

If the FPU is not enabled ($FPRS.ife = 0$ or $PSTATE.ife = 0$) or if no FPU is present, an attempt to execute an PDISTN instruction causes an `fp_disabled` exception.

**Exceptions**
`fp_disabled`

**See Also**
PDIST on page 284
### 7.103 Population Count

**Description**

POPC counts the number of ‘1’ bits in $R[rs2]$ if $i = 0$, or the number of ‘1’ bits in $\text{sign_ext}(simm13)$ if $i = 1$, and stores the count in $R[rd]$. This instruction does not modify the condition codes.

**V9 Compatibility Note**

Instruction bits 18 through 14 must be zero for POPC. Other encodings of this field ($rs1$) may be used in future versions of the SPARC architecture for other instructions.

**Programming Note**

POPC can be used to “find first bit set” from the least-significant (right) end (or, equivalently, Trailing Zero Count) in an integer register. A ‘C’-language program illustrating how POPC can be used for this purpose follows:

```c
int ffs(x) /* finds first 1 bit, counting from the LSB */
unsigned long long int x;
{
    return popc(in ^ (∼(-x))); /* for nonzero x */
}
```

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

```assembly
neg %X, %NEG_X ! -x(2’s complement)
xnor %X, %NEG_X, %TEMP ! ^ -x (exclusive nor)
popc %TEMP, %RESULT ! result = popc(x ^ -x)
movz %X, %g0, %RESULT ! %RESULT should be 0 for %X=0
```

where $X$, $NEG_X$, $TEMP$, and $RESULT$ are integer registers.

**Example computation:**

- $X = \ldots 00101000$ ! 1st ‘1’ bit from right is
- $-X = \ldots 11011000$ ! bit 3 (4th bit)
- $\sim -X = \ldots 00100111$
- $X \ ^ {\ } \ ^ {\ } \ ^ {\ } \ ^ {\ } \sim -X = \ldots 00001111$
- $\text{popc}(X \ ^ {\ } \ ^ {\ } \ ^ {\ } \ ^ {\ } \sim -X) = 4$

**Programming Note**

Another means of implementing “Trailing Zero Count” is:

`or TZCNT(x) = POPC((\text{not}(x)) \text{ and } (x - 1))`

**Programming Note**

POPC can be used to “centrifuge” all the ‘1’ bits in a register to the least significant end of a destination register. Assembly-language code illustrating how POPC can be used for this purpose follows:

```assembly
popc %X, %DEST
cmp %X, -1 ! Test for pattern of all 1’s
mov -1, %TEMP ! Constant -1 -> temp register
sllx %TEMP, %DEST, %DEST ! (shift count of 64 same as 0)
not %DEST !
movcc %xcc, -1, %DEST ! If src was -1, result is -1
```

where $X$, $TEMP$, and $DEST$ are integer registers.
POPC

**Programming Note**
POPC is a “64-bit-only” instruction; there is no version of this instruction that operates on just the less-significant 32 bits of its source operand.

An attempt to execute a POPC instruction when either instruction bits 18:14 are nonzero, or \( i = 0 \) and instruction bits 12:5 are nonzero causes an `illegal_instruction` exception.

**Exceptions**
- `illegal_instruction`

**See Also**
- *Leading Zeroes Count* on page 253
7.104 Prefetch

**Description**

A PREFETCH[A] instruction provides a hint to the virtual processor that software expects to access a particular address in memory in the near future, so that the virtual processor may take action to reduce the latency of accesses near that address. Typically, execution of a prefetch instruction initiates
PREFETCH

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 ($\text{ASI\_PRIMARY}[\text{\_LITTLE}]$).

PREFETCHA instructions access an alternate address space. If $i = 0$, the address space identifier (ASI) to be used for the instruction is in the imm_asi field. If $i = 1$, the ASI is found in the ASI register.

A prefetch operates much the same as a regular load operation, but with certain important differences. In particular, a 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))

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

```
prefix [address], prefix_fc
prefix [address + 63], prefix_fc
```

Variants of the prefetch instruction can be used to prepare the memory system for different types of accesses.

IMPL. DEP. #103-V9-Ms10(b): An implementation may implement none, some, or all of the defined PREFETCH[A] variants. It is implementation-dependent whether each variant is (1) not implemented and executes as a NOP, (2) is implemented and supports the full semantics for that variant, or (3) is implemented and only supports the simple common-case prefetching semantics for that variant.
7.104.1 Exceptions

Prefetch instructions `PREFETCH` and `PREFETCHA` generate exceptions under the conditions detailed in TABLE 7-17. Only the implementation-dependent prefetch variants (see TABLE 7-16) may generate an exception under conditions not listed in this table; the predefined variants only generate the exceptions listed here.

**TABLE 7-17** 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><code>PREFETCH</code></td>
<td>i = 0 and instruction bits 12:5 are nonzero</td>
<td><code>illegal_instruction</code></td>
</tr>
<tr>
<td>any</td>
<td><code>PREFETCHA</code></td>
<td>reference to an ASI in the range (0_{16}..7F_{16}), while in nonprivileged mode (privileged_action condition)</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>any</td>
<td><code>PREFETCHA</code></td>
<td>reference to an ASI in range (30_{16}..7F_{16}), while in privileged mode (privileged_action condition)</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-3 (weak)</td>
<td><code>PREFETCH[A]</code></td>
<td>condition detected for MMU miss</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-4</td>
<td><code>PREFETCH[A]</code></td>
<td>variant unimplemented</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>0-4</td>
<td><code>PREFETCHA</code></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><code>PREFETCH[A]</code></td>
<td>condition detected for <code>DAE_invalid_asi</code> (see following table), <code>DAE_privilege_violation</code>, <code>DAE_nc_page</code> ((TTE.cp = 0) or ((fcn = 0) and TTE.cv = 0)), <code>DAE_nfo_page</code>, or <code>DAE_side_effect_page</code> (TTE.e = 1)</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>4, 20-23 (strong)</td>
<td><code>PREFETCH[A]</code></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(<em>{16})–0F(</em>{16}))</td>
<td><code>PREFETCH[A]</code></td>
<td>(always)</td>
<td><code>illegal_instruction</code></td>
</tr>
<tr>
<td>16 (10(_{16}))</td>
<td><code>PREFETCH[A]</code></td>
<td>variant unimplemented</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>17-31 (11(<em>{16})–1F(</em>{16}))</td>
<td><code>PREFETCH[A]</code></td>
<td>variant unimplemented</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>24 (18(_{16}))</td>
<td><code>PREFETCH[A]</code></td>
<td>variant unimplemented</td>
<td>executes as NOP</td>
</tr>
<tr>
<td>25 (19(_{16}))</td>
<td><code>PREFETCH[A]</code></td>
<td>variant unimplemented</td>
<td>executes as NOP</td>
</tr>
</tbody>
</table>

**ASIs valid for `PREFETCHA` (all other ASIs are invalid)**

| \(04_{16}\) | `ASI_NUCLEUS` | \(0C_{16}\) | `ASI_NUCLEUS_LITTLE` |
| \(10_{16}\) | `ASI_AS_IF_USER_PRIMARY` | \(18_{16}\) | `ASI_AS_IF_USER_PRIMARY_LITTLE` |
| \(11_{16}\) | `ASI_AS_IF_USER_SECONDARY` | \(19_{16}\) | `ASI_AS_IF_USER_SECONDARY_LITTLE` |
| \(14_{16}\) | `ASI_REAL` | \(1C_{16}\) | `ASI_REAL_LITTLE` |
| \(80_{16}\) | `ASI_PRIMARY` | \(88_{16}\) | `ASI_PRIMARY_LITTLE` |
| \(81_{16}\) | `ASI_SECONDARY` | \(89_{16}\) | `ASI_SECONDARY_LITTLE` |
| \(82_{16}\) | `ASI_PRIMARY_NO_FAULT` | \(8A_{16}\) | `ASI_PRIMARY_NO_FAULT_LITTLE` |
| \(83_{16}\) | `ASI_SECONDARY_NO_FAULT` | \(8B_{16}\) | `ASI_SECONDARY_NO_FAULT_LITTLE` |
7.104.2 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) the underlying hardware will invest to perform the prefetch. If the prefetch is speculative (software believes the data will probably be needed, but isn’t sure), a Weak prefetch will initiate data movement if the operation can be performed quickly, but abort the prefetch and behave like a NOP if it turns out that performing the full prefetch will be time-consuming. If software has very high confidence that data being prefetched will subsequently be accessed, then a Strong prefetch will ensure that the prefetch operation will continue, even if the prefetch operation does become time-consuming.

From the virtual processor’s perspective, the difference between a Weak and a Strong prefetch is whether the prefetch is allowed to perform a time-consuming operation in order to complete. If a time-consuming operation is required, a Weak prefetch will abandon the operation and behave like a NOP while a Strong prefetch will pay the cost of performing the time-consuming operation so it can finish initiating the requested data movement.

Behavioral differences among loads, strong prefetches, and weak prefetches are compared in TABLE 7-18.

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

‡ The PREFETCH[A] instruction aborts, appearing to behave like a NOP instruction.

7.104.3 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 Oracle SPARC Architecture 2011.

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 Oracle SPARC 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).
PREFETCH

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

The intent of these variants is to cause movement of data into the cache nearest the virtual processor.

There are Weak and Strong versions of this prefetch variant; fcn = 0 is Weak and fcn = 20 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

**Programming Note** The intended use of this variant is for streaming relatively small amounts of data into the primary data cache of the virtual processor.

7.104.3.2 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.

7.104.3.3 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.

There are Weak and Strong versions of this prefetch variant; fcn = 2 is Weak and fcn = 22 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

**Programming Note** An example use of this variant is to initialize a cache line, in preparation for a partial write.

**Implementation Note** On a multiprocessor system, this variant indicates that exclusive ownership of the addressed data is needed. Therefore, it may have the additional effect of obtaining exclusive ownership of the addressed cache line.

7.104.3.4 Prefetch for One Write (fcn = 3, 23(1716))

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.

**Implementation Note** On a multiprocessor system, this variant indicates that exclusive ownership of the addressed data is needed. Therefore, it may have the additional effect of obtaining exclusive ownership of the addressed cache line.
7.104.3.5 Prefetch Page (fcn = 4)

In a virtual memory system, the intended action of this variant is for hardware (or privileged or hyperprivileged software) to initiate asynchronous mapping of the referenced virtual address (assuming that it is legal to do so).

**Programming Note** Prefetch Page is used is to avoid a later page fault for the given address, or at least to shorten the latency of a page fault.

In a non-virtual-memory system or if the addressed page is already mapped, this variant has no effect.

**Implementation Note** The mapping required by Prefetch Page may be performed by privileged software, hyperprivileged software, or hardware.

7.104.3.6 Prefetch to Nearest Unified Cache (fcn = 17(11_{16}))

The intent of this variant is to cause movement of data from memory into the nearest unified (combined instruction and data) cache. At the successful completion of this variant, the selected line from memory will be in the unified cache in the shared state, and in caches (if any) below it in the cache hierarchy.

Prefetch to Nearest Unified Cache is a Strong prefetch variant.

7.104.4 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.

7.104.5 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.

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
7.105 Read Ancillary State Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rs1</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
<th>Added</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, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>D3</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>1</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDCCR</td>
<td>2</td>
<td>Read Condition Codes register (CCR)</td>
<td>rd %ccr, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>RDASI</td>
<td>3</td>
<td>Read ASI register</td>
<td>rd %asi, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>RDTICK&lt;sup&gt;P,dis,Hdis&lt;/sup&gt;</td>
<td>4</td>
<td>Read TICK register</td>
<td>rd %tick, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>RDPC</td>
<td>5</td>
<td>Read Program Counter (PG)</td>
<td>rd %pc, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>RDFPRS</td>
<td>6</td>
<td>Read Floating-Point Registers Status (FPRS) register</td>
<td>rd %fprs, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>7–14</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>13 (0D&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>14 (0E&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See text</td>
<td>15 (F&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>MEMBAR or Reserved; see text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>16–18</td>
<td>Reserved (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDGSR</td>
<td>19 (13&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Read General Status register (GSR)</td>
<td>rd %gsr, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>20–21</td>
<td>Reserved (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDSOFTINT&lt;sup&gt;P&lt;/sup&gt;</td>
<td>22 (16&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Read per-virtual processor Soft Interrupt register</td>
<td>rd %softint, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>23 (17&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDSTICK&lt;sup&gt;P,dis,Hdis&lt;/sup&gt;</td>
<td>24 (18&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Read System Tick Register (STICK)</td>
<td>rd %tick†, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>RDSTICK_CMPRP&lt;sup&gt;P&lt;/sup&gt;</td>
<td>25 (19&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Read System Tick Compare register (STICK_CMPR)</td>
<td>rd %tick_cmpr†, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>RDCFR</td>
<td>26 (1A&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Read Compatibility Features register (CFR)</td>
<td>rd %cfr, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A2</td>
<td>OSA 2011</td>
</tr>
<tr>
<td>—</td>
<td>27 (1B&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Reserved (used for PAUSE, via WRasr)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>28 (1C&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>29 (1D&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>30 (1E&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>31 (1F&lt;sub&gt;16&lt;/sub&gt;)</td>
<td>Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† The original assembly language names for %tick and %tick_cmpr were, respectively, %sys_tick and %sys_tick_cmpr, which are now deprecated. Over time, assemblers will support the new %tick and %tick_cmpr names for these registers (which are consistent with %tick). In the meantime, some existing assemblers may only recognize the original names.
RDAsr

Description

The Read Ancillary State Register (RDAsr) instructions copy the contents of the state register specified by $rs1$ into $R[rd]$.

An RDAsr instruction with $rs1 = 0$ is a (deprecated) RDY instruction (which should not be used in new software).

The RDY instruction is deprecated. It is recommended that all instructions that reference the Y register be avoided.

RDPC copies the contents of the PC register into $R[rd]$. If $PSTATE.am = 0$, the full 64-bit address is copied into $R[rd]$. If $PSTATE.am = 1$, only a 32-bit address is saved; $PC[31:0]$ is copied to $R[rd][31:0]$ and $R[rd][63:32]$ is set to 0. (closed impl. dep. #125-V9-Cs10)

RDFPRS waits for all pending FPops and loads of floating-point registers to complete before reading the FPRS register.

The following values of $rs1$ are reserved for future versions of the architecture: 1, 7–13, 14, 16-18, 20-21, and 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.

**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 Oracle SPARC Architecture, since the PSR, WIM, and TBR registers do not exist.

See *Ancillary State Registers* on page 48 for more detailed information regarding ASR registers.

Exceptions.

An attempt to execute a RDAsr instruction when any of the following conditions are true causes an *illegal_instruction* exception:

- $rs1$ contains a reserved value (that is not assigned for implementation-dependent use)
- instruction bits 13:0 are nonzero

An attempt to execute aRDSTICK_CMPR, or RDSOFTINT instruction in nonprivileged mode ($PSTATE.priv = 0$) causes a *privileged_opcode* exception (impl. dep. #250-U3-Cs10).

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

Nonprivileged software can read the STICK register by using the RDSTICK instruction, but only when nonprivileged access to STICK is enabled. If nonprivileged access is disabled, an attempt by nonprivileged software to read the STICK register causes a *privileged_action* exception. See *System Tick (STICK) Register (ASR 24)* on page 56 for details.

Privileged software can read the STICK register with the RDSTICK instruction, but only when privileged access to STICK is enabled by hyperprivileged software. An attempt by privileged software to read the STICK register when privileged access is disabled causes a *privileged_action* exception. See *System Tick (STICK) Register (ASR 24)* on page 56 for details.

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

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

In privileged mode (PSTATE.priv = 1), the following cause a privileged_action exception:
■ execution of RDTICK or RDSTICK when privileged access to TICK and STICK is disabled

Implementation

Note
RD.asr shares an opcode with MEMBAR; RD.asr is distinguished by rs1 = 15 or rd = 0 or (i = 0, and bit 12 = 0).

Exceptions
illegal_instruction
privileged_opcode
fp_disabled
privileged_action

See Also
RDPR on page 298
WR.asr on page 357
7.106  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>1010</td>
<td>rd</td>
<td>rdpr %tpc, rsrd</td>
<td>A2?</td>
</tr>
<tr>
<td>TPC</td>
<td>0</td>
<td>rdpr</td>
<td>tpc</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>TNPC</td>
<td>1</td>
<td>rdpr</td>
<td>tnpc</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>TSTATE</td>
<td>2</td>
<td>rdpr</td>
<td>tstate</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>TT</td>
<td>3</td>
<td>rdpr</td>
<td>tt</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>TICK</td>
<td>4</td>
<td>rdpr</td>
<td>tick</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>TBA</td>
<td>5</td>
<td>rdpr</td>
<td>tba</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>PSTATE</td>
<td>6</td>
<td>rdpr</td>
<td>pstate</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>TL</td>
<td>7</td>
<td>rdpr</td>
<td>tl</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>PIL</td>
<td>8</td>
<td>rdpr</td>
<td>pil</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>CWP</td>
<td>9</td>
<td>rdpr</td>
<td>cwp</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>CANSAVE</td>
<td>10</td>
<td>rdpr</td>
<td>cansave</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>CANRESTORE</td>
<td>11</td>
<td>rdpr</td>
<td>canrestore</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>CLEANWIN</td>
<td>12</td>
<td>rdpr</td>
<td>cleanwin</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>OTHERWIN</td>
<td>13</td>
<td>rdpr</td>
<td>otherwin</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>WSTATE</td>
<td>14</td>
<td>rdpr</td>
<td>wstate</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL</td>
<td>16</td>
<td>rdpr</td>
<td>gl</td>
<td>rsrd</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**  
The rs1 field in the instruction determines the privileged register that is read. There are \( \text{MAXPTL} \) 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 contains a reserved value (see above)
- \( 0 \leq \text{rs1} \leq 3 \) (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**  
In nonprivileged mode, illegal_instruction exception due to \( 0 \leq \text{rs1} \leq 3 \) 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 Oracle SPARC 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 295
WRPR on page 360
7.107 RESTORE

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 ($\text{mod } N\_REG\_WINDOWS$) to restore the register window that was in use prior to the last SAVE instruction executed by the current process. It also updates the state of the register windows by decrementing CANRESTORE and incrementing CANSAVE.

If the register window to be restored has been spilled (CANRESTORE = 0), then a fill trap is generated. The trap vector for the fill trap is based on the values of OTHERWIN and WSTATE, as described in Trap Type for Spill/Fill Traps on page 432. 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.

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 307
### 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) and (OTHERWIN = 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 64) 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.
- 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.

### Exceptions

illegal_instruction
privileged_opcode

### See Also

ALLCLEAN on page 115
INVALW on page 225
NORMALW on page 280
OTHERW on page 282
SAVED on page 309
7.109 RETRY

<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>RETRY$^p$</td>
<td>11 1110</td>
<td>Return from Trap (retry trapped instruction)</td>
<td>retry</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

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).

**Programming Note**

The DONE and RETRY instructions are used to return from privileged trap handlers.

If the saved TPC[TL] and TNPC[TL] were not altered by trap handler software, RETRY causes execution to resume at the instruction that originally caused the trap (“retrying” it).

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

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

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system.

**IMPL. DEP. #417-S10**: If (1) TSTATE[TL].pstate.am = 1 and (2) a RETRY instruction is executed (which sets PSTATE.am to ‘1’ by restoring the value from TSTATE[TL].pstate.am to PSTATE.am), it is implementation dependent whether the RETRY instruction masks (zeroes) the more-significant 32 bits of the values it places into PC and NPC.

**Exceptions.** An attempt to execute the RETRY instruction when either of the following conditions is true causes an illegal_instruction exception:

- instruction bits 18:0 are nonzero
- TL = 0 and the virtual processor is in privileged mode (PSTATE.priv = 1)

An attempt to execute a RETRY instruction in nonprivileged mode (PSTATE.priv = 0) causes a privileged_opcode exception.

**Implementation Note**

In nonprivileged mode, illegal_instruction exception due to TL = 0 does not occur. The privileged_opcode exception occurs instead, regardless of the current trap level (TL).

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

Note that since PSTATE.tct is automatically set to 0 during entry to a trap handler, the execution of a RETRY instruction at the end of a trap handler will not cause a control_transfer_instruction exception unless trap handler software has explicitly set PSTATE.tct to 1. During execution of the RETRY instruction, the value of PSTATE.tct is restored from TSTATE.
RETRY

**Programming Note**

RETRY should *not* normally be used to return from the trap handler for the `control_transfer_instruction` exception itself. See the DONE instruction on page 144 and *Trap on Control Transfer (tct)* on page 68.

**Exceptions**

- `illegal_instruction`
- `privileged_opcode`
- `control_transfer_instruction` (impl. dep. #450-520)

**See Also**

DONE on page 144
The RETURN instruction causes a register-indirect delayed transfer of control to the target address and has the window semantics of a RESTORE instruction; that is, it restores the register window prior to the last SAVE instruction. The target address is “R[rs1] + R[rs2]” if \( i = 0 \), or “R[rs1] + \text{sign_ext}(\text{simm13})” if \( i = 1 \). Registers R[rs1] and R[rs2] come from the old window.

Like other DCTIs, all effects of RETURN (including modification of CWP) are visible prior to execution of the delay slot instruction.

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

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

A RETURN instruction may cause a window_fill exception as part of its RESTORE semantics.

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system. However, if a control_transfer_instruction trap occurs, the full 64-bit (nonmasked) address of the RETURN instruction is written into TPC[TL].

A RETURN instruction causes a mem_address_not_aligned exception if either of the two least-significant bits of the target address is nonzero.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then RETURN generates a control_transfer_instruction exception instead of causing a control transfer.

### 7.110 RETURN

<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>RETURN</td>
<td>11 1001</td>
<td>Return</td>
<td>return address</td>
<td>A1</td>
</tr>
</tbody>
</table>

**Description**

To reexecute the trapped instruction when returning from a user trap handler, use the RETURN instruction in the delay slot of a JMPL instruction, for example:

```
jmp %l6,%g0  !Trapped PC supplied to user trap handler
return %l7  !Trapped NPC supplied to user trap handler
```

A routine that uses a register window may be structured either as:

```
save %sp,-framesize,%sp
. . .
ret      !"ret" is shorthand for "jmpl %i7 + 8,%g0"
restore  : A useful instruction in the delay slot, such as
          : "restore %o2,%l2,%o0"
or as:
save %sp,-framesize,%sp
. . .
return %i7 + 8
nop      !Instead of "nop", could do some useful work in the
          : caller’s window, for example,"or %o1,%o2,%o0"
```

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

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

A RETURN instruction may cause a window_fill exception as part of its RESTORE semantics.

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system. However, if a control_transfer_instruction trap occurs, the full 64-bit (nonmasked) address of the RETURN instruction is written into TPC[TL].

A RETURN instruction causes a mem_address_not_aligned exception if either of the two least-significant bits of the target address is nonzero.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then RETURN generates a control_transfer_instruction exception instead of causing a control transfer.
Exceptions

illegal_instruction
fill_n_normal \((n = 0\text{--}7)\)
fill_n_other \((n = 0\text{--}7)\)
mem_address_not_aligned
control_transfer_instruction (impl. dep. #450-S20)
7.111 SAVE

The SAVE instruction provides the routine executing it with a new register window. The \textit{out} registers from the old window become the \textit{in} registers of the new window. The contents of the \textit{out} and the \textit{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 \textit{old} window (that is, the window addressed by the original CWP) and the sum is written into \( R[rd] \) of the \textit{new} window (that is, the window addressed by the new CWP).

\textbf{Note} \quad \textit{CWP} arithmetic is performed modulo the number of implemented windows, \( N_{\text{REG WINDOWS}} \).

**Description (Effect on Privileged State)**

If a SAVE instruction does not trap, it increments the \textit{CWP} (mod \( N_{\text{REG WINDOWS}} \)) to provide a new register window and updates the state of the register windows by decrementing \textit{CANSAVE} and incrementing \textit{CANRESTORE}.

If the new register window is occupied (that is, \textit{CANSAVE} = 0), a spill trap is generated. The trap vector for the spill trap is based on the value of \textit{OTHERWIN} and \textit{WSTATE}. The spill trap handler is invoked with the \textit{CWP} set to point to the window to be spilled (that is, old \textit{CWP} + 2).

An attempt to execute a SAVE instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an \textit{illegal_instruction} exception.
SAVE

If \( \text{CANSAVE} \neq 0 \), the SAVE instruction checks whether the new window needs to be cleaned. It causes a `clean_window` trap if the number of unused clean windows is zero, that is, \( (\text{CLEANWIN} - \text{CANRESTORE}) = 0 \). The `clean_window` trap handler is invoked with the CWP set to point to the window to be cleaned (that is, old CWP + 1).

**Programming Note**

The vectoring of spill traps can be controlled by setting the value of the OTHERWIN and WSTATE registers appropriately. The spill handler normally will end with a SAVED instruction followed by a RETRY instruction.

**Exceptions**

- `illegal_instruction`
- `spill_n_normal (n = 0–7)`
- `spill_n_other (n = 0–7)`
- `clean_window`

**See Also**

RESTORE on page 300
### 7.112 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>A1</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.

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) and (OTHERWIN = 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 64) 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

#### See Also
ALLCLEAN on page 115
INVALW on page 225
NORMALW on page 280
OTHERW on page 282
RESTORED on page 302
7.113 Signed Divide (64-bit ÷ 32-bit)

The SDIV and SDIVcc instructions are deprecated and should not be used in new software. The SDIVX instruction should be used instead.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDIVD</td>
<td>00 1111</td>
<td>Signed Integer Divide</td>
<td>sdiv rs1, reg_or_imm, reg =D3</td>
<td></td>
</tr>
<tr>
<td>SDIVccD</td>
<td>01 1111</td>
<td>Signed Integer Divide and modify cc's</td>
<td>sdivcc rs1, reg_or_imm, reg =D3</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The signed divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If \( i = 0 \), they compute \((Y :: R[rs1][31:0]) ÷ R[rs2][31:0]\). 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.

**Signed Divide**

Signed divide (SDIV, SDIVcc) assumes a signed integer doubleword dividend \((Y :: \text{lower 32 bits of } R[rs1])\) and a signed integer word divisor (lower 32 bits of \( R[rs2] \) or lower 32 bits of \( \text{sign} \text{_ext}(\text{simm13}) \)) and computes a signed integer word quotient \( R[rd] \).

Signed division rounds an inexact quotient toward zero. For example, \(-7 ÷ 4\) equals the rational quotient of \(-1.75\), which rounds to \(-1\) (not \(-2\)) when rounding toward zero.

The result of a signed divide can overflow the low-order 32 bits of the destination register \( R[rd] \) under certain conditions. When overflow occurs, the largest appropriate signed integer is returned as the quotient in \( R[rd] \). The conditions under which overflow occurs and the value returned in \( R[rd] \) under those conditions are specified in TABLE 7-19.

**TABLE 7-19** SDIV / SDIVcc 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^{31} )</td>
<td>( 2^{31} - 1 ) (0000 0000 7FFF FFFF&lt;sub&gt;16&lt;/sub&gt;)</td>
</tr>
<tr>
<td>Rational quotient ( ≤ -2^{31} - 1 )</td>
<td>( -2^{31} ) (FFFF FFFF 8000 0000&lt;sub&gt;16&lt;/sub&gt;)</td>
</tr>
</tbody>
</table>

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

SDIV does not affect the condition code bits. SDIVcc writes the integer condition code bits as shown in the following table. Note that negative (N) and zero (Z) are set according to the value of R[rd] after it has been set to reflect overflow, if any.

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

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

**Exceptions**
- illegal_instruction
- division_by_zero

**See Also**
- RDY on page 295
- UDIV[cc] on page 353
## 7.114 SETHI

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rd</th>
<th>op2</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETHI</td>
<td>0</td>
<td>100</td>
<td>Set High 22 Bits of Low Word</td>
<td>sethi const22, reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sethi %hi(value), reg&lt;sub&gt;rd&lt;/sub&gt;</td>
<td></td>
</tr>
</tbody>
</table>

### Description
SETHI zeroes the least significant 10 bits and the most significant 32 bits of \(R[rd]\) and replaces bits 31 through 10 of \(R[rd]\) with the value from its imm22 field.

SETHI does not affect the condition codes.

Some SETHI instructions with \(rd = 0\) have special uses:
- \(rd = 0\) and imm22 = 0: defined to be a NOP instruction (described in No Operation)
- \(rd = 0\) and imm22 ≠ 0 may be used to trigger hardware performance counters in some Oracle SPARC Architecture implementations (for details, see implementation-specific documentation).

### Programming Note
The most common form of 64-bit constant generation is creating stack offsets whose magnitude is less than \(2^{32}\). The code below can be used to create the constant 0000 0000 ABCD 1234<sub>16</sub>:

```
sethi %hi(0xabcd1234),%o0
or %o0, 0x234, %o0
```

The following code shows how to create a negative constant. **Note:**
The immediate field of the xor instruction is sign extended and can be used to place 1’s in all of the upper 32 bits. For example, to set the negative constant FFFF FFFF ABCD 1234<sub>16</sub>:

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

### Exceptions
None
7.115 Secure Hash Operations

The Secure Hash instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

The SHA1, SHA256, and SHA512 instructions operate on 64-bit floating-point registers and process an entire 512-bit block (SHA1 or SHA256) or a 1024-bit block (SHA512). The locations of the Initialization Vector (IV), Data, and Result values in the floating-point register file are described below. To compute the hash over multiple blocks, the result from the previous hash instruction is used as the IV for the next block. Software must appropriately pad the final block as specified by the given algorithm.

SHA1:
\[
\]
\[
\]
\[
\]

SHA256:
\[
\]
\[
\]
\[
\]

SHA512:
\[
\]
\[
\]
\[
\]

If \( rD \neq 0 \), \( rs1 \neq 0 \), or \( rs2 \neq 0 \), an attempt to execute a SHA1, SHA256, or SHA512 instruction causes an illegal_instruction exception.

If CFR.sha1 = 0, CFR.sha256 = 0, or CFR.sha512 = 0, an attempt to execute a SHA1, SHA256, or SHA512 instruction (respectively) causes a compatibility_feature exception.
Exceptions \textit{fp\_disabled}

**Programming Note** Software must check that the corresponding capability bit in the CFR register (sha1, sha256, or sha512) = 1 before executing one of these instructions. If the corresponding capability bit in CFR = 0, then software should assume that an attempt to execute the hash instruction either (1) will generate an \textit{illegal\_instruction} exception because it is not implemented in hardware, or (2) will execute, but perform some other operation.

Therefore, if the corresponding capability bit in CFR = 0, software should perform the hash operation by other means, such as using a software implementation, a crypto coprocessor, or another set of instructions which implement the desired function.
7.116 Set Interval Arithmetic Mode

<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>010000001</td>
<td>Set the interval arithmetic mode fields in the GSR</td>
<td>siam siam_mode</td>
<td>A1</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 GSR.im is set to 1, all subsequent floating-point instructions requiring round mode settings derive rounding-mode information from the General Status Register (GSR.irnd) instead of the Floating-Point State Register (FSR.rd).

**Note**  
When GSR.im = 1, the processor operates in standard floating-point mode regardless of the setting of FSR.ns.

An attempt to execute a SIAM instruction when instruction bits 29:25, 18:14, or 4:3 are nonzero causes an `illegal_instruction` exception.

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

**Exceptions**  
`illegal_instruction`  
`fp_disabled`
7.117 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 7-20 shows the shift count encodings for all values of $i$ and $x$.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>x</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>10 0101</td>
<td>0</td>
<td>Shift Left Logical – 32 bits</td>
<td>sll reg_rs1, reg_or_shcnt, reg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>SRL</td>
<td>10 0110</td>
<td>0</td>
<td>Shift Right Logical – 32 bits</td>
<td>srl reg_rs1, reg_or_shcnt, reg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>SRA</td>
<td>10 0111</td>
<td>0</td>
<td>Shift Right Arithmetic– 32 bits</td>
<td>sra reg_rs1, reg_or_shcnt, reg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>SLLX</td>
<td>10 0101</td>
<td>1</td>
<td>Shift Left Logical – 64 bits</td>
<td>sllx reg_rs1, reg_or_shcnt, reg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>SRLX</td>
<td>10 0110</td>
<td>1</td>
<td>Shift Right Logical – 64 bits</td>
<td>srlx reg_rs1, reg_or_shcnt, reg_rd</td>
<td>A1</td>
</tr>
<tr>
<td>SRAX</td>
<td>10 0111</td>
<td>1</td>
<td>Shift Right Arithmetic – 64 bits</td>
<td>srax reg_rs1, reg_or_shcnt, reg_rd</td>
<td>A1</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 $R[rd]$.

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 $R[rd]$. 

### TABLE 7-20 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>

rd | op3 | rs1 | i=0 | x | — | rs2
10 | | | | | | 1
rd | op3 | rs1 | i=0 | x | — | shcnt32
10 | | | | | | 1
rd | op3 | rs1 | i=1 | x=0 | — | shcnt64
10 | | | | | | 1
31 30 29 25 24 19 18 14 13 12 11 6 5 4 0
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 \text{reg}rs1, 0, \text{reg}rd” can be used to convert a 32-bit value to 64 bits, with sign extension into the upper word. “srl \text{reg}rs1, 0, \text{reg}rd” can be used to clear the upper 32 bits of \text{R}[rd].

An attempt to execute a SLL, SRL, or SRA instruction when instruction bits 11:5 are nonzero causes an \textit{illegal\_instruction} exception.

An attempt to execute a SLLX, SRLX, or SRAX instruction when either of the following conditions exist causes an \textit{illegal\_instruction} exception:

- \text{i = 0 or x = 0 and instruction bits 11:5 are nonzero}
- \text{x = 1 and instruction bits 11:6 are nonzero}

**Exceptions**

\textit{illegal\_instruction}
7.118 Signed Multiply (32-bit)

The SMUL and SMULcc instructions are deprecated and should not be used in new software. The MULX instruction should be used instead.

<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>SMUL</td>
<td>00 1011</td>
<td>Signed Integer Multiply</td>
<td>smul rs1, reg_or_imm, reg</td>
<td>D3</td>
</tr>
<tr>
<td>SMULcc</td>
<td>01 1011</td>
<td>Signed Integer Multiply and modify cc's</td>
<td>smulcc rs1, reg_or_imm, reg</td>
<td>D3</td>
</tr>
</tbody>
</table>

Description

The signed multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute “R[rs1][31:0] × R[rs2][31:0]” if i = 0, or “R[rs1][31:0] × sign_ext (simm13)[31:0]” if i = 1. They write the 32 most significant bits of the product into the Y register and all 64 bits of the product into R[rd].

Signed multiply instructions (SMUL, SMULcc) operate on signed integer word operands and compute a signed integer doubleword product.

SMUL does not affect the condition code bits. SMULcc writes the integer condition code bits, icc and xcc, as shown below.

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

Note

32-bit negative (icc.n) and zero (icc.z) condition codes are set according to the less significant word of the product, not according to the full 64-bit result.

Programming Notes

32-bit overflow after SMUL or SMULcc is indicated by Y ≠ (R[rd] >> 31), where “>>” indicates 32-bit arithmetic right-shift.

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

Exceptions

illegal_instruction

See Also

UMUL[cc] on page 355
## 7.119 Store Integer

### Description
The store integer instructions (except store doubleword) copy the whole extended (64-bit) integer, the less significant word, the least significant halfword, or the least significant byte of \( R[rd] \) into memory.

These instructions access memory using the implicit ASI (see page 81). 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 (notably, STX) integer instruction operates atomically.

An attempt to execute a store integer instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an `illegal_instruction` exception.

Any of the following conditions cause a `mem_address_not_aligned` exception:
- an attempt to execute STH when the effective address is not halfword-aligned
- an attempt to execute STW when the effective address is not word-aligned
- an attempt to execute STX when the effective address is not doubleword-aligned

### Exceptions
- `illegal_instruction`
- `mem_address_not_aligned`
- `VA_watchpoint`

### See Also
STTW on page 337

### Instruction Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( \text{op3} )</th>
<th>Operation</th>
<th>( \text{Assembly Language Syntax} )</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STB</td>
<td>00 0101</td>
<td>Store Byte</td>
<td>( stb^{†} , \text{reg}_{rd} [\text{address}] )</td>
<td>A1</td>
</tr>
<tr>
<td>STH</td>
<td>00 0110</td>
<td>Store Halfword</td>
<td>( sth^{†} , \text{reg}_{rs} [\text{address}] )</td>
<td>A1</td>
</tr>
<tr>
<td>STW</td>
<td>00 0100</td>
<td>Store Word</td>
<td>( stw^{◊} , \text{reg}_{rs} [\text{address}] )</td>
<td>A1</td>
</tr>
<tr>
<td>STX</td>
<td>00 1110</td>
<td>Store Extended Word</td>
<td>( stx , \text{reg}_{rd} [\text{address}] )</td>
<td>A1</td>
</tr>
</tbody>
</table>

\( ^{†} \text{synonyms: stub, stab} \quad ^{‡} \text{synonyms: stuh, stsh} \quad ^{◊} \text{synonyms: st, stuw, stsw} \)
7.120 Store Integer into Alternate Space

STBA*ASI, STHA*ASI, STWA*ASI, STXA*ASI

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

\† synonyms: stuba, staba
\‡ synonyms: stuha, stsha
\◊ synonyms: sta, stuwa, stswa

Description

The store integer into alternate space instructions copy the whole extended (64-bit) integer, the less significant word, the least significant halfword, or the least significant byte of \( R[rd]\) into memory.

Store integer to alternate space instructions contain the address space identifier (ASI) to be used for the store in the \( imm_asi \) field if \( i = 0 \), or in the ASI register if \( i = 1 \). The effective address for these instructions is \( R[rs1] + R[rs2] \) if \( i = 0 \), or \( R[rs1] + \text{sign ext}(simm13) \) if \( i = 1 \).

A successful store (notably, STXA) instruction operates atomically.

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

Any of the following conditions cause a mem_address_not_aligned exception:
- an attempt to execute STHA when the effective address is not halfword-aligned
- an attempt to execute STWA when the effective address is not word-aligned
- an attempt to execute STXA when the effective address is not doubleword-aligned

STBA, STHA, and STWA can be used with any of the following ASIs, subject to the privilege mode rules described for the privileged_action exception above. Use of any other ASI with these instructions causes a DAE_invalid_asi exception.

ASIs valid for STBA, STHA, and STWA

04\(_{16}\) ASI_NUCLEUS 0C\(_{16}\) ASI_NUCLEUS_LITTLE
10\(_{16}\) ASI_IF_USER_PRIMARY 18\(_{16}\) ASI_IF_USER_PRIMARY_LITTLE
11\(_{16}\) ASI_IF_USER_SECONDARY 19\(_{16}\) ASI_IF_USER_SECONDARY_LITTLE
14\(_{16}\) ASI_REAL 1C\(_{16}\) ASI_REAL_LITTLE
15\(_{16}\) ASI_REAL_IO 1D\(_{16}\) ASI_REAL_IO_LITTLE
22\(_{16}\) ASI_STBI_AIUP 2A\(_{16}\) ASI_STBI_AIUP_L
23\(_{16}\) ASI_STBI_AIUS 2B\(_{16}\) ASI_STBI_AIUS_L
27\(_{16}\) ASI_STBI_N 2F\(_{16}\) ASI_STBI_N_L
80\(_{16}\) ASI_PRIMARY 88\(_{16}\) ASI_PRIMARY_LITTLE
81\(_{16}\) ASI_SECONDARY 89\(_{16}\) ASI_SECONDARY_LITTLE
STXA can be used with any ASI (including, but not limited to, the above two lists), unless it either (a) violates the privilege mode rules described for the privileged_action exception above or (b) is used with any of the following ASIs, which causes a DAE_invalid_asi exception.

### ASIs invalid for STXA (cause DAE_invalid_asi exception)

<table>
<thead>
<tr>
<th>ASIs</th>
<th>16</th>
<th>16</th>
<th>17</th>
<th>16</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY</td>
<td>1E</td>
<td>E1</td>
<td>1F</td>
<td>E1</td>
<td>E1</td>
</tr>
<tr>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE</td>
<td>1E</td>
<td>E1</td>
<td>1F</td>
<td>E1</td>
<td>E1</td>
</tr>
<tr>
<td>ASI_TWIX_AIUP</td>
<td>22</td>
<td>22</td>
<td>2A</td>
<td>2A</td>
<td>2A</td>
</tr>
<tr>
<td>ASI_TWIX_AIUS</td>
<td>23</td>
<td>23</td>
<td>2B</td>
<td>2B</td>
<td>2B</td>
</tr>
<tr>
<td>ASI_TWIX_REAL</td>
<td>26</td>
<td>26</td>
<td>2E</td>
<td>2E</td>
<td>2E</td>
</tr>
<tr>
<td>ASI_TWIX_N</td>
<td>27</td>
<td>27</td>
<td>2F</td>
<td>2F</td>
<td>2F</td>
</tr>
<tr>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY</td>
<td>1E</td>
<td>E1</td>
<td>1F</td>
<td>E1</td>
<td>E1</td>
</tr>
<tr>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE</td>
<td>1E</td>
<td>E1</td>
<td>1F</td>
<td>E1</td>
<td>E1</td>
</tr>
<tr>
<td>ASI_PRIMARY_NO_FAULT</td>
<td>82</td>
<td>82</td>
<td>8A</td>
<td>8A</td>
<td>8A</td>
</tr>
<tr>
<td>ASI_SECONDARY_NO_FAULT</td>
<td>83</td>
<td>83</td>
<td>8B</td>
<td>8B</td>
<td>8B</td>
</tr>
<tr>
<td>ASI_PST8_PRIMARY</td>
<td>C0</td>
<td>C0</td>
<td>C8</td>
<td>C8</td>
<td>C8</td>
</tr>
<tr>
<td>ASI_PST8_SECONDARY</td>
<td>C1</td>
<td>C1</td>
<td>C9</td>
<td>C9</td>
<td>C9</td>
</tr>
<tr>
<td>ASI_PST16_PRIMARY</td>
<td>C2</td>
<td>C2</td>
<td>C8</td>
<td>C8</td>
<td>C8</td>
</tr>
<tr>
<td>ASI_PST16_SECONDARY</td>
<td>C3</td>
<td>C3</td>
<td>C9</td>
<td>C9</td>
<td>C9</td>
</tr>
<tr>
<td>ASI_PST32_PRIMARY</td>
<td>C4</td>
<td>C4</td>
<td>CC</td>
<td>CC</td>
<td>CC</td>
</tr>
<tr>
<td>ASI_PST32_SECONDARY</td>
<td>C5</td>
<td>C5</td>
<td>CD</td>
<td>CD</td>
<td>CD</td>
</tr>
<tr>
<td>ASI_FL8_PRIMARY</td>
<td>D0</td>
<td>D0</td>
<td>D8</td>
<td>D8</td>
<td>D8</td>
</tr>
<tr>
<td>ASI_FL8_SECONDARY</td>
<td>D1</td>
<td>D1</td>
<td>D9</td>
<td>D9</td>
<td>D9</td>
</tr>
<tr>
<td>ASI_FL16_PRIMARY</td>
<td>D2</td>
<td>D2</td>
<td>DA</td>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>ASI_FL16_SECONDARY</td>
<td>D3</td>
<td>D3</td>
<td>DB</td>
<td>DB</td>
<td>DB</td>
</tr>
<tr>
<td>ASI_BLOCK_COMMIT_PRIMARY</td>
<td>E0</td>
<td>E0</td>
<td>E1</td>
<td>E1</td>
<td>E1</td>
</tr>
<tr>
<td>ASI_BLOCK_COMMIT_SECONDARY</td>
<td>F0</td>
<td>F0</td>
<td>F8</td>
<td>F8</td>
<td>F8</td>
</tr>
<tr>
<td>ASI_BLOCK_SECONDARY</td>
<td>F1</td>
<td>F1</td>
<td>F9</td>
<td>F9</td>
<td>F9</td>
</tr>
</tbody>
</table>

### V8 Compatibility

**Note** The SPARC V8 STA instruction was renamed STWA in the SPARC V9 architecture.

### Exceptions

- `mem_address_not_aligned` (all except STBA)
- `privileged_action`
- `VA_watchpoint`
- `DAE_invalid_asi`
- `DAE_privilege_violation`
- `DAE_nfo_page`

### See Also

- LDA on page 228
- STTWA on page 339
### Block Store [VIS1]

The STBLOCKFD instructions are deprecated and should not be used in new software. A sequence of STDF instructions should be used instead.

The STBLOCKFD instruction is intended to be a processor-specific instruction, which may or may not be implemented in future Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

#### Instruction	ASI Value	Operation	Assembly Language Syntax	Class

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASI Value</th>
<th>Operation Description</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STBLOCKFD</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>A1</td>
</tr>
<tr>
<td></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>A1</td>
</tr>
<tr>
<td></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>A1</td>
</tr>
<tr>
<td></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>A1</td>
</tr>
<tr>
<td></td>
<td>F016</td>
<td>64-byte block store to primary address space</td>
<td>stda freg, [regaddr] #ASI_BLK_P</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td>F116</td>
<td>64-byte block store to secondary address space</td>
<td>stda freg, [regaddr] #ASI_BLK_S</td>
<td>A1</td>
</tr>
<tr>
<td></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>A1</td>
</tr>
<tr>
<td></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>A1</td>
</tr>
<tr>
<td></td>
<td>E016</td>
<td>64-byte block commit store to primary address space</td>
<td>stda freg, [regaddr] #ASI_BLK_COMMIT_P</td>
<td>B1</td>
</tr>
<tr>
<td></td>
<td>E116</td>
<td>64-byte block commit store to secondary address space</td>
<td>stda freg, [regaddr] #ASI_BLK_COMMIT_S</td>
<td>B1</td>
</tr>
</tbody>
</table>

#### Description

A block store instruction references one of several special block-transfer ASIs. Block-transfer ASIs allow block stores to be performed accessing the same address space as normal stores. Little-endian ASIs (those with an ‘L’ suffix) access data in little-endian format; otherwise, the access is assumed to be big-endian. Byte swapping is performed separately for each of the eight double-precision registers accessed by the instruction.

#### Programming Note

The block store instruction, STBLOCKFD, and its companion, LDBLOCKFD, were originally defined to provide a fast mechanism for block-copy operations. However, in modern implementations they are rarely much faster than a sequence of regular loads and stores, so are now deprecated.
STBLOCKF (deprecated)

STBLOCKF\(^D\) 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\(^D\) 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\(^D\) instructions may or may not allow for case (2) above. Such software should be checked to verify that either it always waits for STBLOCKF\(^D\) 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\(^D\) operation is in progress.

A Block Store only guarantees atomicity for each 64-bit (8-byte) portion of the 64 bytes that it stores.

A Block Store with Commit forces the data to be written to memory and invalidates copies in all caches present. As a result, a Block Store with Commit maintains coherency with the I-cache\(^1\). It does not, however, flush instructions that have already been fetched into the pipeline before executing the modified code. If a Block Store with Commit is used to write modified instructions, a FLUSH instruction must still be executed to guarantee that the instruction pipeline is flushed. (See Synchronizing Instruction and Data Memory on page 394 for more information.)

ASIs \(E_{016}\) and \(E_{116}\) are only used for block store-with-commit operations; they are not available for use by block load operations. See Block Load and Store ASIs on page 410 for more information.

Software should assume the following (where “load operation” includes load, load-store, and LDBLOCKF\(^D\) instructions and “store operation” includes store, load-store, and STBLOCKF\(^D\) instructions):

- A STBLOCKF\(^D\) 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\(^D\) 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\(^D\) and subsequent load operations.

- A STBLOCKF\(^D\) 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\(^D\) and subsequent store operations.

- STBLOCKFs do not follow register dependency interlocks, as do ordinary stores.

[Programming Note] STBLOCKF\(^D\) is intended to be a processor-specific instruction (see the warning at the top of page 322). If STBLOCKF\(^D\) 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\(^D\)) instruction are implementation dependent:

- The memory ordering model that STBLOCKF\(^D\) 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\(^D\) (the recommended behavior), or only on accesses to the first eight bytes.

\(^1\) Even if all data stores on a given implementation coherently update the instruction cache (see page 389), stores (other than Block Store with Commit) on SPARC V9 implementations in general do not maintain coherency between instruction and data caches.
STBLOCKF (deprecated)

- Whether STBLOCKFs to non-cacheable (TTE.cp = 0) pages execute in strict program order or not. If not, a STBLOCKFD to a non-cacheable page causes an illegal_instruction exception.
- Whether STBLOCKFD follows register dependency interlocks (as ordinary stores do).
- Whether a non-Commit STBLOCKFD forces the data to be written to memory and invalidates copies in all caches present (as the Commit variants of STBLOCKFD do).
- Any other restrictions on the behavior of STBLOCKFD, 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.ief = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute a STBLOCKFD 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), STBLOCKFD causes a privileged_action exception.

An access caused by STBLOCKFD may trigger a VA_watchpoint exception (impl. dep. #411-S10).

Implementation Note | STBLOCKFD shares an opcode with the STDFA, STPARTIALF, and STSHORTF instructions; they are distinguished by the ASI used.

Exceptions  
illegal_instruction  
fp_disabled  
mem_address_not_aligned  
privileged_action  
VA_watchpoint  
(DAE_privilege_violation)  
(DAE_nfo_page)

See Also  
LDBLOCKFD on page 230  
STDF on page 325
7.122 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>
</table>
| STF         | 10 0100 | 0–31 | Store Floating-Point register | st 
Freg[rd] [address] | A1 |
| STDF        | 10 0111 | rd | Store Double Floating-Point register | std 
Freg[rd] [address] | A1 |
| STQF        | 10 0110 | rd | Store Quad Floating-Point register | stq 
Freg[rd] [address] | C3 |

† Encoded floating-point register value, as described on page 51.

Description
The store single floating-point instruction (STF) copies the contents of the 32-bit floating-point register F_S[rd] into memory.

The store double floating-point instruction (STDF) copies the contents of 64-bit floating-point register F_D[rd] into a word-aligned doubleword in memory. The unit of atomicity for STDF is 4 bytes (one word).

The store quad floating-point instruction (STQF) copies the contents of 128-bit floating-point register F_Q[rd] into a word-aligned quadword in memory. The unit of atomicity for STQF is 4 bytes (one word).

These instructions access memory using the implicit ASI (see page 81). The effective address for these instructions is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + sign_ext(simm13)” if i = 1.

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

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STF or STDF instruction causes an fp_disabled exception.

Any of the following conditions cause an exception:
- an attempt to execute STF, STDF, or STQF when the effective address is not word-aligned causes a mem_address_not_aligned exception
- an attempt to execute STDF when the effective address is word-aligned but not doubleword-aligned 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)).
- an attempt to execute STQF when the effective address is word-aligned but not quadword-aligned causes an STQF_mem_address_not_aligned exception. In this case, trap handler software must emulate the STQF instruction and return (impl. dep. #112-V9-Cs10(a)).

Programming Note
Some compilers issued sequences of single-precision stores for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned stores is expected to be fast, compilers should issue sets of single-precision stores only when they can determine that double- or quadword operands are not properly aligned.
STF / STDF / STQF

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

\textbf{Implementation Note} Since Oracle SPARC Architecture 2011 processors do not implement in hardware instructions (including STQF) that refer to quad-precision floating-point registers, the \texttt{STQF\_mem\_address\_not\_aligned} and \texttt{fp\_exception\_other} (with \texttt{FSR.ftt} = \texttt{invalid\_fp\_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.

\textbf{Exceptions} \texttt{illegal\_instruction}  
\texttt{fp\_disabled}  
\texttt{STDF\_mem\_address\_not\_aligned}  
\texttt{STQF\_mem\_address\_not\_aligned} (not used in Oracle SPARC Architecture 2011)  
\texttt{mem\_address\_not\_aligned}  
\texttt{fp\_exception\_other} (FSR.ftt = invalid_fp_register (STQF only))  
\texttt{VA\_watchpoint}  
\texttt{DAE\_privilege\_violation}  
\texttt{DAE\_nfo\_page}

\textbf{See Also}  
Load Floating-Point Register on page 233  
Block Store on page 322  
Store Floating-Point into Alternate Space on page 327  
Store Floating-Point State Register (Lower) on page 330  
Store Short Floating-Point on page 335  
\texttt{DAE\_invalid\_asi} \texttt{DAE\_invalid\_asi} Store Partial Floating-Point on page 332  
Store Floating-Point State Register on page 341
7.123 Store Floating-Point into Alternate Space

### Description

The store single floating-point into alternate space instruction (STFA) copies the contents of the 32-bit floating-point register \( F_S[\text{rd}] \) into memory.

The store double floating-point into alternate space instruction (STDFA) copies the contents of 64-bit floating-point register \( F_D[\text{rd}] \) into a word-aligned doubleword in memory. The unit of atomicity for STDFA is 4 bytes (one word).

The store quad floating-point into alternate space instruction (STQFA) copies the contents of 128-bit floating-point register \( F_Q[\text{rd}] \) into a word-aligned quadword in memory. The unit of atomicity for STQFA is 4 bytes (one word).

Store floating-point into alternate space instructions contain the address space identifier (ASI) to be used for the load in the \( \text{imm}_\text{asi} \) field if \( i = 0 \) or in the ASI register if \( i = 1 \). The effective address for these instructions is “\( R[\text{rs1}] + R[\text{rs2}] \)” if \( i = 0 \), or “\( R[\text{rs1}] + \text{sign_ext}(\text{simm13}) \)” if \( i = 1 \).

### Exceptions

If the floating-point unit is not enabled (\( \text{FPRS}.\text{lef} = 0 \) or \( \text{PSTATE}.\text{pfl} = 0 \)) or if the FPU is not present, then an attempt to execute a STFA or STDFA instruction causes an \( \text{fp_disabled} \) exception.

Any of the following conditions cause an exception:

- an attempt to execute STFA, STDFA, or STQFA when the effective address is not word-aligned causes a \( \text{mem_address_not_aligned} \) exception
- an attempt to execute STDFA when the effective address is word-aligned but not doubleword-aligned causes an \( \text{STDF_mem_address_not_aligned} \) exception. In this case, trap handler software must emulate the STDFA instruction and return (impl. dep. #110-V9-Cs10(a)).
- an attempt to execute STQFA when the effective address is word-aligned but not quadword-aligned causes an \( \text{STQF_mem_address_not_aligned} \) exception. In this case, trap handler software must emulate the STQFA instruction and return (impl. dep. #112-V9-Cs10(a)).

---

### Tables

#### Instruction Set

<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(^{\text{PAsi}})</td>
<td>11 0100</td>
<td>0–31</td>
<td>Store Floating-Point Register to Alternate Space</td>
<td>sta ( freg_{rd} [\text{regaddr}] \text{imm}<em>\text{asi} ) sta ( freg</em>{rd} [\text{reg_plus_imm}] %\text{asi} )</td>
<td>A1</td>
</tr>
<tr>
<td>STDFA(^{\text{PAsi}})</td>
<td>11 0111 †</td>
<td>Store Double Floating-Point Register to Alternate Space</td>
<td>stda ( freg_{rd} [\text{regaddr}] \text{imm}_\text{asi} )</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>STQFA(^{\text{PAsi}})</td>
<td>11 0110 †</td>
<td>Store Quad Floating-Point Register to Alternate Space</td>
<td>stqa ( freg_{rd} [\text{regaddr}] \text{imm}_\text{asi} )</td>
<td>C3</td>
<td></td>
</tr>
</tbody>
</table>

\(^{†}\) Encoded floating-point register value, as described on page 51.
STFA / STDFA / STQFA

**Implementation Note**: STQFA can be used with any of the following ASIs, subject to the privilege mode rules described for the `privileged_action` exception above. Use of any other ASI with these instructions causes a `DAE_invalid_asi` exception.

<table>
<thead>
<tr>
<th>ASIs valid for STFA and STQFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>04_{16} ASI_NUCLEUS</td>
</tr>
<tr>
<td>10_{16} ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>11_{16} ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>14_{16} ASI_REAL</td>
</tr>
<tr>
<td>15_{16} ASI_REAL_IO</td>
</tr>
<tr>
<td>22_{16} ASI_STBI_AIUP</td>
</tr>
<tr>
<td>23_{16} ASI_STBI_AIUS</td>
</tr>
<tr>
<td>27_{16} ASI_STBI_N</td>
</tr>
<tr>
<td>80_{16} ASI_PRIMARY</td>
</tr>
<tr>
<td>81_{16} ASI_SECONDARY</td>
</tr>
<tr>
<td>E2_{16} ASI_STBI_P</td>
</tr>
<tr>
<td>E3_{16} ASI_STBI_S</td>
</tr>
</tbody>
</table>

`STQFA` can be used with any of the following ASIs, subject to the privilege mode rules described for the `privileged_action` exception above. Use of any other ASI with the `STDFA` instruction causes a `DAE_invalid_asi` exception.

<table>
<thead>
<tr>
<th>ASIs valid for STDFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>04_{16} ASI_NUCLEUS</td>
</tr>
<tr>
<td>10_{16} ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>11_{16} ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>14_{16} ASI_REAL</td>
</tr>
<tr>
<td>15_{16} ASI_REAL_IO</td>
</tr>
<tr>
<td>16_{16} ASI_BLOCK_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>17_{16} ASI_BLOCK_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>80_{16} ASI_PRIMARY</td>
</tr>
<tr>
<td>81_{16} ASI_SECONDARY</td>
</tr>
<tr>
<td>C0_{16} ASI_PST8_PRIMARY</td>
</tr>
<tr>
<td>C1_{16} ASI_PST8_SECONDARY</td>
</tr>
<tr>
<td>C2_{16} ASI_PST16_PRIMARY</td>
</tr>
<tr>
<td>C3_{16} ASI_PST16_SECONDARY</td>
</tr>
</tbody>
</table>
**Exceptions**

- `fp_disabled`
- `STDF_mem_address_not_aligned`
- `STQF_mem_address_not_aligned` (STQFA only) (not used in UA-2011)
- `mem_address_not_aligned`
- `fp_exception_other` (FSR.flt = invalid_fp_register (STQFA only))
- `privileged_action`
- `VA_watchpoint`
- `DAE_invalid_asi`
- `DAE_privilege_violation`
- `DAE_nfo_page`

**See Also**

- Load Floating-Point from Alternate Space on page 235
- Block Store on page 322
- Store Floating-Point on page 325
- Store Short Floating-Point on page 335
- `DAE_invalid_asi` `eDAE_invalid_asi` `eStore Partial Floating-Point` on page 332.

---

**ASIs valid for STDFA**

<table>
<thead>
<tr>
<th>C4₁₆ ASI_PST32_PRIMARY *</th>
<th>CC₁₆ ASI_PST32_PRIMARY_LITTLE *</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5₁₆ ASI_PST32_SECONDARY *</td>
<td>CD₁₆ ASI_PST32_SECONDARY_LITTLE *</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D0₁₆ ASI_FL8_PRIMARY ‡</th>
<th>D8₁₆ ASI_FL8_PRIMARY_LITTLE ‡</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1₁₆ ASI_FL8_SECONDARY ‡</td>
<td>D9₁₆ ASI_FL8_SECONDARY_LITTLE ‡</td>
</tr>
<tr>
<td>D2₁₆ ASI_FL16_PRIMARY ‡</td>
<td>DA₁₆ ASI_FL16_PRIMARY_LITTLE ‡</td>
</tr>
<tr>
<td>D3₁₆ ASI_FL16_SECONDARY ‡</td>
<td>DB₁₆ ASI_FL16_SECONDARY_LITTLE ‡</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E0₁₆ ASI_BLOCK_COMMIT_PRIMARY †</th>
<th>E1₁₆ ASI_BLOCK_COMMIT_SECONDARY †</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0₁₆ ASI_BLOCK_PRIMARY †</td>
<td>F8₁₆ ASI_BLOCK_PRIMARY_LITTLE †</td>
</tr>
<tr>
<td>F1₁₆ ASI_BLOCK_SECONDARY †</td>
<td>F9₁₆ ASI_BLOCK_SECONDARY_LITTLE †</td>
</tr>
</tbody>
</table>

† If this ASI is used with the opcode for STDFA, the STBLOCKFD instruction is executed instead of STFA. For behavior of STBLOCKFD, see Block Store on page 322.

‡ 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 335.

* If this ASI is used with the opcode for STDFA, the STPARTIALF instruction is executed instead of STDFA. For behavior of STPARTIALF, see `DAE_invalid_asi` `eDAE_invalid_asi` `eStore Partial Floating-Point` on page 332.
7.124 Store Floating-Point State Register (Lower)

The STFSR instruction is deprecated and should not be used in new software. The STXFSR instruction should be used instead.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>rd</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STFSR0</td>
<td>10 0101</td>
<td>0</td>
<td>Store Floating-Point State Register (Lower)</td>
<td>st %fsr, [address]</td>
<td>D2</td>
</tr>
<tr>
<td></td>
<td>10 0101</td>
<td>1-31</td>
<td>(see page 341)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The Store Floating-point State Register (Lower) instruction (STFSR) waits for any currently executing FPop instructions to complete, and then it writes the less-significant 32 bits of FSR into memory.

After writing the FSR to memory, STFSR zeroes FSR.ftt.

**V9 Compatibility Note**

FSR.ftt should not be zeroed until it is known that the store will not cause a precise trap.

STFSR accesses memory using the implicit ASI (see page 81). The effective address for this instruction is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + sign_ext(simm13)” if i = 1.

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

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STFSR instruction causes an fp_disabled exception.

STFSR causes a mem_address_not_aligned exception if the effective memory address is not word-aligned.

**V9 Compatibility Note**

Although STFSR is deprecated, Oracle SPARC Architecture implementations continue to support it for compatibility with existing SPARC V8 software. The STFSR instruction is defined to store only the less-significant 32 bits of the FSR into memory, while STXFSR allows SPARC V9 software to store all 64 bits of the FSR.

**Implementation Note**

STFSR shares an opcode with the STXFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = 102, op3 = 10 01012 opcode with an invalid rd value causes an illegal_instruction exception.

**Exceptions**

illegal_instruction
fp_disabled
mem_address_not_aligned
VA_watchpoint
DAE_privilegeViolation
DAE_nfo_page
STFSR (Deprecated)

See Also

Store Floating-Point on page 325
Store Floating-Point State Register on page 341
Description

The partial store instructions are selected by one of the partial store ASIs with the STDFA instruction. Two 32-bit, four 16-bit, or eight 8-bit values from the 64-bit floating-point register \( F_0[rd] \) are conditionally stored at the address specified by \( R[rs1] \), using the mask specified in \( R[rs2] \). STPARTIALF has the effect of merging selected data from its source register, \( F_0[rd] \), into the existing data at the corresponding destination memory locations.

The mask value in \( R[rs2] \) has the same format as the result specified by the edge-handling and pixel compare instructions (see Edge Handling Instructions (no CC) on page 147, Partitioned Signed Compare on page 201, and Partitioned Unsigned Compare on page 203). The most significant bit of the mask (not of
the entire register) corresponds to the most significant part of F[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.

![Diagram of 8-bit partial store mask](image1)

![Diagram of 16-bit partial store mask](image2)

![Diagram of 32-bit partial store mask](image3)

**FIGURE 7-45** Mask Format for Partial Store

**Exceptions.**

**IMPL. DEP. #__**: It is implementation-dependent whether any of the following exceptions are suppressed when the store-mask in R[rs2] = 0. In particular, a store mask with value 0 may or may not prevent a `mem_address_not_aligned` exception or data access exception from occurring, if the conditions for triggering such an exception are otherwise met.

A Partial Store instruction can cause a virtual watchpoint exception when the following conditions are met:

- The virtual address in R[rs1] matches the address in the VA Data Watchpoint Register.
- The byte store mask in R[rs2] indicates that a byte, halfword or word is to be stored.
- The Virtual (Physical) Data Watchpoint Mask in ASI_DCU_WATCHPOINT_CONTROL_REG indicates that one or more of the bytes to be stored at the watched address is being watched.

For data watchpoints of partial stores in Oracle SPARC Architecture 2011, the byte store mask (R[rs2]) in the Partial Store instruction is ignored, and a watchpoint exception can occur even if the mask is zero (that is, no data will be written to memory). The ASI_DCU_WATCHPOINT_CONTROL_REG Data Watchpoint masks are only checked for nonzero value (watchpoint enabled) (impl. dep. #249).

An attempt to execute a STPARTIALF instruction when i = 1 causes an `illegal_instruction` exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STPARTIALF instruction causes an `fp_disabled` exception.

Any of the following conditions cause an exception:

- an attempt to execute STPARTIALF when the effective address is not word-aligned causes a `mem_address_not_aligned` exception
STPARTIALF

An attempt to execute STPARTIALF when the effective address is word-aligned but not doubleword-aligned causes an **STDF_mem_address_not_aligned** exception. In this case, trap handler software must emulate the STPARTIALF 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 \( C_{016} - C_{16} \) and \( C_{816} - C_{D16} \) 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 237.

**Implementation Note:** STPARTIALF shares an opcode with the STBLOCKF\(^D\), STDFA, and STSHORTF instructions; they are distinguished by the ASI used.

**Exceptions**

- **illegal_instruction**
- **fp_disabled**
- **mem_address_not_aligned**
- **VA_watchpoint** (see text)
- **DAE_privilege_violation**
- **DAE_nc_page**
- **DAE_nfo_page**
CHAPTER 7 • Instructions

7.126 Store Short Floating-Point

Description
The short floating-point store instruction allows 8- and 16-bit stores to be performed from the floating-point registers. Short stores access the low-order 8 or 16 bits of the register.

Little-endian ASIs transfer data in little-endian format from memory; otherwise, memory is assumed to be big-endian. Short floating-point stores are typically used with the `FALIGNDATAg` instruction to assemble or store 64 bits on noncontiguous components.

Implementation Note
STSHORTF shares an opcode with the `STBLOCKF D, STDFA`, and `STPARTIALF` instructions; they are distinguished by the ASI used.

If the floating-point unit is not enabled (`FPRS.fef = 0` or `PSTATE.pef = 0`) or if the FPU is not present, then an attempt to execute a STSHORTF instruction causes an `fp_disabled` exception.

An 8-bit STSHORTF (using ASI D0, D1, D8, or D9) can be performed to an arbitrary memory address (no alignment requirement).

An attempt to execute a 16-bit STSHORTF (using ASI D2, D3, DA, or DB) when the effective address is not halfword-aligned causes a `mem_address_not_aligned` exception.

Exceptions
`fp_disabled`
`mem_address_not_aligned`
`VA_watchpoint`
`DAE_privilege_violation`
`DAE_nfo_page`
See Also

LDSHORTF on page 240

Align Data (using GSR.align) on page 151
STTW (Deprecated)

7.127 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>STTWD</td>
<td>00 0111</td>
<td>Store Integer Twin Word</td>
<td>sttw reg, [address]</td>
<td>D2</td>
</tr>
</tbody>
</table>

† The original assembly language syntax for this instruction used an “std” instruction mnemonic, which is now deprecated. Over time, assemblers will support the new “sttw” mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original “std” mnemonic.

Description

The store integer twin word instruction (STTW) copies two words from an R register pair into memory. The least significant 32 bits of the even-numbered R register are written into memory at the effective address, and the least significant 32 bits of the following odd-numbered R register are written into memory at the “effective address + 4”.

The least significant bit of the rd field of a store twin word instruction is unused and should always be set to 0 by software.

STTW accesses memory using the implicit ASI (see page 81). 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 Oracle SPARC Architecture 2011 implementations.)

An attempt to execute an STTW instruction when either of the following conditions exist causes an illegal_instruction exception:

- destination register number rd is an odd number (is misaligned)
- i = 0 and instruction bits 12:5 are nonzero

STTW causes a mem_address_not_aligned exception if the effective address is not doubleword-aligned.

With respect to little-endian memory, an STTW instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.
**STTW (Deprecated)**

**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, an STX instruction should be used for the memory access in the emulation code to preserve atomicity. Emulation software should examine TSTATE[TL].pstate.cle (and, if appropriate, TTE.ie) to determine the endianness of the emulated memory access.

Note that the value of TTE.ie is not saved during a trap. Therefore, if it is examined in the emulation trap handler, that should be done as quickly as possible, to minimize the window of time during which the value of TTE.ie could possibly be changed from the value it had at the time of the attempted execution of STTW.

**Exceptions**

- unimplemented_STTW (not used in Oracle SPARC Architecture 2011)
- illegal_instruction
- mem_address_not_aligned
- VA_watchpoint
- DAE_privilegeViolation
- DAE_nfo_page

**See Also**

- STW/STX on page 319
- STTWA on page 339
### 7.128 Store Integer Twin Word into Alternate Space

The STTWA instruction is deprecated and should not be used in new software. The STXA instruction should be used instead.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>STTWA</td>
<td>01 0111</td>
<td>Store Twin Word into Alternate Space</td>
<td>\texttt{sttwa reg} \texttt{rd} \texttt{regaddr} \texttt{imm} \texttt{asi} \texttt{sttwa reg} \texttt{rd} \texttt{reg_plus_imm} \texttt{%asi}</td>
<td>D2, Y3†</td>
</tr>
</tbody>
</table>

†The original assembly language syntax for this instruction used an “\texttt{stda}” instruction mnemonic, which is now deprecated. Over time, assemblers will support the new “\texttt{sttwa}” mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original “\texttt{stda}” mnemonic.

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

**Description**

The store twin word integer into alternate space instruction (STTWA) copies two words from an \texttt{R} register pair into memory. The least significant 32 bits of the even-numbered \texttt{R} register are written into memory at the effective address, and the least significant 32 bits of the following odd-numbered \texttt{R} register are written into memory at the “effective address + 4”.

The least significant bit of the \texttt{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 \texttt{imm} \texttt{asi} field if \(i = 0\), or in the ASI register if \(i = 1\). The effective address for these instructions is “\texttt{R[rs1]} + \texttt{R[rs2]}” if \(i = 0\), or “\texttt{R[rs1]} + \texttt{sign_ext (simm13)}” if \(i = 1\).

A successful store twin word instruction operates atomically.

With respect to little-endian memory, an STTWA instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.

**IMPL. DEP. #108-V9b:** It is implementation dependent whether STTWA is implemented in hardware. If not, an attempt to execute it will cause a \texttt{unimplemented_STTWA} exception. (STTWA is implemented in hardware in all Oracle SPARC Architecture 2011 implementations.)

An attempt to execute an STTWA instruction with a misaligned (odd) destination register number \texttt{rd} causes an \texttt{illegal_instruction} exception.

STTWA causes a \texttt{mem_address_not_aligned} exception if the effective address is not doubleword-aligned.

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

STTWA can be used with any of the following ASIs, subject to the privilege mode rules described for the privileged_action exception above. Use of any other ASI with this instruction causes a DAE_invalid_asi exception (impl. dep. #300-U4-Cs10).

<table>
<thead>
<tr>
<th>ASIs valid for STTWA</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 04_{16} ) ASI_NUCLEUS</td>
</tr>
<tr>
<td>( 10_{16} ) ASI_AS_IF_USER_PRIMARY</td>
</tr>
<tr>
<td>( 11_{16} ) ASI_AS_IF_USER_SECONDARY</td>
</tr>
<tr>
<td>( 14_{16} ) ASI_REAL</td>
</tr>
<tr>
<td>( 15_{16} ) ASI_REAL_IO</td>
</tr>
<tr>
<td>( 22_{16} ) ASI_STBI_AIUP</td>
</tr>
<tr>
<td>( 23_{16} ) ASI_STBI_AIUS</td>
</tr>
<tr>
<td>( 27_{16} ) ASI_STBI_N</td>
</tr>
<tr>
<td>( 80_{16} ) ASI_PRIMARY</td>
</tr>
<tr>
<td>( 81_{16} ) ASI_SECONDARY</td>
</tr>
<tr>
<td>( E2_{16} ) ASI_STBI_P</td>
</tr>
<tr>
<td>( E3_{16} ) ASI_STBI_S</td>
</tr>
</tbody>
</table>

**Programming Note**

Nontranslating ASIs (see page 397) may only be accessed using STXA (not STTWA) instructions. If an STTWA referencing a nontranslating ASI is executed, per the above table, it generates a DAE_invalid_asi exception (impl. dep. #300-U4-Cs10).

**Programming Notes**

STTWA 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 STTWA.

If STTWA is emulated in software, an STXA instruction should be used for Emulation software should examine TSTATE[TL].pstate.clie (and, if appropriate, TTE.ie) to determine the endianness of the emulated memory access.

Note that the value of TTE.ie is not saved during a trap. Therefore, if it is examined in the emulation trap handler, that should be done as quickly as possible, to minimize the window of time during which the value of TTE.ie could possibly be changed from the value it had at the time of the attempted execution of STTWA.

**Exceptions**

unimplemented_STTW
illegal_instruction
mem_address_not_aligned
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nfo_page

**See Also**

STWA/STXA on page 320
STTW on page 337
**STXFSR**

### 7.129 Store Floating-Point State Register

<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>STXFSR</td>
<td>10 0101</td>
<td>1</td>
<td>Store Floating-Point State register</td>
<td>stx %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>

**Description**

The store floating-point state register instruction (STXFSR) waits for any currently executing FPop instructions to complete, and then it writes all 64 bits of the FSR into memory.

STXFSR zeroes FSR.ftt after writing the FSR to memory.

**Implementation Note**

FSR.ftt should not be zeroed by STXFSR until it is known that the store will not cause a precise trap.

STXFSR accesses memory using the implicit ASI (see page 81). The effective address for this instruction is “R[rs1] + R[rs2]” if \( i = 0 \), or “R[rs1] + sign_ext(simm13)” if \( i = 1 \).

**Exceptions.** An attempt to execute a STXFSR instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

If the floating-point unit is not enabled (FPRS.ief = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STXFSR instruction causes an fp_disabled exception.

If the effective address is not doubleword-aligned, an attempt to execute an STXFSR instruction causes a mem_address_not_aligned exception.

**Implementation Note**

STXFSR shares an opcode with the (deprecated) STFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the \( op = 102, op3 = 10 01012 \) opcode with an invalid rd value causes an illegal_instruction exception.

**See Also**

Load Floating-Point State Register on page 251
Store Floating-Point on page 325
Store Floating-Point State Register (Lower) on page 330
7.130 Subtract

These instructions compute “R[rs1] – R[rs2]” if \( i = 0 \), or “R[rs1] – sign_ext(simm13)” if \( i = 1 \), and write the difference into R[rd].

SUBC and SUBCc (“SUBtract with carry”) also subtract the CCR register’s 32-bit carry (icc.c) bit; that is, they compute “R[rs1] – R[rs2] – icc.c” or “R[rs1] – sign_ext(simm13) – icc.c” and write the difference into R[rd].

SUBcc and SUBCc modify the integer condition codes (CCR.icc and CCR.xcc). A 32-bit overflow (CCR.icc.v) occurs on subtraction if bit 31 (the sign) of the operands differs and bit 31 (the sign) of the difference differs from R[rs1]{31}. A 64-bit overflow (CCR.xcc.v) occurs on subtraction if bit 63 (the sign) of the operands differs and bit 63 (the sign) of the difference differs from R[rs1]{63}.

A SUBcc instruction with \( rd = 0 \) can be used to effect a signed or unsigned integer comparison. See the cmp synthetic instruction in Appendix C, Assembly Language Syntax.

SUBC and SUBCc read the 32-bit condition codes’ carry bit (CCR.icc.c), not the 64-bit condition codes’ carry bit (CCR.xcc.c).

An attempt to execute a SUB instruction when \( i = 0 \) and instruction bits 12:5 are nonzero causes an illegal_instruction exception.
7.131 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.

### 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 81). The effective address for these instructions is “\( R[rs1] + R[rs2] \)” if \( i = 0 \), or “\( R[rs1] + \text{sign_ext}(\text{simm13}) \)” if \( i = 1 \).

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

If the effective address is not word-aligned, an attempt to execute a SWAP instruction causes a mem_address_not_aligned exception.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

### Exceptions

- illegal_instruction
- mem_address_not_aligned
- VA_watchpoint
- DAE_privilegeViolation
- DAE_nfo_page

---

**Opcode** | op3 | Operation | Assembly Language Syntax | Class
---|---|---|---|---
SWAP | 00 1111 | Swap Register with Memory | swap [address], reg, rd | D2

<table>
<thead>
<tr>
<th>11</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i=0</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>11</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i=1</th>
<th>simm13</th>
</tr>
</thead>
</table>
7.132 Swap Register with Alternate Space Memory

**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 effective address for this instruction is “\( R[rs1] + R[rs2] \)” if \( i = 0 \), or “\( R[rs1] + \text{sign_ext}(\text{simm}13) \)” if \( i = 1 \).

This instruction causes a \textit{mem_address_not_aligned} exception if the effective address is not word-aligned. It causes a \textit{privileged_action} exception if PSTATE.priv = 0 and bit 7 of the ASI is 0.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep #120-V9).

If the effective address is not word-aligned, an attempt to execute a SWAPA instruction causes a \textit{mem_address_not_aligned} exception.

In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0, this instruction causes a \textit{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 \textit{privileged_action} exception.

SWAPA can be used with any of the following ASIs, subject to the privilege mode rules described for the \textit{privileged_action} exception above. Use of any other ASI with this instruction causes a \textit{DAE_invalid_asi} exception.

**ASIs valid for SWAPA**

- \( 04_{16} \text{ ASI_NUCLEUS} \)
- \( 10_{16} \text{ ASI_AS_IF_USER_PRIMARY} \)
- \( 11_{16} \text{ ASI_AS_IF_USER_SECONDARY} \)
- \( 14_{16} \text{ ASI_REAL} \)
- \( 80_{16} \text{ ASI_PRIMARY} \)
- \( 81_{16} \text{ ASI_SECONDARY} \)

---

**Opcode**

<table>
<thead>
<tr>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>SWAPAD, PASI</td>
<td>\text{swapa} \ [regaddr] \ \text{imm}<em>\text{asi}, \ \text{reg}</em>\text{rd} \</td>
<td>\text{D2}, \ Y3‡</td>
</tr>
<tr>
<td>00</td>
<td>SWAPAD, PASI</td>
<td>\text{swapa} \ [reg_plus_imm] \ %asi, \ \text{reg}_\text{rd} \</td>
<td>\text{D2}, \ Y3‡</td>
</tr>
</tbody>
</table>

‡ \( Y3 \) for restricted ASIs (00_{16}-7F_{16}); \( D2 \) for unrestricted ASIs (80_{16}-FF_{16})
Exceptions

- `mem_address_not_aligned`
- `privileged_action`
- `VA_watchpoint`
- `DAE_invalid_asi`
- `DAE_privilege_violation`
- `DAE_nc_page`
- `DAE_nfo_page`
### 7.133 Tagged Add

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

#### Description

This instruction computes a sum that is \( R[rs1] + R[rs2] \) if \( i = 0 \), or \( R[rs1] + \text{sign_ext}(\text{simm13}) \) if \( i = 1 \).

TADDcc modifies the integer condition codes (icc and xcc).

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the addition generates 32-bit arithmetic overflow (that is, both operands have the same value in bit 31 and bit 31 of the sum is different).

If a TADDcc causes a tag overflow, the 32-bit overflow bit (CCR.icc.v) is set to 1; if TADDcc does not cause a tag overflow, CCR.icc.v is set to 0.

In either case, the remaining integer condition codes (both the other CCR.icc bits and all the CCR.xcc bits) are also updated as they would be for a normal ADD instruction. In particular, the setting of the CCR.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). CCR.xcc.v is set based on the 64-bit arithmetic overflow condition, like a normal 64-bit add.

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

#### Exceptions

illegal_instruction

#### See Also

TADDccTVD on page 347
TSUBcc on page 351
7.134 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.

**SPARC V8 Compatibility Note**

TADDccTV traps based on the 32-bit overflow condition, just as in the SPARC V8 architecture. Although the tagged add instructions set the 64-bit condition codes CCR.xcc, there is no form of the instruction that traps on the 64-bit overflow condition.

**Exceptions**

*illegal_instruction*

*tag_overflow*

**See Also**

TADDcc on page 346

TSUBccTV on page 352
## 7.135 Trap on Integer Condition Codes (Tcc)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>cond</th>
<th>Operation</th>
<th>cc Test Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA</td>
<td>11</td>
<td>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</td>
<td>0000</td>
<td>Trap Never</td>
<td>tn i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TNE</td>
<td>11</td>
<td>1001</td>
<td>Trap on Not Equal</td>
<td>not Z tne† i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TE</td>
<td>11</td>
<td>0001</td>
<td>Trap on Equal</td>
<td>Z te‡ i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TG</td>
<td>11</td>
<td>1010</td>
<td>Trap on Greater</td>
<td>not (Z or (N xor V)) tg i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TLE</td>
<td>11</td>
<td>0010</td>
<td>Trap on Less or Equal</td>
<td>Z or (N xor V) tle i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TGE</td>
<td>11</td>
<td>1011</td>
<td>Trap on Greater or Equal</td>
<td>not (N xor V) tge i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TL</td>
<td>11</td>
<td>0011</td>
<td>Trap on Less</td>
<td>N xor V tl i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TGU</td>
<td>11</td>
<td>1100</td>
<td>Trap on Greater, Unsigned</td>
<td>not (C or Z) tgu i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TLEU</td>
<td>11</td>
<td>0100</td>
<td>Trap on Less or Equal, Unsigned</td>
<td>(C or Z) tleu i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TCC</td>
<td>11</td>
<td>1101</td>
<td>Trap on Carry Clear</td>
<td>not C tcc◊ i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TCS</td>
<td>11</td>
<td>0101</td>
<td>Trap on Carry Set</td>
<td>C tcs V i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TPOS</td>
<td>11</td>
<td>1110</td>
<td>Trap on Positive or zero</td>
<td>not N tpos i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TNEG</td>
<td>11</td>
<td>0110</td>
<td>Trap on Negative</td>
<td>N tneg i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TVC</td>
<td>11</td>
<td>1111</td>
<td>Trap on Overflow Clear</td>
<td>not V tvc i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
<tr>
<td>TVS</td>
<td>11</td>
<td>0111</td>
<td>Trap on Overflow Set</td>
<td>V tvs i_or_x_cc, software_trap_number</td>
<td>A1</td>
</tr>
</tbody>
</table>

† synonym: tnz ‡ synonym: tz ◊ synonym: tgeu ∇ synonym: tlu

### Condition Codes Evaluated

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

<table>
<thead>
<tr>
<th>10 — cond</th>
<th>op3</th>
<th>rs1</th>
<th>i=0 cc1 cc0</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 — cond</td>
<td>op3</td>
<td>rs1</td>
<td>i=1 cc1 cc0</td>
<td>—</td>
<td>imm_trap_#</td>
</tr>
</tbody>
</table>

### Condition Codes Evaluated

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

---

© Oracle SPARC Architecture 2011 • Draft D1.0.0, 12 Jan 2016
**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 \( "R[rs1] + \text{imm\_trap\_#}" \). 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 \( "R[rs1] + \text{imm\_trap\_#}" \). 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, because SWTN is a 7-bit value in nonprivileged mode)</td>
</tr>
<tr>
<td></td>
<td>(256 \leq \text{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 \text{TT} \leq 383)</td>
<td>(384 \leq \text{TT} \leq 511)</td>
</tr>
</tbody>
</table>

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

**Exceptions.** An attempt to execute a Tcc instruction when any of the following conditions exist causes an illegal_instruction exception:

- instruction bit 29 is nonzero
- \( i = 0 \) and instruction bits 10:5 are nonzero
- \( i = 1 \) and instruction bits 10:8 are nonzero
- \( cc0 = 1 \)

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

If a Tcc instruction causes a trap_instruction trap, 256 plus the SWTN value is written into TT[TL]. Then the trap is taken and the virtual processor performs the normal trap entry procedure, as described in Trap Processing on page 432.
Exceptions

- illegal_instruction
- control_transfer_instruction (impl. dep. #450-S20)
- trap_instruction (0 ≤ SWTN ≤ 127)
- htrap_instruction (128 ≤ SWTN ≤ 255)
7.136 Tagged Subtract

This instruction computes \( R[rs1] - R[rs2] \) if \( i = 0 \), or \( R[rs1] - \text{sign}_\text{ext}(\text{simm13}) \) 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 \textit{illegal\_instruction} exception.

**Exceptions**

\textit{illegal\_instruction}

**See Also**

TADDcc on page 346
TSUBccTV\textsuperscript{D} on page 352
7.137 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.

### Exceptions
illegal_instruction

### Tag Overflow

### See Also
TADDccTV\[^D\] on page 347
TSUBcc on page 351
7.138 Unsigned Divide (64-bit ÷ 32-bit)

The unsigned divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If \( i = 0 \), they compute \( (Y::R[rs1][31:0]) \div R[rs2][31:0] \). Otherwise (that is, if \( i = 1 \)), the divide instructions compute \( (Y::R[rs1][31:0]) \div (\text{sign ext}(\text{simm13})[31:0]) \). In either case, if overflow does not occur, the less significant 32 bits of the integer quotient are sign- or zero-extended to 64 bits and are written into \( R[rd] \).

The contents of the \( Y \) register are undefined after any 64-bit by 32-bit integer divide operation.

**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 \( (\text{sign ext}(\text{simm13})[31:0]) \) 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).

The result of an unsigned divide instruction can overflow the less significant 32 bits of the destination register \( R[rd] \) under certain conditions. When overflow occurs, the largest appropriate unsigned integer is returned as the quotient in \( R[rd] \). The condition under which overflow occurs and the value returned in \( R[rd] \) under this condition are specified in TABLE 7-21.

<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^{32} )</td>
<td>( 2^{32} - 1 )</td>
</tr>
<tr>
<td></td>
<td>( \text{(0000 0000 FFFF FFFF)}_{16} )</td>
</tr>
</tbody>
</table>
UDIV, UDIVcc  (Deprecated)

UDIV does not affect the condition code bits. UDIVcc writes the integer condition code bits as shown in the following table. Note that negative (N) and zero (Z) are set according to the value of R[rd] after it has been set to reflect overflow, if any.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Effect on bit of UDIVcc instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc.n</td>
<td>Set if R[rd][31] = 1</td>
</tr>
<tr>
<td>icc.z</td>
<td>Set if R[rd][31:0] = 0</td>
</tr>
<tr>
<td>icc.v</td>
<td>Set if overflow (per TABLE 7-21)</td>
</tr>
<tr>
<td>icc.c</td>
<td>Zero</td>
</tr>
<tr>
<td>xcc.n</td>
<td>Set if R[rd][63] = 1</td>
</tr>
<tr>
<td>xcc.z</td>
<td>Set if R[rd][63:0] = 0</td>
</tr>
<tr>
<td>xcc.v</td>
<td>Zero</td>
</tr>
<tr>
<td>xcc.c</td>
<td>Zero</td>
</tr>
</tbody>
</table>

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

Exceptions  
- illegal_instruction
- division_by_zero

See Also  
- RDY on page 295
- SDIV[cc] on page 310,
- UMUL[cc] on page 355
7.139  Unsigned Multiply (32-bit)

The UMUL and UMULcc instructions are deprecated and should not be used in new software. The MULX instruction should be used instead.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>op3</th>
<th>Operation</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMULD</td>
<td>00</td>
<td>Unsigned Integer Multiply</td>
<td>umul  reg_{rs1}, reg_{or_imm}, reg_{rd}</td>
<td>D3</td>
</tr>
<tr>
<td>UMULccD</td>
<td>01</td>
<td>Unsigned Integer Multiply and modify cc’s</td>
<td>umulcc reg_{rs1}, reg_{or_imm}, reg_{rd}</td>
<td>D3</td>
</tr>
</tbody>
</table>

**Description**

The unsigned multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute “\(R[rs1]\{31:0\} \times R[rs2]\{31:0\}\)” if \(i=0\), or “\(R[rs1]\{31:0\} \times \text{sign_ext}(simm13)\{31:0\}\)” if \(i=1\). They write the 32 most significant bits of the product into the \(Y\) register and all 64 bits of the product into \(R[rd]\).

Unsigned multiply instructions (UMUL, UMULcc) operate on unsigned integer word operands and compute an unsigned integer doubleword product.

UMUL does not affect the condition code bits. UMULcc writes the integer condition code bits, icc and xcc, as shown below.

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

**Note**

32-bit negative (icc.n) and zero (icc.z) condition codes are set according to the less significant word of the product, not according to the full 64-bit result.

**Programming Notes**

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

An attempt to execute a UMUL or UMULcc instruction when \(i = 0\) and instruction bits 12:5 are nonzero causes an illegal_instruction exception.

**Exceptions**

illegal_instruction

**See Also**

RDY on page 295
SMUL\[cc\] on page 318,
UDIV\[cc\] on page 353
### 7.140 Integer Multiply High (64-bit) \[VIS 3\]

The UMULXHI instruction is new and is not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, it currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

<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>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMULXHI</td>
<td>0001 0110</td>
<td>Unsigned integer multiply yielding i64 i64 i64</td>
<td>umulxhi</td>
<td>regrs1, regrs2, regrd</td>
<td>C1</td>
<td>OSA 2011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Description

UMULXHI computes “R[rs1] × R[rs2]” and writes the upper 64-bits of the 128-bit product into R[rd]. The two 64-bit operands are treated as unsigned integer values.

UMULXHI does not modify any condition codes.

\[
\begin{array}{c}
\text{rs1} \\
\times \\
\text{rs2}
\end{array}
\]

\[
\begin{array}{c}
\text{rd} \\
\text{UMULXHI} \\
\text{MULX}
\end{array}
\]

#### FIGURE 7-46 UMULXHI

UMULXHI, in conjunction with MULX, can be used to multiply very large numbers (e.g., 1024b × 1024b) together. Most of the required multiplications are unsigned. However, the few signed 64b X unsigned 64b and signed 64b X signed 64b “MULXHI” products can be calculated via UMULXHI and other standard SPARC V9 instructions:

Signed 64 bit × signed 64 bit \([127:64]\) =

\[
\text{UMULXHI product} - (R[rs1][63] × R[rs2] + R[rs2][63] × R[rs1])
\]

Unsigned 64 bit × signed 64 bit\([127:64]\) =

\[
\text{UMULXHI product} - (R[rs2][63] × R[rs1])
\]

#### Exceptions

None
## 7.141 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;13&lt;/sup&gt;</td>
<td>0</td>
<td>Write Y register (deprecated)</td>
<td>wr reg&lt;sub&gt;y&lt;/sub&gt;, reg_or_imm, %y</td>
<td>D3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRCCCR</td>
<td>2</td>
<td>Write Condition Codes register</td>
<td>wr reg&lt;sub&gt;y&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;y&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;y&lt;/sub&gt;, reg_or_imm, %fprs</td>
<td>A1</td>
</tr>
<tr>
<td></td>
<td>7–13</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>(0E&lt;sub&gt;16&lt;/sub&gt;) Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>(0F&lt;sub&gt;16&lt;/sub&gt;) used at higher privilege level</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16–18</td>
<td>Reserved (impl. dep. #8-V8-Cs20, #9-(10-12&lt;sub&gt;16&lt;/sub&gt;V8-Cs20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRGSR</td>
<td>19</td>
<td>(13&lt;sub&gt;16&lt;/sub&gt;) Write General Status register (GSR)</td>
<td>wr reg&lt;sub&gt;y&lt;/sub&gt;, reg_or_imm, %gsr</td>
<td>A1</td>
</tr>
<tr>
<td>WROSOFTINT_SET&lt;sup&gt;P&lt;/sup&gt;</td>
<td>20</td>
<td>(14&lt;sub&gt;16&lt;/sub&gt;) Set bits of per-virtual processor Soft Interrupt register</td>
<td>wr reg&lt;sub&gt;y&lt;/sub&gt;, reg_or_imm, %softint_set</td>
<td>N–</td>
</tr>
<tr>
<td>WROSOFTINT_CLR&lt;sup&gt;P&lt;/sup&gt;</td>
<td>21</td>
<td>(15&lt;sub&gt;16&lt;/sub&gt;) Clear bits of per-virtual processor Soft Interrupt register</td>
<td>wr reg&lt;sub&gt;y&lt;/sub&gt;, reg_or_imm, %softint_clr</td>
<td>N–</td>
</tr>
<tr>
<td>WROSOFTINT&lt;sup&gt;P&lt;/sup&gt;</td>
<td>22</td>
<td>(16&lt;sub&gt;16&lt;/sub&gt;) Write per-virtual processor Soft Interrupt register</td>
<td>wr reg&lt;sub&gt;y&lt;/sub&gt;, reg_or_imm, %softint</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>(17&lt;sub&gt;16&lt;/sub&gt;) Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>(18&lt;sub&gt;16&lt;/sub&gt;) 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>(19&lt;sub&gt;16&lt;/sub&gt;) Write System Tick Compare register</td>
<td>wr reg&lt;sub&gt;y&lt;/sub&gt;, reg_or_imm, %stick_cmpr†</td>
<td>N–</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>(1A&lt;sub&gt;16&lt;/sub&gt;) Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>(1B&lt;sub&gt;16&lt;/sub&gt;) Pause virtual processor</td>
<td>wr reg&lt;sub&gt;y&lt;/sub&gt;, reg_or_imm, %pause</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>(1C&lt;sub&gt;16&lt;/sub&gt;) Implementation dependent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>(1D&lt;sub&gt;16&lt;/sub&gt;) Implementation dependent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>(1E&lt;sub&gt;16&lt;/sub&gt;) Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>(1F&lt;sub&gt;16&lt;/sub&gt;) Implementation dependent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† The original assembly language names for %stick and %stick_cmpr were, respectively, %sys_tick and %sys_tick_cmpr, which are now deprecated. Over time, assemblers will support the new %tick and %tick_cmpr names for these registers (which are consistent with %tick). In the meantime, some existing assemblers may only recognize the original names.
**WRasr**

**Description**

The WRasr instructions each store a value to the writable fields of the ancillary state register (ASR) specified by `rd`.

The value stored by these instructions (other than the implementation-dependent variants) is as follows: if `i = 0`, store the value \( \text{"R[rs1] xor R[rs2]"} \); if `i = 1`, store \( \text{"R[rs1] xor sign_ext(simm13)"} \).

*Note* | The operation is exclusive-or.

The WRasr instruction with `rd = 0` is a (deprecated) WRY instruction (which should not be used in new software). WRY is not a delayed-write instruction; the instruction immediately following a WRY observes the new value of the `Y` register.

**The WRY instruction is deprecated. It is recommended that all instructions that reference the Y register be avoided.**

WRCCR, WRFPRS, and WRASI are not delayed-write instructions. The instruction immediately following a WRCCR, WRFPRS, or WRASI observes the new value of the CCR, FPRS, or ASI register.

WRFPRS waits for any pending floating-point operations to complete before writing the FPRS register.

WRPAUSE writes a value to the `PAUSE` register, requesting a temporary suspension of instruction execution on the virtual processor. The value written to `PAUSE` indicates the requested number of processor cycles to pause. See Pause on page 283 and Pause Count (PAUSE) Register (ASR 27) on page 60 for details.

**IMPL. DEP. # 48-V8-Cs20**: WRasr instructions with `rd` of 16-18, 28, 29, or 31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRasr instruction using one of those `rd` values, the following are implementation dependent:
- the interpretation of bits 18:0 in the instruction
- the operation(s) performed (for example, `xor`) to generate the value written to the ASR
- whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20), and
- whether an attempt to execute the instruction causes an `illegal_instruction` exception.

**V9 Compatibility Notes**

Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers.

The SPARC V8 WRIER, WRPSR, WRWIM, and WRTBR instructions do not exist in the Oracle SPARC Architecture because the IER, PSR, TBR, and WIM registers do not exist in the Oracle SPARC Architecture.

See Ancillary State Registers on page 48 for more detailed information regarding ASR registers.

**Exceptions.** An attempt to execute a WRasr instruction when any of the following conditions exist causes an `illegal_instruction` exception:
- `i = 0` and instruction bits 12:5 are nonzero
- `rd` contains a reserved value (that is not assigned for implementation-dependent use)
- `rd = 15` and \((rs1 \neq 0) \text{ or } (i = 0)\)

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

If the floating-point unit is not enabled (`FPRS.hdf = 0` or `PSTATE.pef = 0`) or if the FPU is not present, then an attempt to execute a WRGSR instruction causes an `fp_disabled` exception.

**Implementation Note** | WRasr to ASR 27 (1B16) is a `PAUSE` instruction. See Pause on page 283 for details.
WRasr

Exceptions
- illegal_instruction
- privileged_opcode
- fp_disabled

See Also
- PAUSE on page 283
- RDasr on page 295
- WRPR on page 360
7.1.42 Write Privileged Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op3</th>
<th>Operation</th>
<th>rd</th>
<th>Assembly Language Syntax</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRPR &amp;</td>
<td>11 0010</td>
<td>Write Privileged register</td>
<td></td>
<td></td>
<td>A1</td>
</tr>
<tr>
<td>TPC</td>
<td>0</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %tpc</td>
<td></td>
</tr>
<tr>
<td>TNPC</td>
<td>1</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %tnpc</td>
<td></td>
</tr>
<tr>
<td>TSTATE</td>
<td>2</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %tstate</td>
<td></td>
</tr>
<tr>
<td>TT</td>
<td>3</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %tt</td>
<td></td>
</tr>
<tr>
<td>(illegal_instruction)</td>
<td>4</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %tba</td>
<td></td>
</tr>
<tr>
<td>TBA</td>
<td>5</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %tba</td>
<td></td>
</tr>
<tr>
<td>PSTATE</td>
<td>6</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %pstate</td>
<td></td>
</tr>
<tr>
<td>TL</td>
<td>7</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %tl</td>
<td></td>
</tr>
<tr>
<td>PIL</td>
<td>8</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %pil</td>
<td></td>
</tr>
<tr>
<td>CWP</td>
<td>9</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %cwp</td>
<td></td>
</tr>
<tr>
<td>CANSAVE</td>
<td>10</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %cansave</td>
<td></td>
</tr>
<tr>
<td>CANRESTORE</td>
<td>11</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %canrestore</td>
<td></td>
</tr>
<tr>
<td>CLEANWIN</td>
<td>12</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %cleanwin</td>
<td></td>
</tr>
<tr>
<td>OTHERWIN</td>
<td>13</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %otherwin</td>
<td></td>
</tr>
<tr>
<td>WSTATE</td>
<td>14</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %wstate</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GL</td>
<td>16</td>
<td>wrpr</td>
<td>reg</td>
<td>reg_or_imm, %gl</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>16–22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>24–31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

This instruction generates a source value of \( \text{"R[rs1] xor R[rs2]" if } i = 0, \text{ or } \text{"R[rs1] xor sign_ext(simm13)" if } i = 1. \) It stores that source value 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 \( \text{MAXPTL} \) 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).

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.

A WRPR to TL only stores a value to TL; it does not cause a trap, cause a return from a trap, or alter any machine state other than TL and state (such as PC, NPC, TICK, etc.) that is indirectly modified by every instruction.

**Programming Note** | A WRPR of TL can be used to read the values of TPC, TNPC, and TSTATE for any trap level; however, software must take care that traps do not occur while the TL register is modified.

\( \text{MAXPTL} \) is the maximum value that may be written by a WRPR to TL; an attempt to write a larger value results in \( \text{MAXPTL} \) being written to TL. For details, see TABLE 5-21 on page 72.
**WRPR**

\( \text{MAXPGL} \) is the maximum value that may be written by a WRPR to GL; an attempt to write a larger value results in \( \text{MAXPGL} \) being written to GL. For details, see TABLE 5-22 on page 74.

An attempt to use a WRPR instruction to write a value greater than \( N_{\text{REG\_WINDOWS}} - 1 \) to CANSAVE, CANRESTORE, OTHERWIN, or CLEANWIN causes an implementation-dependent value in the range \( 0 \leq N_{\text{REG\_WINDOWS}} - 1 \) to be written to the register (impl. dep. #126-V9-Ms10Cs40)

**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
- \( rd = 4 \)
- \( rd \) contains a reserved value (see above)
- \( 0 \leq rd \leq 3 \) (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.

**Exceptions**
- privileged_opcode
- illegal_instruction

**See Also**
- RDPR on page 298
- WRasr on page 357
The XMULX instructions are new and are not guaranteed to be implemented on all Oracle SPARC Architecture implementations. Therefore, they currently should only be used in platform-specific software (such as system-supplied runtime libraries or in software created by an implementation-aware runtime code generator).

### Description

The XMULX instruction performs a 64-bit by 64-bit bitwise (XOR) multiplication. An XOR multiply uses the XOR operation instead of the ADD operation when combining partial products.

XMULX computes $R[rs1]$ xor-multiply $R[rs2]$ and writes the less-significant 64 bits of the 128-bit result into $R[rd]$.

XMULXHI computes $R[rs1]$ xor-multiply $R[rs2]$ and writes the more-significant 64 bits of the 128-bit result into $R[rd]$.

Neither instruction modifies any condition codes.

### Exceptions

None

### See Also

......
7.144 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 Oracle SPARC Architecture 2011

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 365.
- **Underflow Behavior** on page 366.
- **Integer Overflow Definition** on page 367.
- **Floating-Point Nonstandard Mode** on page 367.
- **Arithmetic Result Tables** on page 368.

Exceptions are discussed in this chapter on the assumption that instructions are implemented in hardware. If an instruction is implemented in software, it may not trigger hardware exceptions but its behavior as observed by nonprivileged software (other than timing) must be the same as if it was implemented in hardware.

### 8.1 Traps Inhibiting Results

As described in *Floating-Point State Register (FSR)* on page 42 and elsewhere, when a floating-point trap occurs, the following conditions are true:

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

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

An Oracle SPARC Architecture virtual processor detects tininess before rounding occurs. (impl. dep. #55-V8-Cs10)

TABLE 8-1 summarizes what happens when an exact unrounded value $u$ satisfying

$$0 \leq |u| \leq \text{smallest normalized number}$$

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

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

<table>
<thead>
<tr>
<th>Underflow trap:</th>
<th>$ufm = 1$</th>
<th>$ufm = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u = r$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r$ is minimum normal</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>$r$ is subnormal</td>
<td>UF</td>
<td>None</td>
</tr>
<tr>
<td>$r$ is zero</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>$u \neq r$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r$ is minimum normal</td>
<td>UF</td>
<td>$NX$</td>
</tr>
<tr>
<td>$r$ is subnormal</td>
<td>UF</td>
<td>$NX$</td>
</tr>
<tr>
<td>$r$ is zero</td>
<td>UF</td>
<td>$NX$</td>
</tr>
</tbody>
</table>

$UF = \text{fp\_exception\_ieee\_754}$ trap with $cexc.ufc = 1$

$NX = \text{fp\_exception\_ieee\_754}$ trap with $cexc.nxc = 1$

$uf = cexc.ufc = 1$, $aexc.ufa = 1$, no $\text{fp\_exception\_ieee\_754}$ trap

$nx = cexc.nxc = 1$, $aexc.nxa = 1$, no $\text{fp\_exception\_ieee\_754}$ trap
8.2.1 Trapped Underflow Definition (ufm = 1)

Since tininess is detected before rounding, trapped underflow occurs when the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format.

| Note | The wrapped exponent results intended to be delivered on trapped underflows and overflows in IEEE 754 are irrelevant to the Oracle SPARC Architecture at the hardware, and privileged software levels. If they are created at all, it would be by user software in a nonprivileged-mode trap handler. |

8.2.2 Untrapped Underflow Definition (ufm = 0)

Untrapped underflow occurs when the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format and the correctly rounded result in the destination format is inexact.

8.3 Integer Overflow Definition

- F<sdaq>TOi — When a NaN, infinity, large positive argument ≥ 2^{31} or large negative argument ≤ −(2^{31} + 1) is converted to an integer, the invalid_current (nvc) bit of FSR.cexc is set to 1, and if the floating-point invalid trap is enabled (FSR.tem.nvm = 1), the fp_exception_IEEE_754 exception is raised. If the floating-point invalid trap is disabled (FSR.tem.nvm = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is 2^{31} − 1; if the sign bit of the operand is 1, the result is −2^{31}.

- F<sdaq>TOx — When a NaN, infinity, large positive argument ≥ 2^{63}, or large negative argument ≤ −(2^{63} + 1) is converted to an extended integer, the invalid_current (nvc) bit of FSR.cexc is set to 1, and if the floating-point invalid trap is enabled (FSR.tem.nvm = 1), the fp_exception_IEEE_754 exception is raised. If the floating-point invalid trap is disabled (FSR.tem.nvm = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is 2^{63} − 1; if the sign bit of the operand is 1, the result is −2^{63}.

8.4 Floating-Point Nonstandard Mode

If implemented, floating-point nonstandard mode is enabled by setting FSR.ns = 1 (see Nonstandard Floating-Point (ns) on page 43).

An Oracle SPARC Architecture 2011 processor may choose to implement nonstandard floating-point mode in order to obtain higher performance in certain circumstances. For example, when FSR.ns = 1 an implementation that processes fully normalized operands more efficiently than subnormal operands may convert a subnormal floating-point operand or result to zero.

The behavior of the following instructions is required to follow IEEE Std. 754 at all times, regardless of the value of FSR.ns: FADD, FDIV, FLCMP FMAF, FHADD, FHSUB, FNHADD, FMUL, FNADD, FNMUL, FSQRT, and FSUB.
Assuming that nonstandard floating-point mode is implemented, the effects of $\text{FSR}.\text{ns} = 1$ are as follows:

- **IMPL. DEP. #18-V8-Ms10(a):** When $\text{FSR}.\text{ns} = 1$ and a floating-point source operand is subnormal, an implementation may treat the subnormal operand as if it were a floating-point zero value of the same sign. The cases in which this replacement is performed are implementation dependent. However, if it occurs,
  1. it should not apply to \texttt{FLCMP}, \texttt{FABS}, \texttt{FMOV}, or \texttt{FNEG} instructions and
  2. \texttt{FADD}, \texttt{FSUB}, \texttt{FCMPE}, and \texttt{FCMP} should give identical treatment to subnormal source operands.

 Treating a subnormal source operand as zero may generate an IEEE 754 floating-point “inexact”, “division by zero”, or “invalid” condition (see Current Exception (\texttt{cexc}) on page 46). Whether the generated condition(s) trigger an \texttt{fp_exception_ieee_754} exception or not depends on the setting of $\text{FSR}.\text{tem}$.

- **IMPL. DEP. #18-V8-Ms10(b):** When a floating-point operation generates a subnormal result value, an Oracle SPARC Architecture 2011 implementation may either write the result as a subnormal value or replace the subnormal result by a floating-point zero value of the same sign and generate IEEE 754 floating-point “inexact” and “underflow” conditions. Whether these generated conditions trigger an \texttt{fp_exception_ieee_754} exception or not depends on the setting of $\text{FSR}.\text{tem}$.

- **IMPL. DEP. #18-V8-Ms10(c):** If an \texttt{FPop} generates an intermediate result value, the intermediate value is subnormal, and $\text{FSR}.\text{ns} = 1$, it is implementation dependent whether (1) the operation continues, using the subnormal value (possibly with some loss of accuracy), or (2) the virtual processor replaces the subnormal intermediate value with a floating-point zero value of the same sign, generates IEEE 754 floating-point “inexact” and “underflow” conditions, completes the instruction, and writes a final result (possibly with some loss of accuracy). Whether generated IEEE conditions trigger an \texttt{fp_exception_ieee_754} exception or not depends on the setting of $\text{FSR}.\text{tem}$.

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

## 8.5 Arithmetic Result Tables

This section contains detailed tables, showing the results produced by various floating-point operations, depending on their source operands.

Notes on source types:
- $\text{N}_n$: a number in $\text{F}[\text{rsn}]$, which may be normal or subnormal.
- $\text{QNaN}_n$ and $\text{SNaN}_n$ are Quiet and Signaling Not-a-Number values in $\text{F}[\text{rsn}]$, respectively.

Notes on result types:
- $\text{R}$: (rounded) result of operation, which may be normal, subnormal, zero, or infinity. May also cause OF, UF, NX, unfinished.
- $\text{dQNaN}$ is the generated default Quiet NaN (sign = 0, exponent = all 1s, fraction = all 1s). The sign of the default Quiet NaN is zero to distinguish it from storage initialized to all ones.
8.5.1 Floating-Point Add (FADD) and Add and Halve (FHADD)

TABLE 8-2 Floating-Point Add operation \( (F[rs1] + F[rs2]) \)
and Floating-Point Add and Halve operation \( ((F[rs1] + F[rs2]) \div 2) \)

<table>
<thead>
<tr>
<th>F[rs2]</th>
<th>(-\infty)</th>
<th>(-N2)</th>
<th>(-0)</th>
<th>(+0)</th>
<th>(+N2)</th>
<th>(+\infty)</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-N1)</td>
<td>(-R)</td>
<td>(-N1)</td>
<td>(\pm R^*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-0)</td>
<td>(-N2)</td>
<td>(-0)</td>
<td>(\pm 0^{**})</td>
<td>(+N2)</td>
<td></td>
<td></td>
<td>QNaN2</td>
<td></td>
</tr>
<tr>
<td>(+0)</td>
<td>(\pm 0^{**})</td>
<td></td>
<td>(+0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+N1)</td>
<td>(\pm R^*)</td>
<td></td>
<td>(+N1)</td>
<td>(+R)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\infty)</td>
<td>dQNaN, NV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN1</td>
<td>QNaN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN1</td>
<td>QNaN1, NV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* if \(N1 = -N2\), then **

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

For the FADD instructions, \(R\) may be any number; its generation may cause OF, UF, and/or NX.
For the FHADD instructions, \(R\) may be any number; its generation may cause UF and/or NX.
Floating-point add is not commutative when both operands are NaN.
8.5.2 Floating-Point Negative Add (FNADD) and Negative Add and Halve (FNHADD)

For the FNADD instructions, R may be any number; its generation may cause OF, UF, and/or NX.

For the FNHADD instructions, R may be any number; its generation may cause UF and/or NX.

Floating-point negative add is not commutative when both operands are NaN.

Note that rounding occurs after the negation. Thus, FNADD is not equivalent to FADD followed by FNEG when the rounding mode is towards ±∞.

8.5.3 Floating-Point Subtract (FSUB) and Subtract and Halve (FHSUB)

For the FNADD instructions, R may be any number; its generation may cause OF, UF, and/or NX.

For the FNHADD instructions, R may be any number; its generation may cause UF and/or NX.

Floating-point negative add is not commutative when both operands are NaN.

Note that rounding occurs after the negation. Thus, FNADD is not equivalent to FADD followed by FNEG when the rounding mode is towards ±∞.
For the FSUB instructions, R may be any number; its generation may cause OF, UF, and/or NX.
For the FHSUB instructions, R may be any number; its generation may cause UF and/or NX.
Note that \(-x \neq 0 - x\) when \(x\) is zero or NaN.

### 8.5.4 Floating-Point Multiply

**TABLE 8-5**  
Floating-Point Multiply operation \((F[rs1] \times F[rs2])\)

<table>
<thead>
<tr>
<th>(F[rs1])</th>
<th>(-\infty)</th>
<th>(-N2)</th>
<th>(-\infty)</th>
<th>(+0)</th>
<th>(+N2)</th>
<th>(+\infty)</th>
<th>(\text{QNaN2})</th>
<th>(\text{SNaN2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>+0</td>
<td>dQNaN, NV</td>
<td>(-R)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-N1)</td>
<td></td>
<td></td>
<td>+R</td>
<td></td>
<td>-R</td>
<td></td>
<td>QNaN2</td>
<td>QNaN2, NV</td>
</tr>
<tr>
<td>0</td>
<td>dQNaN, NV</td>
<td>+0</td>
<td>-0</td>
<td>-0</td>
<td>dQNaN, NV</td>
<td></td>
<td>QNaN2</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>NV</td>
<td>-0</td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+N1</td>
<td>(-R)</td>
<td></td>
<td>+R</td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
<td>QNaN2, NV</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(-\infty)</td>
<td>dQNaN, NV</td>
<td>(-R)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{QNaN1})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN1</td>
<td></td>
</tr>
<tr>
<td>(\text{SNaN1})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN1</td>
<td></td>
</tr>
</tbody>
</table>

R may be any number; its generation may cause OF, UF, and/or NX.
Floating-point multiply is not commutative when both operands are NaN.
FsMULd (FdMULq) never causes OF, UF, or NX.
A NaN input operand to FsMULd (FdMULq) must be widened to produce a double-precision (quad-
precision) NaN output, by filling the least-significant bits of the NaN result with zeros.

### 8.5.5 Floating-Point Negative Multiply (FNMUL)

**TABLE 8-6**  
Floating-Point Negative Multiply operation \((- (F[rs1] \times F[rs2]))\)

<table>
<thead>
<tr>
<th>(F[rs1])</th>
<th>(-\infty)</th>
<th>(-N2)</th>
<th>(-\infty)</th>
<th>(+0)</th>
<th>(+N2)</th>
<th>(+\infty)</th>
<th>(\text{QNaN2})</th>
<th>(\text{SNaN2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>-R</td>
<td></td>
<td>dQNaN, NV</td>
<td>+R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-N1)</td>
<td></td>
<td>-R</td>
<td>+R</td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
<td>QNaN2, NV</td>
</tr>
<tr>
<td>-0</td>
<td>dQNaN, NV</td>
<td>-0</td>
<td>+0</td>
<td>-0</td>
<td>dQNaN, NV</td>
<td></td>
<td>QNaN2</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>NV</td>
<td>+0</td>
<td>-0</td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
<td>QNaN2, NV</td>
</tr>
<tr>
<td>+N1</td>
<td>+R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN2</td>
<td>QNaN2, NV</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>dQNaN, NV</td>
<td>(-R)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{QNaN1})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN1</td>
<td></td>
</tr>
<tr>
<td>(\text{SNaN1})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN1</td>
<td></td>
</tr>
</tbody>
</table>

R may be any number; its generation may cause OF, UF, and/or NX.
Floating-point negative multiply is not commutative when both operands are NaNs.
Note that rounding occurs after the negation. Thus, FNMUL is not equivalent to FMUL followed by FNEG when the rounding mode is towards \( \pm \infty \).

FNSMULd never causes OF, UF, or NX.

A NaN input operand to FNSMULd must be widened to produce a double-precision NaN output, by filling the least-significant bits of the NaN result with zeros.

8.5.6 Floating-Point Multiply-Add (FMAf)

First refer to the Floating-Point Multiply table (TABLE 8-5 on page 371) to select a row in the table below.

**TABLE 8-7** Floating-Point Multiply-Add \( ((F_{rs1} \times F_{rs2}) + F_{rs3}) \)

<table>
<thead>
<tr>
<th>( F_{rs1} \times F_{rs2} )</th>
<th>( F_{rs3} )</th>
<th>(-\infty)</th>
<th>(-N)</th>
<th>(-0)</th>
<th>(+0)</th>
<th>(+N)</th>
<th>(+\infty)</th>
<th>( QNaN3 )</th>
<th>( SNaN3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td>(-N)</td>
<td>(-0)</td>
<td>(+0)</td>
<td>(+N)</td>
<td>(+\infty)</td>
<td>( dQNaN, NV )</td>
<td>( QNaN3 )</td>
</tr>
<tr>
<td>(-N)</td>
<td>(-N)</td>
<td>(-N)</td>
<td>(-0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>(+N3)</td>
<td>(+\infty)</td>
<td>( dQNaN, NV )</td>
<td>( QNaN3 )</td>
</tr>
<tr>
<td>(-0)</td>
<td>(-0)</td>
<td>(-N3)</td>
<td>(-0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>(+N3)</td>
<td>(+\infty)</td>
<td>( dQNaN, NV )</td>
<td>( QNaN3 )</td>
</tr>
<tr>
<td>(+0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>(+\infty)</td>
<td>( dQNaN, NV )</td>
<td>( QNaN3 )</td>
</tr>
<tr>
<td>(+N)</td>
<td>(+N)</td>
<td>(+N)</td>
<td>(+N)</td>
<td>(+N)</td>
<td>(+N)</td>
<td>(+N)</td>
<td>(+\infty)</td>
<td>( dQNaN, NV )</td>
<td>( QNaN3 )</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>(+\infty)</td>
<td>( QNaN3 )</td>
<td>( QNaN3 )</td>
</tr>
</tbody>
</table>

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

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

*** if \( \text{FSR}.nvm = 1 \), \( \text{FSR}.nvc \leftarrow 1 \), the trap occurs, and \( \text{FSR}.aexc \) is left unchanged; otherwise, \( \text{FSR}.nvm = 0 \) and \( \text{FSR}.nvc \leftarrow 1 \).

In the above table, \( R \) may be any number; its generation may cause OF, UF, and/or NX.

The multiply operation in fused floating-point multiply-add (FMADD) instructions cannot cause inexact, underflow, or overflow exceptions.

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

8.5.7 Floating-Point Negative Multiply-Add (FNMAADD)

First refer to the Floating-Point Multiply table (TABLE 8-5 on page 371) to select a row in the table below.
R may be any number; its generation may cause OF, UF, and/or NX.

The multiply operation in fused floating-point negative multiply-add (FNMADD) instructions cannot cause inexact, underflow, or overflow exceptions.

Note that rounding occurs after the negation. Thus, when the rounding mode is towards ±∞, FNMADD is not equivalent to FMADD followed by FNEG.

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

### 8.5.8 Floating-Point Multiply-Subtract (FMSUB)

First refer to the Floating-Point Multiply table (TABLE 8-5 on page 371) to select a row in the table below.
R may be any number; its generation may cause OF, UF, and/or NX.

The multiply operation in fused floating-point multiply-subtract (FMSUB) instructions cannot cause inexact, underflow, or overflow exceptions.

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

### 8.5.9 Floating-Point Negative Multiply-Subtract (FNMSUB)

First refer to the Floating-Point Multiply table (TABLE 8-5) to select a row in the table below.
R may be any number; its generation may cause OF, UF, and/or NX.

The multiply operation in fused floating-point negative multiply-subtract (FNMSUB) instructions cannot cause inexact, underflow, or overflow exceptions.

Note that rounding occurs after the negation. Thus, FNMSUB is not equivalent to FMSUB followed by FNEG when the rounding mode is towards ±∞.

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

---

**TABLE 8-10** Floating-Point Negative Multiply-Subtract (−(F[rs1] × F[rs2]) + F[rs3])

<table>
<thead>
<tr>
<th>F[rs3]</th>
<th>−∞</th>
<th>−N3</th>
<th>−0</th>
<th>+0</th>
<th>+N3</th>
<th>+∞</th>
<th>QNaN3</th>
<th>SNaN3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F[rs1] × F[rs2]</td>
<td>dQNaN, NV</td>
<td>±R*</td>
<td>+N</td>
<td>+R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−∞</td>
<td>+N3</td>
<td>±0*</td>
<td>+0</td>
<td>+N3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−0</td>
<td>-N3</td>
<td>-0</td>
<td>±0*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>-R</td>
<td>-N</td>
<td>±R*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>dQNaN, NV</td>
<td>QNaN3</td>
<td>QNaN3, NV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* if N = N3, then **

** result is +0 unless rounding mode is round to −∞, in which case the result is −0

*** if FSR.nvm = 1, FSR.nvc ← 1, the trap occurs, and FSR.aexc is left unchanged; otherwise, FSR.nvm = 0 so FSR.nva ← 1 and for FNMSUB FSR.nvc ← 1.
### 8.5.10 Floating-Point Divide (FDIV)

**TABLE 8-11**  Floating-Point Divide operation \((F[rs1] \div F[rs2])\)

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

R may be any number; its generation may cause OF, UF, and/or NX.

### 8.5.11 Floating-Point Square Root (FSQRT)

**TABLE 8-12**  Floating-Point Square Root operation \((\sqrt{F[rs2]}\))

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

R may be any number; its generation may cause NX.

Square root cannot cause DZ, OF, or UF.
8.5.12 Floating-Point Compare (FCMP, FCMPE)

TABLE 8-13 Floating-Point Compare (FCMP, FCMPE) operation (F[rs1] ? F[rs2])

<table>
<thead>
<tr>
<th>F[rs2]</th>
<th>−∞</th>
<th>−N2</th>
<th>−0</th>
<th>+0</th>
<th>+N2</th>
<th>+∞</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−N1</td>
<td>1</td>
<td>0, 1, 2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−0</td>
<td>0</td>
<td>0</td>
<td>0, 1, 2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+N1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>3</td>
<td>3, NV*</td>
<td>3, NV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F[rs1]

* NV for FCMPE, but not for FCMP.

TABLE 8-14 FSR/fcc Encoding for Result of FCMP, FCMPE

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

NaN is considered to be unequal to anything else, even the identical NaN bit pattern.

FCMP/FCMPE cannot cause DZ, OF, UF, NX.

8.5.13 Floating-Point Lexicographic Compare (FLCMP)

TABLE 8-15 Lexicographic Compare (FLCMP) operation (F[rs1] ?_L F[rs2])

<table>
<thead>
<tr>
<th>F[rs2]</th>
<th>−∞</th>
<th>−N2</th>
<th>−0</th>
<th>+0</th>
<th>+N2</th>
<th>+∞</th>
<th>QNaN2</th>
<th>SNaN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−N1</td>
<td>1</td>
<td>0, 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−0</td>
<td>0</td>
<td>0</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+N1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F[rs1]
FLCMP does not cause any floating-point exceptions (NV, DZ, OF, UF, NX, or \textit{fp\_exception\_other} (with FSR\_ftt = unfinished\_FPop)).

FSR\_cexc and FSR\_aexc are unchanged by FLCMP.

### 8.5.14 Floating-Point to Floating-Point Conversions
\((F{s|d|q}TO{s|d|q})\)

#### TABLE 8-16  FSR\_fxc Encoding for Result of FLCMP

<table>
<thead>
<tr>
<th>fxc result</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\geq)</td>
</tr>
<tr>
<td>1</td>
<td>(&lt;)</td>
</tr>
<tr>
<td>2</td>
<td>(\text{F[rs1]}) is NaN and (\text{F[rs2]}) is not NaN</td>
</tr>
<tr>
<td>3</td>
<td>(\text{F[rs2]}) is NaN</td>
</tr>
</tbody>
</table>

FLCMP does not cause any floating-point exceptions (NV, DZ, OF, UF, NX, or \textit{fp\_exception\_other} (with FSR\_ftt = unfinished\_FPop)).

FSR\_cexc and FSR\_aexc are unchanged by FLCMP.

---

**Note** Encoding of the condition code result for FLCMP is different from the encoding for FCMP/FCMPE. Furthermore, for this operation, the sign of zero is significant: \(-0 < +0\).

**TABLE 8-16  FSR\_fxc Encoding for Result of FLCMP**

<table>
<thead>
<tr>
<th>fxc result</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\geq)</td>
</tr>
<tr>
<td>1</td>
<td>(&lt;)</td>
</tr>
<tr>
<td>2</td>
<td>(\text{F[rs1]}) is NaN and (\text{F[rs2]}) is not NaN</td>
</tr>
<tr>
<td>3</td>
<td>(\text{F[rs2]}) is NaN</td>
</tr>
</tbody>
</table>

---

### TABLE 8-17  Floating-Point to Float-Point Conversions (convert(\(F[rs2]\)))

\[
\begin{array}{cccccccc}
\text{F[rs2]} & -\text{SNaN2} & -\text{QNaN2} & -\infty & -\text{N2} & -0 & +0 & +\text{N2} & +\infty & +\text{QNaN2} & +\text{SNaN2} \\
-\text{QSNaN2}, & -\text{QNaN2} & -\infty & -\text{R} & -0 & +0 & +\text{R} & +\infty & +\text{QNaN2} & +\text{QSNaN2}, & +\text{SNaN2}, \\
\text{NV} & & & & & & & & & & NV \\
\end{array}
\]

For Fs\_TOd:
- the least-significant fraction bits of a normal number are filled with zero to fit in double-precision format
- the least-significant bits of a NaN result operand are filled with zero to fit in double-precision format

For Fs\_TOq and Fd\_TOq:
- the least-significant fraction bits of a normal number are filled with zero to fit in quad-precision format
- the least-significant bits of a NaN result operand are filled with zero to fit in quad-precision format

For Fq\_TOs and Fd\_TOs:
- the fraction is rounded according to the current rounding mode
- the lower-order bits of a NaN source are discarded to fit in single-precision format; this discarding is not considered a rounding operation, and will not cause an NX exception

For Fq\_TOd:
- the fraction is rounded according to the current rounding mode
- the least-significant bits of a NaN source are discarded to fit in double-precision format; this discarding is not considered a rounding operation, and will not cause an NX exception
8.5.15 Floating-Point to Integer Conversions (F<s | d | q>TO<i | x>)

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

R may be any integer, and may cause NV, NX.

Float-to-Integer conversions are always treated as round-toward-zero (truncated).

These operations are invalid (due to integer overflow) under the conditions described in Integer Overflow Definition on page 367.

| NV | • SNaN operand  
• QNaN operand  
• ±∞ operand  
• integer overflow |
| NX | • non-integer source (truncation occurred) |

8.5.16 Integer to Floating-Point Conversions (F<i | x>TO<s | d | q>)

<table>
<thead>
<tr>
<th>F&lt;rs2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-int 0 +int</td>
</tr>
<tr>
<td>-R +0 +R</td>
</tr>
</tbody>
</table>

R may be any number; its generation may cause NX.
| NX | • FxTOd, FxTOs, FiTOs (possible loss of precision)  
• not applicable to FiTOd, FxTOq, or FiTOq (FSR.cexc will always be cleared) |
The Oracle SPARC 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.

This chapter contains a great deal of theoretical information so that the discussion of the Oracle SPARC Architecture TSO memory model has sufficient background.

This chapter describes memory models in these sections:

- Memory Location Identification on page 381.
- Memory Accesses and Cacheability on page 381.
- Memory Addressing and Alternate Address Spaces on page 383.
- SPARC V9 Memory Model on page 386.
- The Oracle SPARC Architecture Memory Model — TSO on page 389.
- Nonfaulting Load on page 395.
- Store Coalescing on page 396.

### 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 Oracle SPARC Architecture: cacheable and noncacheable accesses. The manner in which addresses are differentiated is implementation dependent. In some implementations, it is indicated in the page translation entry (TTE.cp).

Although SPARC V9 does not specify memory ordering between cacheable and noncacheable accesses, the Oracle SPARC Architecture maintains TSO ordering between memory references regardless of their cacheability.

9.2.1.1 Cacheable Accesses

Accesses within the coherence domain are called cacheable accesses. They have these properties:

- Data reside in real memory locations.
- Accesses observe supported cache coherency protocol(s).
- The cache line size is \(2^n\) bytes (where \(n \geq 4\)), and can be different for each cache.

9.2.1.2 Noncacheable Accesses

Noncacheable accesses are outside of the coherence domain. They have the following properties:

- Data might not reside in real memory locations. Accesses may result in programmer-visible side effects. An example is memory-mapped I/O control registers.
- Accesses do not observe supported cache coherency protocol(s).
- The smallest unit in each transaction is a single byte.

The Oracle SPARC Architecture MMU optionally includes an attribute bit in each page translation, TTE.e, which when set signifies that this page has side effects.

Noncacheable accesses without side effects (TTE.e = 0) are processor-consistent and obey TSO memory ordering. In particular, processor consistency ensures that a noncacheable load that references the same location as a previous noncacheable store will load the data from the previous store.

Noncacheable accesses with side effects (TTE.e = 1) are processor consistent and are strongly ordered. These accesses are described in more detail in the following section.

9.2.1.3 Noncacheable Accesses with Side-Effect

Loads, stores, and load-stores to I/O locations might not behave with memory semantics. Loads and stores could have side effects; for example, a read access could clear a register or pop an entry off a FIFO. A write access could set a register address port so that the next access to that address will read or write a particular internal register. Such devices are considered order sensitive. Also, such devices may only allow accesses of a fixed size, so store merging of adjacent stores or stores within a 16-byte region would cause an error (see Store Coalescing on page 396).
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 393.

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 Oracle SPARC Architecture virtual processors, noncacheable and side-effect accesses do not observe supported cache coherency protocols (impl. dep. #120).

Non-faulting loads (using ASI_PRIMARY_NO_FAULT[Little] or ASI_SECONDARY_NO_FAULT[Little]) with the TTE.e bit = 1 cause a DAE_side_effect_page trap.

Prefetches to noncacheable addresses result in nops.

The processor does speculative instruction memory accesses and follows branches that it predicts are taken. Instruction addresses mapped by the MMU can be accessed even though they are not actually executed by the program. Normally, locations with side effects or that generate timeouts or bus errors are not mapped as instruction addresses by the MMU, so these speculative accesses will not cause problems.

**IMPL. DEP. #118-V9:** The manner in which I/O locations are identified is implementation dependent.

**IMPL. DEP. #120-V9:** The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent.

**V9 Compatibility Note:** Operations to I/O locations are not guaranteed to be sequentially consistent among themselves, as they are in SPARC V8.

Systems supporting SPARC V8 applications that use memory-mapped I/O locations must ensure that SPARC V8 sequential consistency of I/O locations can be maintained when those locations are referenced by a SPARC V8 application. The MMU either must enforce such consistency or cooperate with system software or the virtual processor to provide it.

**IMPL. DEP. #121-V9:** An implementation may choose to identify certain addresses and use an implementation-dependent memory model for references to them.

### 9.3 Memory Addressing and Alternate Address Spaces

An address in SPARC V9 is a tuple consisting of an 8-bit address space identifier (ASI) and a 64-bit byte-address offset within the specified address space. Memory is byte-addressed, with halfword accesses aligned on 2-byte boundaries, word accesses (which include instruction fetches) aligned on 4-byte boundaries, extended-word and doubleword accesses aligned on 8-byte boundaries, and quadword quantities aligned on 16-byte boundaries. With the possible exception of the cases described in Memory Alignment Restrictions on page 77, 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 Oracle SPARC Architecture supports the following types of memory addressing:

Virtual Addresses (VA). Virtual addresses are addresses produced by a virtual processor that maps all systemwide, program-visible memory. Virtual addresses can be presented in nonprivileged mode and privileged mode.

Real addresses (RA). A real address is provided to privileged software to describe the underlying physical memory allocated to it. Translation storage buffers (TSBs) maintained by privileged software are used to translate privileged or nonprivileged mode virtual addresses into real addresses. MMU bypass addresses in privileged mode are also real addresses.

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

9.3.2 Memory Address Spaces

The Oracle SPARC 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:

real_address = context_ID :: virtual_address

9.3.3 Address Space Identifiers

The virtual processor provides an address space identifier with every address. This ASI may serve several purposes:

- To identify which of several distinguished address spaces the 64-bit address offset is addressing.

---

1 Two exceptions to this are the special ASI_TWIN_DW_NUCLEUS[_L] and ASI_TWINX_REAL[_L] which provide hardware support for an atomic quad load to be used for TTE loads from TSBs.
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 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 399.

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 Oracle SPARC Architecture processor.

The memory models specify the possible order relationships between memory-reference instructions issued by a virtual processor and the order and visibility of those instructions as seen by other virtual processors. The memory model is intimately intertwined with the program execution model for instructions.

#### 9.4.1 SPARC V9 Program Execution Model

The SPARC V9 strand model of a virtual processor consists of three units: an Issue Unit, a Reorder Unit, and an Execute Unit, as shown in FIGURE 9-1.

The Issue Unit reads instructions over the instruction path from memory and issues them in program order to the Reorder Unit. Program order is precisely the order determined by the control flow of the program and the instruction semantics, under the assumption that each instruction is performed independently and sequentially.

Issued instructions are collected and potentially reordered in the Reorder Unit, and then dispatched to the Execute Unit. Instruction reordering allows an implementation to perform some operations in parallel and to better allocate resources. The reordering of instructions is constrained to ensure that the results of program execution are the same as they would be if the instructions were performed in program order. This property is called processor self-consistency.
Processor self-consistency requires that the result of execution, in the absence of any shared memory interaction with another virtual processor, be identical to the result that would be observed if the instructions were performed in program order. In the model in FIGURE 9-1, instructions are issued in program order and placed in the reorder buffer. The virtual processor is allowed to reorder instructions, provided it does not violate any of the data-flow constraints for registers or for memory.

The data-flow order constraints for register reference instructions are these:

1. An instruction that reads from or writes to a register cannot be performed until all earlier instructions that write to that register have been performed (read-after-write hazard; write-after-write hazard).

2. An instruction cannot be performed that writes to a register until all earlier instructions that read that register have been performed (write-after-read hazard).

For memory-reference instructions, the data-flow order constraints 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 392 and The Oracle SPARC Architecture Memory Model — TSO on page 389 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 in the SPARC V9 specification, Formal Specificatin of the Memory Models, for more information.
9.4.2 Virtual Processor/Memory Interface Model

Each Oracle SPARC 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 Memory Paths: Multiprocessor System](image)

Data caches are maintained by hardware so their contents always appear to be consistent (coherent). Instruction caches are not required to be kept consistent with data caches and therefore require explicit program (software) action to ensure consistency when a program modifies an executing instruction stream. See Synchronizing Instruction and Data Memory on page 394 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 may complete in any order. Once initiated, all memory operations are performed atomically: loads from one location all see the same value, and the result of stores is visible to all potential requestors at the same instant.

The order of memory operations observed at a single location is a total order that preserves the partial orderings of each virtual processor’s transactions to this address. There may be many legal total orders for a given program’s execution.

---

1. The model described here is only a model; implementations of Oracle SPARC Architecture systems are unconstrained as long as their observable behaviors match those of the model.
The Oracle SPARC Architecture Memory Model — TSO

The Oracle SPARC Architecture is a model that specifies the behavior observable by software on Oracle SPARC 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 Oracle SPARC Architecture virtual processors implement TSO ordering. The PSO and RMO models from SPARC V9 are not described in this Oracle SPARC Architecture specification. Oracle SPARC Architecture 2011 processors do not implement the PSO memory model directly, but all software written to run under PSO will execute correctly on an Oracle SPARC Architecture 2011 processor (using the TSO model).

Whether memory models represented by \texttt{PSTATE.mm} = 10_2 or 11_2 are supported in an Oracle SPARC Architecture processor is implementation dependent (impl. dep. #113-V9-Ms10). If the 10_2 model is supported, then when \texttt{PSTATE.mm} = 10_2 the implementation must correctly execute software that adheres to the RMO model described in \textit{The SPARC Architecture Manual-Version 9}. If the 11_2 model is supported, its definition is implementation dependent and will be described in implementation-specific documentation.

Programs written for Relaxed Memory Order will work in both Partial Store Order and Total Store Order. Programs written for Partial Store Order will work in Total Store Order. Programs written for a weak model, such as RMO, may execute more quickly when run on hardware directly supporting that model, since the model exposes more scheduling opportunities, but use of that model may also require extra instructions to ensure synchronization. Multiprocessor programs written for a stronger model will behave unpredictably if run in a weaker model.

Machines that implement sequential consistency (also called "strong ordering" or "strong consistency") automatically support programs written for TSO. Sequential consistency is not a SPARC V9 memory model. In sequential consistency, the loads, stores, and atomic load-stores of all virtual processors are performed by memory in a serial order that conforms to the order in which these instructions are issued by individual virtual processors. A machine that implements sequential consistency may deliver lower performance than an equivalent machine that implements TSO order. Although particular SPARC V9 implementations may support sequential consistency, portable software must not rely on the sequential consistency memory model.

Memory Model Selection

The active memory model is specified by the 2-bit value in \texttt{PSTATE.mm}. The value 00_2 represents the TSO memory model; increasing values of \texttt{PSTATE.mm} indicate increasingly weaker (less strongly ordered) memory models.

Writing a new value into \texttt{PSTATE.mm} causes subsequent memory reference instructions to be performed with the order constraints of the specified memory model.
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 Oracle SPARC 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 Oracle SPARC 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 387), 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, #StoreLoad, and #StoreStore.

In addition to the above TSO rules, the following rules apply to Oracle SPARC 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 Oracle SPARC Architecture Memory Model — TSO on page 389) is desired.
- Accesses that have side effects are all strongly ordered with respect to each other.
- A MEMBAR #LookasideD 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 Oracle SPARC Architecture virtual processor running in TSO mode, to ensure that the operations appear to complete in program 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 Oracle SPARC 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>#</td>
<td>#</td>
<td>#</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>store</td>
<td>M²</td>
<td>#</td>
<td>#</td>
<td>M</td>
<td>S</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>S</td>
<td>M</td>
<td>S</td>
</tr>
<tr>
<td>atomic</td>
<td>#</td>
<td>#</td>
<td>M</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>M</td>
<td>M</td>
<td>S</td>
<td>S</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>S</td>
<td>M</td>
<td>M</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>load_nc_e</td>
<td>#</td>
<td>#</td>
<td>S</td>
<td>#¹</td>
<td>#¹</td>
<td>#¹</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>store_nc_e</td>
<td>S</td>
<td>#</td>
<td>S</td>
<td>#¹</td>
<td>#¹</td>
<td>M²</td>
<td>#¹</td>
<td>M</td>
<td>M</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>load_nc_ne</td>
<td>#</td>
<td>#</td>
<td>S</td>
<td>#¹</td>
<td>#¹</td>
<td>#¹</td>
<td>#¹</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>store_nc_ne</td>
<td>S</td>
<td>#</td>
<td>S</td>
<td>M²</td>
<td>#¹</td>
<td>M²</td>
<td>#¹</td>
<td>M</td>
<td>M</td>
<td>S</td>
<td>S</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.

Note that transitivity applies; if operation X is always ordered before operation Y (#" in TABLE 9-2) and operation Y is always ordered before operation Z (again, "#" in the table), then the sequence of operations X ... Y ... Z may safely be executed with no intervening MEMBAR, even if the table shows that a MEMBAR is normally needed between X and Z. For example, a MEMBAR is normally needed between a store and a load ("M" in TABLE 9-2); however, the sequence "store ... atomic ... load" may be executed safely with no intervening MEMBAR because stores are always ordered before atomics and atomics are always ordered before loads.
9.5.4 Hardware Primitives for Mutual Exclusion

In addition to providing memory-ordering primitives that allow programmers to construct mutual-exclusion mechanisms in software, the Oracle SPARC Architecture provides some hardware primitives for mutual exclusion:

- Compare and Swap (CASA and CASXA)
- Load Store Unsigned Byte (LDSTUB and LDSTUBA)
- Swap (SWAP and SWAPA)

Each of these instructions has the semantics of both a load and a store in all three memory models. They are all atomic, in the sense that no other store to the same location can be performed between the load and store elements of the instruction. All of the hardware mutual-exclusion operations conform to the TSO memory model and may require barrier instructions to ensure proper data visibility.

Atomic load-store instructions can be used only in the cacheable domains (not in noncacheable I/O addresses). An attempt to use an atomic load-store instruction to access a noncacheable page results in a DAE_nc_page exception.

The atomic load-store alternate instructions can use a limited set of the ASIs. See the specific instruction descriptions for a list of the valid ASIs. An attempt to execute an atomic load-store alternate instruction with an invalid ASI results in a DAE_invalid_asi exception.

9.5.4.1 Compare-and-Swap (CASA, CASXA)

Compare-and-swap is an atomic operation that compares a value in a virtual processor register to a value in memory and, if and only if they are equal, swaps the value in memory with the value in a second virtual processor register. Both 32-bit (CASA) and 64-bit (CASXA) operations are provided. The compare-and-swap operation is atomic in the sense that once it begins, no other virtual processor can access the memory location specified until the compare has completed and the swap (if any) has also completed and is potentially visible to all other virtual processors in the system.

Compare-and-swap is substantially more powerful than the other hardware synchronization primitives. It has an infinite consensus number; that is, it can resolve, in a wait-free fashion, an infinite number of contending processes. Because of this property, compare-and-swap can be used to construct wait-free algorithms that do not require the use of locks.

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 FF₁₆ 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 Oracle SPARC Architecture provides some level of programmer control over memory ordering and synchronization through the MEMBAR and FLUSH instructions.

MEMBAR serves two distinct functions in SPARC V9. One variant of the MEMBAR, the ordering MEMBAR, provides a way for the programmer to control the order of loads and stores issued by a virtual processor. The other variant of MEMBAR, the sequencing MEMBAR, enables the programmer
to explicitly control order and completion for memory operations. Sequencing MEMBARs are needed only when a program requires that the effect of an operation becomes globally visible rather than simply being scheduled. Because both forms are bit-encoded into the instruction, a single MEMBAR can function both as an ordering MEMBAR and as a sequencing MEMBAR.

The SPARC V9 instruction set architecture does not guarantee consistency between instruction and data spaces. A problem arises when instruction space is dynamically modified by a program writing to memory locations containing instructions (Self-Modifying Code). Examples are Lisp, debuggers, and dynamic linking. The FLUSH instruction synchronizes instruction and data memory after instruction space has been modified.

### 9.5.5.1 Ordering MEMBAR Instructions

Ordering MEMBAR instructions induce an ordering in the instruction stream of a single virtual processor. Sets of loads and stores that appear before the MEMBAR in program order are ordered with respect to sets of loads and stores that follow the MEMBAR in program order. Atomic operations (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 (#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 in the SPARC V9 specification, Formal Specification of the Memory Models, for a formal description of the <m relationship.

**TABLE 9-3** Ordering Relationships Selected by Mask

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

**Implementation Note** | An Oracle SPARC Architecture 2011 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 (deprecated)** — 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.

- **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.

- **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 (deprecated)</td>
<td>#LookasideD</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 Oracle SPARC Architecture 2011 implementations, MEMBAR #LookasideD and MEMBAR #MemIssue are typically implemented as a MEMBAR #Sync.

For more details, see the MEMBAR instruction on page 256 of Chapter 7, Instructions.

9.5.5.3 **Synchronizing Instruction and Data Memory**

The SPARC V9 memory models do not require that instruction and data memory images be consistent at all times. The instruction and data memory images may become inconsistent if a program writes into the instruction stream. As a result, whenever instructions are modified by a program in a context where the data (that is, the instructions) in the memory and the data cache hierarchy may be inconsistent with instructions in the instruction cache hierarchy, some special programmatic (software) action must be taken.

The FLUSH instruction will ensure consistency between the in-flight instruction stream and the data references in the virtual processor executing FLUSH. The programmer must ensure that the modification sequence is robust under multiple updates and concurrent execution. Since, in general, loads and stores may be performed out of order, appropriate MEMBAR and FLUSH instructions must be interspersed as needed to control the order in which the instruction data are modified.

The FLUSH instruction ensures that subsequent instruction fetches from the doubleword target of the FLUSH by the virtual processor executing the FLUSH appear to execute after any loads, stores, and atomic load-stores issued by the virtual processor to that address prior to the FLUSH. FLUSH acts as a barrier for instruction fetches in the virtual processor on which it executes and has the properties of a store with respect to MEMBAR operations.
The latency between the execution of FLUSH on one virtual processor and the point at which the modified instructions have replaced outdated instructions in a multiprocessor is implementation dependent.

Because FLUSH is designed to act on a doubleword and because, on some implementations, FLUSH may trap to system software, it is recommended that system software provide a user-callable service routine for flushing arbitrarily sized regions of memory. On some implementations, this routine would issue a series of FLUSH instructions; on others, it might issue a single trap to system software that would then flush the entire region.

On an Oracle SPARC 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 Oracle SPARC Architecture implementation may ignore the address in the operands of a FLUSH instruction.

Oracle SPARC 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 166 of Chapter 7, Instructions.

9.6 Nonfaulting Load

A nonfaulting load behaves like a normal load, with the following exceptions:
- A nonfaulting load from a location with side effects (TTE.e = 1) causes a DAE_side_effect_page exception.
- A nonfaulting load from a page marked for nonfault access only (TTE.nfo = 1) is allowed; other types of accesses to such a page cause a DAE_nfo_page exception.
- These loads are issued with ASI_PRIMARY_NO_FAULT[_LITTLE] or ASI_SECONDARY_NO_FAULT[_LITTLE]. A store with a NO_FAULTASI causes a DAE_invalid_asi exception.

Typically, optimizers use nonfaulting loads to move loads across conditional control structures that guard their use. This technique potentially increases the distance between a load of data and the first use of that data, in order to hide latency. The technique allows more flexibility in instruction scheduling and improves performance in certain algorithms by removing address checking from the critical code path.

For example, when following a linked list, nonfaulting loads allow the null pointer to be accessed safely in a speculative, read-ahead fashion; the page at virtual address 016 can safely be accessed with no penalty. The TTE.nfo bit marks pages that are mapped for safe access by nonfaulting loads but that can still cause a trap by other, normal accesses.
Thus, programmers can trap on “wild” pointer references—many programmers count on an exception being generated when accessing address 0x0 to debug software—while benefiting from the acceleration of nonfaulting access in debugged library routines.

9.7 Store Coalescing

Cacheable stores may be coalesced with adjacent cacheable stores within an 8 byte boundary offset in the store buffer to improve store bandwidth. Similarly non-side-effect-noncacheable stores may be coalesced with adjacent non-side-effect noncacheable stores within an 8-byte boundary offset in the store buffer.

In order to maintain strong ordering for I/O accesses, stores with side-effect attribute (e bit set) will not be combined with any other stores.

Stores that are separated by an intervening MEMBAR #SYNC will not be coalesced.
Address Space Identifiers (ASIs)

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

- Address Space Identifiers and Address Spaces on page 397.
- ASI Values on page 397.
- ASI Assignments on page 398.
- Special Memory Access ASIs on page 406.

10.1 Address Space Identifiers and Address Spaces

An Oracle SPARC Architecture processor provides an address space identifier (ASI) with every address sent to memory. The ASI does the following:

- Distinguishes between different address spaces
- Provides an attribute that is unique to an address space
- Maps internal control and diagnostics registers within a virtual processor

The memory management unit uses a 64-bit virtual address and an 8-bit ASI to generate a memory, I/O, or internal register address.

10.2 ASI Values

The range of address space identifiers (ASIs) is $00_{16}$–$FF_{16}$. That range is divided into restricted and unrestricted portions. ASIs in the range $80_{16}$–$FF_{16}$ are unrestricted; they may be accessed by software running in any privilege mode.

ASIs in the range $00_{16}$–$7F_{16}$ are restricted; they may only be accessed by software running in a mode with sufficient privilege for the particular ASI. ASIs in the range $00_{16}$–$2F_{16}$ may only be accessed by software running in privileged or hyperprivileged mode and ASIs in the range $30_{16}$–$7F_{16}$ may only be accessed by software running in hyperprivileged mode.

In SPARC V9, the range of ASIs was evenly divided into restricted ($00_{16}$–$7F_{16}$) and unrestricted ($80_{16}$–$FF_{16}$) halves.

An attempt by nonprivileged software to access a restricted (privileged or hyperprivileged) ASI ($00_{16}$–$7F_{16}$) causes a privileged_action trap.

An attempt by privileged software to access a hyperprivileged ASI ($30_{16}$–$7F_{16}$) also causes a privileged_action trap.
An ASI can be categorized based on how it affects the MMU’s treatment of the accompanying address, into one of three categories:

- A **Translating ASI** (the most common type) causes the accompanying address to be treated as a virtual address (which is translated by the MMU).
- A **Non-translating ASI** is not translated by the MMU; instead the address is passed through unchanged. Nontranslating ASIs are typically used for accessing internal registers.
- A **Real ASI** causes the accompanying address to be treated as a real address. An access using a Real ASI can cause exception(s) only visible in hyperprivileged mode. Real ASIs are typically used by privileged software for directly accessing memory using real (as opposed to virtual) addresses.

Implementation-dependent ASIs may or may not be translated by the MMU. See implementation-specific documentation for detailed information about implementation-dependent ASIs.

### 10.3 ASI Assignments

Every load or store address in an Oracle SPARC 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 Oracle SPARC Architecture implementations and some are only present in some implementations.

An ASI marked with a closed bullet (●) is required to be implemented on all Oracle SPARC Architecture 2011 processors.

An ASI marked with an open bullet (❍) is defined by the Oracle SPARC Architecture 2011 but is not necessarily implemented in all Oracle SPARC Architecture 2011 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 the supplied virtual address is decoded by the virtual processor.

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

ASIs marked “Reserved” are set aside for use in future revisions to the architecture and are not to be used by implementations. ASIs marked “implementation dependent” may be used for implementation-specific purposes.
Attempting to access an address space described as “Implementation dependent” in TABLE 10-1 produces implementation-dependent results.

<table>
<thead>
<tr>
<th>ASI Value</th>
<th>req’d (●)</th>
<th>ASI Name (and Abbreviation)</th>
<th>Access Type(s)</th>
<th>Virtual Address (VA)</th>
<th>V/ non-T/R</th>
<th>Shared/per strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0016–0116</td>
<td>☉</td>
<td>—</td>
<td>_2,12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>0216–0316</td>
<td>☉</td>
<td>—</td>
<td>_2,12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>0416</td>
<td>●</td>
<td>ASI_NUCLEUS (ASI_N)</td>
<td>RW²,⁴ (decoded)</td>
<td>V</td>
<td>—</td>
<td>—</td>
<td>Implicit address space, nucleus context, TL &gt; 0</td>
</tr>
<tr>
<td>0516–0616</td>
<td>☉</td>
<td>—</td>
<td>_2,12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>0716–0816</td>
<td>☉</td>
<td>—</td>
<td>_2,12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>0916</td>
<td>●</td>
<td>ASI_NUCLEUS_LITTLE (ASI_NL)</td>
<td>RW²,⁴ (decoded)</td>
<td>V</td>
<td>—</td>
<td>—</td>
<td>Implicit address space, nucleus context, TL &gt; 0, little-endian</td>
</tr>
<tr>
<td>0A16–0B16</td>
<td>☉</td>
<td>—</td>
<td>_2,12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>0C16</td>
<td>●</td>
<td>ASI AS_IF_USER_PRIMARY (ASI_AIUP)</td>
<td>RW²,⁴ (decoded)</td>
<td>V</td>
<td>—</td>
<td>—</td>
<td>Primary address space, as if user (nonprivileged)</td>
</tr>
<tr>
<td>0D16–0F16</td>
<td>☉</td>
<td>—</td>
<td>_2,12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Secondary address space, as if user (nonprivileged)</td>
</tr>
<tr>
<td>1016–1116</td>
<td>●</td>
<td>ASI AS_IF_USER_SECONDARY (ASI_AIUS)</td>
<td>RW²,⁴ (decoded)</td>
<td>V</td>
<td>—</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>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>1416</td>
<td>●</td>
<td>ASI_REAL</td>
<td>RW²,⁴ (decoded)</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>Real address</td>
</tr>
<tr>
<td>1516</td>
<td>●</td>
<td>ASI_REAL_IO</td>
<td>RW²,⁴ (decoded)</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>Real address, noncacheable, with side effect (deprecated)</td>
</tr>
<tr>
<td>1616</td>
<td>●</td>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)</td>
<td>RW²,⁴,¹⁴ (decoded)</td>
<td>R 8</td>
<td>—</td>
<td>—</td>
<td>Primary address space, block load/store, as if user (nonprivileged)</td>
</tr>
<tr>
<td>1716</td>
<td>●</td>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS)</td>
<td>RW²,⁴,¹⁴ (decoded)</td>
<td>R 8</td>
<td>—</td>
<td>—</td>
<td>Secondary address space, block load/store, as if user (nonprivileged)</td>
</tr>
<tr>
<td>1816</td>
<td>●</td>
<td>ASI AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)</td>
<td>RW²,⁴,¹⁴ (decoded)</td>
<td>V</td>
<td>—</td>
<td>—</td>
<td>Primary address space, as if user (nonprivileged), little-endian</td>
</tr>
<tr>
<td>1916</td>
<td>●</td>
<td>ASI AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)</td>
<td>RW²,⁴,¹⁴ (decoded)</td>
<td>V</td>
<td>—</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>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>1C16</td>
<td>●</td>
<td>ASI_REAL_LITTLE</td>
<td>RW²,⁴ (decoded)</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>Real address, little-endian</td>
</tr>
<tr>
<td>1D16</td>
<td>●</td>
<td>ASI_REAL_IO_LITTLE</td>
<td>RW²,⁴ (decoded)</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>Real address, noncacheable, with side effect, little-endian (deprecated)</td>
</tr>
<tr>
<td>1E16</td>
<td>●</td>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUPL)</td>
<td>RW²,⁴,¹⁴ (decoded)</td>
<td>V</td>
<td>—</td>
<td>—</td>
<td>Primary address space, block load/store, as if user (nonprivileged), little-endian</td>
</tr>
<tr>
<td>1F16</td>
<td>●</td>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL)</td>
<td>RW²,⁴,¹⁴ (decoded)</td>
<td>V</td>
<td>—</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 ( ● )</td>
<td>ASI Name (and Abbreviation)</td>
<td>Access Type(s)</td>
<td>Virtual Address (VA)</td>
<td>V/non-T/R</td>
<td>Shared per strand</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>20₁₆</td>
<td></td>
<td>ASI_SCRATCHPAD</td>
<td>RW²,⁶</td>
<td>(decoded; non-T per strand)</td>
<td></td>
<td></td>
<td>Privileged Scratchpad registers; implementation dependent¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0₁₆</td>
<td></td>
<td></td>
<td>Scratchpad Register 0¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8₁₆</td>
<td></td>
<td></td>
<td>Scratchpad Register 1¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10₁₆</td>
<td></td>
<td></td>
<td>Scratchpad Register 2¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18₁₆</td>
<td></td>
<td></td>
<td>Scratchpad Register 3¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20₁₆</td>
<td></td>
<td></td>
<td>Scratchpad Register 4¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28₁₆</td>
<td></td>
<td></td>
<td>Scratchpad Register 5¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30₁₆</td>
<td></td>
<td></td>
<td>Scratchpad Register 6¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38₁₆</td>
<td></td>
<td></td>
<td>Scratchpad Register 7¹</td>
</tr>
<tr>
<td>2¹₆</td>
<td></td>
<td>ASI_MMU, ASI_MMU_CONTEXTID</td>
<td>RW²,⁶</td>
<td>(decoded; non-T per strand)</td>
<td></td>
<td></td>
<td>MMU context registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8₁₆</td>
<td></td>
<td></td>
<td>I/D MMU Primary Context ID register 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10₁₆</td>
<td></td>
<td></td>
<td>I/D MMU Secondary Context ID register 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>108₁₆</td>
<td></td>
<td></td>
<td>I/D Primary Context ID register 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110₁₆</td>
<td></td>
<td></td>
<td>I/D MMU Secondary Context ID register 1</td>
</tr>
<tr>
<td>22₁₆</td>
<td></td>
<td>ASI_TWINX_AS_IF_USERPRIMARY (ASI_TWINX_AIUP)</td>
<td>R²,⁷,¹¹</td>
<td>(decoded)</td>
<td>V</td>
<td>_</td>
<td>Primary address space, 128-bit atomic load twin extended word, as if user (nonprivileged)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASI_ST_BLKINIT_AS_IF_USERPRIMARY (ASI_STBI_AIUP)</td>
<td>W²,²⁰</td>
<td>(decoded)</td>
<td>V</td>
<td>_</td>
<td>Primary address space, block-initializing store, as if user (nonprivileged); may be evicted from cache quickly (installed in cache with LRU [least-recently-used] or equivalent attribute)</td>
</tr>
<tr>
<td>2³₁₆</td>
<td></td>
<td>ASI_TWINX_AS_IF_USERSECONDARY (ASI_TWINX_AIUS)</td>
<td>R²,⁷,¹¹</td>
<td>(decoded)</td>
<td>V</td>
<td>_</td>
<td>Secondary address space, 128-bit atomic load twin extended word, as if user (nonprivileged); may be evicted from cache quickly (installed in cache with LRU [least-recently-used] or equivalent attribute)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASI_ST_BLKINIT_AS_IF_USERSECONDARY (ASI_STBI_AIUS)</td>
<td>W²,²⁰</td>
<td>(decoded)</td>
<td>V</td>
<td>_</td>
<td>Secondary address space, block-initializing store, as if user (nonprivileged)</td>
</tr>
<tr>
<td>2⁴₁₆</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent¹</td>
</tr>
<tr>
<td>ASI Value</td>
<td><code>req'd</code></td>
<td><code>opt'l</code></td>
<td>ASI Name (and Abbreviation)</td>
<td>Access Type(s)</td>
<td>Virtual Address (VA)</td>
<td>V/ non-T/ R</td>
<td>Shared/per strand</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>----------------------------</td>
<td>---------------</td>
<td>---------------------</td>
<td>------------</td>
<td>------------------</td>
</tr>
<tr>
<td>2516</td>
<td>○</td>
<td>●</td>
<td>ASI_QUEUE</td>
<td>(see below)</td>
<td>(decoded; see below)</td>
<td>non-T/ R per strand</td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td>RW²,⁶</td>
<td>3C0₁₆</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td>RW²,⁶,₁⁷</td>
<td>3C₈₁₆</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td>RW²,⁶</td>
<td>3D₀₁₆</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td>RW²,⁶,₁⁷</td>
<td>3D₈₁₆</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td>RW²,⁶</td>
<td>3E₀₁₆</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td>RW²,⁶,₁⁷</td>
<td>3E₈₁₆</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td>RW²,⁶</td>
<td>3F₀₁₆</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 10-1** Oracle SPARC Architecture ASIs (3 of 7)

<table>
<thead>
<tr>
<th>ASI Value</th>
<th><code>req'd</code></th>
<th><code>opt'l</code></th>
<th>ASI Name (and Abbreviation)</th>
<th>Access Type(s)</th>
<th>Virtual Address (VA)</th>
<th>V/ non-T/ R</th>
<th>Shared/per strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2616</td>
<td>○</td>
<td></td>
<td>ASI_TWINX_REAL (ASI_TWINX_R)</td>
<td>R²,⁷,¹¹</td>
<td>(decoded) R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td>ASI_QUAD_LDD_REAL²</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2716</td>
<td>○</td>
<td></td>
<td>ASI_TWINX_NUCLEUS (ASI_TWINX_N)</td>
<td>R²,⁷,¹¹</td>
<td>(decoded) V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td>ASI_ST_BLKINIT_NUCLEUS (ASI_STBI_N)</td>
<td>(decoded) V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2816</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td>2,¹²</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2916</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td>2,¹²</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2A16</td>
<td>○</td>
<td></td>
<td>ASI_TWINX_AS_IF_USER_ PRIMARY_LITTLE (ASI_TWINX_AIUP_L)</td>
<td>R²,⁷,¹¹</td>
<td>(decoded) V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td>ASI_ST_BLKINIT_AS_IF_USER_ PRIMARY_LITTLE (ASI_STBI_AIUPL)</td>
<td>(decoded) V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2B16</td>
<td>○</td>
<td></td>
<td>ASI_TWINX_AS_IF_USER_ SECONDARY_LITTLE (ASI_TWINX_AIUS_L)</td>
<td>R²,⁷,¹¹</td>
<td>(decoded) V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td>ASI_ST_BLKINIT_AS_IF_USER_ SECONDARY_LITTLE (ASI_STBI_AIUS_L)</td>
<td>(decoded) V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2C16</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td>²</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D16</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td>²,¹²</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASI Value</td>
<td>req’d(●)</td>
<td>opt’d(❍)</td>
<td>ASI Name (and Abbreviation)</td>
<td>Access Type(s)</td>
<td>Virtual Address (VA)</td>
<td>V/ non-T/R</td>
<td>Shared per strand</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
<td>-----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>------------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>2E16</td>
<td></td>
<td></td>
<td>ASI_TWINX_REAL_LITTLE (ASI_TWINX_REAL_L)</td>
<td>R^{2,7,11} (decoded)</td>
<td>R</td>
<td></td>
<td>128-bit atomic twin-extended-word load from real address, little-endian</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ASI_QUAD_LDD_REAL_LITTLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2F16</td>
<td></td>
<td></td>
<td>ASI_TWINX_NUCLEUS_LITTLE (ASI_TWINX_NL)</td>
<td>R^{2,7,11} (decoded)</td>
<td>V</td>
<td></td>
<td>Nucleus context, 128-bit atomic load twin-extended-word, little-endian</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ASI_ST_BLKINIT_NUCLEUS_LITTLE (ASI_STBI_NL)</td>
<td>W^{2,20} (decoded)</td>
<td>V</td>
<td></td>
<td>Nucleus context, block-initializing store, little-endian; may be evicted from cache quickly (installed in cache with LRU [least-recently-used] or equivalent attribute)</td>
<td></td>
</tr>
<tr>
<td>30_{16}–7F_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reserved for use in hyperprivilege mode</td>
</tr>
<tr>
<td>42_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>43_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>45_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>46_{16}–48_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>4A_{16}–4B_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4C_{16}</td>
<td></td>
<td></td>
<td>Error Status and Enable Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>75_{16}–7F_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>80_{16}</td>
<td></td>
<td></td>
<td>ASI_PRIMARY (ASI_P)</td>
<td>RW^4 (decoded)</td>
<td>V</td>
<td></td>
<td>Implicit primary address space</td>
<td></td>
</tr>
<tr>
<td>81_{16}</td>
<td></td>
<td></td>
<td>ASI_SECONDARY (ASI_S)</td>
<td>RW^4 (decoded)</td>
<td>V</td>
<td></td>
<td>Secondary address space</td>
<td></td>
</tr>
<tr>
<td>82_{16}</td>
<td></td>
<td></td>
<td>ASI_PRIMARY_NO_FAULT (ASI_PNF)</td>
<td>R^{2,11} (decoded)</td>
<td>V</td>
<td></td>
<td>Primary address space, no fault</td>
<td></td>
</tr>
<tr>
<td>83_{16}</td>
<td></td>
<td></td>
<td>ASI_SECONDARY_NO_FAULT (ASI_SNF)</td>
<td>R^{2,11} (decoded)</td>
<td>V</td>
<td></td>
<td>Secondary address space, no fault</td>
<td></td>
</tr>
<tr>
<td>84_{16}–85_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>86_{16}–87_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88_{16}</td>
<td></td>
<td></td>
<td>ASI_PRIMARY_LITTLE (ASI_PL)</td>
<td>RW^4 (decoded)</td>
<td>V</td>
<td></td>
<td>Implicit primary address space, little-endian</td>
<td></td>
</tr>
<tr>
<td>89_{16}</td>
<td></td>
<td></td>
<td>ASI_SECONDARY_LITTLE (ASI_SL)</td>
<td>RW^4 (decoded)</td>
<td>V</td>
<td></td>
<td>Secondary address space, little-endian</td>
<td></td>
</tr>
<tr>
<td>8A_{16}</td>
<td></td>
<td></td>
<td>ASI_PRIMARY_NO_FAULT_LITTLE (ASI_PNFL)</td>
<td>R^{2,11} (decoded)</td>
<td>V</td>
<td></td>
<td>Primary address space, no fault, little-endian</td>
<td></td>
</tr>
<tr>
<td>8B_{16}</td>
<td></td>
<td></td>
<td>ASI_SECONDARY_NO_FAULT_LITTLE (ASI_SNFL)</td>
<td>R^{2,11} (decoded)</td>
<td>V</td>
<td></td>
<td>Secondary address space, no fault, little-endian</td>
<td></td>
</tr>
<tr>
<td>8C_{16}–8F_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90_{16}–9C_{16}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9D_{16}–9F_{16}</td>
<td></td>
<td></td>
<td></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>V/ non-T/ R</td>
<td>Shared /per strand</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>-------------</td>
<td>-------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>A0&lt;sub&gt;16&lt;/sub&gt;–AF&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td></td>
<td>_16</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>B0&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PIC</td>
<td>RW</td>
<td>0&lt;sub&gt;16&lt;/sub&gt;</td>
<td>non-T</td>
<td>per strand</td>
<td>Performance Instrumentation Counter 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RW</td>
<td>8&lt;sub&gt;16&lt;/sub&gt;</td>
<td>non-T</td>
<td>per strand</td>
<td>Performance Instrumentation Counter 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RW</td>
<td>10&lt;sub&gt;16&lt;/sub&gt;</td>
<td>non-T</td>
<td>per strand</td>
<td>Performance Instrumentation Counter 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RW</td>
<td>18&lt;sub&gt;16&lt;/sub&gt;</td>
<td>non-T</td>
<td>per strand</td>
<td>Performance Instrumentation Counter 3</td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>●</td>
<td></td>
<td>_16</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>B2&lt;sub&gt;16&lt;/sub&gt;–BF&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td></td>
<td>_15</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent&lt;sup&gt;1&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>C0&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST8_PRIMARY (ASI_PST8_P)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 8×8-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C1&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST8_SECONDARY (ASI_PST8_S)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 8×8-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C2&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST16_PRIMARY (ASI_PST16_P)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 4×16-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C3&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST16_SECONDARY (ASI_PST16_S)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 4×16-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C4&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST32_PRIMARY (ASI_PST32_P)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 2×32-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C5&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST32_SECONDARY (ASI_PST32_S)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 2×32-bit partial store</td>
<td></td>
</tr>
<tr>
<td>C6&lt;sub&gt;16&lt;/sub&gt;–C7&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td></td>
<td>_15</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent&lt;sup&gt;1&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>C8&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 8×8-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>C9&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 8×8-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CA&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 4×16-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CB&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 4×16-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CC&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 2×32-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CD&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL)</td>
<td>RW,10,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 2×32-bit partial store, little-endian</td>
<td></td>
</tr>
<tr>
<td>CE&lt;sub&gt;16&lt;/sub&gt;–CF&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td></td>
<td>_15</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent&lt;sup&gt;1&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>D0&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_FL8_PRIMARY (ASI_FL8_P)</td>
<td>RW,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, one 8-bit floating-point load/store</td>
<td></td>
</tr>
<tr>
<td>D1&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_FL8_SECONDARY (ASI_FL8_S)</td>
<td>RW,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, one 8-bit floating-point load/store</td>
<td></td>
</tr>
<tr>
<td>D2&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_FL16_PRIMARY (ASI_FL16_P)</td>
<td>RW,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, one 16-bit floating-point load/store</td>
<td></td>
</tr>
<tr>
<td>D3&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td>ASI_FL16_SECONDARY (ASI_FL16_S)</td>
<td>RW,14</td>
<td>(decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, one 16-bit floating-point load/store</td>
<td></td>
</tr>
<tr>
<td>D4&lt;sub&gt;16&lt;/sub&gt;–D7&lt;sub&gt;16&lt;/sub&gt;</td>
<td>●</td>
<td></td>
<td>_15</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent&lt;sup&gt;1&lt;/sup&gt;</td>
<td></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>V/ non-T/R</td>
<td>Shared per strand</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>----------------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>-----------</td>
<td>------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>D8₁₆</td>
<td>●</td>
<td>ASI_FL₈_PRIMARY_LITTLE (ASI_FL₈_PL)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, one 8-bit floating point load/store, little-endian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D₉₁₆</td>
<td>●</td>
<td>ASI_FL₈_SECONDARY_LITTLE (ASI_FL₈_SL)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, one 8-bit floating point load/store, little-endian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dₐ₁₆</td>
<td>●</td>
<td>ASI_FL₁₆_PRIMARY_LITTLE (ASI_FL₁₆_PL)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, one 16-bit floating-point load/store, little-endian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D₉₁₆</td>
<td>●</td>
<td>ASI_FL₁₆_SECONDARY_LITTLE (ASI_FL₁₆_SL)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Second address space, one 16-bit floating point load/store, little-endian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dₐ₁₆</td>
<td>●</td>
<td>ASI_FL₁₆_PRIMARY_LITTLE (ASI_FL₁₆_P)</td>
<td>W₁₈ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Implementation dependent¹</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₀₁₆</td>
<td>●</td>
<td>ASI_BLOCK_COMMIT_PRIMARY (ASI_BLK_COMMIT_P)</td>
<td>W₈,₁₀,₁₄ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 8x8-byte block store commit operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₁₁₆</td>
<td>●</td>
<td>ASI_BLOCK_COMMIT_SECONDARY (ASI_BLK_COMMIT_S)</td>
<td>W₈,₁₀,₁₄ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 8x8-byte block store commit operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₂₁₆</td>
<td>●</td>
<td>ASI_TWINX_PRIMARY (ASI_TWINX_P)</td>
<td>R¹⁹ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 128-bit atomic load twin extended word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₃₁₆</td>
<td>●</td>
<td>ASI_TWINX_SECONDARY (ASI_TWINX_S)</td>
<td>R¹⁹ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 128-bit atomic load twin extended-word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₄₁₆-</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₅₁₆</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₆₁₆</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EA₁₆</td>
<td>●</td>
<td>ASI_TWINX_PRIMARY_LITTLE (ASI_TWINX_P)</td>
<td>R¹⁹ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, 128-bit atomic load twin extended word, little endian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₉₁₆</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EB₁₆</td>
<td>●</td>
<td>ASI_TWINX_SECONDARY_LITTLE (ASI_TWINX_P)</td>
<td>R¹⁹ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 128-bit atomic load twin extended word, little endian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC₁₆-</td>
<td>●</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent¹</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>V</td>
<td>—</td>
<td>Primary address space, 8x8-byte block load/store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F₁₁₆</td>
<td>●</td>
<td>ASI_BLOCK_SECONDARY (ASI_BLK_S)</td>
<td>RW₈,₁₄ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Secondary address space, 8x8-byte block load/store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F₂₁₆</td>
<td>●</td>
<td>ASI_ST_BLKINIT_PRIMARY_MRU (ASI_STBI_PM)</td>
<td>W₂₀ (decoded)</td>
<td>V</td>
<td>—</td>
<td>Primary address space, block-initializing store (installed in cache with MRU [most-recently-used] or equivalent attribute)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 10-1  Oracle SPARC Architecture ASIs  (7 of 7)

<table>
<thead>
<tr>
<th>ASI Value</th>
<th>req'd ()</th>
<th>ASI Name (and Abbreviation)</th>
<th>Access Type(s)</th>
<th>Virtual Address (VA)</th>
<th>V/ non-T</th>
<th>Shared/per strand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F316</td>
<td>()</td>
<td>ASI_ST_BLKINIT_SECONDARY_MRU (ASI_STBI_SM)</td>
<td>W20 (decoded)</td>
<td>V—</td>
<td>—</td>
<td>Secondary address space, block-initializing store (installed in cache with MRU [most-recently-used] or equivalent attribute)</td>
<td></td>
</tr>
<tr>
<td>F416—</td>
<td>()</td>
<td>ASI_BLOCK_PRIMARY_LITTLE (ASI_BLK_PL)</td>
<td>RW8,14 (decoded)</td>
<td>V—</td>
<td>—</td>
<td>Primary address space, 8x8-byte block load/store, little endian</td>
<td></td>
</tr>
<tr>
<td>F616—</td>
<td>()</td>
<td>ASI_BLOCK_SECONDARY_LITTLE (ASI_BLK_SL)</td>
<td>RW8,14 (decoded)</td>
<td>V—</td>
<td>—</td>
<td>Secondary address space, 8x8-byte block load/store, little endian</td>
<td></td>
</tr>
<tr>
<td>FA16</td>
<td>()</td>
<td>ASI_ST_BLKINIT_PRIMARY_LITTLE_MR U (ASI_STBI_PLM)</td>
<td>W20 (decoded)</td>
<td>V—</td>
<td>—</td>
<td>Primary address space, block-initializing store, little endian (installed in cache with MRU [most-recently-used] or equivalent attribute)</td>
<td></td>
</tr>
<tr>
<td>FB16</td>
<td>()</td>
<td>ASI_ST_BLKINIT_SECONDARY_LITTLE MRU (ASI_STBI_SLM)</td>
<td>W20 (decoded)</td>
<td>V—</td>
<td>—</td>
<td>Secondary address space, block-initializing store, little endian (installed in cache with MRU [most-recently-used] or equivalent attribute)</td>
<td></td>
</tr>
<tr>
<td>FC16—</td>
<td>()</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent1</td>
<td></td>
</tr>
<tr>
<td>FF16</td>
<td>()</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Implementation dependent1</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, LDFA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDX, PREFETCHA, STBA, STTWA, STDFA, STFA, STHA, STWA, STXA, SWAPA).

5 May be used with all of the following load alternate and store alternate instructions: LDTWA, LDDFA, LDFA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDX, STBA, STTWA, STDFA, STFA, STHA, STWA, STXA.

Use with an atomic alternate or prefetch alternate instruction ( CASA, CASXA, LDSTUBA, SWAPA or PREFETCHA) causes a DAE_invalid_asi exception.

6 May only be used in a LDXA or STXA instruction for RW ASIs, LDFA for read-only ASIs and STXA for write-only ASIs. Use of LDFA for write-only ASIs, STXA for read-only ASIs, or any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a DAE_invalid_asi exception.

7 May only be used in an LDTXA instruction. Use of this ASI in any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a DAE_invalid_asi exception.
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}, and 17_{16} (ASI\_\_*AS\_IF\_USER\_* )

These ASI are intended to be used in accesses from privileged mode, but are processed as if they were issued from nonprivileged mode. Therefore, they are subject to privilege-related exceptions. They are distinguished from each other by the context from which the access is made, as described in TABLE 10-2.

When one of these ASIs is specified in a load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- In any other privilege mode:
  - If U/DMMU TTE.p = 1, a *DAE_privilege_violation* exception occurs
  - Otherwise, the access occurs and its endianness is determined by the U/DMMU TTE.ie bit. If U/DMMU TTE.ie = 0, the access is big-endian; otherwise, it is little-endian.
10.4.2 ASIs 18\textsubscript{16}, 19\textsubscript{16}, 1E\textsubscript{16}, and 1F\textsubscript{16} (ASI\_\_*AS\_IF\_USER\_\_*\_LITTLE)

These ASIs are little-endian versions of ASIs 10\textsubscript{16}, 11\textsubscript{16}, 16\textsubscript{16}, and 17\textsubscript{16} (ASI\_\_*AS\_IF\_USER\_\_*), described in section 10.4.1. Each operates identically to the corresponding non-little-endian ASI, except that if an access occurs its endianness is the opposite of that for the corresponding non-little-endian ASI.

These ASI are intended to be used in accesses from privileged mode, but are processed as if they were issued from nonprivileged mode. Therefore, they are subject to privilege-related exceptions. They are distinguished from each other by the context from which the access is made, as described in TABLE 10-3.

When one of these ASIs is specified in a load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a privileged\_action exception occurs
- In any other privilege mode:
  - If U/DMMU TTE.|e| = 1, a DAE\_privilege\_violation exception occurs
  - Otherwise, the access occurs and its endianness is determined by the U/DMMU TTE.|e| bit. If U/DMMU TTE.|e| = 0, the access is little-endian; otherwise, it is big-endian.

<table>
<thead>
<tr>
<th>ASI</th>
<th>Names</th>
<th>Addressing (Context)</th>
<th>Endianness of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>18\textsubscript{16}</td>
<td>ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)</td>
<td>Virtual (Primary)</td>
<td>Little-endian when U/DMMU TTE.</td>
</tr>
<tr>
<td>19\textsubscript{16}</td>
<td>ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
<tr>
<td>1E\textsubscript{16}</td>
<td>ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUPL)</td>
<td>Virtual (Primary)</td>
<td></td>
</tr>
<tr>
<td>1F\textsubscript{16}</td>
<td>ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL)</td>
<td>Virtual (Secondary)</td>
<td></td>
</tr>
</tbody>
</table>

10.4.3 ASI 14\textsubscript{16} (ASI\_REAL)

When ASI\_REAL is specified in any load alternate, store alternate or prefetch alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a privileged\_action exception occurs
- In any other privilege mode:
- VA is passed through to RA
- During the address translation, context values are disregarded.
- The endianness of the access is determined by the U/DMMU TTE.ie bit; if U/DMMU TTE.ie = 0, the access is big-endian, otherwise it is little-endian.

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

10.4.4 ASI 15₁₆ (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
- Used with any other load alternate or store alternate instruction, in privileged mode:
  - VA is passed through to RA
  - During the address translation, context values are disregarded.
  - The endianness of the access is determined by the U/DMMU TTE.ie bit; if U/DMMU TTE.ie = 0, the access is big-endian, otherwise it is little-endian.

10.4.5 ASI 1C₁₆ (ASI_REAL_LITTLE)

ASI_REAL_LITTLE is a little-endian version of ASI 14₁₆ (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 1D₁₆ (ASI_REAL_IO_LITTLE)

ASI_REAL_IO_LITTLE is a little-endian version of ASI 15₁₆ (ASI_REAL_IO). It operates identically to ASI_REAL_IO, except if an access occurs, its endianness the opposite of that for ASI_REAL_IO.

10.4.7 ASIs 22₁₆, 23₁₆, 26₁₆, 27₁₆, 2A₁₆, 2B₁₆, 2E₁₆, 2F₁₆ (Privileged Load Integer Twin Extended Word)

ASIs 22₁₆, 23₁₆, 26₁₆, 2A₁₆, 2B₁₆, 2E₁₆, 2F₁₆ exist for use with the (nonportable) LDTXₐ instruction as atomic Load Integer Twin Extended Word operations (see Load Integer Twin Extended Word from Alternate Space on page 249). These ASIs are distinguished by the context from which the access is made and the endianness of the access, as described in TABLE 10-4.
When these ASIs are used with LDTXA, a `mem_address_not_aligned` exception is generated if the operand address is not 16-byte aligned.

If these ASIs are used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a `DAE_invalid_asi` exception is always generated and `mem_address_not_aligned` is not generated.

### 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 249). These two ASIs are distinguished by the endianness of the access, as described in TABLE 10-5.

<table>
<thead>
<tr>
<th>ASI</th>
<th>Name</th>
<th>Addressing (Context)</th>
<th>Endianness of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>2616</td>
<td>ASI_TWIX_REAL (ASI_TWIX_R)</td>
<td>Real (—)</td>
<td>Big-endian when U/DMMU TTE.ie = 0; little-endian when U/DMMU TTE.ie = 1</td>
</tr>
<tr>
<td>2E16</td>
<td>ASI_TWIX_REAL_LITTLE (ASI_TWIX_REAL_L)</td>
<td>Real (—)</td>
<td>Little-endian when U/DMMU TTE.ie = 0; big-endian when U/DMMU TTE.ie = 1</td>
</tr>
</tbody>
</table>

When these ASIs are used with LDTXA, a `mem_address_not_aligned` exception is generated if the operand address is not 16-byte aligned.
If these ASIs are used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or 
PREFETCHA instruction, a $\text{DAE\_invalid\_asi}$ exception is always generated and 
$\text{mem\_address\_not\_aligned}$ is not generated.

**Compatibility Note** These ASIs replaced ASIs $34_{16}$ and $3C_{16}$ used in earlier 
UltraSPARC implementations; see the Compatibility Note on 
page 553 for details.

### 10.4.9 ASIs E2$_{16}$, E3$_{16}$, EA$_{16}$, EB$_{16}$

**(Nonprivileged Load Integer Twin Extended Word)**

ASIs $E2_{16}$, $E3_{16}$, $EA_{16}$, and $EB_{16}$ exist for use with the (nonportable) LDTXA instruction as atomic 
Load Integer Twin Extended Word operations (see Load Integer Twin Extended Word from Alternate Space 
on page 249). These ASIs are distinguished by the address space accessed (Primary or Secondary) and 
the endianness of the access, as described in TABLE 10-6.

**TABLE 10-6** Load Integer Twin Extended Word ASIs

<table>
<thead>
<tr>
<th>ASI</th>
<th>Names</th>
<th>Addressing (Context)</th>
<th>Endianness of Access</th>
</tr>
</thead>
</table>
| $E2_{16}$ | \texttt{ASI\_TWINX\_PRIMARY} (\texttt{ASI\_TWINX\_P}) | Virtual (Primary)   | Big-endian when U/DMMU $\text{TTE.ie}=0$, 
little-endian when U/DMMU $\text{TTE.ie}=1$ |
| $E3_{16}$ | \texttt{ASI\_TWINX\_SECONDARY} (\texttt{ASI\_TWINX\_S}) | Virtual (Secondary) | Little-endian when U/DMMU $\text{TTE.ie}=0$, 
big-endian when U/DMMU $\text{TTE.ie}=1$ |
| $EA_{16}$ | \texttt{ASI\_TWINX\_PRIMARY\_LITTLE} (\texttt{ASI\_TWINX\_PL}) | Virtual (Primary)   | Big-endian when U/DMMU $\text{TTE.ie}=0$, 
little-endian when U/DMMU $\text{TTE.ie}=1$ |
| $EB_{16}$ | \texttt{ASI\_TWINX\_SECONDARY\_LITTLE} (\texttt{ASI\_TWINX\_SL}) | Virtual (Secondary) | Little-endian when U/DMMU $\text{TTE.ie}=0$, 
big-endian when U/DMMU $\text{TTE.ie}=1$ |

When these ASIs are used with LDTXA, a $\text{mem\_address\_not\_aligned}$ 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 $\text{DAE\_invalid\_asi}$ exception is always generated and 
$\text{mem\_address\_not\_aligned}$ is not generated.

### 10.4.10 Block Load and Store ASIs

ASIs $16_{16}$, $17_{16}$, $1E_{16}$, $1F_{16}$, $E0_{16}$, $E1_{16}$, $F0_{16}$, $F1_{16}$, $F8_{16}$, and $F9_{16}$ exist for use with LDDFA and STDFA 
instructions as Block Load (LDBLOCKF) and Block Store (STBLOCKF) operations (see Block Load on 
page 230 and Block Store on page 322).

When these ASIs are used with the LDDFA (STDFA) opcode for Block Load (Store), a $\text{mem\_address\_not\_aligned}$ exception is generated if the operand address is not 64-byte aligned.

ASIs $E0_{16}$ and $E1_{16}$ are only defined for use in Block Store with Commit operations (see page 322). 
Neither ASI $E0_{16}$ nor $E1_{16}$ should be used with the LDDFA opcode; however, if either is used, the 
resulting behavior is specified in the LDDFA instruction description on page 236.

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 $\text{DAE\_invalid\_asi}$ exception is always generated and 
$\text{mem\_address\_not\_aligned}$ is not generated.
10.4.11 Partial Store ASIs

ASI s C016–C516 and C816–CD16 exist for use with the STDFA instruction as Partial Store (STPARTIALF) operations (see DAE_invalid_asi eDAE_invalid_asi eStore Partial Floating-Point on page 332).

When these ASIs are used with STDFA for Partial Store, a mem_address_not_aligned exception is generated if the operand address is not 8-byte aligned and an illegal_instruction exception is generated if $i = 1$ in the instruction and the ASI register contains one of the Partial Store ASIs.

If one of these ASIs is used with a Store Alternate instruction other than STDFA, a Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a DAE_invalid_asi exception is generated and mem_address_not_aligned, LDDF_mem_address_not_aligned, and illegal_instruction (for $i = 1$) are not generated.

ASI s C016–C516 and C816–CD16 are only defined for use in Partial Store operations (see page 332). 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 237.

10.4.12 Short Floating-Point Load and Store ASIs

ASI s D016–D316 and D816–DB16 exist for use with the LDDFA and STDFA instructions as Short Floating-point Load and Store operations (see Load Floating-Point Register on page 233 and Store Floating-Point on page 325).

When ASI D216, D316, DA16, or DB16 is used with LDDFA (STDFA) for a 16-bit Short Floating-point Load (Store), a mem_address_not_aligned exception is generated if the operand address is not halfword-aligned.

If any of these ASIs are used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a DAE_invalid_asi exception is always generated and mem_address_not_aligned is not generated.

10.5 ASI-Accessible Registers

In this section the scratchpad registers are described.

A list of Oracle SPARC Architecture 2011 ASIs is shown in TABLE 10-1 on page 399.

10.5.1 Privileged Scratchpad Registers (ASI_SCRATCHPAD) 

An Oracle SPARC 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.

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>20&lt;sub&gt;16&lt;/sub&gt;</td>
<td>00&lt;sub&gt;16&lt;/sub&gt;</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>08&lt;sub&gt;16&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10&lt;sub&gt;16&lt;/sub&gt;</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18&lt;sub&gt;16&lt;/sub&gt;</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20&lt;sub&gt;16&lt;/sub&gt;</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28&lt;sub&gt;16&lt;/sub&gt;</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30&lt;sub&gt;16&lt;/sub&gt;</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>38&lt;sub&gt;16&lt;/sub&gt;</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 `DAE_invalid_asi` exception).

**V9 Compatibility Note**: Privileged scratchpad registers are an Oracle SPARC Architecture extension to SPARC V9.

## 10.6 ASI Changes in the Oracle SPARC Architecture

The following Compatibility Note summarize ASI changes in the Oracle SPARC Architecture.

**Compatibility Note**: The names of several ASIs used in earlier UltraSPARC implementations have changed in Oracle SPARC Architecture. Their functions have not changed; just their names have changed.

<table>
<thead>
<tr>
<th>ASI#</th>
<th>Previous UltraSPARC</th>
<th>Oracle SPARC Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>14&lt;sub&gt;16&lt;/sub&gt;</td>
<td>ASI_PHYS_USE_EC</td>
<td>ASI_REAL</td>
</tr>
<tr>
<td>17&lt;sub&gt;16&lt;/sub&gt;</td>
<td>ASI_PHYS_BYPASS_EC_WITH_EBIT</td>
<td>ASI_REAL_IO</td>
</tr>
<tr>
<td>1C&lt;sub&gt;16&lt;/sub&gt;</td>
<td>ASI_PHYS_USE_EC_LITTLE (ASI_PHYS_USE_EC_L)</td>
<td>ASI_REAL_LITTLE</td>
</tr>
<tr>
<td>1D&lt;sub&gt;16&lt;/sub&gt;</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>
Performance Instrumentation

This chapter describes the architecture for performance monitoring hardware on Oracle SPARC Architecture processors. The architecture is based on the design of performance instrumentation counters in previous Oracle SPARC Architecture processors, with an extension for the selective sampling of instructions.

11.1 High-Level Requirements

11.1.1 Usage Scenarios

The performance monitoring hardware on Oracle SPARC Architecture processors addresses the needs of various kinds of users. There are four scenarios envisioned:

- System-wide performance monitoring. In this scenario, someone skilled in system performance analysis (e.g., a Systems Engineer) is using analysis tools to evaluate the performance of the entire system. An example of such a tool is cpustat. The objective is to obtain performance data relating to the configuration and behavior of the system, e.g., the utilization of the memory system.

- Self-monitoring of performance by the operating system. In this scenario the OS is gathering performance data in order to tune the operation of the system. Some examples might be:
  - (a) determining whether the processors in the system should be running in single- or multi-stranded mode.
  - (b) determining the affinity of a process to a processor by examining that process's memory behavior.

- Performance analysis of an application by a developer. In this scenario a developer is trying to optimize the performance of a specific application, by altering the source code of the application or the compilation options. The developer needs to know the performance characteristics of the components of the application at a coarse grain, and where these are problematic, to be able to determine fine-grained performance information. Using this information, the developer will alter the source or compilation parameters, re-run the application, and observe the new performance characteristics. This process is repeated until performance is acceptable, or no further improvements can be found.

An example might be that a loop nest is measured to be not performing well. Upon closer inspection, the developer determines that the loop has poor cache behavior, and upon more detailed inspection finds a specific operation which repeatedly misses the cache. Reorganizing the code and/or data may improve the cache behavior.

- Monitoring of an application’s performance, e.g., by a Java Virtual Machine. In this scenario the application is not executing directly on the hardware, but its execution is being mediated by a piece of system software, which for the purposes of this document is called a Virtual Machine. This may
be a Java VM, or a binary translation system running software compiled for another architecture, or for an earlier version of the Oracle SPARC Architecture. One goal of the VM is to optimize the behavior of the application by monitoring its performance and dynamically reorganizing the execution of the application (e.g., by selective recompilation of the application).

This scenario differs from the previous one principally in the time allowed to gather performance data. Because the data are being gathered during the execution of the program, the measurements must not adversely affect the performance of the application by more than, say, a few percent, and must yield insight into the performance of the application in a relatively short time (otherwise, optimization opportunities are deferred for too long). This implies an observation mechanism which is of very low overhead, so that many observations can be made in a short time.

In contrast, a developer optimizing an application has the luxury of running or re-running the application for a considerable period of time (minutes or even hours) to gather data. However, the developer will also expect a level of precision and detail in the data which would overwhelm a virtual machine, so the accuracy of the data required by a virtual machine need not be as high as that supplied to the developer.

Scenarios 1 and 2 are adequately dealt with by a suitable set of performance counters capable of counting a variety of performance-related events. Counters are ideal for these situations because they provide low-overhead statistics without any intrusion into the behavior of the system or disruption to the code being monitored. However, counters may not adequately address the latter two scenarios, in which detailed and timely information is required at the level of individual instructions. Therefore, Oracle SPARC Architecture processors may also implement an instruction sampling mechanism.

11.1.2 Metrics

There are two classes of data reported by a performance instrumentation mechanism:

- **Architectural performance metrics.** These are metrics related to the observable execution of code at the architectural level (Oracle SPARC Architecture). Examples include:
  - The number of instructions executed
  - The number of floating point instructions executed
  - The number of conditional branch instructions executed

- **Implementation performance metrics.** These describe the behavior of the microprocessor in terms of its implementation, and would not necessarily apply to another implementation of the architecture.

In optimizing the performance of an application or system, attention will first be paid to the first class of metrics, and so these are more important. Only in performance-critical cases would the second class receive attention, since using these metrics requires a fairly extensive understanding of the specific implementation of the Oracle SPARC Architecture.

11.1.3 Accuracy Requirements

Accuracy requirements for performance instrumentation vary depending on the scenario. The requirements are complicated by the possibly speculative nature of Oracle SPARC Architecture processor implementations. For example, an implementation may include in its cache miss statistics the misses induced by speculative executions which were subsequently flushed, or provide two separate statistics, one for the misses induced by flushed instructions and one for misses induced by retired instructions. Although the latter would be desirable, the additional implementation complexity of associating events with specific instructions is significant, and so all events may be counted without distinction. The instruction sampling mechanism may distinguish between instructions that retired and those that were flushed, in which case sampling can be used to obtain statistical estimates of the frequencies of operations induced by mis-speculation.
For critical performance measurements, architectural event counts must be accurate to a high degree (1 part in $10^5$). Which counters are considered performance-critical (and therefore accurate to 1 part in $10^5$) are specified in implementation-specific documentation.

Implementation event counts must be accurate to 1 part in $10^3$, not including the speculative effects mentioned above. An upper bound on counter skew must be stated in implementation-specific documentation.

**Programming Note** Increasing the time between counter reads will mitigate the inaccuracies that could be introduced by counter skew (due to speculative effects).

## 11.2 Performance Counters and Controls

The performance instrumentation hardware provides performance instrumentation counters (PICs). The number and size of performance counters is implementation dependent, but each performance counter register contains at least one 32-bit counter. It is implementation dependent whether the performance counter registers are accessed as ASRs or are accessed through ASIs.

There are one or more performance counter control registers (PCRs) associated with the counter registers. It is implementation dependent whether the PCRs are accessed as ASRs or are accessed through ASIs.

Each counter in a counter register can count one kind of event at a time. The number of the kinds of events that can be counted is implementation dependent. For each performance counter register, the corresponding control register is used to select the event type being counted. A counter is incremented whenever an event of the matching type occurs. A counter may be incremented by an event caused by an instruction which is subsequently flushed (for example, due to mis-speculation). Counting of events may be controlled based on privilege mode or on the strand in which they occur. Masking may be provided to allow counting of subgroups of events (for example, various occurrences of different opcode groups).

A field that indicates when a counter has overflowed must be present in either each performance instrumentation counter or in a separate performance counter control register.

Performance counters are usually provided on a per-strand basis.

### 11.2.1 Counter Overflow

Overflow of a counter is recorded in the overflow-indication field of either a performance instrumentation counter or a separate performance counter control register.

**Programming Note** Counter overflow traps can also be used for sampling, by setting the initial counter value so that an interrupt occurs $n$ counts later.

Counter overflow traps are provided so that large counts can be maintained in software, beyond the range directly supported in hardware. The counters continue to count after an overflow, and software can utilize the overflow traps to maintain additional high-order bits.
Traps

A trap is a vectored transfer of control to software running in a privilege mode (see page 418) 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 Oracle SPARC 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 418.
- Virtual Processor States and Traps on page 419.
- Trap Categories on page 419.
- Trap Control on page 423.
- Trap-Table Entry Addresses on page 424.
- Trap Processing on page 432.
- Exception and Interrupt Descriptions on page 434.
Register Window Traps on page 439.

12.1 Virtual Processor Privilege Modes

An Oracle SPARC 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 12-1 Virtual Processor Privilege Modes

<table>
<thead>
<tr>
<th>PSTATE.priv</th>
<th>Virtual Processor Privilege Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nonprivileged</td>
</tr>
<tr>
<td>1</td>
<td>Privileged</td>
</tr>
</tbody>
</table>

A trap is delivered to the virtual processor in either privileged mode or hyperprivileged mode; in which mode the trap is delivered depends on:

- Its trap type
- The trap level (TL) at the time the trap is taken
- The privilege mode at the time the trap is taken

Traps detected in nonprivileged and privileged mode can be delivered to the virtual processor in privileged mode or hyperprivileged mode.

TABLE 12-4 on page 427 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 424 for details.

The maximum trap level at which privileged software may execute is MAXPTL (which, on an Oracle SPARC Architecture 2011 virtual processor, is 2).

Notes

Execution in nonprivileged mode with TL > 0 is an invalid condition that privileged software should never allow to occur.
CHAPTER 12 • Traps

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 $\text{MAXPTL} = 2$ in an Oracle SPARC 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
12.3.1 Precise Traps

A precise trap is induced by a particular instruction and occurs before any program-visible state has been changed by the trap-inducing instructions. When a precise trap occurs, several conditions must be true:

- The PC saved in TPC[TL] points to the instruction that induced the trap and the NPC saved in TNPC[TL] points to the instruction that was to be executed next.
- All instructions issued before the one that induced the trap have completed execution.
- Any instructions issued after the one that induced the trap remain unexecuted.

Among the actions that trap handler software might take when processing a precise trap are:

- Return to the instruction that caused the trap and reexecute it by executing a RETRY instruction (PC ← old PC, NPC ← old NPC).
- Emulate the instruction that caused the trap and return to the succeeding instruction by executing a DONE instruction (PC ← old NPC, NPC ← old NPC + 4).
- Terminate the program or process associated with the trap.

12.3.2 Deferred Traps

A deferred trap is also induced by a particular instruction, but unlike a precise trap, a deferred trap may occur after program-visible state has been changed. Such state may have been changed by the execution of either the trap-inducing instruction itself or by one or more other instructions.

There are two classes of deferred traps:

- **Termination deferred traps** — The instruction (usually a store) that caused the trap has passed the retirement point of execution (the TPC has been updated to point to an instruction beyond the one that caused the trap). The trap condition is an error that prevents the instruction from completing and its results becoming globally visible. A termination deferred trap has high trap priority, second only to the priority of resets.

  *Programming Note* | Not enough state is saved for execution of the instruction stream to resume with the instruction that caused the trap. Therefore, the trap handler must terminate the process containing the instruction that caused the trap.

- **Restartable deferred traps** — The program-visible state has been changed by the trap-inducing instruction or by one or more other instructions after the trap-inducing instruction.

  *SPARC V9 Compatibility Note* | A restartable deferred trap is the “deferred trap” defined in the SPARC V9 specification.

The fundamental characteristic of a restartable deferred trap is that the state of the virtual processor on which the trap occurred may not be consistent with any precise point in the instruction sequence being executed on that virtual processor. When a restartable deferred trap occurs, TPC[TL] and TNPC[TL] contain a PC value and an NPC value, respectively, corresponding to a point in the instruction sequence being executed on the virtual processor. This PC may correspond to the trap-inducing instruction or it may correspond to an instruction following the trap-inducing instruction. With a restartable deferred trap, program-visible updates may be missing from instructions prior to the instruction to which TPC[TL] refers. The missing updates are limited to instructions in the range from (and including) the actual trap-inducing instruction up to (but not including) the instruction to which TPC[TL] refers. By definition, the instruction to which TPC[TL] refers has not yet executed, therefore it cannot have any updates, missing or otherwise.
With a restartable deferred trap there must exist sufficient information to report the error that caused the deferred trap. If system software can recover from the error that caused the deferred trap, then there must be sufficient information to generate a consistent state within the processor so that execution can resume. Included in that information must be an indication of the mode (nonprivileged, privileged, or hyperprivileged) in which the trap-inducing instruction was issued.

How the information necessary for repairing the state to make it consistent state is maintained and how the state is repaired to a consistent state are implementation dependent. It is also implementation dependent whether execution resumes at the point of the trap-inducing instruction or at an arbitrary point between the trap-inducing instruction and the instruction pointed to by the TPC[TL], inclusively.

Associated with a particular restartable deferred trap implementation, the following must exist:

- An instruction that causes a potentially outstanding restartable deferred trap exception to be taken as a trap
- Instructions with sufficient privilege to access the state information needed by software to emulate the restartable deferred trap-inducing instruction and to resume execution of the trapped instruction stream.

Programming Note: Resuming execution may require the emulation of instructions that had not completed execution at the time of the restartable deferred trap, that is, those instructions in the deferred-trap queue.

Software should resume execution with the instruction starting at the instruction to which TPC[TL] refers. Hardware should provide enough information for software to recreate virtual processor state and update it to the point just before execution of the instruction to which TPC[TL] refers. After software has updated virtual processor state up to that point, it can then resume execution by issuing a RETRY instruction.

**IMPL. DEP. #32-V8-Ms10:** Whether any restartable deferred traps (and, possibly, associated deferred-trap queues) are present is implementation dependent.

Among the actions software can take after a restartable deferred trap are these:

- Emulate the instruction that caused the exception, emulate or cause to execute any other execution-deferred instructions that were in an associated restartable deferred trap state queue, and use RETRY to return control to the instruction at which the deferred trap was invoked.
- Terminate the program or process associated with the restartable deferred trap.

A deferred trap (of either of the two classes) is always delivered to the virtual processor in hyperprivileged mode.

### 12.3.3 Disrupting Traps

#### 12.3.3.1 Disrupting versus Precise and Deferred Traps

A *disrupting trap* is caused by a condition (for example, an interrupt) rather than directly by a particular instruction. This distinguishes it from *precise* and *deferred* traps.

When a disrupting trap has been serviced, trap handler software normally arranges for program execution to resume where it left off. This distinguishes disrupting traps from reset traps, since a reset trap vectors to a unique reset address and execution of the program that was running when the reset occurred is generally not expected to resume.

When a disrupting trap occurs, the following conditions are true:
1. The PC saved in TPC[TL] points to an instruction in the disrupted program stream and the NPC value saved in TNPC[TL] points to the instruction that was to be executed after that one.

2. All instructions issued before the instruction indicated by TPC[TL] have retired.

3. The instruction to which TPC[TL] refers and any instruction(s) that were issued after it remain unexecuted.

A disrupting trap may be due to an interrupt request directly related to a previously-executed instruction, for example, when a previous instruction sets a bit in the SOFTINT register.

12.3.3.2 Causes of Disrupting Traps

A disrupting trap may occur due to either an interrupt request or an error not directly related to instruction processing. The source of an interrupt request may be either internal or external. An interrupt request can be induced by the assertion of a signal not directly related to any particular virtual processor or memory state, for example, the assertion of an “I/O done” signal.

A condition that causes a disrupting trap persists until the condition is cleared.

12.3.3.3 Conditioning of Disrupting Traps

How disrupting traps are conditioned is affected by:

- The privilege mode in effect when the trap is outstanding, just before the trap is actually taken (regardless of the privilege mode that was in effect when the exception was detected).
- The privilege mode for which delivery of the trap is destined

**Outstanding in Nonprivileged or Privileged mode, destined for delivery in Privileged mode.** An outstanding disrupting trap condition in either nonprivileged mode or privileged mode and destined for delivery to privileged mode is held pending while the Interrupt Enable (ie) field of PSTATE is zero (PSTATE.ie = 0). interrupt_level_\_n interrupts are further conditioned by the Processor Interrupt Level (PIL) register. An interrupt is held pending while either PSTATE.ie = 0 or the condition’s interrupt level is less than or equal to the level specified in PIL. When delivery of this disrupting trap is enabled by PSTATE.ie = 1, it is delivered to the virtual processor in privileged mode if TL < MAXPTL (2, in Oracle SPARC Architecture 2011 implementations).

**Outstanding in Nonprivileged or Privileged mode, destined for delivery in Hyperprivileged mode.** An outstanding disrupting trap condition detected while in either nonprivileged mode or privileged mode and destined for delivery in hyperprivileged mode is never masked; it is delivered immediately.

The above is summarized in TABLE 12-3.

**TABLE 12-3** Conditioning of Disrupting Traps

<table>
<thead>
<tr>
<th>Type of Disrupting Trap Condition</th>
<th>Current Virtual Processor Privilege Mode</th>
<th>Disposition of Disrupting Traps, based on privilege mode in which the trap is destined to be delivered</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt_level__n</td>
<td>Nonprivileged or Privileged</td>
<td>Held pending while PSTATE.ie = 0 or interrupt level ≤ PIL</td>
</tr>
<tr>
<td>All other disrupting traps</td>
<td>Nonprivileged or Privileged</td>
<td>Held pending while PSTATE.ie = 0 or 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 \((PC \leftarrow \text{old } PC, \text{NPC} \leftarrow \text{old NPC})\).
- 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 434 for identification of which traps are precise.
3. In an Oracle SPARC 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 LDBLOCKFD) 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 421 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 432.
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.fcon, and FSR.aexc fields remain unchanged. However, if an IEEE_754_exception does not result in a trap, then the F register, FSR.fcon, 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 427 and TABLE 12-5 on page 430 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>TL&gt;0</th>
<th>TT[TL]</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

FIGURE 12-2 Privileged Mode Trap Vector Address
12.5.2 Privileged Trap Table Organization

The layout of the privileged-mode trap table (which is accessed using virtual addresses) is illustrated in FIGURE 12-3.

![Figure 12-3: Privileged-mode Trap Table Layout](image)

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 $TT[TL]$ 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 424)

$TT$ values $000_{16}$–$07F_{16}$ are reserved for hardware traps. $TT$ values $100_{16}$–$17F_{16}$ are reserved for software traps (caused by execution of a Tcc instruction) to privileged-mode trap handlers.

**IMPL. DEP. #35-V8-Cs20:** $TT$ values $060_{16}$ to $07F_{16}$ were reserved for implementation_dependent_exception_n exceptions in the SPARC V9 specification, but are now all defined as standard Oracle SPARC 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 Oracle SPARC Architecture 2011. 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 Oracle SPARC Architecture 2011, 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-2011</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>IAE_privilege_violation</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>(used at higher privilege levels)</td>
</tr>
<tr>
<td>●</td>
<td>IAE_unauth_access</td>
</tr>
<tr>
<td>●</td>
<td>IAE_nfa_page</td>
</tr>
<tr>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>illegal_instruction</td>
</tr>
<tr>
<td>●</td>
<td>privileged_opcode</td>
</tr>
<tr>
<td>❍</td>
<td>unimplemented_LDTW</td>
</tr>
<tr>
<td>❍</td>
<td>unimplemented_STTW</td>
</tr>
<tr>
<td>●</td>
<td>DAE_invalid_asi</td>
</tr>
<tr>
<td>❍</td>
<td>DAE_privilege_violation</td>
</tr>
<tr>
<td>●</td>
<td>DAE_nfa_page</td>
</tr>
<tr>
<td>●</td>
<td>DAE_nfo_page</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>Reserved</td>
</tr>
<tr>
<td>●</td>
<td>fp_disabled</td>
</tr>
<tr>
<td>❍</td>
<td>fp_exception_ieee_754</td>
</tr>
<tr>
<td>❍</td>
<td>fp_exception_other</td>
</tr>
<tr>
<td>●</td>
<td>tag_overflow&lt;sup&gt;D&lt;/sup&gt;</td>
</tr>
<tr>
<td>●</td>
<td>clean_window</td>
</tr>
<tr>
<td>Exception or Interrupt Request</td>
<td>UA-2011</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>division_by_zero</td>
<td>○=Opt’l</td>
</tr>
<tr>
<td>compatibility_feature</td>
<td></td>
</tr>
<tr>
<td>DAE_side_effect_page</td>
<td>●</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>mem_address_not_aligned</td>
<td>●</td>
</tr>
<tr>
<td>LDDF_mem_address_not_aligned</td>
<td>●</td>
</tr>
<tr>
<td>STDF_mem_address_not_aligned</td>
<td>●</td>
</tr>
<tr>
<td>privileged_action</td>
<td></td>
</tr>
<tr>
<td>LDQF_mem_address_not_aligned</td>
<td>○</td>
</tr>
<tr>
<td>STQF_mem_address_not_aligned</td>
<td>○</td>
</tr>
<tr>
<td>STTW_exception</td>
<td>○</td>
</tr>
<tr>
<td>BLD_exception</td>
<td>○</td>
</tr>
<tr>
<td>BST_exception</td>
<td>○</td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>○</td>
</tr>
<tr>
<td>interrupt_level_n (n = 1–15)</td>
<td>●</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>●</td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>○</td>
</tr>
<tr>
<td>VA_watchpoint</td>
<td>○</td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>●</td>
</tr>
<tr>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
<td>○</td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>●</td>
</tr>
<tr>
<td>implementation_dependent_exception_n (impl. dep. #35-V8-Cs20)</td>
<td>○</td>
</tr>
<tr>
<td>control_transfer_instruction</td>
<td>○</td>
</tr>
<tr>
<td>instruction_VA_watchpoint</td>
<td>●</td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>○</td>
</tr>
<tr>
<td>(used at higher privilege levels)</td>
<td>●</td>
</tr>
</tbody>
</table>
Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36-V8 on page 432), including relative priorities within a given priority level.

† The trap vector entry (32 bytes) for this trap type plus the next three trap types (total of 128 bytes) are permanently reserved for this exception.

‡ This exception is deprecated, because the only instructions that can generate it have been deprecated.
<table>
<thead>
<tr>
<th>Exception or Interrupt Request</th>
<th>TT (Trap Type)</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>instruction VA_watchpoint</td>
<td>075_{16}</td>
<td>2.05</td>
<td>P (nm)</td>
</tr>
<tr>
<td>IAE_privilege_violation</td>
<td>008_{16}</td>
<td>3.1</td>
<td>H -x-</td>
</tr>
<tr>
<td>IAE_unauth_access</td>
<td>00B_{16}</td>
<td>3.2</td>
<td>H H</td>
</tr>
<tr>
<td>IAE_nfo_page</td>
<td>00C_{16}</td>
<td>3.3</td>
<td>H H</td>
</tr>
<tr>
<td>illegal_instruction</td>
<td>010_{16}</td>
<td>6.2</td>
<td>H H</td>
</tr>
<tr>
<td>unimplemented_LDTW</td>
<td>012_{16}</td>
<td></td>
<td>H H</td>
</tr>
<tr>
<td>unimplemented_STTW</td>
<td>013_{16}</td>
<td>6.3</td>
<td>H H</td>
</tr>
<tr>
<td>compatibility_feature</td>
<td>02F_{16}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>privileged_opcode</td>
<td>011_{16}</td>
<td>7</td>
<td>P (nm)</td>
</tr>
<tr>
<td>fp_disabled</td>
<td>020_{16}</td>
<td>8</td>
<td>P P</td>
</tr>
<tr>
<td>spill_n_normal ((n = 0–7))</td>
<td>080{16}–09F_{16}</td>
<td>9</td>
<td>P P</td>
</tr>
<tr>
<td>spill_n_other ((n = 0–7))</td>
<td>0A0{16}–0BF_{16}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fill_n_normal ((n = 0–7))</td>
<td>0C0{16}–0DF_{16}</td>
<td>9</td>
<td>P P</td>
</tr>
<tr>
<td>fill_n_other ((n = 0–7))</td>
<td>0E0{16}–0FF_{16}</td>
<td>10.1</td>
<td></td>
</tr>
<tr>
<td>clean_window</td>
<td>024{16}–027_{16}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDDF_mem_address_not_aligned</td>
<td>035_{16}</td>
<td>10.1</td>
<td></td>
</tr>
<tr>
<td>STDF_mem_address_not_aligned</td>
<td>036_{16}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDQF_mem_address_not_aligned</td>
<td>038_{16}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STQF_mem_address_not_aligned</td>
<td>039_{16}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mem_address_not_aligned</td>
<td>034_{16}</td>
<td>10.2</td>
<td>H H</td>
</tr>
<tr>
<td>fp_exception_other</td>
<td>022_{16}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp_exception_ieee_754</td>
<td>021_{16}</td>
<td>11.1</td>
<td></td>
</tr>
<tr>
<td>privileged_action</td>
<td>037_{16}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>control_transfer_instruction</td>
<td>074_{16}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 12-5  Exception and Interrupt Requests, by Priority  (2 of 2)

<table>
<thead>
<tr>
<th>UA-2011</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>VA_watchpoint</td>
<td>062_{16}</td>
<td>precise</td>
<td>11.2</td>
<td>P (nm)  P (nm)</td>
</tr>
<tr>
<td>●</td>
<td>DAE_invalid_asi</td>
<td>014_{16}</td>
<td>precise</td>
<td>11.3</td>
<td>H  H</td>
</tr>
<tr>
<td>●</td>
<td>DAE_privilege_violation</td>
<td>015_{16}</td>
<td>precise</td>
<td>12.04</td>
<td>H  H</td>
</tr>
<tr>
<td>●</td>
<td>DAE_nc_page</td>
<td>016_{16}</td>
<td>precise</td>
<td>12.05</td>
<td>H  H</td>
</tr>
<tr>
<td>●</td>
<td>DAE_nfo_page</td>
<td>017_{16}</td>
<td>precise</td>
<td>12.06</td>
<td>H  H</td>
</tr>
<tr>
<td>●</td>
<td>DAE_side_effect_page</td>
<td>030_{16}</td>
<td>precise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>○</td>
<td>STTW_exception</td>
<td>03A_{16}</td>
<td>precise</td>
<td></td>
<td>H  H</td>
</tr>
<tr>
<td>○</td>
<td>BLD_exception</td>
<td>03C_{16}</td>
<td>precise</td>
<td>12.13</td>
<td>H  H</td>
</tr>
<tr>
<td>○</td>
<td>BST_exception</td>
<td>03D_{16}</td>
<td>precise</td>
<td></td>
<td>H  H</td>
</tr>
<tr>
<td>●</td>
<td>tag_overflow(^{D})</td>
<td>023_{16}</td>
<td>precise</td>
<td>14</td>
<td>P (nm)  P (nm)</td>
</tr>
<tr>
<td>●</td>
<td>division_by_zero</td>
<td>028_{16}</td>
<td>precise</td>
<td>15</td>
<td>P (nm)  P (nm)</td>
</tr>
<tr>
<td>✷</td>
<td>trap_instruction</td>
<td>100_{16}-17F_{16}</td>
<td>precise</td>
<td>16.03</td>
<td>-x-</td>
</tr>
<tr>
<td>●</td>
<td>htrap_instruction</td>
<td>180_{16}-1FF_{16}</td>
<td>precise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>●</td>
<td>cpu_mondo</td>
<td>07C_{16}</td>
<td>disrupting</td>
<td>16.08</td>
<td>P (ie)  P (ie)</td>
</tr>
<tr>
<td>●</td>
<td>dev_mondo</td>
<td>07D_{16}</td>
<td>disrupting</td>
<td>16.11</td>
<td>P (ie)  P (ie)</td>
</tr>
<tr>
<td>●</td>
<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>●</td>
<td>resumable_error</td>
<td>07E_{16}</td>
<td>disrupting</td>
<td>33.3</td>
<td>P (ie)  P (ie)</td>
</tr>
<tr>
<td>—</td>
<td>nonresumable_error</td>
<td>07F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

\(^{A}\) Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36-V8 on page 432), 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.

\(^{D}\) 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 Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:6 spill_or_fill</td>
<td>0102 for spill traps; 0112 for fill traps</td>
</tr>
<tr>
<td>5 other</td>
<td>(OTHERWIN ≠ 0)</td>
</tr>
<tr>
<td>4:2 wtype</td>
<td>If (other) then WSTATE.other; else WSTATE.normal</td>
</tr>
</tbody>
</table>

FIGURE 12-4 Trap Type Encoding for Spill/Fill Traps

12.5.4 Trap Priorities

TABLE 12-4 on page 427 and TABLE 12-5 on page 430 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 Oracle SPARC 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>
<thead>
<tr>
<th>Original State</th>
<th>New State, After Receiving Trap or Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute_state</td>
<td>execute_state</td>
</tr>
<tr>
<td>TL &lt; MAXPTL − 1</td>
<td>TL ← TL + 1</td>
</tr>
</tbody>
</table>

### 12.6.1 Normal Trap Processing

A trap is delivered in either privileged mode or hyperprivileged mode, depending on the type of trap, the trap level (TL), and the privilege mode in effect when the exception was detected.

During normal trap processing, the following state changes occur (conceptually, in this order):

- The trap level is updated. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack.
  
  ```
  TL ← TL + 1
  ```

- Existing state is preserved.
  
  ```
  TSTATE[TL].gl ← GL
  TSTATE[TL].ccr ← CCR
  TSTATE[TL].asi ← ASI
  TSTATE[TL].pstate ← PSTATE
  TSTATE[TL].cwp ← CWP
  TPC[TL] ← PC // (upper 32 bits zeroed if PSTATE.am = 1)
  TNPC[TL] ← NPC // (upper 32 bits zeroed if PSTATE.am = 1)
  ```

- The trap type is preserved.
  
  ```
  TT[TL] ← the trap type
  ```

- The Global Level register (GL) is updated. This normally provides access to a fresh set of global registers:
  
  ```
  GL ← min (GL + 1, MAXPGL)
  ```

- The PSTATE register is updated to a predefined state:
  
  ```
  PSTATE.mm is unchanged
  PSTATE.pef ← 1 // if an FPU is present, it is enabled
  PSTATE.am ← 0 // address masking is turned off
  PSTATE.priv ← 1 // the virtual processor enters privileged mode
  PSTATE.cle ← PSTATE.tle // set endian mode for traps
  ```

- For a register-window trap (clean_window, window spill, or window fill) only, CWP is set to point to the register window that must be accessed by the trap-handler software, that is:
  
  ```
  if TT[TL] = 02416 // a clean_window trap
  then CWP ← (CWP + 1) mod N_REG_WINDOWS;
  ```

  ```
  if (08016 ≤ TT[TL] ≤ 0BF16) // window spill trap
  then CWP ← (CWP + CANSAVE + 2) mod N_REG_WINDOWS;
  ```

  ```
  if (0C016 ≤ TT[TL] ≤ 0FF16) // window fill trap
  then CWP ← (CWP − 1) mod N_REG_WINDOWS;
  ```

  For non-register-window traps, CWP is not changed.
Control is transferred into the trap table:

```c
// Note that at this point, TL has already been incremented (above)
if ( (trap is to privileged mode) and (TL ≤ MAXPTL) )
then
    // the trap is handled in privileged mode
    // Note: The expression “(TL > 1)” below evaluates to the
    // value 02 if TL was 0 just before the trap (in which
    // case, TL = 1 now, since it was incremented above,
    // during trap entry). “(TL > 1)” evaluates to 12 if
    // TL was > 0 before the trap.
    PC ← TBA[63:15] :: (TL > 1) :: TT[TL] :: 0 00002
    NPC ← TBA[63:15] :: (TL > 1) :: TT[TL] :: 0 01002
else
    { trap is handled in hyperprivileged mode }
endif
```
Interrupts are ignored as long as PSTATE.ie = 0.

**Programming Note**
State in TPC[n], TNPC[n], TSTATE[n], and TT[n] is only changed autonomously by the processor when a trap is taken while TL = n – 1; however, software can change any of these values with a WRPR instruction when TL = n.

---

### 12.7 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 Oracle SPARC Architecture differs from that shown in The SPARC Architecture Manual—Version 9. Each trap is marked as precise, deferred, disrupting, or reset. Example exception conditions are included for each exception type. Chapter 7, Instructions, enumerates which traps can be generated by each instruction.

The following traps are generally expected to be supported in all Oracle SPARC Architecture 2011 implementations. A given trap is not required to be supported in an implementation in which the conditions that cause the trap can never occur.

- **clean_window** [TT = 02416–02716] (Precise) — A SAVE instruction discovered that the window about to be used contains data from another address space; the window must be cleaned before it can be used.

  **IMPL. DEP. #102-V9:** An implementation may choose either to implement automatic cleaning of register windows in hardware or to generate a clean_window trap, when needed, so that window(s) can be cleaned by software. If an implementation chooses the latter option, then support for this trap type is mandatory.

- **control_transfer_instruction** [TT = 07416] (Precise) — This exception is generated if PSTATE.tct = 1 and the processor determines that a successful control transfer will occur as a result of execution of that instruction. If such a transfer will occur, the processor generates a control_transfer_instruction precise trap (trap type = 7416) instead of completing the control transfer. The pc stored in TPC[TL] is the address of the CTI, and the TNPC[TL] is set to the value of NPC before the CTI is executed. (impl. dep. #450-S20). PSTATE.tct is always set to 0 as part of
normal entry into a trap handler. When this exception occurs in nonprivileged or privileged mode, the trap is delivered in privileged mode. If it occurs in hyperprivileged mode, the trap is delivered in hyperprivileged mode.

- **cpu_mondo** [TT = 07C₁₆] (Disrupting) — This interrupt is generated when another virtual processor has enqueued a message for this virtual processor. It is used to deliver a trap in privileged mode, to inform privileged software that an interrupt report has been appended to the virtual processor’s CPU mondo queue. A direct message between virtual processors is sent via a CPU mondo interrupt. When the CPU mondo queue contains a valid entry, a cpu_mondo exception is sent to the target virtual processor.

  Programming Note It is possible that an implementation may occasionally cause a cpu_mondo interrupt when the CPU Mondo queue is empty (CPU Mondo Queue Head pointer = CPU Mondo Queue Tail pointer). A guest operating system running in privileged mode should handle this by ignoring any CPU Mondo interrupt with an empty queue.

- **compatibility_feature** [TT = 02F₁₆] (Precise) — This exception is generated if an attempt is made to execute an extended-capability instruction that is currently disabled by the Compatibility Feature Register (CFR).

- **DAE_invalid_asi** [TT = 014₁₆] (Precise) — An attempt was made to execute an invalid combination of instruction and ASI. See the instruction descriptions in Chapter 7 for a detailed list of valid ASIs for each instruction that can access alternate address spaces. The following invalid combinations of instruction, ASI, and virtual address cause a DAE_invalid_asi exception:
  - A load, store, load-store, or PREFETCHA instruction with either an invalid ASI or an invalid virtual address for a valid ASI.
  - A disallowed combination of instruction and ASI (see Block Load and Store ASIs on page 410 and Partial Store ASIs on page 411). This includes the following:
    - an attempt to use a (deprecated) atomic quad load ASI (2₄₁₆, 2C₁₆, 3₄₁₆, or 3C₁₆) with any load alternate opcode other than LDTX A’s (which is shared by LDLDA)
    - an attempt to use a nontranslating ASI value with any load or store alternate instruction other than LDXA, LDDFA, STXA, or STDFA
    - an attempt to read from a write-only ASI-accessible register, or load from a store-only ASI (for example, a block commit store ASI, E₀₁₆ or E₁₁₆)
    - an attempt to write to a read-only ASI-accessible register
  - An access to a noncacheable page (TTE.cp = 0) was attempted by an atomic load-store instruction (CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA), an LDTXA instruction, a LDLOCKF D instruction, or a STPARTIALF instruction.

- **DAE_nc_page** [TT = 016₁₆] (Precise) — An access to a noncacheable page (TTE.cp = 0) was attempted by an atomic load-store instruction (CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA), an LDTXA instruction, a LDLOCKF D instruction, or a STPARTIALF instruction.

- **DAE_nfo_page** [TT = 017₁₆] (Precise) — An attempt was made to access a non-faulting-only page (TTE.nfo = 1) by any type of load, store, load-store, or FLUSH instruction with an ASI other than a nonfaulting ASI (PRIMARY_NO_FAULT[...LITTLE] or SECONDARY_NO_FAULT[...LITTLE]).

- **DAE_privilege_violation** [TT = 015₁₆] (Precise) — A privilege violation occurred, due to an attempt to access a privileged page (TTE.p = 1) by any type of load, store, or load-store instruction when executing in nonprivileged mode (PSTATE.priv = 0). This includes the special case of an access by privileged software using one of the ASI_AS_IF_USER_PRIMARY[...LITTLE] or ASI_AS_IF_USER_SECONDARY[...LITTLE] ASIs.
- **DAE_side_effect_page** \([TT = 030_{16}]\) (Precise) — An attempt was made to access a page which may cause side effects (\(TTE.e = 1\)) by any type of load instruction with nonfaulting ASI.

- **dev_mondo** \([TT = 07D_{16}]\) (Disrupting) — This interrupt causes a trap to be delivered in privileged mode, to inform privileged software that an interrupt report has been appended to its device mondo queue. When a virtual processor has appended a valid entry to a target virtual processor’s device mondo queue, it sends a dev_mondo exception to the target virtual processor. The interrupt report contents are device specific.

---

**Programming Note**

It is possible that an implementation may occasionally cause a dev_mondo interrupt when the Device Mondo queue is empty (Device Mondo Queue Head pointer = Device Mondo Queue Tail pointer). A guest operating system running in privileged mode should handle this by ignoring any Device Mondo interrupt with an empty queue.

---

- **disrupting_performance_event** \([TT = 02C_{16}]\) (Disrupting) — One of the virtual processor’s performance control registers is configured for a disrupting performance event and the associated counter has overflowed.

- **division_by_zero** \([TT = 028_{16}]\) (Precise) — An integer divide instruction attempted to divide by zero.

- **fill_n_normal** \([TT = 0C0_{16}–0DF_{16}]\) (Precise)

- **fill_n_other** \([TT = 0E0_{16}–0FF_{16}]\) (Precise)

  A RESTORE or RETURN instruction has determined that the contents of a register window must be restored from memory.

- **fp_disabled** \([TT = 020_{16}]\) (Precise) — An attempt was made to execute an FPop, a floating-point branch, or a floating-point load/store instruction while an FPU was disabled (\(PSTATE.pef = 0\) or \(FPRS.fef = 0\)).

- **fp_exception_ieee_754** \([TT = 021_{16}]\) (Precise) — An FPop instruction generated an IEEE_754_exception and its corresponding trap enable mask (\(FSR.tem\)) bit was 1. The floating-point exception type, IEEE_754_exception, is encoded in the FSR.ftt, and specific IEEE_754_exception information is encoded in FSR.cexc.

- **fp_exception_other** \([TT = 022_{16}]\) (Precise) — An FPop instruction generated an exception other than an IEEE_754_exception. Example: execution of an FPop requires software assistance to complete. The floating-point exception type is encoded in FSR.ftt.

- **htrap_instruction** \([TT = 180_{16}–1FF_{16}]\) (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 438.

- **IAE_nfo_page** \([TT = 00C_{16}]\) (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a memory page which was marked for access only by nonfaulting loads (\(TTE.nfo = 1\)).

- **IAE_privilege_violation** \([TT = 008_{16}]\) (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a privileged memory page (\(TTE.p = 1\)) while the virtual processor was executing in nonprivileged mode.

- **IAE_unauth_access** \([TT = 00B_{16}]\) (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a memory page which was missing “execute” permission (\(TTE.ep = 0\)).

- **illegal_instruction** \([TT = 010_{16}]\) (Precise) — An attempt was made to execute an ILLTRAP instruction, an instruction with an unimplemented opcode, an instruction with invalid field usage, or an instruction that would result in illegal processor state.

Examples of cases in which illegal_instruction is generated include the following:

- **illegal_instruction**:
  - An instruction encoding does not match any of the opcode map definitions (see Appendix A, Opcode Maps).
An instruction is not implemented in hardware.

A reserved instruction field in Tcc instruction is nonzero.

If a reserved instruction field in an instruction other than Tcc is nonzero, an illegal_instruction exception should be, but is not required to be, generated. (See Reserved Opcodes and Instruction Fields on page 91.)

An illegal value is present in an instruction i field.

An illegal value is present in a field that is explicitly defined for an instruction, such as cc2, cc1, cc0, fcn, impl, rcond, or opf_cc.

Illegal register alignment (such as odd rd value in a doubleword load instruction).

Illegal rd value for LDXFSR, STXFSR, or the deprecated instructions LDIFSR or STIFSR.

ILLTRAP instruction.

DONE or RETRY when TL = 0.

All causes of an illegal_instruction exception are described in individual instruction descriptions in Chapter 7, Instructions.

SPARC V9 Compatibility Note: The instruction_access_exception exception from SPARC V9 has been replaced by more specific exceptions, such as IAE_privilege_violation and IAE_unauth_access.

instruction_VA_watchpoint [TT = 07516] (Precise) — The virtual processor has detected that the Program Counter (PC) matches the VA Watchpoint register, when instruction VA watchpoints are enabled and the PC is being translated from a virtual address to a hardware address. If the PC is not being translated from a virtual address (for example, the PC is being treated as a hardware address), then an instruction_VA_watchpoint exception will not be generated, even if a match is detected between the VA Watchpoint register and the PC.

interrupt_level_n [TT = 04116–04F16] (Disrupting) — SOFTINT[n] was set to 1 or an external interrupt request of level n was presented to the virtual processor and n > PIL.

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 55).

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 406.)

nonresumable_error [TT = 07F16] (Disrupting) — This interrupt indicates that there is a valid entry in the nonresumable error queue. This interrupt is not generated by hardware, but is used by hyperprivileged software to inform privileged software that an error report has been appended to the nonresumable error queue.

privileged_action [TT = 03716] (Precise) — An action defined to be privileged has been attempted while in nonprivileged mode (PSTATE.priv = 0), or an action defined to be hyperprivileged has been attempted while in nonprivileged or privileged mode. Examples:

A data access by nonprivileged software using a restricted (privileged or hyperprivileged) ASI, that is, an ASI in the range 0016 to 7F16 (inclusively)

A data access by nonprivileged or privileged software using a hyperprivileged ASI, that is, an ASI in the range 3016 to 7F16 (inclusively)

Execution by nonprivileged software of an instruction with a privileged operand value

An attempt to read the TICK register by nonprivileged software when nonprivileged access to TICK is disabled.

CHAPTER 12 • Traps 437
• An attempt to execute a nonprivileged instruction with an operand value requiring more
privilege than available in the current privilege mode.

privileged_opcode [TT = 01116] (Precise) — An attempt was made to execute a privileged
instruction while in nonprivileged mode (PSTATE.priv = 0).

resumable_error [TT = 07E16] (Disrupting) — There is a valid entry in the resumable error queue.
This interrupt is used to inform privileged software that an error report has been appended to the
resumable error queue, and the current instruction stream is in a consistent state so that execution
can be resumed after the error is handled.

spill_n_normal [TT = 08016-09F16] (Precise)
spill_n_other [TT = 0A016-0BF16] (Precise)
A SAVE or FLUSHW instruction has determined that the contents of a register window must be
saved to memory.

STDF_mem_address_not_aligned [TT = 03616] (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 = 02316] (Precise) (deprecated [C2]) — A TADDccTV or TSUBccTV instruction
was executed, and either 32-bit arithmetic overflow occurred or at least one of the tag bits of the
operands was nonzero.

trap_instruction [TT = 10016-17F16] (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 = 01216] (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 = 01316] (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 = 06216] (Precise) — The virtual processor has detected an attempt to access
(load from or store to) a virtual address specified by the VA Watchpoint register, while VA
watchpoints are enabled and the address is being translated from a virtual address to a hardware
address. If the load or store address is not being translated from a virtual address (for example, the
address is being treated as a real address), then a VA_watchpoint exception will not be generated
even if a match is detected between the VA Watchpoint register and a load or store address.

12.7.1 SPARC V9 Traps Not Used in Oracle SPARC Architecture 2011

The following traps were optional in the SPARC V9 specification and are not used in Oracle SPARC
Architecture 2011:

implementation_dependent_exception_n [TT = 07716 - 07B16] This range of implementa-
dependent exceptions has been replaced by a set of architecturally-defined exceptions. (impl.dep.
#35-V8-Cs20)

LDQF_mem_address_not_aligned [TT = 03816] (Precise) — An attempt was made to execute an
LDQF instruction and the effective address was word aligned but not quadword aligned. Use of
this exception is implementation dependent (impl. dep. #111-V9-Cs10). A separate trap entry for
this exception supports fast software emulation of the LDQF instruction when the effective address
is word aligned but not quadword aligned. See Load Floating-Point Register on page 233. (impl. dep.
#111)

STQF_mem_address_not_aligned [TT = 03916] (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 325. (impl. dep. #112)
12.8 Register Window Traps

Window traps are used to manage overflow and underflow conditions in the register windows, support clean windows, and implement the FLUSHW instruction.

12.8.1 Window Spill and Fill Traps

A window overflow occurs when a SAVE instruction is executed and the next register window is occupied (CANSAVE = 0). An overflow causes a spill trap that allows privileged software to save the occupied register window in memory, thereby making it available for use.

A window underflow occurs when a RESTORE instruction is executed and the previous register window is not valid (CANRESTORE = 0). An underflow causes a fill trap that allows privileged software to load the registers from memory.

12.8.2 clean_window Trap

The virtual processor provides the clean_window trap so that system software can create a secure environment in which it is guaranteed that data cannot inadvertently leak through register windows from one software program to another.

A clean register window is one in which all of the registers, including uninitialized registers, contain either 0 or data assigned by software executing in the address space to which the window belongs. A clean window cannot contain register values from another process, that is, from software operating in a different address space.

Supervisor software specifies the number of windows that are clean with respect to the current address space in the CLEANWIN register. This number includes register windows that can be restored (the value in the CANRESTORE register) and the register windows following CWP that can be used without cleaning. Therefore, the number of clean windows available to be used by the SAVE instruction is

CLEANWIN – CANRESTORE

The SAVE instruction causes a clean_window exception if this value is 0. This behavior allows supervisor software to clean a register window before it is accessed by a user.

12.8.3 Vectoring of Fill/Spill Traps

To make handling of fill and spill traps efficient, the SPARC V9 architecture provides multiple trap vectors for the fill and spill traps. These trap vectors are determined as follows:

- Supervisor software can mark a set of contiguous register windows as belonging to an address space different from the current one. The count of these register windows is kept in the OTHERWIN register. A separate set of trap vectors (fill_n_other and spill_n_other) is provided for spill and fill traps for these register windows (as opposed to register windows that belong to the current address space).

- Supervisor software can specify the trap vectors for fill and spill traps by presetting the fields in the WSTATE register. This register contains two subfields, each three bits wide. The WSTATE.normal field determines one of eight spill (fill) vectors to be used when the register window to be spilled (filled) belongs to the current address space (OTHERWIN = 0). If the OTHERWIN register is nonzero, the WSTATE.other field selects one of eight fill_n_other (spill_n_other) trap vectors.

See Trap-Table Entry Addresses on page 424, 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 (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 441.
■ **Software Interrupt Register** (*SOFTINT*) on page 442.
■ **Interrupt Queues** on page 442.
■ **Interrupt Traps** on page 444.

### 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$P$ Register (ASRs 20, 21, 22) on page 55 for a detailed description of the SOFTINT register.

13.2.1 Setting the Software Interrupt Register

SOFTINT$[n]$ is set to 1 by executing a WRSOFTINT_SET$P$ instruction (WRasr using ASR 20) with a ‘1’ in bit $n$ of the value written (bit $n$ corresponds to interrupt level $n$). The value written to the SOFTINT_SET register is effectively ORed into the SOFTINT register. This approach allows the interrupt handler to set one or more bits in the SOFTINT register with a single instruction.

See SOFTINT_SET$P$ Pseudo-Register (ASR 20) on page 55 for a detailed description of the SOFTINT_SET pseudo-register.

13.2.2 Clearing the Software Interrupt Register

When all interrupts scheduled for service at level $n$ have been serviced, kernel software executes a WRSOFTINT_CLR$P$ instruction (WRasr using ASR 21) with a ‘1’ in bit $n$ of the value written, to clear interrupt level $n$ (impl. dep. 34-V8a). The complement of the value written to the SOFTINT_CLR register is effectively 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.

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.


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

Programming Notes
- 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 Notes
- The SOFTINT mechanism is independent of the “mondo” interrupt mechanism mentioned in Interrupt Queues on page 442. The two mechanisms do not interact.
New interrupt entries are appended to the tail of a queue and privileged software reads them from the head of the queue.

**Programming Note** | Software conventions for cooperative management of interrupt queues and the format of queue entries are specified in the separate Hypervisor API Specification document.

13.3.1 Interrupt Queue Registers

The active contents of each queue are delineated by a 64-bit head register and a 64-bit tail register.

The interrupt queue registers are accessed through ASI ASI_QUEUE (2516). The ASI and address assignments for the interrupt queue registers are provided in TABLE 13-1.

<table>
<thead>
<tr>
<th>Register</th>
<th>ASI</th>
<th>Virtual Address</th>
<th>Privileged mode Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Mondo Queue Head</td>
<td>2516</td>
<td>3C016</td>
<td>RW</td>
</tr>
<tr>
<td>CPU Mondo Queue Tail</td>
<td>2516</td>
<td>3C816</td>
<td>R or RW†</td>
</tr>
<tr>
<td>Device Mondo Queue Head</td>
<td>2516</td>
<td>3D016</td>
<td>RW</td>
</tr>
<tr>
<td>Device Mondo Queue Tail</td>
<td>2516</td>
<td>3D816</td>
<td>R or RW†</td>
</tr>
<tr>
<td>Resumable Error Queue Head</td>
<td>2516</td>
<td>3E016</td>
<td>RW</td>
</tr>
<tr>
<td>Resumable Error Queue Tail</td>
<td>2516</td>
<td>3E816</td>
<td>R or RW†</td>
</tr>
<tr>
<td>Nonresumable Error Queue Head</td>
<td>2516</td>
<td>3F016</td>
<td>RW</td>
</tr>
<tr>
<td>Nonresumable Error Queue Tail</td>
<td>2516</td>
<td>3F816</td>
<td>R or RW†</td>
</tr>
</tbody>
</table>

† see IMPL. DEP.#422-S10

The status of each queue is reflected by its head and tail registers:

- A Queue Head Register indicates the location of the oldest interrupt packet in the queue.
- A Queue Tail Register indicates the location where the next interrupt packet will be stored.

An event that results in the insertion of a queue entry causes the tail register for that queue to refer to the following entry in the circular queue. Privileged code is responsible for updating the head register appropriately when it removes an entry from the queue.

A queue is *empty* when the contents of its head and tail registers are equal. A queue is *full* when the insertion of one more entry would cause the contents of its head and tail registers to become equal.

---

1. “mondo” is a historical term, referring to the name of the original UltraSPARC 1 bus transaction in which these interrupts were introduced.
13.4 Interrupt Traps

The following interrupt traps are defined in the Oracle SPARC Architecture 2011: cpu_mondo, dev_mondo, resumable_error, and nonresumable_error. See Chapter 12, Traps, for details.

Oracle SPARC Architecture 2011 also supports the interrupt_level_n traps defined in the SPARC V9 specification. How interrupts are delivered is implementation-specific; see the relevant implementation-specific Supplement to this specification for details.

Programming Note

By current convention, the format of a Queue Head or Tail register is as follows:

<table>
<thead>
<tr>
<th>head/tail offset</th>
<th>000000</th>
</tr>
</thead>
</table>

Under this convention:
- updating a Queue Head register involves incrementing it by 64 (size of a queue entry, in bytes)
- Queue Head and Tail registers are updated using modular arithmetic (modulo the size of the circular queue, in bytes)
- bits 5:0 always read as zeros, and attempts to write to them are ignored
- the maximum queue offset for an interrupt queue is implementation dependent
- behavior when a queue register is written with a value larger than the maximum queue offset (queue length minus the length of the last entry) is undefined

This is merely a convention and is subject to change.
CHAPTER 14

Memory Management

An Oracle SPARC Architecture Memory Management Unit (MMU) conforms to the requirements set forth in the SPARC V9 Architecture Manual. In particular, it supports a 64-bit virtual address space, simplified protection encoding, and multiple page sizes.

IMPL. DEP. #451-S20: The width of the virtual address supported is implementation dependent. If fewer than 64 bits are supported, then unsupported bits must have the same value as the most significant supported bit. For example, if the model supports 48 virtual address bits, then bits 63:48 must have the same value as bit 47.

This appendix describes the Memory Management Unit, as observed by privileged software, in these sections:

- Virtual Address Translation on page 445.
- Context ID on page 448.
- TSB Translation Table Entry (TTE) on page 449.
- Translation Storage Buffer (TSB) on page 453.

14.1 Virtual Address Translation

The MMUs may support up to eight page sizes: 8 KBytes, 64 KBytes, 512 KBytes, 4 MBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes. 8 KByte, 64 KByte and 4 MByte page sizes must be supported; the other page sizes are optional.

IMPL. DEP. #310-U4: Which, if any, of the following optional page sizes are supported by the MMU in an Oracle SPARC Architecture 2011 implementation is implementation dependent: 512 KBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes.

An Oracle SPARC Architecture MMU supports a 64-bit virtual address (VA) space.

IMPL. DEP. #452-S20: The number of real address (RA) bits supported is implementation dependent. A minimum of 40 bits and maximum of 56 bits can be provided for real addresses (RA). See implementation-specific documentation for details.

In each translation, the virtual page number is replaced by a physical page number, which is concatenated with the page offset to form the full hardware address, as illustrated in FIGURE 14-1 and FIGURE 14-2.

IMPL. DEP. #453-S20: It is implementation dependent whether there is a unified MMU (UMMU) or a separate IMMU (for instruction accesses) and DMMU (for data accesses). The Oracle SPARC Architecture supports both configurations.
FIGURE 14-1 Virtual-to-Real Address Translation for 8-Kbyte, 64-Kbyte, 512-Kbyte, and 4-Mbyte Page Sizes
Privileged software manages virtual-to-real address translations.

Privileged software maintains translation information in an arbitrary data structure, called the *software translation table*.

The Translation Storage Buffer (TSB) is an array of Translation Table Entries which serves as a cache of the software translation table, used to quickly reload the TLB in the event of a TLB miss.

A conceptual view of privileged-mode memory management the MMU is shown in **FIGURE 14-3**. The software translation table is likely to be large and complex. The translation storage buffer (TSB), which acts like a direct-mapped cache, is the interface between the software translation table and the underlying memory management hardware. The TSB can be shared by all processes running on a virtual processor or can be process specific; the hardware does not require any particular scheme. There can be several TSBs.
14.2 Context ID

The MMU supports three contexts:

- Primary Context
- Secondary Context
- Nucleus Context (which has a fixed Context ID value of zero)

The context used for each access depends on the type of access, the ASI used, the current privilege mode, and the current trap level (TL). Details are provided in the following paragraphs and in TABLE 14-1.

For instruction fetch accesses, in nonprivileged and privileged mode when TL = 0 the Primary Context is used; when TL > 0, the Nucleus Context is used.

For data accesses using implicit ASIs, in nonprivileged and privileged mode when TL = 0 the Primary Context is used; when TL > 0, the Nucleus Context is used.

For data accesses using explicit ASIs:

- In nonprivileged mode the Primary Context is used for the ASI_PRIMARY* ASIs, and the Secondary Context is used for the ASI_SECONDARY* ASIs.
- In privileged mode, the Primary Context is used for the ASI_PRIMARY* and the ASI_AS_IF_USER_PRIMARY* ASIs, the Secondary Context is used for the ASI_SECONDARY* and the ASI_AS_IF_USER_SECONDARY* ASIs, and the Nucleus Context is used for ASI_NUCLEUS* ASIs.
The above paragraphs are summarized in TABLE 14-1.

### TABLE 14-1  Context Usage

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Privilege Mode</th>
<th>Primary Context</th>
<th>Secondary Context</th>
<th>Nucleus Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Nonprivileged</td>
<td>(when TL = 0)</td>
<td>— †</td>
<td>(when TL &gt; 0)</td>
</tr>
<tr>
<td></td>
<td>Privileged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data access</td>
<td>Nonprivileged</td>
<td>(when TL = 0)</td>
<td>— †</td>
<td>(when TL &gt; 0)</td>
</tr>
<tr>
<td></td>
<td>Privileged</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASI</td>
<td>Nonprivileged</td>
<td>ASI_PRIMARY*</td>
<td>ASI_SECONDARY*</td>
<td>†</td>
</tr>
<tr>
<td></td>
<td>Privileged</td>
<td>ASI_PRIMARY*</td>
<td>ASI_SECONDARY*</td>
<td>ASI_NUCLEUS*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASI_AS_IF_USER_PRIMARY*</td>
<td>ASI_AS_IF_USER_SECONDARY*</td>
<td></td>
</tr>
</tbody>
</table>

Note: The Oracle SPARC Architecture provides the capability of private and shared contexts. Multiple primary and secondary context IDs, which allow different processes to share TTEs, are defined. See Context ID Registers on page 457 for details.

Programming Note: Privileged software (operating systems) intended to be portable across all Oracle SPARC Architecture implementations should always ensure that, for memory accesses made in privileged mode, private and shared context IDs are set to the same value. The exception to this is privileged-mode accesses using the ASI_AS_IF_USER* ASIs, which remain portable even if the private and shared context IDs differ.

IMPL. DEP. #: The Oracle SPARC Architecture defines a 16-bit context ID. The size of the context ID field is implementation dependent. At least 13 bits must be implemented. If fewer than 16 bits are supported, the unused high order bits are ignored on writes to the context ID, and read as zeros.

### 14.3 TSB Translation Table Entry (TTE)

Each Translation Table Entry (TTE) in a Translation Storage Buffer (TSB) is the equivalent of a page table entry as defined in the Sun4v Architecture Specification; it holds information for a single page mapping. The TTE is divided into two 64-bit words representing the tag and data of the translation. Just as in a hardware cache, the tag is used to determine whether there is a hit in the TSB; if there is a hit, the data are used by either the hardware tablewalker or privileged software.

The TTE configuration is illustrated in FIGURE 14-4 and described in TABLE 14-2.
TABLE 14-2

<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>
</tbody>
</table>

| Data – 63 | v | Valid. If v = 1, then the remaining fields of the TTE are meaningful, and the TTE can be used; otherwise, the TTE cannot be used to translate a virtual address. |

**Programming Note**

The explicit Valid bit is (intentionally) redundant with the software convention of encoding an invalid TTE with an unused context ID. The encoding of the context_id field is necessary to cause a failure in the TTE tag comparison, while the explicit Valid bit in the TTE data simplifies the TTE miss handler.

| Data – 62 | nfo | No Fault Only. If nfo = 1, loads with ASI_PRIMARY_NO_FAULT({LITTLE}) or ASI_SECONDARY_NO_FAULT({LITTLE}) are translated. Any other data access with the D/UMMU TTE.nfo = 1 will trap with a DAE_nfo_page exception. An instruction fetch access to a page with the IMMU TTE.nfo = 1 results in an IAE_nfo_page exception. |

| Data – 61:56 | soft2 | Software-defined field, provided for use by the operating system. The soft2 field can be written with any value in the TSB. Hardware is not required to maintain this field in any TLB (or uTLB), so when it is read from the TLB (uTLB), it may read as zero. |

| Data – 55:13 | taddr | Target address; the underlying address (Real Address [55:13]) to which the MMU will map the page. |

**IMPL. DEP. # 238-U3:** When page offset bits for larger page sizes are stored in the 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.
Invert Endianness. If \( ie = 1 \) for a page, accesses to the page are processed with inverse endianness from that specified by the instruction (big for little, little for big). See page 454 for details.

**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.

**Programming Notes**

1. The primary purpose of this bit is to aid in the mapping of I/O devices (through noncacheable memory addresses) whose registers contain and expect data in little-endian format. Setting TTE\( .ie = 1 \) allows those registers to be accessed correctly by big-endian programs using ordinary loads and stores, such as those typically issued by compilers; otherwise little-endian loads and stores would have been issued by hand-written assembler code.

2. This bit can also be used when mapping cacheable memory. However, cacheable accesses to pages marked with TTE\( .ie = 1 \) may be slower than accesses to the page with TTE\( .ie = 0 \). For example, an access to a cacheable page with TTE\( .ie = 1 \) may perform as if there was a miss in the first-level data cache.

**Implementation Note**

Some implementations may require cacheable accesses to pages tagged with TTE\( .ie = 1 \) to bypass the data cache, adding latency to those accesses.

**Data – 11 \( \theta \)**

Side effect. If the side-effect bit is set to 1, loads with \( \text{ASI-primary-no-fault} \), \( \text{ASI-secondary-no-fault} \), and their \( \_\text{little} \) variations will trap for addresses within the page, noncacheable memory accesses other than block loads and stores are strongly ordered against other \( \theta \)-bit accesses, and noncacheable stores are not merged. This bit should be set to 1 for pages that map I/O devices having side effects. Note, also, that the \( \theta \) bit causes the prefetch instruction to be treated as a nop, but does not prevent normal (hardware) instruction prefetching.

**Note 1:** The \( \theta \) bit does not force a noncacheable access. It is expected, but not required, that the \( cp \) and \( cv \) bits will be set to 0 when the \( \theta \) bit is set to 1. If both the \( cp \) and \( cv \) bits are set to 1 along with the \( \theta \) bit, the result is undefined.

**Note 2:** The \( \theta \) bit and the \( nfo \) bit are mutually exclusive; both bits should never be set to 1 in any TTE.
The cacheable-in-physically-indexed-cache bit and cacheable-in-virtually-indexed-cache bit determine the cacheability of the page. Given an implementation with a physically indexed instruction cache, a virtually indexed data cache, and a physically indexed unified second-level cache, the following table illustrates how the \( cp \) and \( cv \) bits could be used:

<table>
<thead>
<tr>
<th>Cacheable (( cp:cv ))</th>
<th>Meaning of TTE when placed in:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I-TLB (Instruction Cache PA-indexed)</td>
</tr>
<tr>
<td>00, 01</td>
<td>Noncacheable</td>
</tr>
<tr>
<td>10</td>
<td>Cacheable L2-cache, I-cache</td>
</tr>
<tr>
<td>11</td>
<td>Cacheable L2-cache, I-cache</td>
</tr>
</tbody>
</table>

The MMU does not operate on the cacheable bits but merely passes them through to the cache subsystem. The \( cv \) bit in the IMMU is read as zero and ignored when written.

**IMPL. DEP. #226-U3:** Whether the \( cv \) bit is supported in hardware is implementation dependent in the Oracle SPARC 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 (\( \text{PSTATE}_{.\text{priv}} = 0 \)), then the MMU signals an \( \text{IAE\_privilege\_violation} \) exception or \( \text{DAE\_privilege\_violation} \) 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 \( \text{IAE\_unauth\_access} \) exception.

**IMPL. DEP. #___:** An Oracle SPARC Architecture ITLB implementation may elect to not implement the \( ep \) bit, and instead present the \( \text{IAE\_unauth\_access} \) exception if there is an attempt to load an ITLB entry with \( ep = 0 \) during a hardware tablewalk. In this case, the MMU miss trap handler software must also detect the \( ep = 0 \) case when the IMMU miss is handled by software.

- **Data – 6 \( w \):** Writable. If \( w = 1 \), the page mapped by this TTE has write permission granted. Otherwise, write permission is not granted.

- **Data – 5:4 \( \text{soft} \):** Software-defined field, provided for use by the operating system. The \( \text{soft} \) 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.
14.4 Translation Storage Buffer (TSB)

The Translation Storage Buffer (TSB) is an array of Translation Table Entries managed entirely by privileged software. It serves as a cache of the software translation table, used to quickly reload the hardware translation table (TLB) in the event of a TLB miss.

14.4.1 TSB Cacheability and Consistency

The TSB exists as a data structure in memory and therefore can be cached. Indeed, the speed of the TLB miss handler relies on the TSB accesses hitting the level-2 cache at a substantial rate. This policy may result in some conflicts with normal instruction and data accesses, but the dynamic sharing of the level-2 cache resource will provide a better overall solution than that provided by a fixed partitioning.

Programming Note | When software updates the TSB, it is responsible for ensuring that the store(s) used to perform the update are made visible in the memory system (for access by subsequent loads, stores, and load-stores) by use of an appropriate MEMBAR instruction. Making a TSB update visible to fetches of instructions subsequent to the store(s) that updated the TSB may require execution of instructions such as FLUSH, DONE, or RETRY, in addition to the MEMBAR.

14.4.2 TSB Organization

The TSB is arranged as a direct-mapped cache of TTEs.

In each case, \( n \) least significant bits of the respective virtual page number are used as the offset from the TSB base address, with \( n \) equal to log base 2 of the number of TTEs in the TSB.

The TSB organization is illustrated in FIGURE 14-5. The constant \( n \) can range from 512 to an implementation-dependent number.

FIGURE 14-5 TSB Organization
The maximum number of entries in a TSB is implementation-dependent in the Oracle SPARC Architecture (to a maximum of 16 million).

14.5 ASI Value, Context ID, and Endianness Selection for Translation

The selection of the context ID for a translation is the result of a two-step process:

1. The ASI is determined (conceptually by the Integer Unit) from the instruction, ASI register, trap level, the virtual processor endian mode (PSTATE.cle), and the privilege level (PSTATE.priv).

2. The context ID is determined directly from the ASI. The context ID value is read by the context ID selected by the ASI.

The ASI value and endianness (little or big) are determined, according to TABLE 14-3 through TABLE 14-4.

When using the Primary Context ID, the values stored in the Primary Context IDs are used by the Data (or Unified) MMU. The Secondary Context ID is never used for instruction accesses.

The endianness of a data access is specified by three conditions:

- The ASI specified in the opcode or ASI register
- The PSTATE current little-endian bit (cle)
- The TTE “invert endianness” bit (ie). The TTE bit inverts the endianness that is otherwise specified for the access.

**Note:** The D/UMMU ie bit inverts the endianness for all accesses, including alternate space loads, stores, and atomic load-stores that specify an ASI. For example,

```
ldxa [%g1]#ASI_PRIMARY_LITTLE
```

will be big-endian if the ie bit = 1.

Accesses to ASIs which are not translated by the MMU (nontranslating ASIs) are not affected by the TTE.ie bit.

<table>
<thead>
<tr>
<th>TABLE 14-3</th>
<th>ASI Mapping for Instruction Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mode</strong></td>
<td><strong>TL</strong></td>
</tr>
<tr>
<td>Nonprivileged</td>
<td>0</td>
</tr>
<tr>
<td>Privileged</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1–2</td>
</tr>
</tbody>
</table>
The Context ID used by the data and instruction MMUs is determined according to TABLE 14-5. The Context ID selection is not affected by the endianness of the access. For a comprehensive list of ASI values in the ASI map, see Chapter 10, Address Space Identifiers (ASIs).

### TABLE 14-4  ASI Mapping for Data Accesses

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Privilege Mode</th>
<th>TL1/2</th>
<th>PSTATE .cle</th>
<th>TTE .iei</th>
<th>Endianness</th>
<th>ASI Used</th>
<th>Resulting Address Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load, Store, Atomic Load-Store, or Prefetch with implicit ASI</td>
<td>NP</td>
<td>01</td>
<td>0</td>
<td>Big1</td>
<td>Little1</td>
<td>ASI_PRIMARY</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td>NP</td>
<td>01</td>
<td>1</td>
<td>Little1</td>
<td>0</td>
<td>Big1</td>
<td>ASI_PRIMARY_LITTLE</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>0</td>
<td>1</td>
<td>Little1</td>
<td>0</td>
<td>Big1</td>
<td>ASI_PRIMARY</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>0</td>
<td>1</td>
<td>Little1</td>
<td>1</td>
<td>Big1</td>
<td>ASI_NUCLEUS</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>1-21</td>
<td>0</td>
<td>Big1</td>
<td>Little1</td>
<td>ASI_NUCLEUS_LITTLE</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>0-21</td>
<td>any</td>
<td>Big1</td>
<td>Little1</td>
<td>Explicitly specified in instruction</td>
<td>VA</td>
</tr>
<tr>
<td>Load, Store, Atomic Load-Store, or Prefetch alternate with ASI name <em>not</em> ending in _LITTLE</td>
<td>NP</td>
<td>01</td>
<td>any</td>
<td>Big2</td>
<td>Little2</td>
<td>Explicitly specified in instruction RA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>0-21</td>
<td>any</td>
<td>Big1</td>
<td>Little1</td>
<td>ASI_<em>REAL</em> ASI</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>0-21</td>
<td>any</td>
<td>Big</td>
<td>Little</td>
<td>Nontranslating ASIs</td>
<td>—</td>
</tr>
<tr>
<td>Load, Store, Atomic Load-Store, or Prefetch alternate with ASI name ending in _LITTLE</td>
<td>NP</td>
<td>01</td>
<td>any</td>
<td>Little1</td>
<td>Big1</td>
<td>Explicitly specified in instruction RA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>0-21</td>
<td>any</td>
<td>Little1</td>
<td>Big1</td>
<td>Explicitly specified in instruction RA</td>
<td></td>
</tr>
</tbody>
</table>

1. MAXPTL = 2 for Oracle SPARC Architecture 2011 processors. Privilege mode operation is valid only for TL = 0, 1 or 2. Nonprivileged mode operation is valid only for TL = 0. See section 5.6.7 for details.
2. Accesses to nontranslating ASIs are always made in big endian mode, regardless of the setting of TTE .iei. See ASI Values on page 397 for information about nontranslating ASIs.

### TABLE 14-5  IMMU, DMMU and UMMU Context ID Usage

<table>
<thead>
<tr>
<th>ASI Value</th>
<th>Context ID Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASI_<em>NUCLEUS</em> (any ASI name containing the string &quot;NUCLEUS&quot;)</td>
<td>Nucleus (0000&lt;sub&gt;16&lt;/sub&gt; hard-wired)</td>
</tr>
</tbody>
</table>
14.6 SPARC V9 “MMU Attributes”

The Oracle SPARC Architecture MMU complies completely with the SPARC V9 “MMU Attributes” as described in Appendix F.3.2.

With regard to Read, Write and Execute Permissions, SPARC V9 says “An MMU may allow zero or more of read, write and execute permissions, on a per-mapping basis. Read permission is necessary for data read accesses and atomic accesses. Write permission is necessary for data write accesses and atomic accesses. Execute permission is necessary for instruction accesses. At a minimum, an MMU must allow for ‘all permissions’, ‘no permissions’, and ‘no write permission’; optionally, it can provide ‘execute only’ and ‘write only’, or any combination of ‘read/write/execute’ permissions.”

TABLE 14-6 shows how various protection modes can be achieved, if necessary, through the presence or absence of a translation in the instruction or data MMU. Note that this behavior requires specialized TLB-miss handler code to guarantee these conditions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Resultant Protection Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes No Yes 0 0</td>
<td>Read-only1</td>
</tr>
<tr>
<td>No Yes N/A 1 N/A</td>
<td>Execute-only1</td>
</tr>
<tr>
<td>Yes No Yes 0 1</td>
<td>Read/Write1</td>
</tr>
<tr>
<td>Yes Yes Yes 1 0</td>
<td>Read-only/Execute</td>
</tr>
<tr>
<td>Yes Yes Yes 1 1</td>
<td>Read/Write/Execute</td>
</tr>
<tr>
<td>No No No N/A N/A</td>
<td>No Access</td>
</tr>
</tbody>
</table>

1. These protection modes are optional, according to SPARC V9.

14.6.1 Accessing MMU Registers

All internal MMU registers can be accessed directly by the virtual processor through defined ASIs, using LDXA and STXA instructions. Oracle SPARC Architecture-compatible processors do not require a MEMBAR #$sync, FLUSH, DONE, or RETRY instruction after a store to an MMU register for proper operation.
TABLE 14-7 lists the MMU registers and provides references to sections with more details.

### TABLE 14-7  MMU Internal Registers and ASI Operations

<table>
<thead>
<tr>
<th>IMMU ASI</th>
<th>D/UMMU ASI</th>
<th>VA(63:0)</th>
<th>Access</th>
<th>Register or Operation Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2116</td>
<td>816</td>
<td>RW</td>
<td>Primary Context ID 0 register</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>2116</td>
<td>1016</td>
<td>RW</td>
<td>Secondary Context ID 0 register</td>
</tr>
<tr>
<td>2116</td>
<td>10816</td>
<td>RW</td>
<td>Primary Context ID 1 register</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>2116</td>
<td>11016</td>
<td>RW</td>
<td>Secondary Context ID 1 register</td>
</tr>
</tbody>
</table>

### 14.6.2 Context ID Registers

The MMU architecture supports multiple primary and secondary context IDs. The address assignment of the context IDs is shown in TABLE 14-8.

### TABLE 14-8  Context ID ASI Assignments

<table>
<thead>
<tr>
<th>Register</th>
<th>ASI</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Context ID 0</td>
<td>2116</td>
<td>00816</td>
</tr>
<tr>
<td>Primary Context ID 1</td>
<td>2116</td>
<td>10816</td>
</tr>
<tr>
<td>Secondary Context ID 0</td>
<td>2116</td>
<td>01016</td>
</tr>
<tr>
<td>Secondary Context ID 1</td>
<td>2116</td>
<td>11016</td>
</tr>
</tbody>
</table>

**Programming Note**

For platforms that implement more than one primary context ID and one secondary context ID, privileged code must ensure that no more than one page translation is allowed to match at any time. An illustration of erroneous behavior is as follows:

1. An operating system constructs a mapping for virtual address \( A \) valid for context ID \( P \);
2. it then constructs a mapping for address \( A \) for context ID \( Q \).

By setting Primary Context ID 0 to \( P \) and Primary Context ID 1 to \( Q \), both mappings would be active simultaneously, with conflicting translations for address \( A \). Care must be taken not to construct such scenarios.

Oracle SPARC Architecture processors must prevent errors or data corruption due to multiple valid translations for a given virtual address using different contexts. TLBs may need to detect this scenario as a multiple tag hit error and cause an exception for such an access.

The Oracle SPARC Architecture supports up to two primary context IDs and two secondary context IDs, which are shared by the IMMU and D/UMMU. Primary Context ID 0 and Primary Context ID 1 are the primary context IDs, and a TLB entry for a translating primary ASI can match the contextid field with either Primary Context ID 0 or Primary Context ID 1 to produce a TLB hit.

Secondary Context ID 0 and Secondary Context ID 1 are the Secondary Context IDs, and a TLB entry for a translating secondary ASI can match the contextid field with either Secondary Context ID 0 or Secondary Context ID 1 to produce a TLB hit.
The Primary Context ID 0 and Primary Context ID 1 registers are illustrated in FIGURE 14-6, where $pcontext$ is the context ID for the primary address space.

![Primary Context ID](image)

FIGURE 14-6 IMMU, DMMU, and UMMU Primary Context ID 0 and 1

The Secondary Context ID 0 and Secondary Context ID 1 registers are illustrated in FIGURE 14-7, where $scontextid$ is the context ID for the secondary address space.

![Secondary Context ID](image)

FIGURE 14-7 D/UMMU Secondary Context ID 0 and 1

The Nucleus Context ID register is hardwired to zero, as illustrated in FIGURE 14-6.

![Nucleus Context ID](image)

FIGURE 14-8 IMMU, DMMU, and UMMU Nucleus Context ID

**IMPL. DEP. #FIGURE 14-10415-S10:** The size of context ID fields in MMU context registers is implementation-dependent and may range from 13 to 16 bits.
Opcode Maps

This appendix contains the Oracle SPARC Architecture 2011 instruction opcode maps.

In this appendix and in Chapter 7, Instructions, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE 7-1 on page 94. For preferred substitute instructions for deprecated opcodes, see the individual opcodes in Chapter 7 that are labeled “Deprecated”.

In the tables in this appendix, reserved (—) and shaded entries (as defined below) indicate opcodes that are not implemented in Oracle SPARC Architecture 2011 strands.

<table>
<thead>
<tr>
<th>Shading</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An attempt to execute opcode will cause an illegal_instruction exception.</td>
</tr>
</tbody>
</table>

An attempt to execute a reserved opcode behaves as defined in Reserved Opcodes and Instruction Fields on page 91.

### TABLE A-1  op[1:0]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branches and SETHI (See TABLE A-2)</td>
<td>CALL</td>
<td>Arithmetic &amp; Miscellaneous (See TABLE A-3)</td>
<td>Loads/Stores (See TABLE A-4)</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE A-2  op2[2:0] (op = 0)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILLTRAP (bits 29:25 = 0)</td>
<td>BPce (See TABLE A-7)</td>
<td>Biccc (See TABLE A-7)</td>
<td>BPr (bit 28 = 0) (See TABLE A-8)</td>
<td>SETHI, NOP1</td>
<td>FBp (See TABLE A-7)</td>
<td>FBfcd (See TABLE A-7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>— (bits 29:25 ≠ 0)</td>
<td>CBcond (bit 28 = 1)1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. rd = 0, imm22 = 0
### TABLE A-3  \( \text{op3}[5:0] \) (\( \text{op} = 10_2 \))  (1 of 2)

<table>
<thead>
<tr>
<th>( \text{op3}[5:0] )</th>
<th>( \text{op3}{3:0} )</th>
<th>( \text{op3}{5:4} )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ADDcc</td>
<td>TADDcc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>ANDcc</td>
<td>TSUBcc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>ORcc</td>
<td>TADDccTV(^{14} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>XORcc</td>
<td>TSUBccTV(^{14} )</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>SUBcc</td>
<td>MUL(^{17} )</td>
</tr>
<tr>
<td><strong>5</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANDN</td>
<td>ANDNcc</td>
<td>SLL (x = 0), SLLX (x = 1)</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORN</td>
<td>ORNcc</td>
<td>SRL (x = 0), SRLX (x = 1)</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XNOR</td>
<td>XNORcc</td>
<td>SRA (x = 0), SRAX (x = 1)</td>
</tr>
<tr>
<td>op3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-----</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>ADDC</td>
<td>ADDCc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>9</strong></td>
<td>MULX</td>
<td>—</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>UMUL</td>
<td>UMULc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>SMUL</td>
<td>SMULc</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>SUBC</td>
<td>SUBCc</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>UDIV</td>
<td>UDIVc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>SDIV</td>
<td>SDIVc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op3 (3:0)</td>
<td>op3 (5:4)</td>
<td>0</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>0</td>
<td>LDUW</td>
<td>LDUWA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LDUB</td>
<td>LDUBA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LDUH</td>
<td>LDUHA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LDTWD</td>
<td>LDTWA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>STW</td>
<td>STWA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>STB</td>
<td>STBA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>STH</td>
<td>STA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>STTWD</td>
<td>STTWA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>LDSW</td>
<td>LDSWA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>LDSB</td>
<td>LDSBA</td>
</tr>
<tr>
<td>A</td>
<td>LDSH</td>
<td>LDSHA</td>
</tr>
<tr>
<td>B</td>
<td>LDX</td>
<td>LDXA</td>
</tr>
<tr>
<td>C</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>D</td>
<td>LDSTUB</td>
<td>LDSTUBA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>STX</td>
<td>STXA</td>
</tr>
<tr>
<td>F</td>
<td>SWAPD</td>
<td>SWAPA</td>
</tr>
</tbody>
</table>
### TABLE A-5  \( \text{opf}(8:0) \) (\( \text{op} = 10_2, \text{op} 3 = 34_{16} = \text{FPop1} \))

<table>
<thead>
<tr>
<th>\text{opf}(8:4) \</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>00_{16}</td>
<td>—</td>
<td>FMOVs</td>
<td>FMOVd</td>
<td>FMOVq</td>
<td>—</td>
<td>FNEGs</td>
<td>FNEGd</td>
<td>FNEGq</td>
</tr>
<tr>
<td>01_{16}</td>
<td>—</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>
<td>—</td>
</tr>
<tr>
<td>03_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>04_{16}</td>
<td>—</td>
<td>FADDS</td>
<td>FADDD</td>
<td>FADDD</td>
<td>—</td>
<td>FSUBs</td>
<td>FSUBd</td>
<td>FSUBq</td>
</tr>
<tr>
<td>05_{16}</td>
<td>—</td>
<td>FNADDS</td>
<td>FNADDD</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>06_{16}</td>
<td>—</td>
<td>FHADDS</td>
<td>FHADDD</td>
<td>—</td>
<td>—</td>
<td>FHSUBs</td>
<td>FHSUBd</td>
<td>—</td>
</tr>
<tr>
<td>07_{16}</td>
<td>—</td>
<td>FNHADDS</td>
<td>FNHADDD</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>08_{16}</td>
<td>—</td>
<td>FsTOx</td>
<td>FdTOx</td>
<td>FqTOx</td>
<td>FxTOs</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>
<td>—</td>
</tr>
<tr>
<td>0A_{16}</td>
<td>—</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>
<td>—</td>
</tr>
<tr>
<td>0C_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>FiTOs</td>
<td>FdTOs</td>
<td>FqTOs</td>
<td>—</td>
</tr>
<tr>
<td>0D_{16}</td>
<td>—</td>
<td>FsTOi</td>
<td>FdTOi</td>
<td>FqTOi</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0E_{16}–1F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00_{16}</td>
<td>—</td>
<td>FABSs</td>
<td>FABSD</td>
<td>FABSq</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>01_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>02_{16}</td>
<td>—</td>
<td>FSQRTs</td>
<td>FSQRTd</td>
<td>FSQRTq</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>03_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>04_{16}</td>
<td>—</td>
<td>FMULs</td>
<td>FMULd</td>
<td>FMULq</td>
<td>—</td>
<td>FDIVs</td>
<td>FDIVd</td>
</tr>
<tr>
<td>05_{16}</td>
<td>—</td>
<td>FNMULs</td>
<td>FNMULd</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>06_{16}</td>
<td>—</td>
<td>FsMULd</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>07_{16}</td>
<td>—</td>
<td>FNzMULd</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>08_{16}</td>
<td>—</td>
<td>FxTOd</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>FxTOq</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>FiTOd</td>
<td>FsTOd</td>
<td>—</td>
<td>FqTOd</td>
<td>FiTOq</td>
<td>FsTOq</td>
</tr>
<tr>
<td>0D_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0E_{16}–1F_{16}</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<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&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>FMOVs (fcc0)</td>
<td>FMOVd (fcc0)</td>
<td>FMOVq (fcc0)</td>
<td>—</td>
<td>† †</td>
<td>† †</td>
</tr>
<tr>
<td>01&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>02&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>03&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>FMOVs (fcc1)</td>
<td>FMOVd (fcc1)</td>
<td>FMOVq (fcc1)</td>
<td>—</td>
<td>FMOVRsZ ‡</td>
<td>FMOVRdZ ‡</td>
</tr>
<tr>
<td>04&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>FMOVs (fcc1)</td>
<td>FMOVd (fcc1)</td>
<td>FMOVq (fcc1)</td>
<td>—</td>
<td>FMOVRsLEZ ‡</td>
<td>FMOVRdLEZ ‡</td>
</tr>
<tr>
<td>05&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>FCMPs</td>
<td>FCMPd</td>
<td>FCMPq</td>
<td>—</td>
<td>FCMPEs ‡</td>
<td>FCMPEd ‡</td>
</tr>
<tr>
<td>06&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>07&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>08&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>FMOVs (fcc2)</td>
<td>FMOVd (fcc2)</td>
<td>FMOVq (fcc2)</td>
<td>—</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>09&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0A&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>FMOVRsNZ ‡</td>
<td>FMOVRdNZ ‡</td>
</tr>
<tr>
<td>0B&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0C&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>FMOVs (fcc3)</td>
<td>FMOVd (fcc3)</td>
<td>FMOVq (fcc3)</td>
<td>—</td>
<td>FMOVRsGZ ‡</td>
<td>FMOVRdGZ ‡</td>
</tr>
<tr>
<td>0D&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>0E&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>FMOVRsGEZ ‡</td>
<td>FMOVRdGEZ ‡</td>
</tr>
<tr>
<td>0F&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>FMOVs (icc)</td>
<td>FMOVd (icc)</td>
<td>FMOVq (icc)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>11</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>12&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>13&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>14&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>15&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>17&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>18&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>FMOVs (xcc)</td>
<td>FMOVd (xcc)</td>
<td>FMOVq (xcc)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>19&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1B&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1C&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1D&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1E&lt;sub&gt;16&lt;/sub&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1F&lt;sub&gt;16&lt;/sub&gt;</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

<table>
<thead>
<tr>
<th>cond (3:0)</th>
<th>BPcc (\text{op} = 0) (\text{op2} = 1) (\text{bit} 28 = 0)</th>
<th>CBcond (\text{op} = 0) (\text{op2} = 1) (\text{bit} 28 = 1)</th>
<th>Bicc (\text{op} = 0) (\text{op2} = 2)</th>
<th>FBPfcc (\text{op} = 0) (\text{op2} = 5)</th>
<th>FBfcc (\text{op} = 0) (\text{op2} = 6)</th>
<th>Tcc (\text{op} = 2) (\text{op3} = 3A_{16}) (\text{bit} 29 = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BPN</td>
<td>—</td>
<td>BN(^D)</td>
<td>FBPN</td>
<td>FBN(^D)</td>
<td>TN</td>
</tr>
<tr>
<td>1</td>
<td>BPE</td>
<td>C(^*)BE</td>
<td>BE(^D)</td>
<td>FBPNE</td>
<td>FBN(^D)</td>
<td>TE</td>
</tr>
<tr>
<td>2</td>
<td>BPLE</td>
<td>C(^*)BLE</td>
<td>BLE(^D)</td>
<td>FBPLG</td>
<td>FBLG(^D)</td>
<td>TLE</td>
</tr>
<tr>
<td>3</td>
<td>BPL</td>
<td>C(^*)BL</td>
<td>BL(^D)</td>
<td>FBPLU</td>
<td>FBLU(^D)</td>
<td>TL</td>
</tr>
<tr>
<td>4</td>
<td>BPLEU</td>
<td>C(^*)BLEU</td>
<td>BLEU(^D)</td>
<td>FBPL</td>
<td>FBL(^D)</td>
<td>TLEU</td>
</tr>
<tr>
<td>5</td>
<td>BPCS</td>
<td>C(^*)BCS</td>
<td>BCS(^D)</td>
<td>FBPU</td>
<td>FBUG(^D)</td>
<td>TCS</td>
</tr>
<tr>
<td>6</td>
<td>BPNEG</td>
<td>C(^*)BNEG</td>
<td>BNEG(^D)</td>
<td>FBPG</td>
<td>FBG(^D)</td>
<td>TNEG</td>
</tr>
<tr>
<td>7</td>
<td>BPVS</td>
<td>C(^*)BVS</td>
<td>BVS(^D)</td>
<td>FBPU</td>
<td>FBU(^D)</td>
<td>TVS</td>
</tr>
<tr>
<td>8</td>
<td>BPA</td>
<td>—</td>
<td>BA(^D)</td>
<td>FBPA</td>
<td>FBA(^D)</td>
<td>TA</td>
</tr>
<tr>
<td>9</td>
<td>BPNE</td>
<td>C(^*)BNE</td>
<td>BNE(^D)</td>
<td>FBE</td>
<td>FBE(^D)</td>
<td>TNE</td>
</tr>
<tr>
<td>A</td>
<td>BPG</td>
<td>C(^*)BG</td>
<td>BG(^D)</td>
<td>FBPUE</td>
<td>FBU(^D)</td>
<td>TG</td>
</tr>
<tr>
<td>B</td>
<td>BPE</td>
<td>C(^*)BGE</td>
<td>BGE(^D)</td>
<td>FBPGE</td>
<td>FBE(^D)</td>
<td>TEG</td>
</tr>
<tr>
<td>C</td>
<td>BPGU</td>
<td>C(^*)BGU</td>
<td>BGU(^D)</td>
<td>FBPUGE</td>
<td>FBE(^D)</td>
<td>TGU</td>
</tr>
<tr>
<td>D</td>
<td>BPCC</td>
<td>C(^*)BCC</td>
<td>BCC(^D)</td>
<td>FBPE</td>
<td>FBP(^D)</td>
<td>TCC</td>
</tr>
<tr>
<td>E</td>
<td>BPPOS</td>
<td>C(^*)BPOS</td>
<td>BPOS(^D)</td>
<td>FBPU(^E)</td>
<td>FBU(^E)</td>
<td>TPOS</td>
</tr>
<tr>
<td>F</td>
<td>BPVC</td>
<td>C(^*)BVC</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 (\text{op} = 0) (\text{op2} = 3)</th>
<th>MOVr (\text{op} = 2) (\text{op3} = 2F_{16})</th>
<th>FMOVr (\text{op} = 2) (\text{op3} = 35_{16})</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;s</td>
</tr>
<tr>
<td>2</td>
<td>BRLEZ</td>
<td>MOVRLZ</td>
<td>FMOVR(&lt;s</td>
</tr>
<tr>
<td>3</td>
<td>BRLZ</td>
<td>MOVRLZ</td>
<td>FMOVR(&lt;s</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;s</td>
</tr>
<tr>
<td>6</td>
<td>BRGZ</td>
<td>MOVRGZ</td>
<td>FMOVR(&lt;s</td>
</tr>
<tr>
<td>7</td>
<td>BRGEZ</td>
<td>MOVRGEZ</td>
<td>FMOVR(&lt;s</td>
</tr>
</tbody>
</table>

### TABLE A-9

\(cc / \text{opf}_cc\) Fields (MOVcc and FMOVcc)

<table>
<thead>
<tr>
<th>\text{opf}_cc</th>
<th>Condition Code Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{cc2} \text{cc1} \text{cc0}</td>
<td>fcc0</td>
</tr>
</tbody>
</table>

APPENDIX A • Opcode Maps 465
### TABLE A-9  \( cc / opf_{cc} \) Fields (MOVcc and FMOVcc)

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1</td>
<td>—</td>
</tr>
<tr>
<td>1 1 0</td>
<td>xcc</td>
</tr>
<tr>
<td>1 1 1</td>
<td>—</td>
</tr>
</tbody>
</table>

### TABLE A-10  \( cc \) Fields (FBPfcc, FCMP, and FCMPE)

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc0</th>
<th>Condition Code Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>fcc0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>fcc1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>fcc2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>fcc3</td>
</tr>
</tbody>
</table>

### TABLE A-11  \( cc \) Fields (BPcc and Tcc)

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc0</th>
<th>Condition Code Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>icc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>xcc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>Op Code</td>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
</tr>
<tr>
<td>EDGE8cc</td>
<td>ARRAY8</td>
<td>FPCMPIE16</td>
</tr>
<tr>
<td>1</td>
<td>EDGEXN</td>
<td>ADDXC</td>
</tr>
<tr>
<td>2</td>
<td>EDGE8Lcc</td>
<td>ARRAY16</td>
</tr>
<tr>
<td>3</td>
<td>EDGE8LN</td>
<td>ADDXCC</td>
</tr>
<tr>
<td>4</td>
<td>EDGE16cc</td>
<td>ARRAY32</td>
</tr>
<tr>
<td>5</td>
<td>EDGE16N</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>EDGE16Lcc</td>
<td>UMULXHI</td>
</tr>
<tr>
<td>7</td>
<td>EDGE16LN</td>
<td>LZCNT</td>
</tr>
</tbody>
</table>

**TABLE A-13** opf[8:0] for VIS opcodes (op = 102, op3 = 3616) — part 2

<table>
<thead>
<tr>
<th>Op Code</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>EDGE32cc</td>
<td>ALIGNADDRESS</td>
<td>FPCMPGT16</td>
<td>FMULD8SU16</td>
<td>FALIGNDATAg</td>
<td>FPPADDs16</td>
<td>FANDNOT1</td>
<td>FSRC2</td>
</tr>
<tr>
<td>9</td>
<td>EDGE32N</td>
<td>BMASK</td>
<td>FSLAS16</td>
<td>FMULD8UL16</td>
<td>—</td>
<td>FPPADDs16s</td>
<td>FANDNOT1s</td>
</tr>
<tr>
<td>A</td>
<td>EDGE32Lcc</td>
<td>ALIGNADDRESS</td>
<td>FPCMPFEQ16/</td>
<td>FPCMPUFEQ16</td>
<td>FPPADD64</td>
<td>—</td>
<td>FPPADD32</td>
</tr>
<tr>
<td></td>
<td>LITTLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>EDGE32LN</td>
<td>CMASK8</td>
<td></td>
<td></td>
<td>FPPADD64</td>
<td>—</td>
<td>FPPADD32s</td>
</tr>
</tbody>
</table>

**TABLE A-14** opf[8:0] for VIS opcodes (op = 102, op3 = 3616) — part 3

<table>
<thead>
<tr>
<th>Op Code</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>C</td>
<td>—</td>
<td>—</td>
<td>FPCMPGT32</td>
<td>—</td>
<td>BSHUFFLE</td>
<td>FPSUBS16</td>
<td>FXOR</td>
</tr>
<tr>
<td>C</td>
<td>—</td>
<td>—</td>
<td>FPCMPGT32</td>
<td>—</td>
<td>BSHUFFLE</td>
<td>FPSUBS16</td>
<td>FXOR</td>
</tr>
<tr>
<td>D</td>
<td>—</td>
<td>CMASK16</td>
<td>FSLAS32</td>
<td>FPPADD64</td>
<td>FEXPAND</td>
<td>FPSUBS16s</td>
<td>FXORS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(bits 29:25 = 0) and (bits 18:14 = 0))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>—</td>
<td>FSLAS32</td>
<td>FPPADD64</td>
<td>FEXPAND</td>
<td>FPSUBS16s</td>
<td>FXORS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(bits 29:25 ≠ 0) or (bits 18:14 ≠ 0))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>—</td>
<td>—</td>
<td>FPCMPFEQ32/</td>
<td>PDISTD</td>
<td>—</td>
<td>FPSUBS32</td>
<td>FNAND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FPCMPUFEQ32</td>
<td></td>
<td>—</td>
<td>FPSUBS32</td>
<td>FNAND</td>
</tr>
<tr>
<td>F</td>
<td>—</td>
<td>—</td>
<td>CMASK32</td>
<td>PDISTN</td>
<td>—</td>
<td>FPSUBS32S</td>
<td>FNANDs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(bits 29:25 = 0) and (bits 18:14 = 0))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE A-14
opf [8:0] for VIS opcodes (op = 10_2, op3 = 36_16) — part 4

<table>
<thead>
<tr>
<th>opf (3:0)</th>
<th>opf (8:4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0B 09 0A 0B 0C 0D 0E 0F</td>
</tr>
<tr>
<td>1 SIAM</td>
<td>— — — — — — — —</td>
</tr>
<tr>
<td>2</td>
<td>— — — — Reserved Reserved — —</td>
</tr>
<tr>
<td>3</td>
<td>— — — — — — — —</td>
</tr>
<tr>
<td>4</td>
<td>— — — — — — — —</td>
</tr>
<tr>
<td>5</td>
<td>— — — — Reserved — — — —</td>
</tr>
<tr>
<td>6</td>
<td>— — — — — — Reserved Reserved — —</td>
</tr>
<tr>
<td>7</td>
<td>— — — — — — — — — —</td>
</tr>
</tbody>
</table>

### TABLE A-14
opf [8:0] for VIS opcodes (op = 10_2, op3 = 36_16) — part 5

<table>
<thead>
<tr>
<th>opf (3:0)</th>
<th>opf (8:4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>08 09 0A 0B 0C 0D 0E 0F</td>
</tr>
<tr>
<td>9</td>
<td>— — — — Reserved — — — —</td>
</tr>
<tr>
<td>A</td>
<td>— — Reserved — — Reserved Reserved — —</td>
</tr>
<tr>
<td>B</td>
<td>— — Reserved Reserved — — — —</td>
</tr>
<tr>
<td>C</td>
<td>— — — — — — — —</td>
</tr>
<tr>
<td>D</td>
<td>— — — — — — — —</td>
</tr>
<tr>
<td>E</td>
<td>— — — — — — — — — —</td>
</tr>
<tr>
<td>F</td>
<td>— — — — — — — — — —</td>
</tr>
</tbody>
</table>

### TABLE A-14
opf [8:0] for VIS opcodes (op = 10_2, op3 = 36_16) — part 6

<table>
<thead>
<tr>
<th>opf (3:0)</th>
<th>opf (8:4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10 11 12 13 14 15</td>
</tr>
<tr>
<td>1</td>
<td>MOVdTOx FPCMPULE8 3-op AES_KEXPAND0N 1-op MD3N —</td>
</tr>
<tr>
<td>2</td>
<td>MOVsTOuw — 3-op AES_KEXPAND2N 1-op SHA1N FLCMPs</td>
</tr>
<tr>
<td>3</td>
<td>MOVsTOsw — FPCMPN E8/ FPCMPUNE8 — 1-op SHA256N FLCMPd</td>
</tr>
<tr>
<td>4</td>
<td>X MULx — 2-op DES_IP N — —</td>
</tr>
<tr>
<td>5</td>
<td>XMULXHI — 2-op DES_IP N — —</td>
</tr>
<tr>
<td>6</td>
<td>— — 2-op DES_KEXPANDN — —</td>
</tr>
<tr>
<td>7</td>
<td>— — — 3-op CRC32CN — —</td>
</tr>
<tr>
<td>7</td>
<td>— — — 3-op CRC32CN — —</td>
</tr>
</tbody>
</table>
### TABLE A-14  \( \text{opf}[8:0] \) for VIS opcodes (\( \text{op} = 102 \), \( \text{op3} = 36_{16} \)) — part 7

<table>
<thead>
<tr>
<th>( \text{opf} {8:4} )</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>—</td>
<td>MAVxTOD</td>
<td>FPCMPUGT8</td>
<td>—</td>
<td>MPMUL(^N) (rd=0, rs1=0)</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>MAVwTOs</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>MPMUL(^N) (rd=0, rs1=0)</td>
</tr>
<tr>
<td>A</td>
<td>—</td>
<td>—</td>
<td>FPCMPUEQ8/</td>
<td>—</td>
<td>MONTMUL(^N) (rd=0, rs1=0)</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>—</td>
<td>FPCMPUEQ8</td>
<td>—</td>
<td>—</td>
<td>MONTMUL(^N) (rd=0, rs1=0)</td>
</tr>
</tbody>
</table>

### TABLE A-14  \( \text{opf}[8:0] \) for VIS opcodes (\( \text{op} = 102 \), \( \text{op3} = 36_{16} \)) — part 8

<table>
<thead>
<tr>
<th>( \text{opf} {8:4} )</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>1A</th>
<th>1B</th>
<th>1C–1F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>
</tr>
<tr>
<td>2</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>
</tr>
<tr>
<td>4</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>
</tr>
<tr>
<td>6</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>
</tr>
</tbody>
</table>

### TABLE A-14  \( \text{opf}[8:0] \) for VIS opcodes (\( \text{op} = 102 \), \( \text{op3} = 36_{16} \)) — part 9

<table>
<thead>
<tr>
<th>( \text{opf} {8:4} )</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>1A</th>
<th>1B</th>
<th>1C–1F</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</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>
</tr>
<tr>
<td>A</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>
</tr>
<tr>
<td>C</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>
</tr>
<tr>
<td>E</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>
</tr>
</tbody>
</table>

**opf (3:0)**

|     |     |    |    | Reserved | Reserved |
|-----|-----|----|--------|----------|

APPENDIX A • Opcode Maps
### TABLE A-15  \( \text{op5}[3:0] \) (\( \text{op} = 10_2 \), \( \text{op3} = 37_{16} = \text{FMAf} \))

<table>
<thead>
<tr>
<th>( \text{op5}[3:2] ) (var)</th>
<th>( \text{op5}[1:0] ) (size)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>FPMADDX</td>
<td>FMADDs</td>
<td>FMADDd</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>FPMADDXHI</td>
<td>FMSUBs</td>
<td>FMSUBd</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>FNMSUBs</td>
<td>FNMSUBd</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>FNMADDs</td>
<td>FNMADDd</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 473.

The appendix contains these sections:

■ Definition of an Implementation Dependency on page 471.
■ Hardware Characteristics on page 472.
■ Implementation Dependency Categories on page 472.
■ List of Implementation Dependencies on page 472.

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 Oracle SPARC Architecture 2011 Compliance with SPARC V9 Architecture on page 16.

Some elements of the architecture are defined to be implementation dependent. These elements include certain registers and operations that may vary from implementation to implementation; they are explicitly identified as such in this appendix.

Implementation elements (such as instructions or registers) that appear in an implementation but are not defined in this document (or its updates) are not considered to be SPARC V9 elements of that implementation.
B.2 Hardware Characteristics

Hardware characteristics that do not affect the behavior observed by software on SPARC V9 systems are not considered architectural implementation dependencies. A hardware characteristic may be relevant to the user system design (for example, the speed of execution of an instruction) or may be transparent to the user (for example, the method used for achieving cache consistency). The SPARC International document, Implementation Characteristics of Current SPARC V9-based Products, Revision 9.x, provides a useful list of these hardware characteristics, along with the list of implementation-dependent design features of SPARC V9-compliant implementations.

In general, hardware characteristics deal with

- Instruction execution speed
- Whether instructions are implemented in hardware
- The nature and degree of concurrency of the various hardware units constituting a SPARC V9 implementation

B.3 Implementation Dependency Categories

Many of the implementation dependencies can be grouped into four categories, abbreviated by their first letters throughout this appendix:

- **Value (v)**
  The semantics of an architectural feature are well defined, except that a value associated with the feature may differ across implementations. A typical example is the number of implemented register windows (impl. dep. #2-V8).

- **Assigned Value (a)**
  The semantics of an architectural feature are well defined, except that a value associated with the feature may differ across implementations and the actual value is assigned by SPARC International. Typical examples are the impl field of the Version register (VER) (impl. dep. #13-V8) and the FSR.ver field (impl. dep. #19-V8).

- **Functional Choice (f)**
  The SPARC V9 architecture allows implementors to choose among several possible semantics related to an architectural function. A typical example is the treatment of a catastrophic error exception, which may cause either a deferred or a disrupting trap (impl. dep. #31-V8-Cs10).

- **Total Unit (t)**
  The existence of the architectural unit or function is recognized, but details are left to each implementation. Examples include the handling of I/O registers (impl. dep. #7-V8) and some alternate address spaces (impl. dep. #29-V8).

B.4 List of Implementation Dependencies

TABLE B-1 provides a complete list of the SPARC V9 implementation dependencies. The Page column lists the page for the context in which the dependency is defined; bold face indicates the main page on which the implementation dependency is described.
<table>
<thead>
<tr>
<th>Nbr</th>
<th>Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-V8</td>
<td>f</td>
<td>Software emulation of instructions</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whether an instruction complies with Oracle SPARC Architecture 2011 by being implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>2-V8</td>
<td>v</td>
<td>Number of IU registers</td>
<td>17, 34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An Oracle SPARC Architecture implementation may contain from 72 to 640 general-purpose 64-bit R registers. This corresponds to a grouping of the registers into $MAXPGL + 1$ sets of global R registers plus a circular stack of $N_{REG_{WINDOWS}}$ sets of 16 registers each, known as register windows. The number of register windows present ($N_{REG_{WINDOWS}}$) is implementation dependent, within the range of 3 to 32 (inclusive).</td>
<td></td>
</tr>
<tr>
<td>3-V8</td>
<td>f</td>
<td>Incorrect IEEE Std 754-1985 results</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An implementation may indicate that a floating-point instruction did not produce a correct IEEE Std 754-1985 result by generating an $fp_{exception_other}$ exception with $FSR_{ftt} = unfinished_{FPop}$. In this case, software running in a higher privilege mode shall emulate any functionality not present in the hardware.</td>
<td></td>
</tr>
<tr>
<td>4, 5</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>6-V8</td>
<td>f</td>
<td>I/O registers privileged status</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whether I/O registers can be accessed by nonprivileged code is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>7-V8</td>
<td>t</td>
<td>I/O register definitions</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents and addresses of I/O registers are implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>8-V8- Cs20</td>
<td>t</td>
<td>RDasr/WRasr target registers</td>
<td>20, 48, 295, 357</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ancillary state registers (ASRs) in the range 0–27 that are not defined in Oracle SPARC Architecture 2011 are reserved for future architectural use.</td>
<td></td>
</tr>
<tr>
<td>9-V8- Cs20</td>
<td>f</td>
<td>RDasr/WRasr privileged status</td>
<td>20, 48, 295, 357</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The privilege level required to execute each of the implementation-dependent read/write ancillary state register instructions (for ASRs 28–31) is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>10-V8-12-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>13-V8</td>
<td>a</td>
<td>(this implementation dependency applies to execution modes with greater privileges)</td>
<td></td>
</tr>
<tr>
<td>14-V8-15-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>16-V8-Cu3</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>17-V8</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
When FSR.ns = 1, the FPU produces implementation-dependent results that may not correspond to IEEE Standard 754-1985.

a: When FSR.ns = 1 and a floating-point source operand is subnormal, an implementation may treat the subnormal operand as if it were a floating-point zero value of the same sign. The cases in which this replacement is performed are implementation dependent. However, if it occurs,
   (1) it should not apply to FLCMP, FABS, FMOV, or FNEG instructions and
   (2) FADD, FSUB, FLCMPE, and FCMP should give identical treatment to subnormal source operands.

Treating a subnormal source operand as zero may generate an IEEE 754 floating-point “inexact”, “division by zero”, or “invalid” condition (see Current Exception (cexc) on page 46). Whether the generated condition(s) trigger an fp_exception_ieee_754 exception or not depends on the setting of FSR.tem.

b: When a floating-point operation generates a subnormal result value, an Oracle SPARC Architecture implementation may either write the result as a subnormal value or replace the subnormal result by a floating-point zero value of the same sign and generate IEEE 754 floating-point “inexact” and “underflow” conditions. Whether these generated conditions trigger an fp_exception_ieee_754 exception or not depends on the setting of FSR.tem.

c: If an FPop generates an intermediate result value, the intermediate value is subnormal, and FSR.ns = 1, it is implementation dependent whether (1) the operation continues, using the subnormal value (possibly with some loss of accuracy), or (2) the virtual processor replaces the subnormal intermediate value with a floating-point zero value of the same sign, generates IEEE 754 floating-point “inexact” and “underflow” conditions, completes the instruction, and writes a final result (possibly with some loss of accuracy). Whether generated IEEE conditions trigger an fp_exception_ieee_754 exception or not depends on the setting of FSR.tem.

19-V8 a FPU version, FSR.ver
   Bits 19:17 of the FSR, FSR.ver, identify one or more implementations of the FPU architecture.

20-V8–21-V8 Reserved.

22-V8 f FPU tem, cexc, and aexc
   An Oracle SPARC Architecture implementation implements the tem, cexc, and aexc fields in hardware, conformant to IEEE Std 754-1985.

23-V8 Reserved.

24-V8 Reserved.

25-V8 f RDPR of FQ with nonexistent FQ
   An Oracle SPARC 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.

26-V8–28-V8 Reserved.

29-V8 t Address space identifier (ASI) definitions
   In SPARC V9, many ASIs were defined to be implementation dependent. Some of those ASIs have been allocated for standard uses in the Oracle SPARC Architecture. Others remain implementation dependent in the Oracle SPARC Architecture. See ASI Assignments on page 398 and Block Load and Store ASIs on page 410 for details.

30- f ASI address decoding
   In SPARC V9, an implementation could choose to decode only a subset of the 8-bit ASI specifier. In Oracle SPARC 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.
This implementation dependency is no longer used in the Oracle SPARC Architecture, since “catastrophic” errors are now handled using normal error-reporting mechanisms.

Whether any restartable deferred traps (and associated deferred-trap queues) are present is implementation dependent.

In an Oracle SPARC Architecture implementation, all exceptions that occur as the result of program execution are precise.

The method by which an interrupt is removed is now defined in the Oracle SPARC Architecture (see Clearing the Software Interrupt Register on page 442).

How quickly a virtual processor responds to an interrupt request, like all timing-related issues, is implementation dependent.

Trap type (TT) values 060_{16}–07F_{16} were reserved for implementation-dependent exceptions in SPARC V9 but are now all defined as standard Oracle SPARC Architecture exceptions.

The relative priorities of traps defined in the Oracle SPARC 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.

Reserved.

Reserved.

a: If a load floating-point instruction generates an exception that causes a non-precise trap, it is implementation dependent whether the contents of the destination floating-point register(s) or floating-point state register are undefined or are guaranteed to remain unchanged.

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.

b: If a load floating-point alternate instruction generates an exception that causes an illegal_instruction exception

Reserved.

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)
• whether an attempt to execute the instruction causes an illegal_instruction exception

WRAsr instructions with rd of 16-18, 28, 29, or 31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRAsr instruction using one of those rd values, the following are implementation dependent:

• the interpretation of bits 18:0 in the instruction
• the operation(s) performed (for example, xor) to generate the value written to the ASR
• whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20)
• whether an attempt to execute the instruction causes an illegal_instruction exception
In SPARC V9, it is implementation-dependent whether “tininess” (an IEEE 754 term) is detected before or after rounding. In all Oracle SPARC Architecture implementations, tininess is detected before rounding.

The architectural parameter \( \text{MAXPTL} \) 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 \( \text{MAXPTL} = \text{MAXPGL} \) (see impl. dep. #401-S10). Architecturally, \( \text{MAXPTL} \) must be \( \geq 2 \).

An implementation may choose either to implement automatic “cleaning” of register windows in hardware or to generate a \text{clean_window} trap, when needed, for window(s) to be cleaned by software.

The following aspects of the \text{PREFETCH} and \text{PREFETCHA} instructions are implementation dependent:

- \( a \): the attributes of the block of memory prefetched: its size (minimum = 64 bytes) and its alignment (minimum = 64-byte alignment)
- \( b \): whether each defined prefetch variant is implemented (1) as a NOP, (2) with its full semantics, or (3) with common-case prefetching semantics
- \( c \): 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 \text{PREFETCH} and \text{PREFETCHA} instructions used to be (but are no longer) implementation dependent:

- \( d \): while in nonprivileged mode (\text{PSTATE.priv} = 0), an attempt to reference an ASI in the range \( 0_{16}..7F_{16} \) by a \text{PREFETCHA} instruction executes as a NOP; specifically, it does not cause a \text{privileged_action} exception.
- \( e \): \text{PREFETCH} and \text{PREFETCHA} have no observable effect in privileged code
- \( g \): while in privileged mode (\text{PSTATE.priv} = 1), an attempt to reference an ASI in the range \( 30_{16}..7F_{16} \) by a \text{PREFETCHA} instruction executes as a NOP (specifically, it does not cause a \text{privileged_action} exception)

If an accurate count cannot always be returned when \text{TICK} is read, any inaccuracy should be small, bounded, and documented.

An implementation may implement fewer than 63 bits in \text{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 \text{IMPDEP2A} instructions were defined to be completely implementation dependent in SPARC V9. The opcodes that have not been used in this space are now just documented as reserved opcodes.

It is implementation dependent whether \text{LDTW[A]} is implemented in hardware. If not, an attempt to execute an \text{LDTW} instruction will cause an \text{unimplemented_LDTW} exception.

It is implementation dependent whether \text{LDTWA} is implemented in hardware. If not, an attempt to execute an \text{LDTWA} instruction will cause an \text{unimplemented_LDTW} exception.
Unimplemented STTW[A] trap

1. 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.

2. It is implementation dependent whether STDA is implemented in hardware. If not, an attempt to execute an STTWA instruction will cause an *unimplemented_STTW* exception.

LDDF[A]_mem_address_not_aligned

1. 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 Oracle SPARC Architecture processor, the *LDDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDDF instruction)

2. 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 Oracle SPARC Architecture processor, the *LDDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDDFA instruction)

STDF[A]_mem_address_not_aligned

1. 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 Oracle SPARC Architecture processor, the *STDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STDF instruction)

2. 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 Oracle SPARC Architecture processor, the *STDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STDFA instruction)
LDQF[A]_mem_address_not_aligned

- LDQF requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an LDQF instruction may cause an LDQF_mem_address_not_aligned exception. In this case, the trap handler software must emulate the LDQF instruction and return. (In an Oracle SPARC Architecture processor, the LDQF_mem_address_not_aligned exception occurs in this case and trap handler software emulates the LDQF instruction) (this exception does not occur in hardware on Oracle SPARC Architecture 2011 implementations, because they do not implement the LDQF instruction in hardware)

STQF[A]_mem_address_not_aligned

- STQF requires only word alignment in memory. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an STQF instruction may cause an STQF_mem_address_not_aligned exception. In this case, the trap handler software must emulate the STQF instruction and return. (In an Oracle SPARC Architecture processor, the STQF_mem_address_not_aligned exception occurs in this case and trap handler software emulates the STQF instruction) (this exception does not occur in hardware on Oracle SPARC Architecture 2011 implementations, because they do not implement the STQF instruction in hardware)

Implemented memory models

Whether memory models represented by PSTATE.mm = 102 or 112 are supported in an Oracle SPARC 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 Oracle SPARC Architecture implementation that only supports the TSO memory model, PSTATE.mm always reads as zero and attempts to write to it are ignored.
TABLE B-2 provides a list of implementation dependencies that, in addition to those in TABLE B-1, apply to Oracle SPARC Architecture processors. Bold face indicates the main page on which the implementation dependency is described. See Appendix C in the Extensions Documents for further information.

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>200-201</td>
<td>Reserved.</td>
<td>—</td>
</tr>
<tr>
<td>203-U3-</td>
<td>Dispatch Control register (DCR) bits 13:6 and 1</td>
<td>127, 226, 296, 433</td>
</tr>
<tr>
<td>Cs10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>204-U3-</td>
<td>DCR bits 5:3 and 0</td>
<td>206-U3-</td>
</tr>
<tr>
<td>CS10</td>
<td></td>
<td>CASE10</td>
</tr>
<tr>
<td>205-U3-</td>
<td>Instruction Trap Register</td>
<td>206-U3-</td>
</tr>
<tr>
<td>Cs10</td>
<td></td>
<td>CASE10</td>
</tr>
<tr>
<td>206-U3-</td>
<td>SHUTDOWN instruction</td>
<td></td>
</tr>
</tbody>
</table>
### Ordering of errors captured in instruction execution
The order in which errors are captured in instruction execution is implementation dependent. Ordering may be in program order or in order of detection.

### 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.

### TTE support for cv bit
Whether the cv bit is supported in TTE is implementation dependent in the Oracle SPARC Architecture. When the cv bit in TTE is not provided and the implementation has virtually indexed caches, the implementation should support hardware unaliasing for the caches.

### TSB number of entries
The maximum number of entries in a TSB is implementation dependent in the Oracle SPARC Architecture (to a maximum of 16 million).

### TSB 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 Oracle SPARC Architecture. This implementation dependency existed for UltraSPARC III/IV, only to maintain compatibility with the TLB miss handling software of UltraSPARC I/II.

### Reserved

### This implementation dependency no longer applies, as of UltraSPARC Architecture 2005

### This implementation dependency no longer applies, in Oracle SPARC Architecture 2011

### This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.

### This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.

### This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.

### Reserved.

### Reserved.
TABLE B-2  Oracle SPARC Architecture Implementation Dependencies  (3 of 6)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| 244-U3-Cs10 | Data Watchpoint Reliability  
Data Watchpoint traps are completely implementation-dependent in Oracle SPARC Architecture processors. | —    |
| 245-U3-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.          | —    |
| 248-U3   | Conditions for fp_exception_other with unfinished_FPop  
The conditions under which an fp_exception_other exception with floating-point trap type of unfinished_FPop can occur are implementation dependent. An implementation may cause fp_exception_other with unfinished_FPop under a different (but specified) set of conditions. | 45   |
| 249-U3-Cs10 | Data Watchpoint for Partial Store Instruction  
For an STPARTIAL instruction, the following aspects of data watchpoints are implementation dependent: (a) whether data watchpoint logic examines the byte store mask in R[rs2] or it conservatively behaves as if every Partial Store always stores all 8 bytes, and (b) whether data watchpoint logic examines individual bits in the Virtual (Physical) Data Watchpoint Mask in DCUCR to determine which bytes are being watched or (when the Watchpoint Mask is nonzero) it conservatively behaves as if all 8 bytes are being watched. | 334  |
| 250-U3-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2007.          | —    |
| 251     | Reserved.                                                                                       | —    |
| 252-U3-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.          | —    |
| 253-U3-Cs10 | This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.          | —    |
| 255-U3-Cs10 | LDDFA with ASI E0₁₆ or E1₁₆ and misaligned destination register number  
If an LDDFA opcode is used with an ASI of E0₁₆ or E1₁₆ (Block Store Commit ASI, an illegal combination with LDDFA) and a destination register number rd is specified which is not a multiple of 8 (“misaligned” rd), an Oracle SPARC Architecture virtual processor generates an illegal_instruction exception. | 237  |
| 256-U3   | LDDFA with ASI E0₁₆ or E1₁₆ and misaligned memory address  
If an LDDFA opcode is used with an ASI of E0₁₆ or E1₁₆ (Block Store Commit ASI, an illegal combination with LDDFA) and a memory address is specified with less than 64-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the exception generated is DAE_invalid_asi, mem_address_not_aligned, or LDDF_mem_address_not_aligned. | 237  |
| 257-U3   | LDDFA with ASI C0₁₆–C5₁₆ or C8₁₆–CD₁₆ and misaligned memory address  
If an LDDFA opcode is used with an ASI of C0₁₆–C5₁₆ or C8₁₆–CD₁₆ (Partial Store ASIs, which are an illegal combination with LDDFA) and a memory address is specified with less than 8-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the exception generated is DAE_invalid_asi, mem_address_not_aligned, or LDDF_mem_address_not_aligned. | 237  |
| 259–299 | Reserved.                                                                                       | —    |
| 300-U4-Cs10 | Attempted access to ASI registers with LDTWA  
If an LDTWA instruction referencing a non-memory ASI is executed, it generates a DAE_invalid_asi exception. | 247  |
| 301-U4-Cs10 | Attempted access to ASI registers with STTWA  
If an STTWA instruction referencing a non-memory ASI is executed, it generates a DAE_invalid_asi exception. | 340  |
| 302-U4-Cs10 | Scratchpad registers  
An Oracle SPARC Architecture processor includes eight privileged Scratchpad registers (64 bits each, read/write accessible). | 411  |

APPENDIX B  •  Implementation Dependencies  481
### TABLE B-2  Oracle SPARC Architecture Implementation Dependencies  (4 of 6)

<table>
<thead>
<tr>
<th>Nbr</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>303-U4-CS10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>305-U4-CS10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>306-U4-Cs10</td>
<td>Trap type generated upon attempted access to noncacheable page with LDTXA. When an LDTXA instruction attempts access from an address that is not mapped to cacheable memory space, a DAE_nc_page exception is generated.</td>
<td>250</td>
</tr>
<tr>
<td>307-U4-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>308-U3-Cs10</td>
<td>This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.</td>
<td>—</td>
</tr>
<tr>
<td>309-U4-Cs10</td>
<td>Reserved.</td>
<td>—</td>
</tr>
<tr>
<td>310-U4</td>
<td>Large page sizes</td>
<td>445</td>
</tr>
<tr>
<td></td>
<td>Which, if any, of the following optional page sizes are supported by the MMU in an Oracle SPARC Architecture implementation is implementation dependent: 512 KBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes.</td>
<td></td>
</tr>
<tr>
<td>311–319</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>327–399</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>400-S10</td>
<td>Global Level register (GL) implementation</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Although GL is defined as a 4-bit register, an implementation may implement any subset of those bits sufficient to encode the values from 0 to MAXPGL for that implementation. If any bits of GL are not implemented, they read as zero and writes to them are ignored.</td>
<td></td>
</tr>
<tr>
<td>401-S10</td>
<td>Maximum Global Level (MAXPGL)</td>
<td>72, 73</td>
</tr>
<tr>
<td></td>
<td>The architectural parameter MAXPGL is a constant for each implementation; its legal values are from 2 to 15 (supporting from 3 to 16 sets of global registers). In a typical implementation MAXPGL = MAXPTL (see impl. dep. #101-V9-CS10). Architecturally, MAXPTL must be ≥ 2.</td>
<td></td>
</tr>
<tr>
<td>403-S10</td>
<td>Setting of “dirty” bits in FPRS</td>
<td>54, 54</td>
</tr>
<tr>
<td></td>
<td>A “dirty” bit (du or dl) in the FPRS register must be set to ‘1’ if any of its corresponding F registers is actually modified. If an instruction that normally writes to an F register is executed and causes a fp_disabled exception, FPRS.k and FPRS.éli are unchanged. Beyond that, the specific conditions under which a dirty bit is set are implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>404-S10</td>
<td>Scratchpad registers 4 through 7</td>
<td>412</td>
</tr>
<tr>
<td></td>
<td>The degree to which Scratchpad registers 4–7 are accessible to privileged software is implementation dependent. Each may be (1) fully accessible, (2) accessible, with access much slower than to scratchpad register 0–3, or (3) inaccessible (cause a DAE_invalid Asi exception).</td>
<td></td>
</tr>
<tr>
<td>405-S10</td>
<td>Virtual address range</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>An Oracle SPARC Architecture implementation may support a full 64-bit virtual address space or a more limited range of virtual addresses. In an implementation that does not support a full 64-bit virtual address space, the supported range of virtual addresses is restricted to two equal-sized ranges at the extreme upper and lower ends of 64-bit addresses; that is, for n-bit virtual addresses, the valid address ranges are 0 to 2^{n-1} – 1 and 2^{64} – 2^{n-1} to 2^{64} – 1. (see also impl. dep. #451-S20)</td>
<td></td>
</tr>
</tbody>
</table>
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.

**Block Load behavior**

The following aspects of the behavior of block load (LDBLOCKFD) instructions are implementation dependent:
- What memory ordering model is used by LDBLOCKFD (LDBLOCKFD is not required to follow TSO memory ordering)
- Whether LDBLOCKFD 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 LDBLOCKFD appears to execute out of order, or follow LoadLoad ordering (with respect to older loads, younger loads, and other LDBLOCKFs)
- Whether LDBLOCKFD follows register-dependency interlocks, as do ordinary load instructions

- Whether the MMU ignores the side-effect bit (TTE.e) for LDBLOCKFD 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 LDBLOCKFD (the recommended behavior), or only on accesses to the first eight bytes

**Block Store behavior**

The following aspects of the behavior of block store (STBLOCKFD) instructions are implementation dependent:
- The memory ordering model that STBLOCKFD follows (other than as constrained by the rules outlined on page 323).
- Whether VA_watchpoint exceptions are recognized on accesses to all 64 bytes of a STBLOCKFD (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 STBLOCKFD to a non-cacheable page causes a DAE_nc_page exception.
- Whether STBLOCKFD follows register dependency interlocks (as ordinary stores do).
- Whether a non-Commit STBLOCKFD 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 STBLOCKFD accesses (in which case, STBLOCKFs behave as if TTE.e = 0)
- Any other restrictions on the behavior of STBLOCKFD, as described in implementation-specific documentation.

**MEMBAR behavior**

An Oracle SPARC Architecture implementation may define the operation of each MEMBAR variant in any manner that provides the required semantics.

**Load Twin Extended Word behavior**

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.

**Size of ContextID fields**

The size of context ID fields in MMU context registers is implementation-dependent and may range from 13 to 16 bits.
Behavior of DONE and RETRY when TSTATE[TL].pstate.am = 1

If (1) TSTATE[TL].pstate.am = 1 and (2) a DONE or RETRY instruction is executed (which sets PSTATE.am to ‘1’ by restoring the value from TSTATE[TL].pstate.am to PSTATE.am), it is implementation dependent whether the DONE or RETRY instruction masks (zeros) the more-significant 32 bits of the values it places into PC and NPC.

STICK register

a: If an accurate count cannot always be returned when STICK is read, any inaccuracy should be small, bounded, and documented.

b: Implementation dependency 442-S10(b) no longer applies, as of Oracle SPARC Architecture 2015. The bit number m of the least significant implemented bit of the counter field of the STICK register is implementation-dependent, but must be in range 4 to 0, inclusively (that is, 4 ≥ m ≥ 0). Any low-order bits not implemented must read as zero.

Reserved for UltraSPARC Architecture 2005

Availability of control_transfer_instruction exception feature

Availability of the control_transfer_instruction exception feature is implementation dependent. If not implemented, trap type 074 16 is unused, PSTATE.tct always reads as zero, and writes to PSTATE.tct are ignored.

Width of Virtual Addresses supported

The width of the virtual address supported is implementation dependent. If fewer than 64 bits are supported, then the unsupported bits must have the same value as the most significant supported bit. For example, if the model supports 48 virtual address bits, then bits 63:48 must have the same value as bit 47. (see also impl. dep. #405-S10)

Width of Real Addresses supported

The number of real address (RA) bits supported is implementation dependent. A minimum of 40 bits and maximum of 56 bits can be provided for real addresses (RA). See implementation-specific documentation for details.

Unified vs. Split Instruction and Data MMUs

It is implementation dependent whether there is a unified MMU (UMMU) or a separate IMMU (for instruction accesses) and DMMU (for data accesses). The Oracle SPARC Architecture supports both configurations.

Reserved for UltraSPARC Architecture 2007

Maximum Value of PAUSE register (MAX_PAUSE_COUNT)

The maximum number of virtual processor cycles that a virtual processor may be paused by the PAUSE instruction, MAX_PAUSE_COUNT, is 2^B − 1. B, the number of bits used to implement the pause_count field, is implementation dependent but B must be ≥ 13. Therefore MAX_PAUSE_COUNT ≥ 2^{13} − 1 (8191) cycles. If a PAUSE instruction requests a pause of more than MAX_PAUSE_COUNT cycles, the virtual processor must saturate the pause count to its maximum value, that is, write a value of MAX_PAUSE_COUNT to the PAUSE register.

Reserved for Oracle SPARC Architecture 2011

Reserved for future use
Assembly Language Syntax

This appendix supports Chapter 7, *Instructions*. Each instruction description in Chapter 7 includes a table that describes the suggested assembly language format for that instruction. This appendix describes the notation used in those assembly language syntax descriptions and lists some synthetic instructions provided by Oracle SPARC Architecture assemblers for the convenience of assembly language programmers.

The appendix contains these sections:
- **Notation Used** on page 485.
- **Syntax Design** on page 490.
- **Synthetic Instructions** on page 490.

## C.1 Notation Used

The notations defined here are also used in the assembly language syntax descriptions in Chapter 7, *Instructions*.

Items in typewriter font are literals to be written exactly as they appear. Items in italic font are metasymbols that are to be replaced by numeric or symbolic values in actual SPARC V9 assembly language code. For example, "imm asi" would be replaced by a number in the range 0 to 255 (the value of the imm asi bits in the binary instruction) or by a symbol bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, rs8rn2 is a reg (register name) whose binary value will be placed in the rs2 field of the resulting instruction.

### C.1.1 Register Names

*reg.* A reg is an integer register name. It can have any of the following values:\(^1\)

- `%r0-%r31`
- `%g0-%g7` (global registers; same as `%r0-%r7`)
- `%o0-%o7` (out registers; same as `%r8-%r15`)
- `%l0-%l7` (local registers; same as `%r16-%r23`)
- `%i0-%i7` (in registers; same as `%r24-%r31`)
- `%fp` (frame pointer; conventionally same as `%i6`)
- `%sp` (stack pointer; conventionally same as `%o6`)

Subscripts identify the placement of the operand in the binary instruction as one of the following:

---

\(^1\) In actual usage, the `%sp`, `%fp`, `%gn`, `%on`, `%ln`, and `%in` forms are preferred over `%rn`. 

---
reg\_rs1 (rs1 field)
reg\_rs2 (rs2 field)
reg\_rd (rd field)

**freg.** An freg is a floating-point register name. It may have the following values:
%f0, %f1, %f2, ... %f31
%f32, %f34, ... %f60, %f62 (even-numbered only, from %f32 to %f62)
%d0, %d2, %d4, ... %d60, %d62 (%dn, where n mod 2 = 0, only)
%q0, %q4, %q8, ... %q56, %q60 (%qn, where n mod 4 = 0, only)

See Floating-Point Registers on page 38 for a detailed description of how the single-precision,
double-precision, and quad-precision floating-point registers overlap.

Subscripts further identify the placement of the operand in the binary instruction as one of the
following:

\( freg\_rs1 \) (rs1 field)
\( freg\_rs2 \) (rs2 field)
\( freg\_rs3 \) (rs3 field)
\( freg\_rd \) (rd field)

**asr\_reg.** An asr\_reg is an Ancillary State Register name. It may have one of the following values:
%asr16-%asr31

Subscripts further identify the placement of the operand in the binary instruction as one of the
following:

\( asr\_reg\_rs1 \) (rs1 field)
\( asr\_reg\_rd \) (rd field)

**i\_or\_x\_cc.** An i\_or\_x\_cc specifies a set of integer condition codes, those based on either the 32-bit
result of an operation (icc) or on the full 64-bit result (xcc). It may have either of the following
values:
%icc
%xcc

**fccn.** An fccn specifies a set of floating-point condition codes. It can have any of the following
values:
%fcc0
%fcc1
%fcc2
%fcc3

### C.1.2 Special Symbol Names

Certain special symbols appear in the syntax table in typewriter font. They must be written exactly as
they are shown, including the leading percent sign (%).

The symbol names and the registers or operators to which they refer are as follows:

%asi Address Space Identifier (ASI) register
%canrestore Restorable Windows register
%cansave Savable Windows register
%ccr Condition Codes register
%cleanwin Clean Windows register
%cwp Current Window Pointer (CWP) register
%fprs Floating-Point Registers State (FPRS) register
%fss Floating-Point State register
%gsr General Status Register (GSR)
%otherwin Other Windows (OTHERWIN) register
%pc Program Counter (PC) register
%pil Processor Interrupt Level register
%pstate Processor State register
%softint Soft Interrupt register
%softint_clr Soft Interrupt register (clear selected bits)
%softint_set Soft Interrupt register (set selected bits)
%stick† System Timer (STICK) register
%stick_cmpr† System Timer Compare (STICK_CMPR) register
%tba Trap Base Address (TBA) register
%tick Cycle count (TICK) register
%tl Trap Level (TL) register
%tnpc Trap Next Program Counter (TNPC) register
%tpc Trap Program Counter (TPC) register
%tstate Trap State (TSTATE) register
%tt Trap Type (TT) register
%wstate Window State register
%y Y register

† The original assembly language names for %stick and %stick_cmpr were, respectively, %sys_tick and %sys_tick_cmpr, which are now deprecated. Over time, assemblers will support the new %stick and %stick_cmpr names for these registers (which are consistent with %tick). In the meantime, some existing assemblers may only recognize the original names.

The following special symbol names are prefix unary operators that perform the functions described, on an argument that is a constant, symbol, or expression that evaluates to a constant offset from a symbol:

%hh Extracts bits 63:42 (high 22 bits of upper word) of its operand
%hm Extracts bits 41:32 (low-order 10 bits of upper word) of its operand
%hi or %lm Extracts bits 31:10 (high-order 22 bits of low-order word) of its operand
%lo Extracts bits 9:0 (low-order 10 bits) of its operand

For example, the value of "%lo(symbol)" is the least-significant 10 bits of symbol.

Certain predefined value names appear in the syntax table in typewriter font. They must be written exactly as they are shown, including the leading sharp sign (#). The value names and the constant values to which they are bound are listed in TABLE C-1.

**TABLE C-1** Value Names and Values (1 of 2)

<table>
<thead>
<tr>
<th>Value Name in Assembly Language</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>#n_reads</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>#one_read</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>#n_writes</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>#one_write</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>#page</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>#unified</td>
<td>17 (11_{16})</td>
<td></td>
</tr>
<tr>
<td>#n_reads_strong</td>
<td>20 (14_{16})</td>
<td></td>
</tr>
<tr>
<td>#one_read_strong</td>
<td>21 (15_{16})</td>
<td></td>
</tr>
</tbody>
</table>
### C.1.3 Values

Some instructions use operand values as follows:

- **const4**  
  A constant that can be represented in 4 bits
- **const22**  
  A constant that can be represented in 22 bits
- **imm asi**  
  An alternate address space identifier (0–255)
- **siam_mode**  
  A 3-bit mode value for the SIAM instruction
- **simm7**  
  A signed immediate constant that can be represented in 7 bits
- **simm8**  
  A signed immediate constant that can be represented in 8 bits
- **simm10**  
  A signed immediate constant that can be represented in 10 bits
- **simm11**  
  A signed immediate constant that can be represented in 11 bits
- **simm13**  
  A signed immediate constant that can be represented in 13 bits
- **value**  
  Any 64-bit value
- **shcnt32**  
  A shift count from 0–31
- **shcnt64**  
  A shift count from 0–63

### C.1.4 Labels

A label is a sequence of characters that comprises alphabetic letters (a–z, A–Z [with upper and lower case distinct]), underscores (_), dollar signs ($), periods (.), and decimal digits (0–9). A label may contain decimal digits, but it may not begin with one. A local label contains digits only.

### C.1.5 Other Operand Syntax

Some instructions allow several operand syntaxes, as follows:

- **reg plus imm**  
  Can be any of the following:
    - **reg rs1**  
      (equivalent to **reg rs1 + %g0**)
    - **reg rs1 + simm13**
\[
\begin{align*}
\text{reg}_{rs1} - \text{simm13} \\
\text{simm13} \quad \text{(equivalent to } \%g0 + \text{simm13)} \\
\text{simm13} + \text{reg}_{rs1} \text{(equivalent to } \text{reg}_{rs1} + \text{simm13)}
\end{align*}
\]

**address**  
Can be any of the following:

\[
\begin{align*}
\text{reg}_{rs1} \quad \text{(equivalent to } \text{reg}_{rs1} + \%g0) \\
\text{reg}_{rs1} + \text{simm13} \\
\text{reg}_{rs1} - \text{simm13} \\
\text{simm13} \quad \text{(equivalent to } \%g0 + \text{simm13)} \\
\text{simm13} + \text{reg}_{rs1} \text{(equivalent to } \text{reg}_{rs1} + \text{simm13)} \\
\text{reg}_{rs1} + \text{reg}_{rs2}
\end{align*}
\]

**membar_mask**  
Is the following:

\[
\text{const7} \quad \text{A constant that can be represented in 7 bits. Typically, this is an expression involving the logical OR of some combination of } \#\text{Lookaside}, \#\text{MemIssue}, \#\text{Sync}, \#\text{StoreStore}, \#\text{LoadStore}, \#\text{StoreLoad}, \text{and } \#\text{LoadLoad} \text{ (see TABLE 7-13 and TABLE 7-14 on page 257 for a complete list of mnemonics).}
\]

**prefetch_fcn** (prefetch function)  
Can be any of the following:

\[
\begin{align*}
0–31 \\
\text{Predefined constants (the values of which fall in the 0-31 range) useful as } \text{prefetch_fcn} \text{ values can be found in TABLE C-1 on page 487.}
\end{align*}
\]

**regaddr** (register-only address)  
Can be any of the following:

\[
\begin{align*}
\text{reg}_{rs1} \quad \text{(equivalent to } \text{reg}_{rs1} + \%g0) \\
\text{reg}_{rs1} + \text{reg}_{rs2}
\end{align*}
\]

**reg_or_imm** (register or immediate value)  
Can be either of:

\[
\begin{align*}
\text{reg}_{rs2} \\
\text{simm13}
\end{align*}
\]

**reg_or_imm5** (register or immediate value)  
Can be either of:

\[
\begin{align*}
\text{reg}_{rs2} \\
\text{simm5}
\end{align*}
\]

**reg_or_imm10** (register or immediate value)  
Can be either of:

\[
\begin{align*}
\text{reg}_{rs2} \\
\text{simm10}
\end{align*}
\]

**reg_or_imm11** (register or immediate value)  
Can be either of:

\[
\begin{align*}
\text{reg}_{rs2} \\
\text{simm11}
\end{align*}
\]

**reg_or_shcnt** (register or shift count value)  
Can be any of:

\[
\begin{align*}
\text{reg}_{rs2}
\end{align*}
\]
software_trap_number  Can be any of the following:

\[
\begin{align*}
\text{reg}_{\text{rs1}} & \quad \text{(equivalent to } \text{reg}_{\text{rs1}} + %g0) \\
\text{reg}_{\text{rs1}} + \text{reg}_{\text{rs2}} & \\
\text{reg}_{\text{rs1}} + \text{simm8} & \\
\text{reg}_{\text{rs1}} - \text{simm8} & \\
\text{simm8} & \quad \text{(equivalent to } %g0 + \text{simm8}) \\
\text{simm8} + \text{reg}_{\text{rs1}} & \quad \text{(equivalent to } \text{reg}_{\text{rs1}} + \text{simm8})
\end{align*}
\]

The resulting operand value (software trap number) must be in the range 0–255, inclusive.

C.1.6 Comments

Two types of comments are accepted by the SPARC V9 assembler: C-style “/*...*/” comments, which may span multiple lines, and “!/...!” comments, which extend from the “!” to the end of the line.

C.2 Syntax Design

The SPARC V9 assembly language syntax is designed so that the following statements are true:

- The destination operand (if any) is consistently specified as the last (rightmost) operand in an assembly language instruction.
- A reference to the contents of a memory location (for example, in a load, store, or load-store instruction) is always indicated by square brackets ([ ]); a reference to the address of a memory location (such as in a JMPL, CALL, or SETHI) is specified directly, without square brackets.

The follow additional syntax constraints have been adopted for Oracle SPARC Architecture:

- Instruction mnemonics should be limited to a maximum of 15 characters.

C.3 Synthetic Instructions

TABLE C-2 describes the mapping of a set of synthetic (or “pseudo”) instructions to actual instructions. These synthetic instructions are provided by the SPARC V9 assembler for the convenience of assembly language programmers.

Note: Synthetic instructions should not be confused with “pseudo ops,” which typically provide information to the assembler but do not generate instructions. Synthetic instructions always generate instructions; they provide more mnemonic syntax for standard SPARC V9 instructions.

<table>
<thead>
<tr>
<th>Synthetic Instruction</th>
<th>SPARC V9 Instruction(s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>reg_{rs1}, reg_or_imm</td>
<td>subcc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg_{rs1}, reg_or_imm, %g0</td>
</tr>
<tr>
<td>jmp</td>
<td>address</td>
<td>jmpl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>address, %g0</td>
</tr>
<tr>
<td>call</td>
<td>address</td>
<td>jmpl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>address, %o7</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>call address, reg$_{rd}$</td>
<td>jmpl address, reg$_{rd}$</td>
<td>Originally envisioned as an encoding for an “instruction prefetch” operation, but functions as a NOP on all Oracle SPARC Architecture implementations. (See PREFETCH function 17 on page 288 for an alternative method of prefetching instructions.)</td>
</tr>
<tr>
<td>iprefetch label</td>
<td>bn,a,pt %xcc,label</td>
<td></td>
</tr>
<tr>
<td>tst reg$_{rs1}$</td>
<td>orcc %g0, reg$_{rs1}$, %g0</td>
<td>Test.</td>
</tr>
<tr>
<td>ret</td>
<td>jmpl %i7$+$8, %g0</td>
<td>Return from subroutine.</td>
</tr>
<tr>
<td>retl</td>
<td>jmpl %o7$+$8, %g0</td>
<td>Return from leaf subroutine.</td>
</tr>
<tr>
<td>restore</td>
<td>restore %g0, %g0, %g0</td>
<td>Trivial RESTORE.</td>
</tr>
<tr>
<td>save</td>
<td>save %g0, %g0, %g0</td>
<td>Trivial SAVE.</td>
</tr>
<tr>
<td>setuw value, reg$_{rd}$</td>
<td>sethi %hi(value), reg$_{rd}$</td>
<td>(When ((value &amp; 3FF) == 0).)</td>
</tr>
<tr>
<td></td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or %g0, value, reg$_{rd}$</td>
<td>(When 0 ≤ value ≤ 4095).</td>
</tr>
<tr>
<td></td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sethi %hi(value), reg$_{rd}$</td>
<td>(Otherwise)</td>
</tr>
<tr>
<td></td>
<td>or reg$<em>{rd}$, %lo(value), reg$</em>{rd}$</td>
<td>Warning: do not use setuw in the delay slot of a DCTI. synonym for setu.</td>
</tr>
<tr>
<td>set value, reg$_{rd}$</td>
<td>sethi %hi(value), reg$_{rd}$</td>
<td>(When (value &gt; 0) and ((value &amp; 3FF) == 0).)</td>
</tr>
<tr>
<td></td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or %g0, value, reg$_{rd}$</td>
<td>(When 4096 ≤ value ≤ 4095).</td>
</tr>
<tr>
<td></td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sethi %hi(value), reg$_{rd}$</td>
<td>(Otherwise, if (value &lt; 0) and ((value &amp; 3FF) = 0))</td>
</tr>
<tr>
<td>sra reg$<em>{rd}$, %g0, reg$</em>{rd}$</td>
<td>— or —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sethi %hi(value), reg$_{rd}$</td>
<td>(Otherwise, if value &gt; 0)</td>
</tr>
<tr>
<td></td>
<td>or reg$<em>{rd}$, %lo(value), reg$</em>{rd}$</td>
<td>(Otherwise, if value &lt; 0)</td>
</tr>
<tr>
<td>setx value, reg, reg$_{rd}$</td>
<td>sethi %hh(value), reg</td>
<td>Create 64-bit constant.</td>
</tr>
<tr>
<td></td>
<td>or reg, %hm(value), reg</td>
<td>(&quot;reg&quot; is used as a temporary register.)</td>
</tr>
<tr>
<td>sllx reg, 32, reg</td>
<td>sethi %hi(value), reg$_{rd}$</td>
<td>Note: setx optimizations are possible but not enumerated here. The worst case is shown. Warning: do not use setx in the delay slot of a CTI.</td>
</tr>
<tr>
<td></td>
<td>or reg$<em>{rd}$, reg, reg$</em>{rd}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or reg$<em>{rd}$, %lo(value), reg$</em>{rd}$</td>
<td></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>signx reg31, reg32</td>
<td>sra reg31, %g0, reg32</td>
<td>Sign-extend 32-bit value to 64 bits.</td>
</tr>
<tr>
<td>signx reg32</td>
<td>sra reg32, %g0, reg32</td>
<td>One’s complement.</td>
</tr>
<tr>
<td>not reg31, reg32</td>
<td>xnor reg31, %g0, reg32</td>
<td>One’s complement.</td>
</tr>
<tr>
<td>not reg32</td>
<td>xnor reg32, %g0, reg32</td>
<td>Two’s complement.</td>
</tr>
<tr>
<td>neg reg32, reg32</td>
<td>sub %g0, reg32, reg32</td>
<td>Two’s complement.</td>
</tr>
<tr>
<td>cas [reg31], reg32</td>
<td>casa [reg31]#ASI_P, reg32, reg32</td>
<td>Compare and swap.</td>
</tr>
<tr>
<td>casl [reg31], reg32</td>
<td>casa [reg31]#ASI_P_L, reg32, reg32</td>
<td>Compare and swap, little-endian.</td>
</tr>
<tr>
<td>casx [reg31], reg32</td>
<td>casax [reg31]#ASI_P, reg32, reg32</td>
<td>Compare and swap extended.</td>
</tr>
<tr>
<td>casxl [reg31], reg32</td>
<td>casaxl [reg31]#ASI_P_L, reg32, reg32</td>
<td>Compare and swap extended, little-endian.</td>
</tr>
<tr>
<td>inc reg32</td>
<td>add reg32, 1, reg32</td>
<td>Increment by 1.</td>
</tr>
<tr>
<td>incc reg32</td>
<td>addcc reg32, const13, reg32</td>
<td>Increment by 1; set icc &amp; xcc.</td>
</tr>
<tr>
<td>incc const13, reg32</td>
<td>addcc reg32, const13, reg32</td>
<td>Incr by const13; set icc &amp; xcc.</td>
</tr>
<tr>
<td>dec reg32</td>
<td>sub reg32, 1, reg32</td>
<td>Decrement by 1.</td>
</tr>
<tr>
<td>dec const13, reg32</td>
<td>sub reg32, const13, reg32</td>
<td>Decrement by const13.</td>
</tr>
<tr>
<td>decce reg32</td>
<td>subcc reg32, 1, reg32</td>
<td>Decr by const13; set icc &amp; xcc.</td>
</tr>
<tr>
<td>decce const13, reg32</td>
<td>subcc reg32, const13, reg32</td>
<td>Decr by const13; set icc &amp; xcc.</td>
</tr>
<tr>
<td>btst reg_or_imm, reg31</td>
<td>andcc reg31, reg_or_imm, %g0</td>
<td>Bit test.</td>
</tr>
<tr>
<td>bset reg_or_imm, reg32</td>
<td>or reg32, reg_or_imm, reg32</td>
<td>Bit set.</td>
</tr>
<tr>
<td>bclr reg_or_imm, reg32</td>
<td>andn reg32, reg_or_imm, reg32</td>
<td>Bit clear.</td>
</tr>
<tr>
<td>btoe reg_or_imm, reg32</td>
<td>xor reg32, reg_or_imm, reg32</td>
<td>Bit toggle.</td>
</tr>
<tr>
<td>clr reg32</td>
<td>or %g0, %g0, reg32</td>
<td>Clear (zero) register.</td>
</tr>
<tr>
<td>clrb [address]</td>
<td>stb %g0, [address]</td>
<td>Clear byte.</td>
</tr>
<tr>
<td>clrh [address]</td>
<td>sth %g0, [address]</td>
<td>Clear half-word.</td>
</tr>
<tr>
<td>clrw [address]</td>
<td>stw %g0, [address]</td>
<td>Clear word.</td>
</tr>
<tr>
<td>clrx [address]</td>
<td>stx %g0, [address]</td>
<td>Clear extended word.</td>
</tr>
<tr>
<td>clirw reg31, reg32</td>
<td>srl reg31, %g0, reg32</td>
<td>Copy and clear upper word.</td>
</tr>
<tr>
<td>clirw reg32</td>
<td>srl reg32, %g0, reg32</td>
<td>Clear upper word.</td>
</tr>
<tr>
<td>mov reg_or_imm, reg32</td>
<td>or %g0, reg_or_imm, reg32</td>
<td></td>
</tr>
<tr>
<td>mov %y, reg32</td>
<td>rd %y, reg32</td>
<td></td>
</tr>
<tr>
<td>mov %asmn, reg32</td>
<td>rd %asmn, reg32</td>
<td></td>
</tr>
<tr>
<td>mov reg_or_imm, %y</td>
<td>wr %g0, reg_or_imm, %y</td>
<td></td>
</tr>
<tr>
<td>mov reg_or_imm, %asmn</td>
<td>wr %g0, reg_or_imm, %asmn</td>
<td></td>
</tr>
</tbody>
</table>
Index

A
a (annul) instruction field
branch instructions, 120, 123, 125, 152, 155
accesses
cacheable, 382
I/O, 382
restricted ASI, 385
with side effects, 382, 390
accrued exception (aexc) field of FSR register, 46, 424, 474
ADD instruction, 107
ADDC instruction, 107
ADDCc instruction, 107, 317
ADDCcc instruction, 107
address
operand syntax, 489
space identifier (ASI), 397
address mask (am) field of FSTATE register
description, 70
address space, 5, 13
address space identifier (ASI), 5, 381
accessing MMU registers, 456
appended to memory address, 17, 75
architecturally specified, 385
changed in UA
ASI_REAL, 412
ASI_REAL_IO, 412
ASI_REAL_IO_LITTLE, 412
ASI_REAL_LITTLE, 412
definition, 5
coding address space information, 77
explicit, 81
explicitly specified in instruction, 82
implicit, See implicit ASIs
nontranslating, 8, 247, 340
nontranslating ASI, 398
with prefetch instructions, 289
real ASI, 398
restricted, 385, 397
privileged, 385
restriction indicator, 51
SPARC V9 address, 383
translating ASI, 398
unrestricted, 385, 397
address space identifier (ASI) register
for load/store alternate instructions, 51
address for explicit ASI, 81
and LDDA instruction, 235, 246
and LDSTUBA instruction, 243
load integer from alternate space instructions, 228
with prefetch instructions, 289
for register-immediate addressing, 385
restoring saved state, 144, 303
saving state, 417
and STD A instruction, 339
store floating-point into alternate space instructions, 327
store integer to alternate space instructions, 320
and SWAPA instruction, 344
after trap, 21
and TSTATE register, 66
and write state register instructions, 358
addressing modes, 14
ADDX instruction, 108, 109, 128, 130, 140, 205, 255
ADDX instruction (SPARC V8), 107
ADDXc instruction, 109, 128, 130, 140, 205, 255
ADDXcc instruction, 108
ADDXcc instruction (SPARC V8), 107
alias
floating-point registers, 38
aliased, 5
ALIGNADDRESS instruction, 114
ALIGNADDRESS_LITTLE instruction, 114
alignment
data (load/store), 18, 77, 383
doubleword, 18, 77, 383
extended-word, 77
halfword, 18, 77, 383
instructions, 18, 77, 383
integer registers, 237, 244, 247
memory, 383, 437
quadword, 18, 77, 383
word, 18, 77, 383
ALLCLEAN instruction, 115
alternate space instructions, 19, 51
ancillary state registers (ASRs)
access, 48
assembly language syntax, 486
I/O register access, 19
possible registers included, 296, 358
privileged, 20, 473
reading/writing implementation-dependent processor
INDEX

registers, 20, 473
writing to, 358
AND instruction, 115
ANDcc instruction, 115
ANDN instruction, 115
ANDNcc instruction, 115
annul bit
in branch instructions, 125
in conditional branches, 153
annulled branches, 125
application program, 5, 48
architectural direction note, 4
architecture, meaning for SPARC V9, 13
arithmetic overflow, 51
ARRAY16 instruction, 117
ARRAY32 instruction, 117
ARRAY8 instruction, 117
ASI, 5
invalid, and DAE_invalid_asi, 435
ASI register, 49
ASI, See address space identifier (ASI)
ASI_AIUP, 399, 407
ASI_AIUPL, 399, 407
ASI_AIUS, 399, 407
ASI_AIUS_L, 249
ASI_AIUSL, 399, 407
ASI_IF_USER*, 70, 449
ASI_IF_USER_NONFAULT_LITTLE, 386
ASI_IF_USER_PRIMARY, 399, 407, 435, 448
ASI_IF_USER_PRIMARY_LITTLE, 386, 399, 407, 435
ASI_IF_USER_SECONDARY, 386, 399, 407, 435
ASI_IF_USER_SECONDARY_LITTLE, 386, 399, 407, 435
ASI_IF_USER_SECONDARY_NOFAULT_LITTLE, 386
ASI_BLK_AIUP, 399, 407
ASI_BLK_AIUPL, 399, 407
ASI_BLK_AIUS, 399, 407
ASI_BLK_AIUSL, 399, 407
ASI_BLK_COMMIT_P, 404
ASI_BLK_COMMIT_S, 404
ASI_BLK_P, 404
ASI_BLK_PL, 405
ASI_BLK_S, 404
ASI_BLK_SL, 405
ASI_BLK_PRIMARY, 81, 385, 386, 402
ASI_BLK_PRIMARY_LITTLE, 82, 385, 402, 454
ASI_BLK_PRIMARY_NO_FAULT, 383, 395, 402
ASI_BLK_PRIMARY_NO_FAULT_LITTLE, 383, 395, 402, 435
ASI_BLK_PRIMARY_NOFAULT_LITTLE, 386
ASI_PST16_P, 332, 403
ASI_PST16_PL, 332, 403
ASI_PST16_PRIMARY, 403
ASI_PST16_PRIMARY_LITTLE, 403
ASI_PST16_S, 332, 403
ASI_PST16_SECONDARY, 403
ASI_PST16_SECONDARY_LITTLE, 403
ASI_PST16_SL, 332
ASI_PST32_P, 332, 403
ASI_PST32_PL, 332, 403
ASI_PST32_PRIMARY, 403
ASI_PST32_PRIMARY_LITTLE, 403
ASI_PST32_S, 332, 403
ASI_PST32_SECONDARY, 403
ASI_PST32_SECONDARY_LITTLE, 403
ASI_PST32_SL, 332
ASI_PST8_P, 403
ASI_PST8_PL, 403
ASI_PST8_PRIMARY, 403
ASI_PST8_PRIMARY_LITTLE, 403
ASI_PST8_S, 403
ASI_PST8_SECONDARY, 403
ASI_PST8_SECONDARY_LITTLE, 403
ASI_PST8_SL, 332, 403
ASI_QUAD_LDD_REAL (deprecated), 401
ASI_QUAD_LDD_REAL_LITTLE (deprecated), 402
ASI_REAL, 399, 407, 412
ASI_REAL_INVALID, 399, 408, 412
ASI_N, 399
ASI_NL, 399
ASI_NUCLEUS, 81, 82, 399, 448
ASI_NUCLEUS_LITTLE, 82, 399
ASI_P, 402
ASI_PHYS_BYPASS_EC_WITH_EBIT_L, 412
ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE, 412
ASI_PHYS_USE_EC, 412
ASI_PHYS_USE_EC_L, 412
ASI_PHYS_USE_EC_LITTLE, 412
ASI_PIC, 403
ASI_PHY_BYPASS_UNKNOWN_L, 412
ASI_PHY_BYPASS_UNKNOWN_LITTLE, 412
ASI_PRIMARY, 81, 385, 386, 402
ASI_PRIMARY_LITTLE, 82, 385, 402, 454
ASI_PRIMARY_NO_FAULT, 383, 395, 402
ASI_PRIMARY_NO_FAULT_LITTLE, 383, 395, 402, 435
ASI_PRIMARY_NOFAULT_LITTLE, 386
ASI_PST8_PRIMARY, 403
ASI_PST8_PRIMARY_LITTLE, 403
ASI_PST8_S, 403
ASI_PRIMARY_LITTLE, 82, 399
ASI_QUAD_LDD_REAL (deprecated), 401
ASI_QUAD_LDD_REAL_LITTLE (deprecated), 402
ASI_REAL, 399, 407, 412
ASI_REAL_INVALID, 399, 408, 412
ASI_PST8_PRIMARY, 403
ASI_PST8_PRIMARY_LITTLE, 403
ASI_PST8_S, 403
ASI_PST8_SECONDARY, 403
ASI_PST8_SECONDARY_LITTLE, 403
ASI_PST8_SL, 332, 403
ASI_QUAD_LDD_REAL (deprecated), 401
ASI_QUAD_LDD_REAL_LITTLE (deprecated), 402
ASI_REAL, 399, 407, 412
ASI_REAL_INVALID, 399, 408, 412

Index 2
Index 3

ASI_REAL_IO_L, 399
ASI_REAL_IO_LITTLE, 399, 408, 412
ASI_REAL_L, 399
ASI_REAL_LITTLE, 399, 408, 412
ASI_S, 402
ASI_SECONDARY, 402, 448
ASI_SECONDARY_LITTLE, 402
ASI_SECONDARY_NO_FAULT, 395, 402, 435
ASI_SECONDARY_NO_FAULT_LITTLE, 395, 402, 435
ASI_SECONDARY_NOFAULT, 386
ASI_SL, 402
ASI_SNF, 402
ASI_SNFL, 402
ASI_ST_BLKINIT_AS_IF_USER_PRIMARY, 400
ASI_ST_BLKINIT_AS_IF_USER_PRIMARY_LITTLE, 401
ASI_ST_BLKINIT_As_IF_USER_SECONDARY, 400
ASI_ST_BLKINIT_AS_IF_USER_SECONDARY_LITTLE, 401
ASI_ST_BLKINIT_NUCLEUS, 401
ASI_ST_BLKINIT_NUCLEUS_LITTLE, 402
ASI_ST_BLKINIT_PRIMARY, 404
ASI_ST_BLKINIT_PRIMARY_LITTLE, 404
ASI_ST_BLKINIT_REAL, 401
ASI_ST_BLKINIT_REAL_LITTLE, 402
ASI_ST_BLKINIT_SECONDARY, 404
ASI_ST_BLKINIT_SECONDARY_LITTLE, 404
ASI_STBI_AIUP, 400
ASI_STBI_AIUPL, 401
ASI_STBI_AIOUS, 400
ASI_STBI_AIOUS_L, 401
ASI_STBI_N, 401
ASI_STBI_NL, 402
ASI_STBI_P, 404
ASI_STBI_PL, 404
ASI_STBI_PLM, 405
ASI_STBI_PM, 404
ASI_STBI_RL, 402
ASI_STBI_S, 404
ASI_STBI_SL, 404
ASI_STBI_SLM, 405
ASI_STBI_SM, 405
ASI_TWINX_AIUP, 249, 400, 409
ASI_TWINX_AIUPL, 249, 401, 409
ASI_TWINX_AIOUS, 249, 409
ASI_TWINX_AIOUS_L, 401, 409
ASI_TWINX_AS_IF_USER_PRIMARY, 400, 409
ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE, 401, 409
ASI_TWINX_AS_IF_USER_SECONDARY, 400, 409
ASI_TWINX_AS_IF_USER_SECONDARY_LITTLE, 401, 409
ASI_TWINX_N, 249, 401
ASI_TWINX_NL, 249, 402, 409
ASI_TWINX_NUCLEUS, 401, 409
ASI_TWINX_NUCLEUS_L[]_L, 384
ASI_TWINX_NUCLEUS_LITTLE, 402, 409
ASI_TWINX_P, 249, 404
ASI_TWINX_PL, 249, 404
ASI_TWINX_PRIMARY, 404, 410
ASI_TWINX_PRIMARY_LITTLE, 404, 410
ASI_TWINX_R, 401, 409
ASI_TWINX_REAL, 249, 401, 409
ASI_TWINX_REAL_L, 402, 409
ASI_TWINX_REAL_LITTLE, 402, 409
ASI_TWINX_RL, 409
ASI_TWINX_S, 249, 404
ASI_TWINX_SECONDARY, 404, 410
ASI_TWINX_SECONDARY_LITTLE, 404, 410
ASI_TWINX_SL, 249, 404
ASR, 5
asr_reg, 486
atomic
memory operations, 249, 392, 393
store doubleword instruction, 337, 339
store instructions, 319, 320
atomic load-store instructions
compare and swap, 131
load-store unsigned byte, 242, 344
load-store unsigned byte to alternate space, 243
simultaneously addressing doublewords, 343
swap R register with alternate space memory, 344
swap R register with memory, 131, 343
atomicity, 383, 479

B
BA instruction, 120, 465
BCC instruction, 120, 465
bclrg synthetic instruction, 492
BCS instruction, 120, 465
BE instruction, 120, 465
Berkeley RISCs, 15
BG instruction, 120, 465
BGE instruction, 120, 465
BGU instruction, 120, 465
Bicc instructions, 120, 459
big-endian, 5
big-endian byte order, 18, 69, 78
binary compatibility, 15
BL instruction, 120, 465
Bld, See LDBLOCKF instruction
BLE instruction, 120, 465
BLEU instruction, 120, 465
block load instructions, 38, 230, 410
block store instructions, 38, 232, 410
with commit, 236, 323, 410
blocked byte formatting, 117
BMASK instruction, 122
BN instruction, 120, 465
BNE instruction, 120, 465
BNEG instruction, 120, 465
BP instructions, 465
BPA instruction, 123, 465
BPCC instruction, 123, 465
BPCC instructions, 51, 123, 466
BPCS instruction, 123, 465
BPE instruction, 123, 465
BPG instruction, 123, 465
BPG instruction, 123, 465
BPG instruction, 123, 465
BPGU instruction, 123, 465
BPGU instruction, 123, 465
BFL instruction, 123, 465

Index 3
BPLE instruction, 123, 465
BPLEU instruction, 123, 465
BPN instruction, 123, 465
BPNE instruction, 123, 465
BPNEG instruction, 123, 465
BPOS instruction, 120, 465
BPPOS instruction, 123, 465
BP instructions, 125, 465
BPVC instruction, 123, 465
BPVS instruction, 123, 465
branch
  annulled, 125
  delayed, 75
  elimination, 87, 88
  fcc-conditional, 152, 155
  icc-conditional, 120
instructions
  on floating-point condition codes, 152
  on floating-point condition codes with prediction, 154
  on integer condition codes with prediction (BPcc), 123
  on integer condition codes, See Bicc instructions
  when contents of integer register match condition, 125
  prediction bit, 125
  unconditional, 120, 124, 152, 155
  with prediction, 14
BRGEZ instruction, 125
BRGZ instruction, 125
BRLEZ instruction, 125
BRLEQ instruction, 125
BRNZ instruction, 125
BRZ instruction, 125
bsset synthetic instruction, 492
BSHUFFLE instruction, 122
BST, See STBLOCKF instruction
btog synthetic instruction, 492
btst synthetic instruction, 492
BVC instruction, 120, 465
BVS instruction, 120, 465
byte, 5
  addressing, 81
  data format, 23
  order, 18
  order, big-endian, 18
  order, little-endian, 18
byte order
  big-endian, 69
  implicit, 69
  in trap handlers, 423
  little-endian, 69
C
C*BCC instruction, 465
C*BCS instruction, 465
C*BE instruction, 465
C*BG instruction, 465
C*BGE instruction, 465
C*BGU instruction, 465
C*BL instruction, 465
C*BLE instruction, 465
C*BLEU instruction, 465
C*BNE instruction, 465
C*BNEG instruction, 465
C*BPOS instruction, 465
C*BVC instruction, 465
C*BVS instruction, 465
C8BL instruction, 133
cache
  coherency protocol, 382
  data, 388
  instruction, 388
  miss, 293
  nonconsistent instruction cache, 388
  cacheable accesses, 382
  caching, TSB, 453
CALL instruction
  description, 127
  displacement, 20
  does not change CWP, 35
  and JMPL instruction, 226
  writing address into r[15], 37
CALL synthetic instruction, 490, 491
CANRESTORE (restorable windows) register, 62
  and clean_window exception, 88
  and CLEANWIN register, 63, 64, 439
  counting windows, 64
  decremented by RESTORE instruction, 300
  decremented by SAVED instruction, 309
  detecting window underflow, 36
  if registered window was spilled, 300
  incremented by SAVE instruction, 307
  modified by NORMALW instruction, 280
  modified by OTHERW instruction, 282
  range of values, 61, 479
  RESTORE instruction, 88
  specification for RDPR instruction, 298
  specification for WRPR instruction, 360
  window underflow, 439
CANSAVE (savable windows) register, 62
  decremented by SAVE instruction, 307
  detecting window overflow, 36
  FLUSHW instruction, 169
  if equals zero, 88
  incremented by RESTORE, 300
  incremented by SAVED instruction, 309
  range of values, 61, 479
  SAVE instruction, 440
  specification for RDPR instruction, 298
  specification for WRPR instruction, 360
  window overflow, 439
CAS synthetic instruction, 393
CASA instruction, 131
  32-bit compare-and-swap, 392
  alternate space addressing, 18
  and DAE_nc_page exception, 435
  atomic operation, 242
  hardware primitives for mutual exclusion of CASXA, 392
  in multiprocessor system, 243, 343, 344
  r register use, 77
  word access (memory), 77
casn synthetic instructions, 492
CASX synthetic instruction, 392, 393
CASXA instruction, 131
64-bit compare-and-swap, 392
alternate space addressing, 18
and DAE nc_page exception, 435
atomic operation, 243
doubleword access (memory), 77
hardware primitives for mutual exclusion of CASA, 392
in multiprocessor system, 242, 243, 343, 344
R register use, 77
catastrophic error exception, 417
CBcond instructions
opcode maps, 465
cc0 instruction field
branch instructions, 123, 155
floating point compare instructions, 157
move instructions, 268, 465
cc1 instruction field
branch instructions, 123, 155
floating point compare instructions, 157
move instructions, 268, 465
cc2 instruction field
compare and branch instructions, 133
move instructions, 268, 465
CCR (condition codes register), 5
CCR (condition codes) register, 50
32-bit operation (icc) bit of condition field, 50, 51
64-bit operation (xcc) bit of condition field, 50, 51
ADD instructions, 107
ASR for, 49
carry (c) bit of condition fields, 51
icc field, See CCR.icc field
negative (n) bit of condition fields, 50
overflow bit (v) in condition fields, 51
restored by RETRY instruction, 144, 303
saved after trap, 417
saving after trap, 21
TSTATE register, 66
write instructions, 358
xcc field, See CCR.xcc field
zero (z) bit of condition fields, 50
CCR.icc field
add instructions, 107, 346
bit setting for signed division, 311
bit setting for signed/unsigned multiply, 318, 355
bit setting for unsigned division, 354
branch instructions, 120, 124, 269
integer subtraction instructions, 342
logical operation instructions, 116, 281, 363
Tcc instruction, 349
CCR.xcc field
add instructions, 107, 346
bit setting for signed/unsigned divide, 311, 354
bit setting for signed/unsigned multiply, 318, 355
branch instructions, 124, 269
logical operation instructions, 116, 281, 363
subtract instructions, 342
Tcc instruction, 349
CFR register, 59
RDCFR instruction, 295
checksum, for 16-bit TCP/UDP, 156
clean register window, 307, 434
clean window, 5
and window traps, 64, 439
CLEANWIN register, 64
definition, 439
number is zero, 88
trap handling, 440
clean_window exception, 62, 88, 308, 434, 439, 476
CLEANWIN (clean windows) register, 62
CANSAVE instruction, 88
clean window counting, 63
incremented by trap handler, 440
range of values, 61, 63, 479
specification for RDPR instruction, 298
specification for WRPR instruction, 360
specifying number of available clean windows, 439
value calculation, 64
clock cycle, counts for virtual processor, 52
clock tick registers, See Tick and Stick registers
clock-tick register (TICK), 437
cirn synthetic instructions, 492
CMASK16 instruction, 136
CMASK32 instruction, 136
CMASK8 instruction, 136
cmp synthetic instruction, 342, 490
code
self-modifying, 393
coopherence, 5
between processors, 479
data cache, 388
domain, 382
memory, 383
unit, memory, 384
cmpare and branch instructions, 133
cmpare and swap instructions, 131, 133
cmparison instruction, 83, 342
Compatibility Feature register, See CFR
compatibility note, 4
completed (memory operation), 5
compliance
SPARC V9, 456
compliant SPARC V9 implementation, 16
cond instruction field
branch instructions, 120, 123, 152, 155
floating point move instructions, 175
move instructions, 268
condition codes
adding, 346
effect of compare-and-swap instructions, 132
extended integer (xcc), 51
floating-point, 152
icc field, 50
integer, 50
results of integer operation (icc), 51
subtracting, 342, 351
trapping on, 349
xcc field, 50
condition codes register, See CCR register
conditional branches, 120, 152, 155
conditional move instructions, 21
conforming SPARC V9 implementation, 16
consistency
  between instruction and data spaces, 393
  processor, 388, 391
  processor self-consistency, 390
  sequential, 383, 389
  strong, 389
const22 instruction field of ILLTRAP instruction, 224
constants, generating, 312
context, 5
  nucleus, 168
  selection for translation, 454
context identifier, 384
determination of, 454
Context register
  Nucleus, 458
  Primary, 458
  Secondary, 458
clean windows, 63
definition, 6
control transfer
  pseudo-control-transfer via WRPR to PSTATE.am, 71
call/jmpl instructions, 127, 226
done/retry instructions, 145, 303, 349
return, 305
and Tcc instruction, 350
control-transfer instructions, 20
control-transfer instructions (CTIs), 20, 144, 303
conventions
  font, 2
  notational, 2
conversion
  between floating-point formats instructions, 220
  floating-point to integer instructions, 219, 367
  integer to floating-point instructions, 163, 223
  planar to packed, 206
copyback, 5
cpu_mondo exception, 435
CRC32C instruction, 138
cross-call, 5
current exception (cexc) field of FSR register, 46, 90, 474
current window, 6
current window pointer register, see CWP register
current_little_endian (cle) field of PSTATE register, 69, 385
CWBCC instruction, 133
CWBCS instruction, 133
CWBE instruction, 133
CWBG instruction, 133
CWBG instruction, 133
CWBG instruction, 133
CWBL instruction, 133
CWBLE instruction, 133
CWBLE instruction, 133
CWBLEU instruction, 133
CWBN instruction, 133
CWBN instruction, 133
CWBPO instruction, 133
CWBVC instruction, 133
CWBVS instruction, 133
CWP (current window pointer) register
  and instructions
    CALL and Jmpl instructions, 35
    FLUSHW instructions, 169
    RDPR instruction, 298
    RESTORE instruction, 88, 300
    SAVE instruction, 88, 300, 307
    WRPR instruction, 360
  and traps
    after spill trap, 440
    after spill/fill trap, 21
    on window trap, 440
    saved by hardware, 417
CWP (current window pointer) register, 61
cycle, 6
D
D superscript on instruction name, 94
d10_hi instruction field
  compare and branch instructions, 133
d10_lo instruction field
  compare and branch instructions, 133
d16hi instruction field
  branch instructions, 125
d16lo instruction field
  branch instructions, 125
daE_invalid_ASI exception
  with load instructions and ASIs, 237, 409, 410, 411, 412
  with store instructions and ASIs, 237, 409, 410, 411, 412
DAE_invalid_ASI exception, 132, 435
accessing noncacheable page, 392
SWAP/SWAPA, 345
  with load alternate instructions, 132, 229, 236, 239, 240, 243, 247, 320, 321, 328, 332, 340, 344
  with load instructions, 237, 243, 248
with load instructions and ASIs, 237
with nonfaulting load, 395
with store instructions, 243, 321, 329, 340
**DAE_invalid_ASI** exception (replacing SPARC V9
*data_access_exception*), 435
**DAE_invalid_ASI** exception
with load instructions and ASIs, 237
with nonfaulting load, 395
with store instructions, 243, 321, 329, 340
**DAE_nc_page** exception, 435
accessing noncacheable page, 392
with compare-and-swap instructions, 132
with load instructions, 231, 242, 243, 250
with store instructions, 243
with SWAP instructions, 245, 248, 254, 258, 338
with nonfaulting load, 395
with store instructions, 243, 321, 324, 326, 329, 330, 334, 335, 340, 341
with SWAP instruction, 343
with SWAP instructions, 245
**DAE_nc_page** exception (replacing SPARC V9
*data_access_exception*), 435
**DAE_nfo_page** exception, 132, 435
and TTE.nfo, 450
with FLUSH instructions, 168
with LDTXA instructions, 250
with load alternate instructions, 229
with nonfaulting load, 395
with store instructions, 243, 321, 324, 326, 329, 330, 334, 335, 340, 341
with SWAP instruction, 343
with SWAP instructions, 245
**DAE_nfo_page** exception (replacing SPARC V9
*data_access_exception*), 435
**DAE_noncacheable_page** exception
with LDTXA instructions, 250
**DAE_privilege_violation** exception, 132, 435
and TTE.p, 452
with load alternate instructions, 229
with load instructions, 232, 237, 239, 243, 245, 248, 324, 338
with store instructions, 243, 321, 326, 329, 330, 334, 335, 340, 341, 343
with SWAP instructions, 227, 345
**DAE_privilege_violation** exception (replacing SPARC V9
*data_access_exception*), 435
**DAE_side_effect_page**
with nonfaulting loads, 383
**DAE_side_effect_page** exception, 436
with load alternate instructions, 229
with load instructions, 232, 237, 248
with nonfaulting load, 395
**DAE_side_effect_page** exception (replacing SPARC V9
*data_access_exception*), 435
data
access, 6
cache coherence, 388
conversion between SIMD formats, 29
flow order constraints
memory reference instructions, 387
register reference instructions, 387
formats
byte, 23
doubleword, 23
halfword, 23
Int16 SIMD, 30
Int32 SIMD, 30
quadword, 23
tagged word, 23
Uint8 SIMD, 30
word, 23
memory, 394
types
floating-point, 23
signed integer, 23
unsigned integer, 23
width, 23
watchpoint exception, 333
Data Cache Unit Control register, See DCUCR
*data_access_exception* exception (SPARC V9), 435
DCTI couple, 87
DCTI instructions, 6
behavior, 75
RETURN instruction effects, 305
dec synthetic instructions, 492
decccc synthetic instructions, 492
defined trap, 420
distinguishing from disrupting trap, 421
floating-point, 299
restartable
implementation dependency, 421
software actions, 421
delay instruction
and annul field of branch instruction, 152
annulling, 20
conditional branches, 155
DONE instruction, 144
executed after branch taken, 125
following delayed control transfer, 20
RETRY instruction, 303
RETURN instruction, 305
unconditional branches, 155
with conditional branch, 124
delayed branch, 75
delayed control transfer, 125
delayed CTI, See DCTI
denormalized number, 6
deprecated, 6
deprecated exceptions
*tag_overflow*, 438
deprecated instructions
FBA, 152
FBE, 152
FBG, 152
FBGE, 152
FBBL, 152
FBLE, 152
FBLG, 152
FBN, 152
FBNE, 152
FBO, 152
FBU, 152
FBUE, 152
FBUCE, 152
FBUCE, 152
Index 7
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBULE, 152</td>
<td></td>
</tr>
<tr>
<td>LDFSR, 238</td>
<td></td>
</tr>
<tr>
<td>LDTW, 244</td>
<td></td>
</tr>
<tr>
<td>LDTWA, 246</td>
<td></td>
</tr>
<tr>
<td>RDY, 49, 50, 295</td>
<td></td>
</tr>
<tr>
<td>SDIV, 50, 310</td>
<td></td>
</tr>
<tr>
<td>SDIVcc, 50, 310</td>
<td></td>
</tr>
<tr>
<td>SMUL, 50, 318</td>
<td></td>
</tr>
<tr>
<td>SMULcc, 50, 318</td>
<td></td>
</tr>
<tr>
<td>STFSR, 330</td>
<td></td>
</tr>
<tr>
<td>STTW, 337</td>
<td></td>
</tr>
<tr>
<td>STTWA, 339</td>
<td></td>
</tr>
<tr>
<td>SWAP, 343</td>
<td></td>
</tr>
<tr>
<td>SWAPA, 344</td>
<td></td>
</tr>
<tr>
<td>TADDccTV, 347</td>
<td></td>
</tr>
<tr>
<td>TADDccTV, 347</td>
<td></td>
</tr>
<tr>
<td>TSUBccTV, 352</td>
<td></td>
</tr>
<tr>
<td>UDIV, 50, 353</td>
<td></td>
</tr>
<tr>
<td>UDIVcc, 50, 353</td>
<td></td>
</tr>
<tr>
<td>UMUL, 50, 355</td>
<td></td>
</tr>
<tr>
<td>UMULcc, 50, 355</td>
<td></td>
</tr>
<tr>
<td>WRY, 49, 50, 357</td>
<td></td>
</tr>
<tr>
<td>dev_mondo exception, 436</td>
<td></td>
</tr>
<tr>
<td>disp19 instruction field</td>
<td></td>
</tr>
<tr>
<td>branch instructions, 123, 155</td>
<td></td>
</tr>
<tr>
<td>disp22 instruction field</td>
<td></td>
</tr>
<tr>
<td>branch instructions, 120, 152</td>
<td></td>
</tr>
<tr>
<td>disp30 instruction field</td>
<td></td>
</tr>
<tr>
<td>word displacement (CALL), 127</td>
<td></td>
</tr>
<tr>
<td>disrupting trap, 421</td>
<td></td>
</tr>
<tr>
<td>disrupting_performance_event exception, 436</td>
<td></td>
</tr>
<tr>
<td>divide instructions, 20, 278, 310, 353</td>
<td></td>
</tr>
<tr>
<td>division_by_zero exception, 83, 278, 436</td>
<td></td>
</tr>
<tr>
<td>division-by-zero bits of FSR.aexc/FSR.cexc fields, 48</td>
<td></td>
</tr>
<tr>
<td>DMMU context register usage, 455</td>
<td></td>
</tr>
<tr>
<td>Secondary Context register, 458</td>
<td></td>
</tr>
<tr>
<td>DONE instruction, 144</td>
<td></td>
</tr>
<tr>
<td>effect on TNPC register, 66</td>
<td></td>
</tr>
<tr>
<td>effect on TSTATE register, 67</td>
<td></td>
</tr>
<tr>
<td>generating illegal_instruction exception, 437</td>
<td></td>
</tr>
<tr>
<td>modifying CCR.xcc condition codes, 51</td>
<td></td>
</tr>
<tr>
<td>return from trap, 417</td>
<td></td>
</tr>
<tr>
<td>return from trap handler with different GL value, 74</td>
<td></td>
</tr>
<tr>
<td>target address, 20</td>
<td></td>
</tr>
<tr>
<td>doubleword, 6</td>
<td></td>
</tr>
<tr>
<td>addressing, 80</td>
<td></td>
</tr>
<tr>
<td>alignment, 18, 77, 383</td>
<td></td>
</tr>
<tr>
<td>data format, 23</td>
<td></td>
</tr>
<tr>
<td>definition, 6</td>
<td></td>
</tr>
<tr>
<td>EDGE16 instruction, 146</td>
<td></td>
</tr>
<tr>
<td>EDGE16L instruction, 146</td>
<td></td>
</tr>
<tr>
<td>EDGE16LN instruction, 147</td>
<td></td>
</tr>
<tr>
<td>EDGE16N instruction, 147</td>
<td></td>
</tr>
<tr>
<td>EDGE32 instruction, 146</td>
<td></td>
</tr>
<tr>
<td>EDGE32L instruction, 146</td>
<td></td>
</tr>
<tr>
<td>EDGE32LN instruction, 147</td>
<td></td>
</tr>
<tr>
<td>EDGE32N instruction, 147</td>
<td></td>
</tr>
<tr>
<td>EDGE8 instruction, 146</td>
<td></td>
</tr>
<tr>
<td>EDGE8L instruction, 146</td>
<td></td>
</tr>
<tr>
<td>EDGE8LN instruction, 147</td>
<td></td>
</tr>
<tr>
<td>EDGE8N instruction, 147</td>
<td></td>
</tr>
<tr>
<td>emulating multiple unsigned condition codes, 88</td>
<td></td>
</tr>
<tr>
<td>enable floating-point</td>
<td></td>
</tr>
<tr>
<td>See FPRS register, fef field</td>
<td></td>
</tr>
<tr>
<td>See PSTATE register, pef field</td>
<td></td>
</tr>
<tr>
<td>even parity, 6</td>
<td></td>
</tr>
<tr>
<td>exception, 6</td>
<td></td>
</tr>
<tr>
<td>exceptions</td>
<td></td>
</tr>
<tr>
<td>See also individual exceptions</td>
<td></td>
</tr>
<tr>
<td>catastrophic error, 417</td>
<td></td>
</tr>
<tr>
<td>causing traps, 417</td>
<td></td>
</tr>
<tr>
<td>clean_window, 434, 476</td>
<td></td>
</tr>
<tr>
<td>control_transfer_instruction, 434</td>
<td></td>
</tr>
<tr>
<td>cpu_mondo, 435</td>
<td></td>
</tr>
<tr>
<td>DAE_invalid_asi, 435</td>
<td></td>
</tr>
<tr>
<td>DAE_invalid_ASI (replacing SPARC V9 data_access_exception), 435</td>
<td></td>
</tr>
<tr>
<td>DAE_nc_page, 435</td>
<td></td>
</tr>
<tr>
<td>DAE_nc_page (replacing SPARC V9 data_access_exception), 435</td>
<td></td>
</tr>
<tr>
<td>DAE_nfo_page, 435</td>
<td></td>
</tr>
<tr>
<td>DAE_nfo_page (replacing SPARC V9 data_access_exception), 435</td>
<td></td>
</tr>
<tr>
<td>DAE_privilege_violation, 435</td>
<td></td>
</tr>
<tr>
<td>DAE_privilege_violation (replacing SPARC V9 data_access_exception), 435</td>
<td></td>
</tr>
<tr>
<td>DAE_side_effect_page, 436</td>
<td></td>
</tr>
<tr>
<td>DAE_side_effect_page (replacing SPARC V9 data_access_exception), 435</td>
<td></td>
</tr>
<tr>
<td>data_access_exception (SPARC V9), 435</td>
<td></td>
</tr>
<tr>
<td>definition, 417</td>
<td></td>
</tr>
<tr>
<td>dev_mondo, 436</td>
<td></td>
</tr>
<tr>
<td>disrupting_performance_event, 436</td>
<td></td>
</tr>
<tr>
<td>division_by_zero, 436</td>
<td></td>
</tr>
<tr>
<td>fill_n_normal, 436</td>
<td></td>
</tr>
<tr>
<td>fill_n_other, 436</td>
<td></td>
</tr>
<tr>
<td>fp_disabled</td>
<td></td>
</tr>
<tr>
<td>and GSR, 54</td>
<td></td>
</tr>
<tr>
<td>fp_disabled, 436</td>
<td></td>
</tr>
<tr>
<td>fp_exception_ieee_754, 436</td>
<td></td>
</tr>
<tr>
<td>fp_exception_other, 436</td>
<td></td>
</tr>
<tr>
<td>htrap_instruction, 436</td>
<td></td>
</tr>
<tr>
<td>IAE_privilege_violation, 436</td>
<td></td>
</tr>
<tr>
<td>illegal_instruction, 436</td>
<td></td>
</tr>
<tr>
<td>instruction_access_exception (SPARC V9), 437</td>
<td></td>
</tr>
<tr>
<td>interrupt_level_14</td>
<td></td>
</tr>
<tr>
<td>and SOFTINT.int_level, 55</td>
<td></td>
</tr>
<tr>
<td>interrupt_level_14, 437</td>
<td></td>
</tr>
<tr>
<td>interrupt_level_15</td>
<td></td>
</tr>
<tr>
<td>and SOFTINT.int_level, 55</td>
<td></td>
</tr>
<tr>
<td>interrupt_level_n</td>
<td></td>
</tr>
<tr>
<td>and SOFTINT.int_level, 55</td>
<td></td>
</tr>
<tr>
<td>interrupt_level_n, 422, 437</td>
<td></td>
</tr>
<tr>
<td>LDDF_mem_address_not_aligned, 437</td>
<td></td>
</tr>
<tr>
<td>LDQF_mem_address_not_aligned, 438</td>
<td></td>
</tr>
</tbody>
</table>

Index 8
mem_address_not_aligned, 437
nonresumable_error, 437
pending, 21
privileged_action, 437
privileged_opcode
and access to register-window PR state registers, 61, 65, 72, 74
and access to SOFTINT, 55
and access to SOFTINT_CLR, 56
and access to SOFTINT_SET, 56
privileged_opcode, 438
resumable_error, 438
spill_n_normal, 308, 438
spill_n_other, 308, 438
STDF_mem_address_not_aligned, 438
STQF_mem_address_not_aligned, 438
tag_overflow (deprecated), 438
trap_instruction, 438
unimplemented_LDTW, 438
unimplemented_STTW, 438
VA_watchpoint, 438
execute unit, 386
execute_state
trap processing, 432
explicit ASI, 6, 81, 398
extended word, 6
addressing, 80

F
F registers, 6, 17, 90, 365, 424
FABSd instruction, 149, 463, 464
FABSeq instruction, 149, 149, 463, 464
FABSeqs instruction, 149
FADD, 150
FADDd instruction, 150
FADDq instruction, 150, 150, 159
FADDS instruction, 150
FALIGNDATAg instruction, 151
FAND instruction (now FANDd), 215
FANDd instruction, 215
FANDNOT1 instruction (now FANDNOT1d), 215
FANDNOT1d instruction, 215
FANDNOT1S instruction, 215
FANDNOT2 instruction (now FANDNOT2d), 215
FANDNOT2d instruction, 215
FANDNOT2s instruction, 215
FANDS instruction, 215
FBA instruction, 152, 465
FBE instruction, 152, 465
FBfcc instructions, 42, 152, 436, 459, 465
FBG instruction, 152, 465
FBGE instruction, 152, 465
FBLE instruction, 152, 465
FBLE instruction, 152, 465
FBLG instruction, 152, 465
FBN instruction, 152, 465
FBNE instruction, 152, 465
FBO instruction, 152, 465
FBPA instruction, 154, 155, 465
FBPE instruction, 154, 465
FBPfcc instructions, 42, 154, 459, 465, 466
FBPG instruction, 154, 465
FBPGE instruction, 154, 465
FBPL instruction, 154, 465
FBPLE instruction, 154, 465
FBPLAG instruction, 154, 465
FBPN instruction, 154, 155, 465
FBPNE instruction, 154, 465
FBPO instruction, 154, 465
FBPU instruction, 154, 465
FBPUE instruction, 154, 465
FBPUG instruction, 154, 465
FBPUGE instruction, 154, 465
FBPUL instruction, 154, 465
FBPULE instruction, 154, 465
FBU instruction, 152, 465
FBU instruction, 152, 465
FBU instruction, 152, 465
FBU instruction, 152, 465
FBU instruction, 152, 465
FBU instruction, 152, 465
FBU instruction, 152, 465
FBU instruction, 152, 465
fnc-conditional branches, 152, 155
fnc, 6
FCHKSIM16 instruction, 156
FCMP instructions, 466
FCMP* instructions, 42, 157
FCMPEd instruction, 157, 464
FCMPEs instruction, 466
FCMPE* instructions, 42, 157
FCMPED instruction, 157, 464
FCMPEDq instruction, 157, 158, 464
FCMPEDQ16 instruction, 201
FCMPEDQ16, See FPCMPEQ16 instruction
FCMPEDQ32 instruction, 201
FCMPEDQ32, See FPCMPEQ32 instruction
FCMPEDs instruction, 157, 464
FCMPGT16 instruction, 201
FCMPGT16, See FPCMPT16 instruction
FCMPGT32 instruction, 201
FCMPGT32, See FPCMPT32 instruction
FCMPLE16 instruction, 201
FCMPLE16, See FPCMPL16 instruction
FCMPLE32 instruction, 201
FCMPLE32, See FPCMPL32 instruction
FCMPNE16 instruction, 201
FCMPNE16, See FCPMNE16 instruction
FCMPNE32 instruction, 201
FCMPNE32, See FCPMNE32 instruction
FCMPNq instruction, 157, 158, 464
FCMPNq instructions, 157, 464
FCMPLEMENT16 instruction, 203
FCMPLEMENT32 instruction, 203
FCMPNUNE16 instruction, 203
FCMPNUNE32 instruction, 203
cfn instruction field
DONE instruction, 144
PREFETCH, 288
RETRY instruction, 303
FDIVd instruction, 159
FDIVq instruction, 159
FDIVs instructions, 159
FdMULq instruction, 185, 185
FdTOI instruction, 219, 367
FdTOq instruction, 220, 220
FdTOs instruction, 220
FdTOx instruction, 219, 464
feature set, 97
fef field of FPRS register, 53
and access to GSR, 54
and fp_disabled exception, 436
branch operations, 153, 155
byte permutation, 122
comparison operations, 158, 165, 202, 204
component distance, 271, 272, 284, 285
data formatting operations, 160, 192, 206
data movement operations, 269
enabling FPU, 70
FCHKSM16 instruction, 156
floating-point operations, 137, 149, 150, 159, 163, 171, 174,
177, 180, 185, 186, 188, 190, 218, 219, 220, 222, 223, 233,
235, 238, 240, 251
integer arithmetic operations, 181, 198, 200, 209, 212, 217
integer SIMD operations, 173
logical operations, 213, 214, 215
memory operations, 231
read operations, 296, 315, 324
special addressing operations, 114, 151, 325, 327, 330, 333,
335, 341, 358
fef, See FPRS register, fef field
FEXPAND instruction, 160
FEXPAND operation, 160
FHADDd instruction, 161, 189
FHADDs instruction, 161
FHSUBd instruction, 162
FHSUBs instruction, 162
fill handler, 300
fill register window, 436
overflow/underflow, 36
RESTORE instruction, 64, 300, 439
RESTOREd instruction, 89, 440
RETRY instruction, 440
selection of, 439
trap handling, 439, 440
trap vectors, 300
window state, 64
fill_n_normal exception, 301, 306, 436, 436
fill_n_other exception, 301, 306, 436
FiTOd instruction, 163
FiTOq instruction, 163, 163, 223
FiTOs instruction, 163
fixed-point scaling, 181
FLCMPD instruction, 164
FLCMPS instruction, 164
floating point
absolute value instructions, 149
add instructions, 150
compare instructions, 42, 157, 157
condition code bits, 152
condition codes (fcc) fields of FSR register, 45, 152, 155,
157
data type, 23
divided-trap queue (FQ), 299
divide instructions, 159
divide exception, 6
divide encoding type, 44
FPREG register, 358
FSR condition codes, 42
lexicographic compares, 164
move instructions, 174
multiply instructions, 185
multiply-add
See also FMA instructions
multiply-add/subtract, 170, 186, 190
negate instructions, 188
operate (FPop) instructions, 6, 21, 44, 46, 90, 238
registers
destination F, 365
FPREG, See FPRS register
FSR, See FSR register
programming, 41
rounding direction, 43
square root instructions, 218
subtract instructions, 222
trap types, 6
IEEE_754_exception, 44, 45, 46, 48, 365, 366
invalid_fp_register, 222
unfinished_FPop, 44, 45, 48, 150, 159, 185, 218, 220, 222,
365
results after recovery, 45
unimplemented_FPop, 48, 180, 366
traps
defered, 299
precise, 299
floating-point condition codes (fcc) fields of FSR register, 424
floating-point operate (FPop) instructions, 436
floating-point trap types
IEEE_754_exception, 424, 436
floating-point unit (FPU), 7, 17
FLUSH instruction, 167
memory ordering control, 257
FLUSH instruction
memory/instruction synchronization, 166
FLUSH instruction, 166, 394
data access, 6
immediacy of effect, 168
in multiprocessor system, 166
in self-modifying code, 167
latency, 479
flush instruction memory, See FLUSH instruction
flush register windows instruction, 169
FLUSHW instruction, 169, 438
effect, 21
management by window traps, 64, 439
spill exception, 89, 169, 440
FMA instructions
fused, 170
FMADDd instruction, 170
FMADDs instruction, 170
FMEAN16 instruction, 172
Index 10
FMOCVcc instructions
conditionally moving floating-point register contents, 51
conditions for copying floating-point register contents, 87
copying a register, 42
encoding of opf<84> bits, 464
encoding of opf_cc instruction field, 465
encoding of rcond instruction field, 465
floating-point moves, 175
FPop instruction, 90
used to avoid branches, 178, 269
FMVCcdf instruction, 464
FMVCcq instruction, 464
FMVd instruction, 174, 463, 464
FMVDCfcc instructions, 175
FMVdGEZ instruction, 179
FMVdGZ instruction, 179
FMVdLEZ instruction, 179
FMVdLZ instruction, 179
FMVdNZ instruction, 179
FMVdZ instruction, 179
FMVq instruction, 174, 174, 463, 464
FMVQfccc instructions, 175, 177
FMVQqGEZ instruction, 179
FMVQqGZ instruction, 179
FMVQqcc instructions, 175, 177
FMVqLEZ instruction, 179
FMVqLZ instruction, 179
FMVqNZ instruction, 179
FMVqZ instruction, 179
FMVR instructions, 90, 465
FMVRq instructions, 180
FMVRsGZ instruction, 179
FMVRsLEZ instruction, 179
FMVRsLZ instruction, 179
FMVRsnZ instruction, 179
FMVRsZ instruction, 179
FMVs instruction, 174
FMVScce instructions, 177
FMVSfccc instructions, 175
FMVSqGEZ instruction, 179
FMVSqicc instructions, 175
FMVSqxcc instructions, 175
FMVxcc instructions, 175, 177
FMSUBd instruction, 170
FMSUBs instruction, 170
FMUL8SUx16 instruction, 181, 183
FMUL8ULx16 instruction, 181, 183
FMUL8x16 instruction, 181
FMUL8x16AL instruction, 181, 182
FMUL8x16AU instruction, 181, 182
FMULd instruction, 185
FMULD8SUx16 instruction, 181, 184
FMULD8ULx16 instruction, 181, 184
FMULq instruction, 185, 185
FMULs instruction, 185
FNADD instructions, fused, 186
FNADDe instruction, 186
FNADDS instruction, 186
FNAND instruction (now FNANDd), 215
FNANDd instruction, 215
FNANDS instruction, 215
FNEG instructions, 188
FNEGd instruction, 188, 463, 464
FNEGq instruction, 188, 188, 463, 464
FNEGs instruction, 188
FNHADDs instructions, 189
FNMAADDs instruction, 170
FNMAADDs instruction, 170
FNMSUBd instruction, 170
FNMSUBs instruction, 170
FNMUUL instructions
fused, 190
FMULd instruction, 190
FMULs instruction, 190
FNOR instruction (now FNORd), 215
FNORd instruction, 215
FNORS instruction, 215
FNNOT1 instruction (now FNNOT1d), 214
FNNOT1d instruction, 214
FNNOT1S instruction, 214
FNNOT2 instruction (now FNNOT2d), 214
FNNOT2d instruction, 214
FNNOT2S instruction, 214
FNsMULd instruction, 190
FONE instruction (now FONEd), 213
FONEd instruction, 213
FONES instruction, 213
FOR instruction (FORd), 215
FORd instruction, 215
formats, instruction, 76
FORNOT1 instruction (now FORNOT1d), 215
FORNOT1d instruction, 215
FORNOT1S instruction, 215
FORNOT2 instruction (now FORNOT2d), 215
FORNOT2d instruction, 215
FORNOT2S instruction, 215
FORS instruction, 215
fp_disabled exception, 436
absolute value instructions, 149, 150, 222
add and halve instructions, 161, 162, 189
and GSR, 54
checksum instruction, 156
FPop instructions, 90
FPREAls disabled, 53
MOVfTOi instruction, 271
MOVfTOF instruction, 272
pixel component distance, 285
PSTATE.pf not set, 53, 54, 482
with branch instructions, 153, 155
with compare instructions, 202, 204
with conversion instructions, 163, 219, 220, 223
with floating-point arithmetic instructions, 159, 171, 173, 185, 187, 191, 198, 200, 209, 217
with FMOV instructions, 174
with load instructions, 237, 241
with move instructions, 178, 180, 269
with negate instructions, 188
with store instructions, 325, 326, 327, 329, 330, 333, 335, 341, 358
**fp_exception**

Exception, 46

**fp_exception_ieee_754**

“Invalid” exception, 219

**fp_exception_ieee_754 exception**, 436

And tem bit of FSR, 43

Cause encoded in FSR.ftt, 44

FSR.aexc, 46

FSR.cexc, 47

FSR.ftt, 46

Generated by FCMP or FCMPPE, 42

And IEEE 754 overflow/underflow conditions, 46, 47

Trap handler, 366

When FSR.ns = 1, 368, 474

When FSR.tem = 0, 424

When FSR.tem = 1, 424

With floating-point add and halve instructions, 161, 162, 189

With floating-point arithmetic instructions, 150, 159, 171, 185, 187, 191, 222

**fp_exception_other**

Exception, 48, 436

Cause encoded in FSR.ftt, 44

FSUBq instruction, 222

Incorrect IEEE Std 754-1985 result, 90, 473

Never generated by FLCMP instructions, 164

Supervisor handling, 366

Trap type of unfinished_FPop, 45

When quad FPop unimplemented in hardware, 46

With floating-point arithmetic instructions, 159, 171, 185, 187, 191

FPACK instruction, 54

FPACK instructions, 192–195

FPACK16 instruction, 192, 193

FPACK16 operation, 193

FPACK32 instruction, 192, 194

FPACK32 operation, 194

FPACKFIX instruction, 192, 195

FPACKFIX operation, 195

FPADD16 instruction, 196

FPADD16S instruction, 196

FPADD32 instruction, 196

FPADD32S instruction, 196

FPADD64 instruction, 196

FPADDS16 instruction, 199

FPADDS16S instruction, 199

FPADDS32 instruction, 199

FPADDS32S instruction, 199

FPADD16s instruction, 207

FPADD32s instruction, 207

FPADD64 instruction, 207

FPADDS16 instruction, 210

FPADDS32s instruction, 210

FPCMPGT instruction, 201

FPCMPLE16 instruction, 201

FPCMPLE32 instruction, 201

FPCMPNE16 instruction, 202

FPCMPNE32 instruction, 202

FPCMPNE8 instruction, 201

FPCMPUEQ8 instruction, 203

FPCMPUGT8 instruction, 203

FPCMPULE8 instruction, 203

FPCMPUNE8 instruction, 203

FPMERGE instruction, 206

FPop, 7

FPop, See floating-point operate (FPop) instructions

FPRS register

See also floating-point registers state (FPRS) register

FPRS register, 53

ASR summary, 49

Definition, 7

Fef field, 90, 423

RDFPRS instruction, 296

FPRS register fields

dl (dirty lower fp registers), 54

du (dirty upper fp registers), 53

Fef, 53

Fef, See also fef field of FPRS register

FPSUB16 instruction, 207

FPSUB16s instruction, 207

FPSUB32s instruction, 207

FPSUB64 instruction, 207

FPSUBS16 instruction, 210

FPSUBS16S instruction, 210

FPSUBS32s instruction, 210

FPCMPGT instruction, 201

FPCMPLE16 instruction, 201

FPCMPLE32 instruction, 201

FPCMPNE16 instruction, 202

FPCMPNE32 instruction, 202

FPCMPNE8 instruction, 201

FPCMPUEQ8 instruction, 203

FPCMPUGT8 instruction, 203

FPCMPULE8 instruction, 203

FPCMPUNE8 instruction, 203

FPMERGE instruction, 206

FPop, 7

FPRS register

See also floating-point registers state (FPRS) register

FPRS register, 53

ASR summary, 49

Definition, 7

Fef field, 90, 423

RDFPRS instruction, 296

FPRS register fields

dl (dirty lower fp registers), 54

du (dirty upper fp registers), 53

Fef, 53

Fef, See also fef field of FPRS register

FPCSUB16 instruction, 207

FPCSUB16s instruction, 207

FPCSUB32s instruction, 207

FPCSUB64 instruction, 207

FPCSUBS16 instruction, 210

FPCSUBS16S instruction, 210

FPCSUBS32s instruction, 210

FPCMPGT instruction, 201

FPCMPLE16 instruction, 201

FPCMPLE32 instruction, 201

FPCMPNE16 instruction, 202

FPCMPNE32 instruction, 202

FPCMPNE8 instruction, 201

FPCMPUEQ8 instruction, 203

FPCMPUGT8 instruction, 203

FPCMPULE8 instruction, 203

FPCMPUNE8 instruction, 203

FPMERGE instruction, 206

FPop, 7
Index

compliance with IEEE Std 754-1985, 48
LDFSR instruction, 238
reading/writing, 42
values in ft field, 44
writing to memory, 330, 341
FSRC1 instruction (now FSRC1d), 214
FSRC1d instruction, 214
FSRC1S instruction, 214
FSRC2 instruction (now FSRC2d), 214
FSRC2d instruction, 214
FSRC2S instruction, 214
FsTOd instruction, 220
FsTOi instruction, 219, 367
FsTOq instruction, 219, 220
FsTOx instruction, 219, 463, 464
FSUBd instruction, 222
FSUBq instruction, 222, 222
FSUBs instruction, 222
functional choice, implementation-dependent, 472
fused FMA instructions, 170
fused FNADD instructions, 186
fused FNMUL instructions, 190
FXNOR instruction (now FXNORd), 215
FXNORd instruction, 215
FXNORS instruction, 215
FXOR instruction (now FXORd), 215
FXORd instruction, 215
FXORS instruction, 215
FxTOd instruction, 223, 464
FxTOq instruction, 223, 464
FxTOs instruction, 223, 464
FZERO instruction (now FZEROd), 213
FZEROd instruction, 213
GSR (general status) register
  ASR summary, 49
  mask values with CMASK instructions, 136

H
halfword, 7
  alignment, 18, 77, 383
  data format, 23
  hardware
dependency, 472
  traps, 425
  hardware trap stack, 21
  htrap_instruction exception, 350, 436
hyperprivileged, 7

I
i (integer) instruction field
  arithmetic instructions, 278, 281, 310, 318, 353, 355
  floating point load instructions, 233, 235, 238, 251
  flush memory instruction, 166
  flush register instruction, 169
  jump-and-link instruction, 226
  load instructions, 227, 242, 243, 244, 246
  logical operation instructions, 115, 281, 363
  move instructions, 268, 270
  POPC, 286
  PREFETCH, 288
  RETURN, 305
I/O
  access, 382
  memory, 381
  memory-mapped, 382
IAE_privilege_violation exception, 436
  and TTE.p, 452
IAE_unauth_access exception, 452
IEEE 754, 7
IEEE Std 754-1985, 7, 13, 43, 45, 47, 48, 90, 365, 473
IEEE_754_exception floating-point trap type, 7, 44, 45, 46, 48,
  365, 366, 424, 436
IEEE-754 exception, 7
IER register (SPARC V8), 358
illegal_instruction exception, 132, 169, 436
  attempt to write in nonprivileged mode, 57
  DONE/RETRY, 145, 304, 305
  ILLTRAP, 224
  not implemented in hardware, 106
  POPC, 287
  PREFETCH, 294
  RETURN, 306
  with BPr instruction, 125
  with branch instructions, 124, 126
  with CASA and CASXA instructions, 131, 281
  with CBecond instructions, 135
  with DONE instruction, 145
  with FMOV instructions, 174
  with FMOVcc instructions, 178
  with FMOVr instructions, 180
  with load instructions, 38, 232, 234, 244, 247, 252, 411

G
  general status register, See GSR (general status) register
generating constants, 312
GL register, 73
  access, 73
during trap processing, 432
  function, 73
reading with RDPR instruction, 298, 360
relationship to TL, 73
restored during RETRY, 144, 303
SPARC V9 compatibility, 71
  and TSTATE register, 66
  value restored from TSTATE[TL], 74
writing to, 73
global level register, See GL register
global registers, 14, 17, 32, 33, 473
graphics status register, See GSR (general status) register
GSR (general status) register
  fields
    align, 54
    im (interval mode) field, 54
    irnd (rounding), 54
    mask, 54
    scale, 54
with move instructions, 269, 270, 271, 272
with read hyperprivileged register instructions, 298
with read instructions, 296, 298, 361, 475
with store instructions, 237, 326, 330, 337, 338, 339, 340, 341
with Tcc instructions, 350
with TPG register, 65
with TSTATE register, 66
with write instructions, 358, 361
write to ASR 5, 53
write to STICK register, 57
write to TICK register, 52
ILLTRAP instruction, 224, 436
imm_asi instruction field
explicit ASI, providing, 81
floating point load instructions, 243
LDSTUB, 242
load instructions, 243, 244, 246
PREFETCH, 288
immediate CTI, 75
I-MMU
and instruction prefetching, 383
IMMU
context register usage, 455
IMPDEP1 instructions, 90
IMPDEP2 instructions, 90
IMPDEP2A instructions, 476
IMPDEP2B instructions, 90
implementation, 7
implementation dependency, 471
implementation dependent, 7
implementation note, 3, 4
implementation-dependent functional choice, 472
implicit ASI, 7, 81, 398
implicit ASI memory access
LDFSR, 238
LDSTUB, 242
load fp instructions, 233, 235
load integer doubleword instructions, 244
load integer instructions, 227
STD, 337
STFSR, 330
store floating-point instructions, 325, 341
store integer instructions, 319
SWAP, 343
implicit byte order, 69
in registers, 33, 35, 307
inccel synthetic instructions, 492
inexact accrued (nxa) bit of aexc field of FSR register, 366
inexact current (nxc) bit of cexc field of FSR register, 366
inexact mask (nxm) field of FSR.t, 47
inexact quotient, 310, 353
infinity, 367
initiated, 7
input/output (I/O) locations
access by nonprivileged code, 473
behavior, 382
contents and addresses, 473
identifying, 478
order, 382
semantics, 479
value semantics, 382
instruction fields, 7
See also individual instruction fields
definition, 7
instruction group, 7
instruction MMU, See I-MMU
instruction prefetch buffer, invalidation, 167
instruction set architecture (ISA), 7, 7, 14
instruction_access_exception exception (SPARC V9), 437
instruction_va_watchpoint exception, 437
instructions
32-bit wide, 13
alignment, 77
alignment, 18, 114, 383
arithmetic, integer
addition, 107, 108, 346
division, 20, 278, 310, 353
multiplication, 20, 278, 318, 355
subtraction, 342, 351
tagged, 20
array addressing, 117
atomic
CASA/CASXA, 131
load twin extended word from alternate space, 249
load-store, 77, 131, 133, 242, 243, 343, 344
load-store unsigned byte, 242, 243
successful loads, 227, 228, 245, 347
successful stores, 319, 320
branch
branch if contents of integer register match condition, 125
branch on floating-point condition codes, 152, 154
branch on integer condition codes, 120, 123
cache, 388
casting illegal instruction, 224
cast and swap, 131, 133
cast, 83, 342
cast
control-transfer (CTIs), 20, 144, 303
conversion
convert between floating-point formats, 220
convert floating-point to integer, 219
convert integer to floating-point, 163, 223
floating-point to integer, 367
count of number of bits, 286
crypto
AES, 109
Camellia, 128
DES, 140
MD5 hash operation, 255
MONTMUL, 259
MONTSQR, 263
MPMUL, 273
SHA1, SHA256, SHA512 hash operations, 313
device handling, 147
depth, 77
floating point
compare, 42, 157
floating-point add, 150
floating-point divide, 159
floating-point load, 77, 233
floating-point load from alternate space, 235
floating-point load state register, 233, 251
floating-point move, 174, 175, 179
floating-point operate (FPop), 21, 238
floating-point square root, 218
floating-point store, 77, 325
floating-point store to alternate space, 327
floating-point subtract, 222
operate (FPop), 44, 46
short floating-point load, 240
short floating-point store, 335
status of floating-point load, 238

flush instruction memory, 166
flush register windows, 169
formats, 76
integer multiply-add, 205
jump and link, 20, 226
loads
  block load, 230
  floating point, See instructions: floating point
  integer, 77
simultaneously addressing doublewords, 344
  unsigned byte, 131, 242
  unsigned byte to alternate space, 243

logical operations
  64-bit/32-bit, 214, 215
  AND, 115
  logical 1-operand ops on F registers, 213
  logical 2-operand ops on F registers, 214
  logical 3-operand ops on F registers, 215
  logical XOR, 363
OR, 281
memory, 394
moves
  floating point, See instructions: floating point
  move integer register, 267, 270
on condition, 14
MOVXTOD, 271, 272
ordering MEMBAR, 83
PAUSE, 283
permuting bytes specified by GSR.mask, 122
pixel component distance, 284, 284, 285, 285
pixel formatting (PACK), 192
prefetch data, 288
read privileged register, 298
read state register, 20, 295
register window management, 21
reordering, 386
reserved, 91
reserved fields, 105
RETRY
  and restartable deferred traps, 421
RETURN vs. RESTORE, 305
sequencing MEMBAR, 83
set high bits of low word, 312
set interval arithmetic mode, 315
setting GSR.mask field, 122
shift, 19
shift, 316
shift count, 316
SIMD, 10
simultaneous addressing of doublewords, 344
stores
  block store, 322
  floating point, See instructions: floating point
  integer, 77, 319
  integer (except doubleword), 319
  integer into alternate space, 320
  partial, 332
  unsigned byte, 131
  unsigned byte to alternate space, 243
  unsigned bytes, 242
swap R register, 343, 344
synthetic (for assembly language programmers), 490–492
tagged addition, 346
test-and-set, 392
timing, 106
trap on integer condition codes, 348
write privileged register, 360
write state register, 358
XMULX, 362
XMULXHI, 362
integer unit (IU)
  condition codes, 51
definition, 7
description, 16
interrupt
  enable (ie) field of PSTATE register, 422, 423
  level, 72
  request, 7, 21, 417
interrupt_level_14 exception, 55, 437
  and SOFTINT.int_level, 55
interrupt_level_15 exception
  and SOFTINT.int_level, 55
interrupt_level_n exception, 422, 437
  and SOFTINT.reg, 55
  and SOFTINT.int_level, 55
inter-strand operation, 7
intra-strand operation, 7
invalid accrued (nva) bit of aexc field of FSR register, 48
invalid ASI
  and DAE_invalid_asi, 435
invalid current (nvc) bit of cexc field of FSR.ser, 48, 367
invalid mask (nvm) field of FSR.tem, 47, 367
invalid_exception exception, 219
invalid_fp_register floating-point trap type, 180, 188, 222
INVALW instruction, 225
iprefetch synthetic instruction, 491
ISA, 7
ISA, See instruction set architecture
issue unit, 386, 386
issued, 8
italic font, in assembly language syntax, 485
IU, 8
ixc synthetic instructions, 492

J
jmp synthetic instruction, 490
JMPL instruction, 226
  computing target address, 20
does not change CWP, 35
  mem_address_not_aligned exception, 437
reexecuting trapped instruction, 305
jump and link, See JMPL instruction

L
LD instruction (SPARC V8), 227
LDBLOCKF instruction, 230, 410
LDBLOCKF instruction, DAE_nc_page exception, 435
LDD instruction (SPARC V8 and V9), 244
LDDA instruction, 410
LDDA instruction (SPARC V8 and V9), 247
LDDF instruction, 77, 233, 437
LDDF_mem_address_not_aligned exception, 233, 437
  address not doubleword aligned, 477
  address not quadword aligned, 478
LDDF/LDDFA instruction, 77
load instruction with partial store ASI and misaligned address, 237
  with load instructions, 235, 411
  with store instructions, 411
LDDF_mem_not_aligned exception, 41
LDDFA instruction, 235, 334
  alignment, 77
  ASIs for fp load operations, 411
  behavior with block store with Commit ASIs, 236
  behavior with partial store ASIs, 234–??, 237, 237–??, 251–??, 411–??
  causing LDDF_mem_address_not_aligned exception, 77, 437
for block load operations, 410
  used with ASIs, 410
LDF instruction, 41, 233
LDFA instruction, 41, 235
LDFSR instruction, 42, 44, 238, 437
LDQF instruction, 233, 438
LDQF_mem_address_not_aligned exception, 233, 438
  address not quadword aligned, 478
LDQF/LDQFA instruction, 78
LDQF_mem_not_aligned exception, 41
LDQF instruction, 235, 334
LDQF_mem_not_aligned exception, 41
LDQFA instruction, 235
LDSB instruction, 227
LDSBA instruction, 228
LDSH instruction, 227
LDSHA instruction, 228
LDHORF instruction, 240
LDSTUB instruction, 77, 242, 243, 392, 393
  and DAE_nc_page exception, 435
  hardware primitives for mutual exclusion of LDSTUB, 392
LDSTUBA instruction, 242, 243
  alternate space addressing, 18
  and DAE_nc_page exception, 435
  hardware primitives for mutual exclusion of LDSTUBA, 392
LDSW instruction, 227
LDSWA instruction, 228
MAXPG, 17, 32, 33, 72, 73, 73, 482
MAXPT

and MAXPG, 73
instances of TNPc register, 65
instances of TPC register, 65
instances of TSTATE register, 66
instances of TT register, 67
may (keyword), 8
mem_address_not_aligned exception, 132, 437
generated by virtual processor, 237
JMPL instruction, 226
LDTX, 409, 410
load instruction with partial store ASI and misaligned address, 237
RETURN, 305, 306
with CASA instruction, 132
with swap instructions (deprecated), 343, 345
MEMBAR
#Sync
  semantics, 258
instruction
  atomic operation ordering, 393
  FLUSH instruction, 166, 394
  functions, 256, 392–394
  memory ordering, 257
  memory synchronization, 83
  side-effect accesses, 383
  STBAR instruction, 257
mask encodings
  #LoadLoad, 256, 393
  #LoadStore, 256, 393
  #Lookaside, 394
  #Lookaside (deprecated), 257
  #MemIssue, 257, 394
  #StoreLoad, 256, 393
  #StoreStore, 256, 393
  #Sync, 257, 394
predefined constants
  #LoadLoad, 489
  #LoadStore, 489
  #Lookaside (deprecated), 489
  #MemIssue, 489
  #StoreLoad, 489
  #StoreStore, 489
  #Sync, 489
MEMBAR
#Lookaside, 390
#StoreLoad, 390
membar_mask, 489
MemIssue predefined constant, 489
memory
  access instructions, 17, 77
  alignment, 383
  atomic operations, 392
  atomicity, 479
  cached, 381
  coherence, 383, 479
  coherency unit, 384
  data, 394
  instruction, 394
  location, 381
  models, 381
  ordering unit, 384
  real, 382
  reference instructions, data flow order constraints, 387
  synchronization, 257
  virtual address, 381
  virtual address 0, 395
Memory Management Unit
  definition, 8
Memory Management Unit, See MMU
memory model
  mode control, 389
  partial store order (PSO), 389
  relaxed memory order (RMO), 258, 389
  sequential consistency, 389
  strong, 389
  total store order (TSO), 258, 389, 390
  weak, 389
memory model (mm) field of PSTATE register, 69
memory order
  pending transactions, 388
  program order, 386
memory_model (mm) field of PSTATE register, 389
memory-mapped I/O, 382
metrics
  for architectural performance, 414
  for implementation performance, 414
  See also performance monitoring hardware
microcore, 8
MMU
  accessing registers, 456
  contexts, 448
  definition, 8
  page sizes, 445
  SPARC V9 compliance, 456
mode
  nonprivileged, 15
  privileged, 16, 65, 385
MONTMUL instruction, 259
MONTSQR instruction, 263
motion estimation, 284, 285
MOVA instruction, 267
MOVCC instruction, 267
MOVcc instructions, 267
  conditionally moving integer register contents, 51
  conditions for copying integer register contents, 87
  copying a register, 42
  encoding of cond field, 465
  encoding of opf_cc instruction field, 465
  used to avoid branches, 178, 269
MOVCS instruction, 267
move conditionally by BSHUFFLE instruction, 137
move floating-point register if condition is true, 175
move floating-point register if contents of integer register...
satisfy condition, 179
MOVE instruction, 267
move integer register if condition is satisfied
instructions, 267
move integer register if contents of integer register satisfies
condition instructions, 270
move on condition instructions, 14
MOVFA instruction, 268
MOVFE instruction, 268
MOVFG instruction, 268
MOVFLT instruction, 268
MOVFLE instruction, 268
MOVFN instruction, 268
MOVFNZ instruction, 268
MOVFGE instruction, 268
MOVFLE instruction, 268
MOVFUG instruction, 268
MOVFUGE instruction, 268
MOVFNE instruction, 268
MOVFUNE instruction, 268
MOVFUG instruction, 268
multiplication, very large numbers, 356
multiplication, very large numbers, 362
multiply instructions, 278, 318, 355
multiple unsigned condition codes, emulating, 88
multiple unsigned condition codes, emulating, 88
multiply instructions, 278, 318, 355
multiply-add instructions (fused), 170
multiply-add, See FMA instructions
multiply-subtract instructions (fused), 170
multiply-subtract, See FMA instructions
multiple processor synchronization instructions, 131, 343, 344
multiple processor system, 8, 166, 292, 343, 344, 388, 479
MULX instruction, 278
must (keyword), 8

\textit{N}

N superscript on instruction name, 94

\textit{N_REG_WINDOWs}, 9

integer unit registers, 17, 473
RESTORE instruction, 300
SAVE instruction, 307
value of, 32, 61
NaN (not-a-number)
conversion to integer, 367
converting floating-point to integer, 219
signalling, 42, 157, 220
\textit{neg} synthetic instructions, 492
negative add instructions (fused), 186, 190
negative infinity, 367
negative multiply-add instructions (fused), 170
negative multiply-subtract instructions (fused), 170
nested traps, 14
next program counter register, \textit{See NPC} register
NFO, 8
noncacheable
accesses, 382
nonfaulting load, 8, 386
nonfaulting loads
behavior, 395
use by optimizer, 395
non-faulting-only page
illegal access to, 435
non-faulting-only page, illegal access to
and \textit{TTE} nfo, 450
nonleaf routine, 226
nonprivileged, 8
mode, 5, 8, 15, 16, 44
software, 53
\textit{nonresumable\_error} exception, 437
nonstandard floating-point, \textit{See} floating-point status register
\textit{(FSR)} NS field
nontranslating ASI, 8, 247, 340, 398
nonvirtual memory, 293
NOP instruction, 120, 152, 155, 279, 289, 349
normal traps, 425
NORMALW instruction, 280
\textit{not} synthetic instructions, 492
note
architectural direction, 4
compatibility, 4
general, 3
implementation, 3
programming, 3
NPC (next program counter) register, 53
and \textit{PSTATE}.tct, 68
control flow alteration, 11
definition, 8
DONE instruction, 144
instruction execution, 75
relation to TNPC register, 65
RETURN instruction, 303
saving after trap, 21
nucleus context, 168
Nucleus Context register, 458
nucleus software, 9
NUMA, 9
\textit{nvm} (invalid mask) field of \textit{FSR}.\textit{tem}, 47, 367
NWIN, See N_REG_WINDOWS

nxm (inexact mask) field of FSRtem, 47

O
oclet, 9
odd parity, 9
ofm (overflow mask) field of FSRtem, 47
op3 instruction field, 131
  arithmetic instructions, 107, 124, 125, 278, 310, 318, 353, 355
  floating point load instructions, 233, 235, 238, 251
  flush instructions, 166, 169
  jump-and-link instruction, 226
  load instructions, 227, 242, 243, 244, 246
  logical operation instructions, 115, 281, 363
  PREFETCH, 288
  RETURN, 305
opcode
  definition, 9
opf instruction field
  floating point arithmetic instructions, 150, 159, 185, 218
  floating point compare instructions, 157
  floating point conversion instructions, 219, 220, 223
  floating point instructions, 149
  floating point integer conversion, 163
  floating point move instructions, 174
  floating point negate instructions, 188
opf_cc instruction field
  floating point move instructions, 175
  move instructions, 465
opf_low instruction field, 175
optional, 9
OR instruction, 281
ORcc instruction, 281
ordering MEMBAR instructions, 83
ordering unit, memory, 384
ORN instruction, 281
ORNcc instruction, 281
OTHERWIN instruction, 282
OTHERW (other windows) register, 63
  FLUSHW instruction, 169
  keeping consistent state, 64
  modified by OTHERW instruction, 282
  partitioned, 64
  range of values, 61, 479
rd designation for WRPR instruction, 360
rs1 designation for RDPR instruction, 298
SAVE instruction, 307
zeroed by INVALIDW instruction, 225
zeroed by NORMALW instruction, 280
OTHERWIN register trap vectors
  fill/spill traps, 439
  handling spill/fill traps, 439
  selecting spill/fill vectors, 439
out register #7, 37
out registers, 33, 35, 307
overflow
  bits
    (v) in condition fields of CCR, 83
    accrued (ofa) in aexc field of FSR register, 48
    current (ofc) in cexc field of FSR register, 48
    causing spill trap, 439
    tagged add/subtract instructions, 83
    overflow mask (ofm) field of FSRtem, 47
P
p (predict) instruction field of branch instructions, 123, 125, 155
P superscript on instruction name, 94
packed-to-planar conversion, 206
packing instructions, See FPACK instructions
page fault, 293
page table entry (PTE), See translation table entry (TTE)
parity, even, 6
parity, odd, 9
partial store instructions, 332, 411
partial store order (PSO) memory model, 389, 389
partitioned
  addition with saturation, 199
  additions, 196, 207
  shift instructions, 216
  subtraction with saturation, 210
  subtractions, 207
PASI superscript on instruction name, 94
PASR superscript on instruction name, 94
PAUSE instruction, 283
PAUSE register, 60
PC (program counter) register, 9, 49, 53
  after instruction execution, 75
  and PSTATE.tct, 68
  CALL instruction, 127
  changed by NOP instruction, 279
  copied by JMPL instruction, 226
  saving after trap, 21
  set by DONE instruction, 144
  set by RETRY instruction, 303
  Trap Program Counter register, 65
P_dis superscript on instruction or register name, 94
PDIST instruction, 284
PDISTN instruction, 285
pf field of PSTATE register
  and access to GSR, 54
  and fp_disabled exception, 436
  and FPop instructions, 90
  branch operations, 153, 155
  byte permutation, 122
  comparison operations, 158, 165, 202, 204
  component distance, 271, 272, 284, 285
  data formatting operations, 160, 192, 206
  data movement operations, 269
  enabling FPU, 53
  FCHKSM16 instruction, 156
  floating-point operations, 137, 149, 150, 159, 163, 171, 174, 177, 180, 185, 186, 188, 190, 218, 219, 220, 222, 223, 233, 235, 238, 240, 251
  integer arithmetic operations, 181, 198, 200, 206, 207, 209, 212, 217
  integer SIMD operations, 173
logical operations, 213, 214, 215
memory operations, 231
read operations, 296, 315, 324
special addressing operations, 114, 151, 325, 327, 330, 333, 335, 341, 358
trap control, 423

pef, See PSTATE, pef field
performance monitoring hardware
accuracy requirements, 414
classes of data reported, 414
counters and controls, 415
high-level requirements, 413
kinds of user needs, 413
See also instruction sampling
physical processor, 9
PIL (processor interrupt level) register, 72
interrupt conditioning, 422
interrupt request level, 424
interrupt_level_n, 437
specification of register to read, 298
specification of register to write, 360
trap processing control, 423
pipeline, 9
pipeline draining of CPU, 65
pipeline draining of virtual processor, 61
pixel instructions
compare, 201, 203
component distance, 284, 284
formatting, 192
planar-to-packed conversion, 206
POPC instruction, 286
positive infinity, 367
precise floating-point traps, 299
precise trap, 420
conditions for, 420
software actions, 420
vs. disrupting trap, 421
predefined constants
LoadLoad, 489
lookaside (deprecated), 489
MemIssue, 489
StoreLoad, 489
StoreStore, 489
Sync, 489
predict bit, 125
prefetch
for one read, 292
for one write, 292
for several reads, 292
for several writes, 292
page, 293
to nearest unified cache, 293
prefetch data instruction, 288
PREFETCH instruction, 77, 288, 476
prefetch_func, 489
PREFETCHA instruction, 288, 476
and invalid ASI or VA, 435
prefetchable, 9
Primary Context ID 0, 457
Primary Context ID 1, 457

Primary Context register, 458
priority of traps, 423, 432
privileged_action exception
read from TICK register when access disabled, 52
privilege violation, and DAE_privilegeViolation exception, 435
privileged, 9
mode, 16, 65
registers, 65
software, 15, 36, 44, 70, 82, 169, 425, 476
privileged (priv) field of PSTATE register, 71, 132, 145, 228, 231, 235, 236, 243, 247, 320, 324, 328, 339, 344, 385, 437, 438
privileged mode, 9
privileged_action exception, 132, 437
accessing restricted ASIs, 385
read from TICK register when access disabled, 52, 296
restricted ASI access attempt, 82, 397
TICK register access attempt, 51
with CASA instruction, 132
with load alternate instructions, 228, 231, 236, 243, 247, 320, 324, 328, 339, 344
with load instructions, 235
with RDAsr instructions, 297
with read instructions, 297
with store instructions, 329
with swap instructions, 345
privileged_opcode exception, 52, 438
DONE instruction, 145
RETRY instruction, 304
SAVED instruction, 309
with DONE instruction, 145, 298, 303, 361
with write instructions, 361
WRPR in nonprivileged mode, 52
processor, 9
execute unit, 386
issue unit, 386, 386
privilege-mode transition diagram, 419
reorder unit, 386
self-consistency, 386
processor cluster, See processor module
processor consistency, 388, 391
processor interrupt level register, See PIL register
processor self-consistency, 386, 390
processor state register, See PSTATE register
processor states
execute_state, 432
program counter register, See PC register
program counters, saving, 417
program order, 386, 386, 387
programming note, 3
PSO, See partial store order (PSO) memory model
PSR register (SPARC V8), 358
PSTATE register
entering privileged execution mode, 417
restored by RETRY instruction, 144, 303
saved after trap, 417
saving after trap, 21
specification for RDPR instruction, 298
specification for WRPR instruction, 360
and TSTATE register, 66
PSTATE register fields

ag
  unimplemented, 71

am
  CALL instruction, 127
description, 70
masked/unmasked address, 144, 226, 303, 305
cle
  and implicit ASIs, 81
description, 69
masked/unmasked address, 144, 226, 303, 305
ie
  description, 71
  enabling disrupting traps, 422
  interrupt conditioning, 422
  masking disrupting trap, 426
mm
  description, 69
  implementation dependencies, 69, 389, 478
  reserved values, 69
pef
  and FPRS.fef, 70
description, 70
See also pef field of PSTATE register
priv
  access to register-window PR state registers, 65
  accessing restricted ASIs, 385
description, 71
determining mode, 8, 9, 452
tct
  branch instructions, 121, 124, 125, 135, 153, 155
  CALL instruction, 127
description, 68
  DONE instruction, 145
  JMPL instruction, 226
  RETRY instruction, 303, 349
  RETURN instruction, 305
tle
  description, 69
PTE (page table entry), See translation table entry (TTE)

Q
quadword, 9
  alignment, 18, 77, 383
data format, 23
quiet NaN (not-a-number), 42, 157

R
R register, 9
  #15, 37
  special-purpose, 37
  alignment, 244, 247
rational quotient, 353
R-A-W, See read-after-write memory hazard
rcond instruction field
  branch instructions, 125
coding of, 465
move instructions, 270
rd (rounding), 10
rd instruction field, 131
  arithmetic instructions, 107, 124, 125, 278, 310, 318, 353, 355
  floating point arithmetic, 150
  floating point arithmetic instructions, 159, 185, 218
  floating point conversion instructions, 219, 220, 223
  floating point integer conversion, 163
  floating point load instructions, 233, 235, 238, 251
  floating point move instructions, 174, 175
  floating point negate instructions, 188
  floating-point instructions, 149
  jump-and-link instruction, 226
  load instructions, 227, 242, 243, 244, 246
  logical operation instructions, 115, 281, 363
  move instructions, 268, 270
POPC, 286
RDASI instruction, 49, 51, 295
RDAsr instruction, 295
reading I/O registers, 19
  implementation dependencies, 296, 475
  reading ASRs, 48
RDCCCR instruction, 49, 50, 295
RDCFRR instruction, 49, 295
RDFFRR instruction, 49, 53, 295
RDFGSR instruction, 49, 54, 295
RDPC instruction, 49, 295
reading PC register, 53
RDPR instruction, 49, 298
reading GL register, 73
  accessing non-register-window PR state registers, 65
  accessing register-window PR state registers, 61
description, 71
effect on TNPC register, 66
effect on TPC register, 65
effect on TSTATE register, 67
effect on TT register, 67
reading privileged registers, 65
reading PSTATE register, 68
reading the TICK register, 52
registers read, 298
RDSOFTINT instruction, 49, 55, 295
RDSTICK instruction, 49, 57, 295
RDSTICK_CMPR instruction, 49, 295
RDTICK instruction, 49, 52, 295
RDY instruction, 50, 296
read ancillary state register (RDasr) instructions, 295
read state register instructions, 20
real address, 10
real memory, 381, 382
reference MMU, 485
reg, 485
reg_or_imm, 489
reg_plus_imm, 488
regaddr, 489
register reference instructions, data flow order
  constraints, 387
register window, 33, 34

Index  21
register window management instructions, 21
register windows
  clean, 63, 64, 88, 434, 439, 440
  fill, 36, 64, 88, 89, 300, 436, 439, 440
  management of, 15
  overlapping, 35–37
  spill, 36, 64, 88, 89, 307, 438, 439, 440
registers
  See also individual register (common) names
  accessing MMU registers, 456
  address space identifier (ASI), 385
  ASI (address space identifier), 51
  CFR, 59
  CFR (compatibility feature register), 435
  chip-level multithreading, See CMT
  clean windows (CLEANWIN), 62
  clock-tick (TICK), 437
  current window pointer (CWP), 61
  F (floating point), 365, 424
  floating-point, 17
  floating-point registers state (FPRS), 53
  floating-point state (FSR), 42
  global, 14, 17, 32, 33, 33, 473
  global level (GL), 73
  iER (SPARC V8), 358
  in, 33, 35, 307
  local, 33, 35
  next program counter (NPC)
    and PSTATE.tct, 68
  next program counter (NPC), 53
  other windows (OTHERWIN), 63
  out, 33, 35, 307
  out #7, 37
  PAUSE, 60
  processor interrupt level (PIL)
    and SOFTINT, 55
  processor interrupt level (PIL), 72
  program counter (PC)
    and PSTATE.tct, 68
  program counter (PC), 53
  PSR (SPARC V8), 358
  R register #15, 37
  renaming mechanism, 387
  restorable windows (CANRESTORE), 62, 63
  savable windows (CANSAVE), 62
  scratchpad
    privileged, 411
  SOFTINT, 49
  SOFTINT_CLR pseudo-register, 49, 56
  SOFTINT_SET pseudo-register, 49, 55
  STICK
    and STICK_CMPR, 57
  STICK_CMPR
    ASR summary, 49
    int_dis field, 55, 58
    stick_cmp field, 58
    and system software trapping, 57
    STICK_CMPR, 57
  TBR (SPARC V8), 358
  TICK, 52
  trap base address (TBA), 67
  trap base address, See registers: TBA
  trap level (TL), 71
  trap level, See registers: TL
  trap next program counter (TNPC), 65
  trap next program counter, See registers: TNPC
  trap program counter (TPC), 65
  trap program counter, See registers: TPC
  trap state (TSTATE), 66
  trap state, See registers: TSTATE
  trap type (TT), 67, 425
  trap type, See registers: TT
  VA_WATCHPOINT, 437, 438
  visible to software in privileged mode, 65–74
  WIM (SPARC V8), 358
  window state (WSTATE), 63
  window state, See registers: WSTATE
  Y (32-bit multiply/divide), 50
  relaxed memory order (RMO) memory model, 258, 389
  renaming mechanism, register, 387
  reorder unit, 386
  reordering instruction, 386
  reserved, 10
    fields in instructions, 105
    register field, 32
  reset
    reset trap, 421
  restartable deferred trap, 420
  restorable windows register, See CANRESTORE register
  RESTORE instruction, 35, 300–301
    actions, 88
    and current window, 37
    decrementing CWP register, 35
    fill trap, 436, 439
    followed by SAVE instruction, 36
    managing register windows, 21
    operation, 300
    performance trade-off, 300, 307
    and restorable windows (CANRESTORE) register, 62
    restoring register window, 300
    role in register state partitioning, 64
  restore synthetic instruction, 491
  RESTORED instruction, 89, 302
  creating inconsistent window state, 302
  fill handler, 300
  fill trap handler, 89, 440
  register window management, 21
  restricted, 10
  restricted address space identifier, 82
  restricted ASI, 385, 397
  resumable_error exception, 438
  ret/ret1 synthetic instructions, 491
  RETRY instruction, 303
    and restartable deferred traps, 421
    effect on TNPC register, 66
    effect on TPC register, 65
    effect on TSTATE register, 67
    generating illegal_instruction exception, 437
modifying CCR.xcc, 51
reexecuting trapped instruction, 440
restoring gl value in GL, 74
return from trap, 417
returning to instruction after trap, 423
target address, return from privileged traps, 20
RETURN instruction, 305–306
computing target address, 20
fill trap, 436
mem_address_not_aligned exception, 437
operation, 305
reexecuting trapped instruction, 305
RETURN vs. RESTORE instructions, 305
RMO, 10
RMO, See relaxed memory order (RMO) memory model
rounding
for floating-point results, 43
in signed division, 310
rounding direction (rd) field of FSR register, 150, 159, 185, 218, 219, 220, 222, 223
routine, nonleaf, 226
rs1 instruction field, 131
arithmetic instructions, 107, 124, 125, 278, 310, 318, 353, 355
branch instructions, 125
compare and branch instructions, 133
floating point arithmetic instructions, 150, 159, 185
floating point compare instructions, 157
floating point load instructions, 233, 235, 238, 251
flush memory instruction, 166
jump-and-link instruction, 226
load instructions, 227, 242, 243, 244, 246
logical operation instructions, 115, 281, 363
move instructions, 270
PREFETCH, 288
RETURN, 305
rs2 instruction field, 131
arithmetic instructions, 107, 124, 125, 278, 281, 310, 318, 353, 355
branch instructions, 125
compare and branch instructions, 133
floating point arithmetic instructions, 150, 159, 185
floating point compare instructions, 157
floating point conversion instructions, 219, 220, 223
floating point instructions, 149
floating point integer conversion, 163
floating point load instructions, 233, 235, 238, 251
floating point move instructions, 174, 175
floating point negate instructions, 188
flush memory instruction, 166
jump-and-link instruction, 226
load instructions, 227, 244, 246
logical operation instructions, 115, 363
move instructions, 268, 270
POPC, 286
PREFETCH, 288
RTO, 10
RTS, 10
S
savable windows register, See CANSAVE register
SAVE instruction, 35, 307
actions, 88
after RESTORE instruction, 305
clean_window exception, 434, 439
and current window, 37
decrementing CWP register, 35
effect on privileged state, 307
leaf procedure, 226
and local/out registers of register window, 36
managing register windows, 21
no clean window available, 63
number of usable windows, 62
operation, 307
performance trade-off, 307
role in register state partitioning, 64
and savable windows (CANSAVE) register, 62
spill trap, 438, 439, 440
save synthetic instruction, 491
SAVED instruction, 89, 309
creating inconsistent window state, 309
register window management, 21
spill handler, 308, 309
spill trap handler, 89, 440
scaling of the coefficient, 181
scratchpad registers
privileged, 411
SDIVX instruction, 278
Secondary Context ID 0, 457
Secondary Context ID 1, 457
Secondary Context register, 458
self-consistency, processor, 386
self-modifying code, 166, 167, 393
sequencing MEMBAR instructions, 83
sequential consistency, 383, 389
sequential consistency memory model, 389
SETHI instruction, 83, 312
creating 32-bit constant in R register, 19
and NOP instruction, 279
with rd = 0, 312
set? synthetic instructions, 491
SHA1 instruction, 313
SHA256 instruction, 313
SHA512 instruction, 313
shall (keyword), 10
shared memory, 381
shift count encodings, 316
shift instructions, 19
shift instructions, 83, 316
short floating-point load and store instructions, 411
short floating-point load instructions, 240
short floating-point store instructions, 335
should (keyword), 10
SIAM instruction, 315
side effect
accesses, 382
definition, 10
I/O locations, 382
  instruction prefetching, 383
real memory storage, 382
  visible, 382
side-effect page, illegal access to, 436
signalling NaN (not-a-number), 42, 220
  signed integer data type, 23
  synthetic instructions, 492
SIMD, 10
  and CMASK instructions, 137
instruction data formats, 29–30
simm10 instruction field
  move instructions, 270
simm11 instruction field
  move instructions, 268
simm13 instruction field
  floating point
    load instructions, 233, 251
simm13 instruction field
  arithmetic instructions, 278, 281, 310, 318, 353, 355
  floating point load instructions, 235, 238
  flush memory instruction, 166
  jump-and-link instruction, 226
  load instructions, 227, 242, 243, 244, 246
  logical operation instructions, 115, 363
  POPC, 286
  PREFETCH, 288
  RETURN, 305
single instruction/multiple data, See SIMD
SLL instruction, 316
SLLX instruction, 316
SMUL instruction, 50, 318
SMULcc instruction, 50, 318
SOFTWARE register, 49, 55
  clearing, 442
  clearing of selected bits, 56
  communication from nucleus code to kernel code, 442
  scheduling interrupt vectors, 441, 442
  setting, 442
SOFTWARE register fields
  int_level, 55
  sm (stick_int), 55
SOFTWARE CLR pseudo-register, 49, 56
SOFTWARE SET pseudo-register, 49, 55, 56
software
  nucleus, 9
  software translation table, 447
  software trap, 349, 425
  software trap number (SWTN), 349
  software, nonprivileged, 53
  software_trap_number, 490
source operands, 156, 172
SPARC V8 compatibility
  LD, LDUW instructions, 227
  operations to I/O locations, 383
  read state register instructions, 296
  STA instruction renamed, 321
  STBAR instruction, 257
  STD instruction, 245, 246, 338, 340
  tagged subtract instructions, 352
UNIMP instruction renamed, 224
window_overflow exception superseded, 436
write state register instructions, 358
SPARC V9
  compliance, 9, 456
  features, 13
SPARC V9 Application Binary Interface (ABI), 15
speculative load, 11
spill register window, 438
FLUSH instruction, 89
overflow/underflow, 36
RESTORE instruction, 88
SAVE instruction, 64, 88, 307, 439
SAVED instruction, 89, 440
selection of, 439
trap handling, 440
  trap vectors, 307, 440
window state, 64
spill_n_normal exception, 308, 438
  and FLUSHW instruction, 169
spill_n_other exception, 308, 438
  and FLUSHW instruction, 169
SRA instruction, 316
SRAX instruction, 316
SRL instruction, 316
SRLX instruction, 316
stack frame, 307
state registers (ASRs), 48–61
STB instruction, 319
STBA instruction, 320
STBAR instruction, 358, 387, 393
STBLOCF instruction, 322, 410
STDF instruction, 77, 325, 438
STDF_mem_address_not_aligned exception, 325, 327, 334, 438
  and store instructions, 326, 329
STDF/STDFA instruction, 77
STDFA instruction, 327
  alignment, 77
  ASIs for fp store operations, 411
  causing DAE_invalid_ASI exception, 411
  causing mem_address_not_aligned or illegal_instruction exception, 411
  causing STDF_mem_address_not_aligned exception, 77, 438
  for block load operations, 410
  for partial store operations, 411
  used with ASIs, 410
STF instruction, 325
STFA instruction, 327
STFSR instruction, 42, 44, 437
STFSR instruction (deprecated), 330
STH instruction, 319
STHA instruction, 320
STICK register, 49, 52, 56
  and STICK_CMPR, 57
  RDSTICK instruction, 295
STICK_CMPR register, 49, 57, 57
  int_dis field, 55, 58
  RDSTICK_CMPR instruction, 295
stick_cmp field, 58
store
  block, See block store instructions
  partial, See partial store instructions
  short floating-point, See short floating-point store instructions
store buffer
  merging, 382
store instructions, 11, 77
store short floating-point instructions, 335
StoreLoad MEMBAR relationship, 256, 393
StoreLoad predefined constant, 489
stores to alternate space, 19, 51, 81
StoreStore MEMBAR relationship, 256, 393
StoreStore predefined constant, 489
STPARTIALF instruction, 332
STPARTIALF instruction, DAE_nc_page exception, 435
STQF instruction, 78, 325, 438
STQF_mem_address_not_aligned exception, 325, 327, 438
STQF/STQFA instruction, 78
STQFA instruction, 78, 327
strand, 11
strong consistency memory model, 389
strong ordering, 389
Strong Sequential Order, 390
STSHORTF instruction, 335
STTW instruction, 38, 77
STTW instruction (deprecated), 337
STTWA instruction, 38, 77
STWA instruction, 320
STX instruction, 319
STXFSR instruction, 42, 44, 341, 437
SUB instruction, 342, 342
SUBC instruction, 342, 342
SUBcc instruction, 83, 342, 342
SUBCcc instruction, 342, 342
subnormal number, 11
subtract instructions, 342
superscalar, 11
supervisor software
  accessing special protected registers, 18
  definition, 11
SWAP instruction, 18, 343
  accessing doubleword simultaneously with other instructions, 344
  and DAE_nc_page exception, 435
  hardware primitive for mutual exclusion, 392
  identification of R register to be exchanged, 77
  in multiprocessor system, 242, 243
  memory accessing, 343
  ordering by MEMBAR, 393
swap R register
  with alternate space memory instructions, 344
  with memory instructions, 343
SWAPA instruction, 344
accessing doubleword simultaneously with other instructions, 344
alternate space addressing, 18
and DAE_nc_page exception, 435
hardware primitive for mutual exclusion, 392
in multiprocessor system, 242, 243
ordering by MEMBAR, 393
SWTN (software trap number), 349
Sync predefined constant, 489
synchronization, 258
synchronization, 11
synthetic instructions
  mapping to SPARC V9 instructions, 490–492
  for assembly language programmers, 490
mapping
  bclrg, 492
  bset, 492
  btog, 492
  btst, 492
call, 490, 491
cash, 492
clrn, 492
cmp, 490
dec, 492
decccc, 492
inc, 492
inccc, 492
iprefetch, 491
jmp, 490
movn, 492
neg, 492
not, 492
restore, 491
ret/retl, 491
save, 491
setn, 491
signx, 492
tst, 491
vs. pseudo ops, 490
system software
  accessing memory space by server program, 385
  ASIs allowing access to memory space, 386
  FLUSH instruction, 168, 395
  processing exceptions, 385
  trap types from which software must recover, 44
System Tick Compare register, See STICK_CMPR register
System Tick register, See STICK register
T
TA instruction, 348, 465
TADDC instruction, 83, 346
TADDC TV instruction, 83, 438
tag overflow, 83
tag_overflow exception, 83, 346, 347, 351, 352
tag_overflow exception (deprecated), 438
tagged arithmetic, 83
tagged arithmetic instructions, 20
tagged word data format, 23
tagged words, 23
TBA (trap base address) register, 67, 418
   establishing table address, 21, 417
   initialization, 424
   specification for RDPR instruction, 298
   specification for WRPR instruction, 360
   trap behavior, 11
TBR register (SPARC V8), 358
TCC instruction, 348
Tcc instructions, 348
   at TL > 0, 425
   causing trap, 417
   causing trap to privileged trap handler, 425
   CCR register bits, 51
   generating htrap_instruction exception, 436
   generating illegal_instruction exception, 437
   generating trap_instruction exception, 438
   opcode maps, 461, 465, 466
   programming uses, 349
   trap table space, 21
   vector through trap table, 417
TCS instruction, 348, 465
TE instruction, 348, 465
termination deferred trap, 420
   test-and-set instruction, 392
TG instruction, 348, 465
TGE instruction, 348, 465
TGU instruction, 348, 465
   thread, 11
TICK register, 49
   counter field, 52, 476
   inaccuracies between two readings of, 476, 484
   specification for RDPR instruction, 298
   timer registers, See TICK register and STICK register
   timing of instructions, 106
   tininess (floating-point), 48
TL (trap level) register, 71, 418
   affect on privilege level to which a trap is delivered, 424
   and implicit ASIs, 81
   displacement in trap table, 417
   executing RESTORED instruction, 302
   executing SAVED instruction, 309
   indexing for WRPR instruction, 360
   indexing privileged register after RDPR, 298
   setting register value after WRPR, 360
   specification for RDPR instruction, 298
   specification for WRPR instruction, 360
   and TBA register, 424
   and TPC register, 65
   and TSTATE register, 66
   and TT register, 67
   use in calculating privileged trap vector address, 424
   and WSTATE register, 63
TL instruction, 348, 465
TLB
   and 3-dimensional arrays, 119
   miss
   reloading TLB, 447
   specialized miss handler code, 456
TLE instruction, 348, 465
TLEU instruction, 348, 465
TN instruction, 348, 465
TNE instruction, 348, 465
TNEG instruction, 348, 465
TNPC (trap next program counter) register, 65
   saving NPC, 420
   specification for RDPR instruction, 298
   specification for WRPR instruction, 360
TNPG (trap-saved next program counter) register, 11
   total order, 388
   total store order (TSO) memory model, 69, 258, 382, 389, 389, 390
TPC (trap program counter) register, 11, 65
   address of trapping instruction, 299
   number of instances, 65
   specification for RDPR instructions, 298
   specification for WRPR instruction, 360
TPOS instruction, 348, 465
   trailing zero count instruction (how to synthesize), 286
   translating ASI, 398
Translation Storage Buffer (TSB), 449
Translation Table Entry, See TTE
trap
   See also exceptions and traps
   nonecacheable accesses, 383
   when taken, 11
   trap enable mask (tem) field of FSR register, 423, 424, 474
   trap handler
      privileged mode, 425
      regular/nonfaulting loads, 8
      returning from, 144, 303
      user, 45, 367
   trap level register, See TL register
   trap next program counter register, See TNPC register
   Trap on Control Transfer
      and instructions
         Bicc, 121, 135
         BPcc, 124
         BPr, 125
         CALL, 127
         DONE, 145, 303, 349
         FBfcc, 153, 155
         JMPL, 226
      tct field of PSTATE register, 68
      trap on integer condition codes instructions, 348
      trap program counter register, See TPG register
      trap state register, See TSTATE register
      trap type (TT) register, 425
      trap type register, See TT register
      trap_instruction (ISA) exception, 349, 350, 438
      trap_little_endian (tle) field of PSTATE register, 69
   traps, 11
   See also exceptions and individual trap names
   categories
      deferred, 419, 420, 421
      disrupting, 419, 421
      precise, 419, 420, 421
      priority, 423, 432
      reset, 420, 421
      restartable
      implementation dependency, 421
restartable deferred, 420
termination deferred, 420
caused by undefined feature/behavior, 12
causes, 21, 21
definition, 21, 417
hardware, 425
hardware stack, 14
level specification, 71
model stipulations, 423
nested, 14
normal, 425
processing, 432
software, 349, 425
stack, 433
vector address, specifying, 68
TSB, 11, 453
cacheability, 453
caching, 453
organization, 453
TSB (Translation Storage Buffer), 449
TSO, 11
TSO, See total store order (TSO) memory model
tst synthetic instruction, 491
tstate, See trap state (TSTATE) register
TSUBcc instruction, 83, 351
TSUBccTV instruction, 83, 438
TT (trap type) register, 67
and privileged trap vector address, 424
reserved values, 475
specification for RDPR instruction, 298
specification for WRPR instruction, 360
tstate, See trap state (TSTATE) register
TSUBcc instruction, 83, 351
TSUBccTV instruction, 83, 438
TTE (trap state) register, 66
DONE instruction, 144, 303
registers saved after trap, 21
restoring GL value, 74
specification for RDPR instruction, 298
specification for WRPR instruction, 360
tstate, See trap state (TSTATE) register
TSUBcc instruction, 83, 351
TSUBccTV instruction, 83, 438
TT (trap type) register, 67
and privileged trap vector address, 424
reserved values, 475
specification for RDPR instruction, 298
specification for WRPR instruction, 360
and Tcc instructions, 349
transferring trap control, 425
window spill/fill exceptions, 63
WRPR instruction, 360
TTE, 11
context ID field, 450
cp (cacheability) field, 382
cp field, 435, 451, 452
cv field, 451, 452
e field, 382, 395, 436, 451
ie field, 451
nfo field, 395, 435, 450, 451
p field, 435, 452
soft2 field, 450
sz (size) field, 453
taddr field, 450
v field, 450
va_tag field, 450
w field, 452
TTE in TSB, 449
TVC instruction, 348, 465
TVS instruction, 348, 465
typewriter font, in assembly language syntax, 485
TZZCNT instruction (how to synthesize), 286

U
UDIV instruction, 50, 353
UDIVcc instruction, 50, 353
UDIVX instruction, 278
ufm (underflow mask) field of FSR.tem, 47
UltraSPARC, previous ASIs
ASI_PHY_BYPASS_EC_WITH_EBIT_L, 412
ASI_PHY_BYPASS_EC_WITH_EBIT, 412
ASI_PHY_BYPASS_EC_WITH_EBIT_LITTLE, 412
ASI_PHY_USE_EC, 412
ASI_PHY_USE_EC_L, 412
ASI_PHY_USE_EC_LITTLE, 412
UMUL instruction, 50
UMUL instruction (deprecated), 355
UMULcc instruction, 50
UMULcc instruction (deprecated), 355
UMULXHI instruction, 356
unassigned, 11
unconditional branches, 120, 124, 152, 155
undefined, 12
underflow
bits of FSR register
accrued (ufa) bit of aexc field, 48, 366
current (ufc) bit of cexc field, 48
mask (ufm) bit of FSR.tem, 48
mask (ufm) bit of tem field, 367
detection, 36
occurrence, 439
underflow mask (ufm) field of FSR.tem, 47
unfinished_FPop floating-point trap type, 45, 150, 159, 185, 218, 220, 222, 365
handling, 48
in normal computation, 44
results after recovery, 45
UNIMP instruction (SPARC V8), 224
unimplemented, 12
unimplemented_FPop floating-point trap type, 366
handling, 48
unimplemented_LDTW exception, 245, 438
unimplemented_STTW exception, 338, 438
uniprocessor system, 12
unrestricted, 12
unrestricted ASI, 397
unsigned integer data type, 23
user application program, 12
user trap handler, 45, 367

V
VA, 12
VA_watchpoint exception, 132, 438
VA_WATCHPOINT register, 437, 438
value clipping, See FPACK instructions
value semantics of input/output (I/O) locations, 382
virtual address, 381
address 0, 395
virtual address, 12
virtual core, 12
virtual memory, 293
virtual processor, pipeline draining, 61
VIS, 12
VIS instructions
encoding, 467, 468, 469
implicitly referencing GSR register, 54
Visual Instruction Set, See VIS instructions

W
W-A-R, See write-after-read memory hazard
watchpoint comparator, 70
W-A-W, See write-after-write memory hazard
WIM register (SPARC V8), 358
window fill exception, See also fill正常 exception
window fill trap handler, 21
window overflow, 36, 439
window spill exception, See also spill正常 exception
window spill trap handler, 21
window state register, See WSTATE register
window underflow, 439
window, clean, 307
window_fill exception, 63, 88
RETURN, 305
window_spill exception, 63
word, 12
alignment, 18, 77, 383
data format, 23
WRASI instruction, 49, 51, 357
WRasr instruction, 357
accessing I/O registers, 19
try to write to ASR 5 (PC), 53
cannot write to PC register, 53
implementation dependencies, 475
writing ASRs, 48
WRCCR instruction, 49, 50, 51, 357
WRFPRS instruction, 49, 53, 357
WRGSR instruction, 49, 54, 357
WRIER instruction (SPARC V8), 358
write ancillary state register (WRasr) instructions, 357
write ancillary state register instructions, See WRasr instruction
write privileged register instruction, 360
write-after-read memory hazard, 387
WRPAUSE instruction, 357
WRPR instruction
accessing non-register-window PR state registers, 65
accessing register-window PR state registers, 61
and register-window PR state registers, 61
effect on TNPC register, 66
effect on TPC register, 65
effect on TSTATE register, 67
effect on TT register, 67
writing the TICK register, 52
writing to GL register, 73
writing to PSTATE register, 68
writing to TICK register, 52
WRPSR instruction (SPARC V8), 358
WRSOFTINT instruction, 49, 55, 357
WRSOFTINT_CLR instruction, 49, 55, 56, 357, 442
WRSOFTINT_SET instruction, 49, 55, 357, 442
WRSTICK_CMPR instruction, 49, 357
WRTBR instruction (SPARC V8), 358
WRWIM instruction (SPARC V8), 358
WRY instruction, 49, 50, 357
WSTATE (window state) register
description, 63
and fill/spill exceptions, 439
normal field, 439
other field, 439
overview, 61
reading with RDPR instruction, 298
spill exception, 169
spill trap, 307
writing with WRPR instruction, 360

X
XMULX instruction, 362
XMULXHI instruction, 362
XNOR instruction, 363
XNORcc instruction, 363
XOR instruction, 363
XORcc instruction, 363

Y
Y register, 49, 50
content after divide operation, 310, 353
divide operation, 310, 353
unsigned multiply results, 318, 355
WRY instruction, 358
Y register (deprecated), 50

Z
zero virtual address, 395