UltraSPARC Architecture 2007

One Architecture
... Multiple Innovative Implementations

Draft D0.9.3b, 20 Oct 2009

Privilege Levels: *Privileged and Nonprivileged*

Distribution: *Public*

Part No: 950-5554-14
Revision: Draft D0.9.3b, 20 Oct 2009

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

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

Contents

Preface

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

What's New?

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

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

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

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

UltraSPARC Architecture 2007 includes the following changes since :

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

In addition, architectural features are now tagged with Software Classes and Implementation Classes[1]. Software Classes provide a new, high-level view of the expected architectural longevity and portability of software that references those features. Implementation Classes give an indication of how efficiently each feature is likely to be implemented across current and future UltraSPARC Architecture processor implementations. This information provides guidance that should be

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

particularly helpful to programmers who write in assembly language or those who write tools that generate SPARC instructions. It also provides the infrastructure for defining clear procedures for adding and removing features from the architecture over time, with minimal software disruption.

Acknowledgements

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

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

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

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

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

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

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

— *David Weaver*
UltraSPARC Architecture Principal Engineer and specification editor

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

CHAPTER **1**

Document Overview

This chapter discusses:

- **Navigating UltraSPARC Architecture 2007** on page 1.
- **Fonts and Notational Conventions** on page 2.
- **Reporting Errors in this Specification** on page 4.

1.1 Navigating *UltraSPARC Architecture 2007*

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

If you are familiar with the SPARC V9 architecture but not UltraSPARC Architecture 2007, note that UltraSPARC Architecture 2007 conforms to the SPARC V9 Level 1 architecture (and most of Level 2), with numerous extensions — particularly with respect toVIS instructions.

This specfication is structured as follows:

- Chapter 2, *Definitions*, which defines key terms used throughout the specification
- Chapter 3, *Architecture Overview*, provides an overview of UltraSPARC Architecture 2007
- Chapter 4, *Data Formats*, describes the supported data formats
- Chapter 5, *Registers*, describes the register set
- Chapter 6, *Instruction Set Overview*, provides a high-level description of the UltraSPARC Architecture 2007 instruction set
- Chapter 7, *Instructions*, describes the UltraSPARC Architecture 2007 instruction set in great detail
- Chapter 8, *IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007*, describes the trap model
- Chapter 9, *Memory* describes the supported memory model
- Chapter 10, *Address Space Identifiers (ASIs)*, provides a complete list of supported ASIs
- Chapter 11, *Performance Instrumentation* describes the architecture for performance monitoring hardware
- Chapter 12, *Traps*, describes the trap model
- Chapter 13, *Interrupt Handling*, describes how interrupts are handled
- Chapter 14, *Memory Management*, describes MMU operation
- Appendix A, *Opcode Maps*, provides the overall picture of how the instruction set is mapped into opcodes
- Appendix B, *Implementation Dependencies*, describes all implementation dependencies

- Appendix C, *Assembly Language Syntax*, describes extensions to the SPARC assembly language syntax; in particular, synthetic instructions are documented in this appendix

1.2 Fonts and Notational Conventions

Fonts are used as follows:

- *Italic* font is used for emphasis, book titles, and the first instance of a word that is defined.
- *Italic* font is also used for terms where substitution is expected, for example, "`fcc`*n*", "virtual processor *n*", or "*reg_plus_imm*".
- *Italic sans serif* font is used for exception and trap names. For example, "The *privileged_action* exception...."
- lowercase helvetica font is used for register field names (named bits) and instruction field names, for example: "The rs1 field contains...."
- UPPERCASE HELVETICA font is used for register names; for example, FSR.
- `TYPEWRITER` (Courier) font is used for literal values, such as code (assembly language, C language, ASI names) and for state names. For example: `%f0`, ASI_PRIMARY, `execute_state`.
- When a register field is shown along with its containing register name, they are separated by a period ('.'), for example, "FSR.cexc".
- UPPERCASE words are acronyms or instruction names. Some common acronyms appear in the glossary in Chapter 2, *Definitions*. **Note:** Names of some instructions contain both upper- and lower-case letters.
- An underscore character joins words in register, register field, exception, and trap names. **Note:** Such words may be split across lines at the underbar without an intervening hyphen. For example: "This is true whenever the integer_condition_
code field...."

The following notational conventions are used:

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

- A double period (..) indicates any *single* intermediate value between two given end values is possible. For example, NAME[2..0] indicates four forms of NAME exist: NAME, NAME2, NAME1, and NAME0; whereas NAME<2..0> indicates that three forms exist: NAME2, NAME1, and NAME0. (Contrast with the use of the colon, above)
- A vertical bar (|) separates mutually exclusive alternatives inside square brackets ([ ]), angle brackets (< >), or curly braces ({ }). For example, “NAME[A|B]” expands to “NAME, NAMEA, NAMEB” and “NAME<A|B>” expands to "NAMEA, NAMEB".
- The asterisk (*) is used as a wild card, encompassing the full set of valid values. For example, FCMP* refers to FCMP with all valid suffixes (in this case, FCMP<s|d|q> and FCMPE<s|d|q>). An asterisk is typically used when the full list of valid values either is not worth listing (because it has little or no relevance in the given context) or the valid values are too numerous to list in the available space.
- The slash (/) is used to separate paired or complementary values in a list, for example, “the LDBLOCKF/STBLOCKF instruction pair ....”
- The double colon (**::**) is an operator that indicates concatenation (typically, of bit vectors). Concatenation strictly strings the specified component values into a single longer string, in the order specified. The concatenation operator performs no arithmetic operation on any of the component values.

1.2.1 Implementation Dependencies

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

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

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

1.2.2 Notation for Numbers

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

1.2.3 Informational Notes

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

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

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

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

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

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

1.3 Reporting Errors in this Specification

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

CHAPTER **2**

Definitions

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

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

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

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

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

ASI Address space identifier.

ASR Ancillary State register.

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

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

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

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

CCR Abbreviation for Condition Codes Register.

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

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

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

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

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

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

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

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

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

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

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

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

DCTI Delayed control transfer instruction.

denormalized number Synonym for **subnormal number**.

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

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

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

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

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

explicit ASI An ASI that 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*n* One of the floating-point condition code fields fcc0, fcc1, fcc2, or fcc3.

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

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

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

floating-point operate instructions Instructions that perform floating-point calculations, as defined in *Floating-Point Operate (FPop) Instructions* on page 85. FPop instructions do not include FBfcc instructions, loads and stores between memory and the F registers, or non-floating-point operations that read or write F registers.

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

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

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

FPRS Floating-Point Register State register.

FPU Floating-Point Unit.

FSR Floating-Point Status register.

GL Global Level register.

GSR General Status register.

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

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

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

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

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

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

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

initiated Synonym for **issued**.

instruction field A bit field within an instruction word.

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

instruction set architecture A set that defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. Does not define clock cycle times, cycles per instruction, data paths, etc. This specification defines the UltraSPARC Architecture 2007 instruction set architecture.

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

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

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

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

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

ISA Instruction set architecture.

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

IU Integer Unit.

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

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

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

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

Memory Management Unit The address translation hardware in an UltraSPARC Architecture implementation that translates 64-bit virtual address into underlying hardware addresses. The MMU is composed of the ASRs and ASI registers used to manage address translation. See also **context real address**, and **virtual address**.

MMU Abbreviation for **Memory Management Unit**.

multiprocessor system A system containing more than one processor.

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

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

NFO Nonfault access only.

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

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

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

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

NPC Next program counter.

npt Nonprivileged trap.

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

NUMA Nonuniform memory access.

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

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

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

opcode A bit pattern that identifies a particular instruction.

optional A feature not required for UltraSPARC Architecture 2007 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. See also **processor**, **strand**, **thread**, and **virtual processor**.

prefetchable (1) An attribute of a memory location that indicates to an MMU that PREFETCH operations to that location may be applied.
(2) A memory location condition for which the system designer has determined that no undesirable effects will occur if a PREFETCH operation to that location is allowed to succeed. Typically, normal memory is prefetchable.

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

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

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

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

processor core Synonym for **physical core**.

processor module Synonym for **processor**.

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

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

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

RA Real address.

RAS Reliability, Availability, and Serviceability

RAW Read After Write (hazard)

rd Rounding direction.

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

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

A reserved instruction field must read as 0, unless the implementation supports extended instructions within the field. The behavior of an UltraSPARC Architecture 2007 virtual processor when it encounters a nonzero value in a reserved instruction field is as defined in *Reserved Opcodes and Instruction Fields* on page 86.

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

A reserved field within a register reads as 0 in current implementations and, when written by software, should always be written with values of that field previously read from that register or with the value zero (as described in *Reserved Register Fields* on page 32).

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

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

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

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

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

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

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

shall Synonym for **must**.

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

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

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

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

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

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

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

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

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

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

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

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

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

TBA Trap base address.

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

TNPC Trap-saved next program counter.

TPC Trap-saved program counter.

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

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

TSO Total Store Order (a memory model).

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

UA-2007 UltraSPARC Architecture 2007

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

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 VIS™ 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.

CHAPTER 3

Architecture Overview

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

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

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

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

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

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

3.1 The UltraSPARC Architecture 2007

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

3.1.1 Features

The UltraSPARC Architecture 2007, like its ancestor SPARC V9, includes the following principal features:

- **A linear 64-bit address space** with 64-bit addressing.
- **32-bit wide instructions** — These are aligned on 32-bit boundaries in memory. Only load and store instructions access memory and perform I/O.

- **Few addressing modes** — A memory address is given as either "register + register" or "register + immediate".
- **Triadic register addresses** — Most computational instructions operate on two register operands or one register and a constant and place the result in a third register.
- **A large windowed register file** — At any one instant, a program sees 8 global integer registers plus a 24-register window of a larger register file. The windowed registers can be used as a cache of procedure arguments, local values, and return addresses.
- **Floating point** — The architecture provides an IEEE 754-compatible floating-point instruction set, operating on a separate register file that provides 32 single-precision (32-bit), 32 double-precision (64-bit), and 16 quad-precision (128-bit) overlayed registers.
- **Fast trap handlers** — Traps are vectored through a table.
- **Multiprocessor synchronization instructions** — Multiple variations of atomic load-store memory operations are supported.
- **Predicted branches** — The branch with prediction instructions allows the compiler or assembly language programmer to give the hardware a hint about whether a branch will be taken.
- **Branch elimination instructions** — Several instructions can be used to eliminate branches altogether (for example, Move on Condition). Eliminating branches increases performance in superscalar and superpipelined implementations.
- **Hardware trap stack** — A hardware trap stack is provided to allow nested traps. It contains all of the machine state necessary to return to the previous trap level. The trap stack makes the handling of faults and error conditions simpler, faster, and safer.

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

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

3.1.2 Attributes

UltraSPARC Architecture 2007 is a processor *instruction set architecture* (ISA) derived from SPARC V8 and SPARC V9, which in turn come from a reduced instruction set computer (RISC) lineage. As an architecture, UltraSPARC Architecture 2007 allows for a spectrum of processor and system *implementations* at a variety of price/performance points for a range of applications, including scientific/engineering, programming, real-time, and commercial applications.

3.1.2.1 Design Goals

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

3.1.2.2 Register Windows

The UltraSPARC Architecture 2007 architecture is derived from the SPARC architecture, which was formulated at Sun Microsystems in 1984 through1987. The SPARC architecture is, in turn, based on the RISC I and II designs engineered at the University of California at Berkeley from 1980 through 1982. The SPARC "register window" architecture, pioneered in the UC Berkeley designs, allows for straightforward, high-performance compilers and a reduction in memory load/store instructions.

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

3.1.3 System Components

The UltraSPARC Architecture 2007 allows for a spectrum of subarchitectures, such as cache system.

3.1.3.1 Binary Compatibility

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

Although different UltraSPARC Architecture 2007 systems can execute nonprivileged programs at different rates, they will generate the same results as long as they are run under the same memory model. See Chapter 9, *Memory*, for more information.

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

3.1.3.2 UltraSPARC Architecture 2007 MMU

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

3.1.3.3 Privileged Software

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

3.1.4 Architectural Definition

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

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

3.1.5 UltraSPARC Architecture 2007 Compliance with SPARC V9 Architecture

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

3.1.6 Implementation Compliance with UltraSPARC Architecture 2007

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

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

3.2 Processor Architecture

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

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

3.2.1 Integer Unit (IU)

An UltraSPARC Architecture 2007 implementation’s integer unit contains the general-purpose registers and controls the overall operation of the virtual processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU.

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

3.2.2 Floating-Point Unit (FPU)

An UltraSPARC Architecture 2007 implementation's FPU has thirty-two 32-bit (single-precision) floating-point registers, thirty-two 64-bit (double-precision) floating-point registers, and sixteen 128-bit (quad-precision) floating-point registers, some of which overlap.

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

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

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

3.3 Instructions

Instructions fall into the following basic categories:

- Memory access
- Integer arithmetic / logical / shift
- Control transfer
- State register access
- Floating-point operate
- Conditional move
- Register window management
- SIMD (single instruction, multiple data) instructions

These classes are discussed in the following subsections.

3.3.1 Memory Access

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

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

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

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

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

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

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

3.3.1.1 Memory Alignment Restrictions

A memory access on an UltraSPARC Architecture virtual processor must typically be aligned on an address boundary greater than or equal to the size of the datum being accessed. An improperly aligned address in a load, store, or load-store in instruction may trigger an exception and cause a subsequent trap. For details, see *Memory Alignment Restrictions* on page 73.

3.3.1.2 Addressing Conventions

The UltraSPARC Architecture uses big-endian byte order by default: the address of a quadword, doubleword, word, or halfword is the address of its most significant byte. Increasing the address means decreasing the significance of the unit being accessed. All instruction accesses are performed using big-endian byte order.

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

Addressing conventions are illustrated in FIGURE 6-2 on page 75 and FIGURE 6-3 on page 77.

3.3.1.3 Addressing Range

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

3.3.1.4 Load/Store Alternate

Versions of load/store instructions, the *load/store alternate* instructions, can specify an arbitrary 8-bit address space identifier for the load/store data access.
Access to alternate spaces 00_{16}–$2F_{16}$ is restricted to privileged software, access to alternate spaces 30_{16}–$7F_{16}$ is restricted to hyperprivileged software, and access to alternate spaces 80_{16}–FF_{16} is unrestricted. Some of the ASIs are available for implementation-dependent uses. Privileged software can use the implementation-dependent ASIs to access special protected registers, such as cache control registers, virtual processor state registers, and other processor-dependent or system-dependent values. See *Address Space Identifiers (ASIs)* on page 76 for more information.

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

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

3.3.1.5 Separate Instruction and Data Memories

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

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

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

3.3.1.6 Input/Output (I/O)

The UltraSPARC Architecture assumes that input/output registers are accessed through load/store alternate instructions, normal load/store instructions, or read/write Ancillary State Register instructions (RDasr, WRasr).

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

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

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

3.3.1.7 Memory Synchronization

Two instructions are used for synchronization of memory operations: FLUSH and MEMBAR. Their operation is explained in *Flush Instruction Memory* on page 133 and *Memory Barrier* on page 201, respectively.

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

3.3.2 Integer Arithmetic / Logical / Shift Instructions

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

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

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

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

3.3.3 Control Transfer

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

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

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

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

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

3.3.4 State Register Access

3.3.4.1 Ancillary State Registers

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

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

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

3.3.4.2 PR State Registers

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

3.3.5 Floating-Point Operate

Floating-point operate (FPop) instructions perform all floating-point calculations; they are register-to-register instructions that operate on the floating-point registers. FPops compute a result that is a function of one , two, or three source operands. The groups of instructions that are considered FPops are listed in *Floating-Point Operate (FPop) Instructions* on page 85.

3.3.6 Conditional Move

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

3.3.7 Register Window Management

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

3.3.8 SIMD

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

3.4 Traps

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

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

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

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

CHAPTER 4

Data Formats

The UltraSPARC Architecture recognizes these fundamental data types:

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

The widths of the data types are as follows:

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

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

Data formats are described in these sections:

- **Integer Data Formats** on page 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 4-1 Signed Integer, Unsigned Integer, and Tagged Format Ranges

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

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

TABLE 4-2 Integer Doubleword/Extended-word Alignment

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

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

The data types are illustrated in the following subsections.

4.1.1 Signed Integer Data Types

Figures in this section illustrate the following signed data types:

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

4.1.1.1 Signed Integer Byte, Halfword, and Word

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

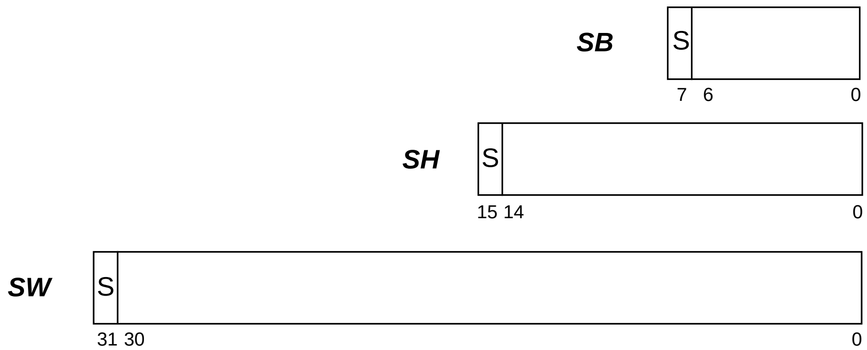


FIGURE 4-1 Signed Integer Byte, Halfword, and Word Data Formats

4.1.1.2 Signed Integer Doubleword (64 bits)

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

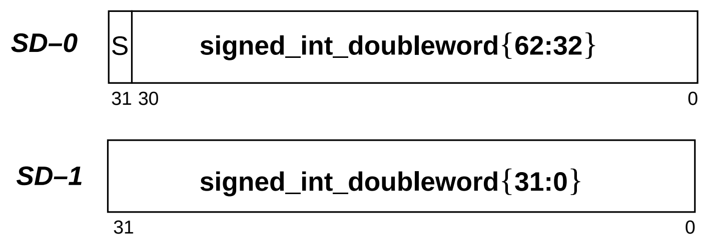


FIGURE 4-2 Signed Integer Double Data Format

4.1.1.3 Signed Integer Extended-Word (64 bits)

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

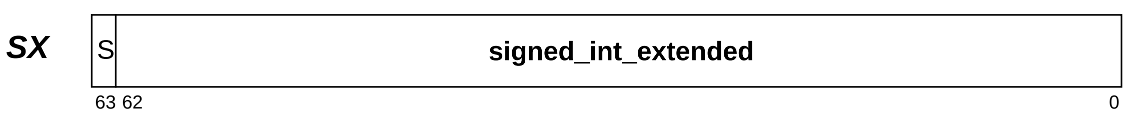


FIGURE 4-3 Signed Integer Extended-Word Data Format

4.1.2 Unsigned Integer Data Types

Figures in this section illustrate the following unsigned data types:

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

4.1.2.1 Unsigned Integer Byte, Halfword, and Word

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

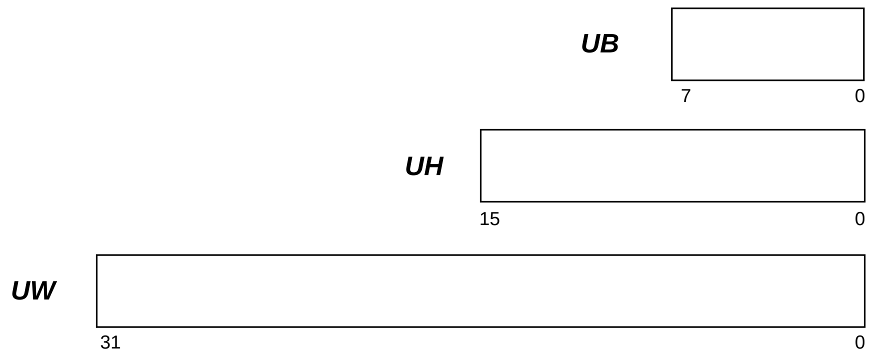


FIGURE 4-4 Unsigned Integer Byte, Halfword, and Word Data Formats

4.1.2.2 Unsigned Integer Doubleword (64 bits)

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

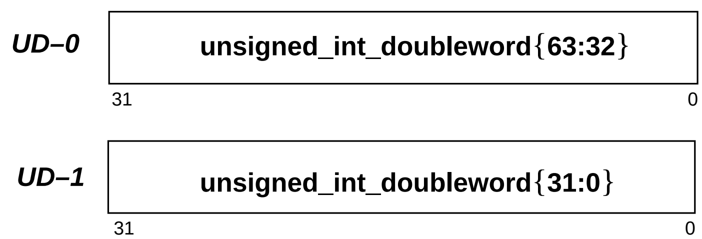


FIGURE 4-5 Unsigned Integer Double Data Format

4.1.2.3 Unsigned Extended Integer (64 bits)

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

FIGURE 4-6 Unsigned Extended Integer Data Format

4.1.3 Tagged Word (32 bits)

FIGURE 4-7 illustrates the tagged word data format.

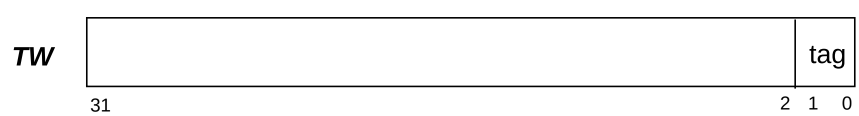


FIGURE 4-7 Tagged Word Data Format

4.2 Floating-Point Data Formats

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

4.2.1 Floating Point, Single Precision (32 bits)

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

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

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

s = sign (1 bit)
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
– ∞ (negative infinity)	s = 1; e = 255 (max); f = .000--00
+ ∞ (positive infinity)	s = 0; e = 255 (max); f = .000--00

4.2.2 Floating Point, Double Precision (64 bits)

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

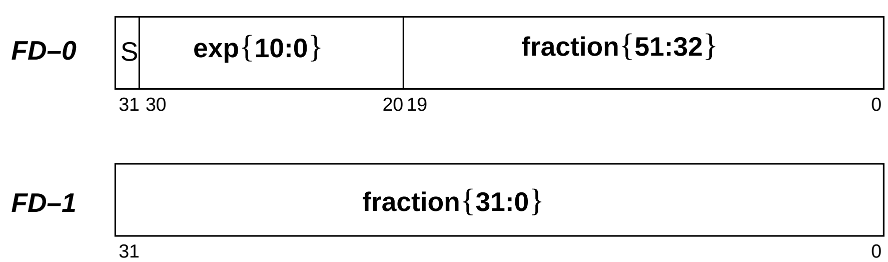


FIGURE 4-9 Floating-Point Double-Precision Data Format

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

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

4.2.3 Floating Point, Quad Precision (128 bits)

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

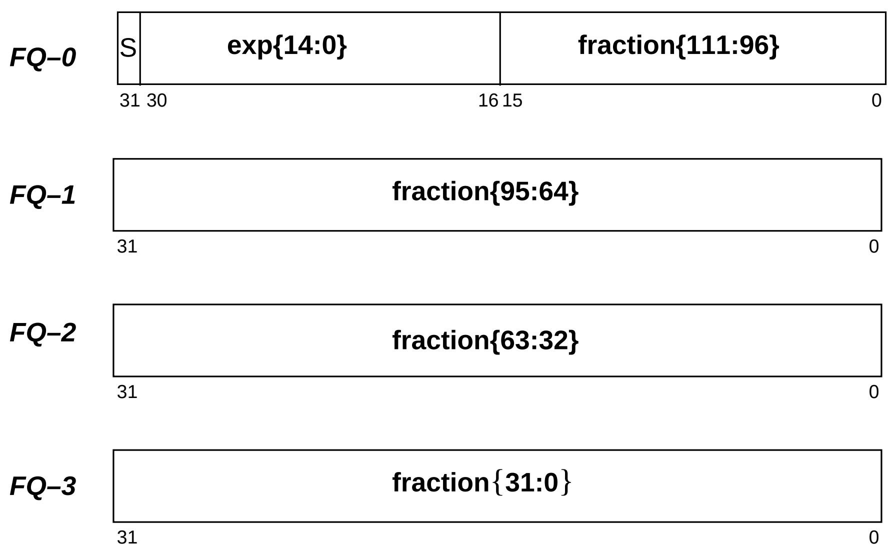


FIGURE 4-10 Floating-Point Quad-Precision Data Format

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

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

TABLE 4-5 Floating-Point Quad-Precision Format Definition *(Continued)*

s = sign (1 bit) e = biased exponent (15 bits) f = fraction (112 bits) *u* = undefined	
Quiet NaN	s = u; e = 32767 (max); f = .1uu--uu
– ∞ (negative infinity)	s = 1; e = 32767 (max); f = .000--00
+ ∞ (positive infinity)	s = 0; e = 32767 (max); f = .000--00

4.2.4 Floating-Point Data Alignment in Memory and Registers

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

TABLE 4-6 Floating-Point Doubleword and Quadword Alignment

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

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

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

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

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

4.3 SIMD Data Formats

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

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

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

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

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

4.3.1 Uint8 SIMD Data Format

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

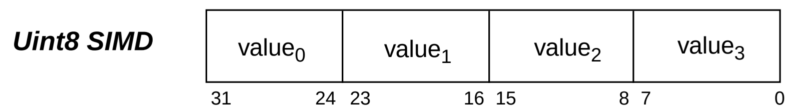


FIGURE 4-11 Uint8 SIMD Data Format

4.3.2 Int16 SIMD Data Formats

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

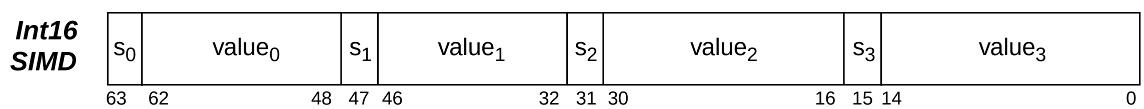


FIGURE 4-12 Int16 SIMD Data Format

4.3.3 Int32 SIMD Data Format

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

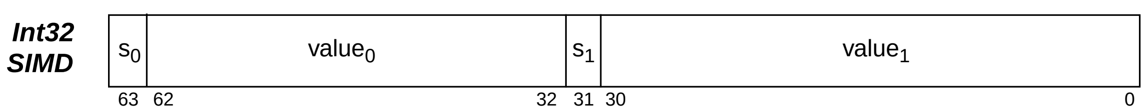


FIGURE 4-13 Int32 SIMD Data Format

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

CHAPTER 5

Registers

The following registers are described in this chapter:

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

5.1 Reserved Register Fields

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

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

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

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

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

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

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

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

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

5.2 General-Purpose R Registers

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

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

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

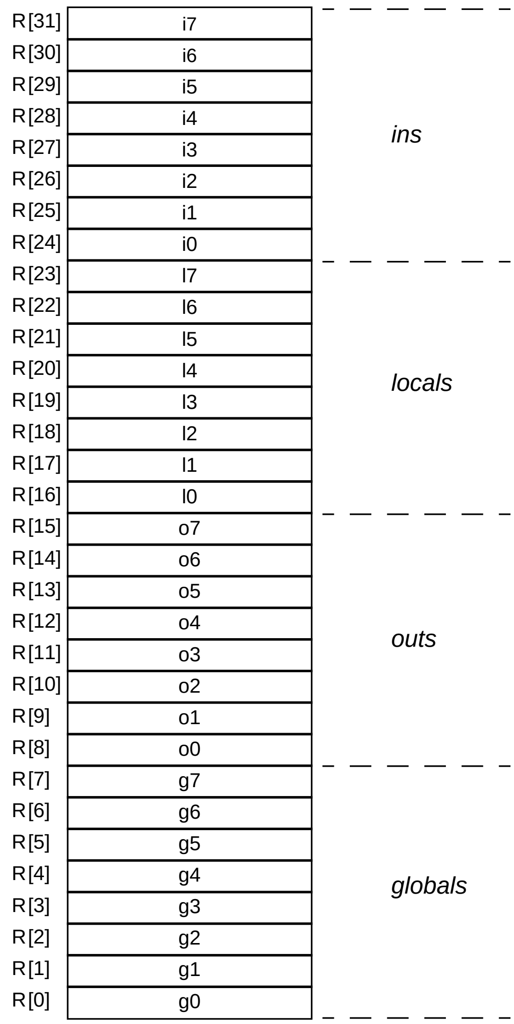


FIGURE 5-1 General-Purpose Registers (as Visible at Any Given Time)

5.2.1 Global R Registers A1

Registers R[0]–R[7] refer to a set of eight registers called the *global* registers (labelled g0 through g7). At any time, one of *MAXPGL* +1 sets of eight registers is enabled and can be accessed as the current set of global registers. The currently enabled set of global registers is selected by the GL register. See *Global Level Register (glP) (PR 16)* on page 69.

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

5.2.2 Windowed R Registers (A1)

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

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

TABLE 5-1 Window Addressing

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

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

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

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

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

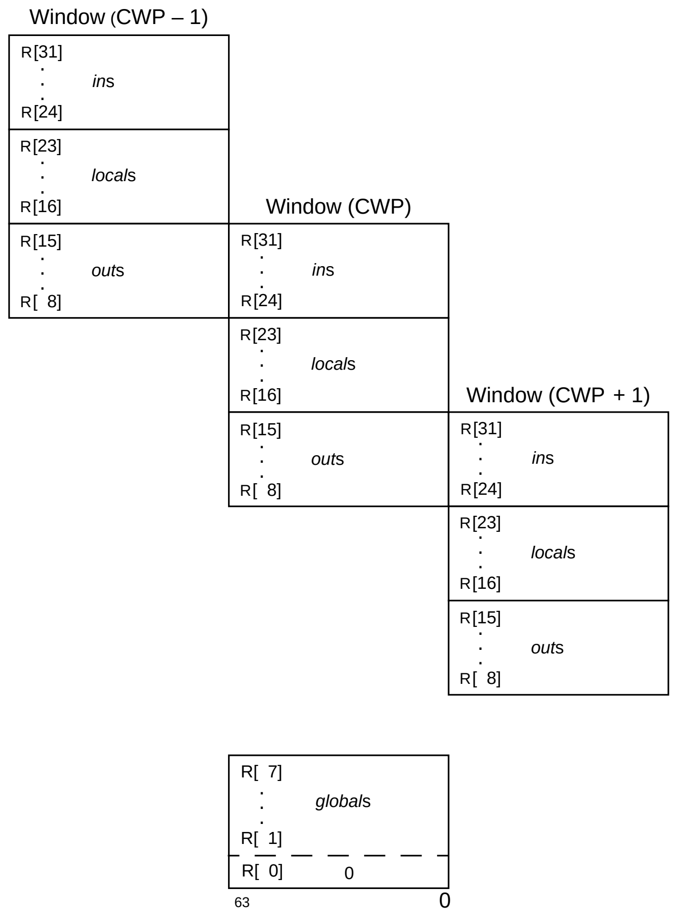


FIGURE 5-2 Three Overlapping Windows and Eight Global Registers

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

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

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

Programming Note | Since the procedure call instructions (CALL and JMPL) do not change the CWP, a procedure can be called without changing the window. See the section "Leaf-Procedure Optimization" in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes*

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

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

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

When a new register window is made visible through use of a SAVE instruction, the *local* and *out* registers are guaranteed to contain either zeroes or valid data from the current context. If software executes a RESTORE and later executes a SAVE, then the contents of the resulting window's *local* and *out* registers are not guaranteed to be preserved between the RESTORE and the SAVE[1]. Those registers may even have been written with "dirty" data, that is, data created by software running in a different context. However, if the clean_window protocol is being used, system software must guarantee that registers in the current window after a SAVE always contains only zeroes or valid data from that context. See *Clean Windows (cleanwinP) Register (PR 12)* on page 59, *Savable Windows (cansaveP) Register (PR 10)* on page 59, and *Restorable Windows (canrestoreP) Register (PR 11)* on page 59.

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

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

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

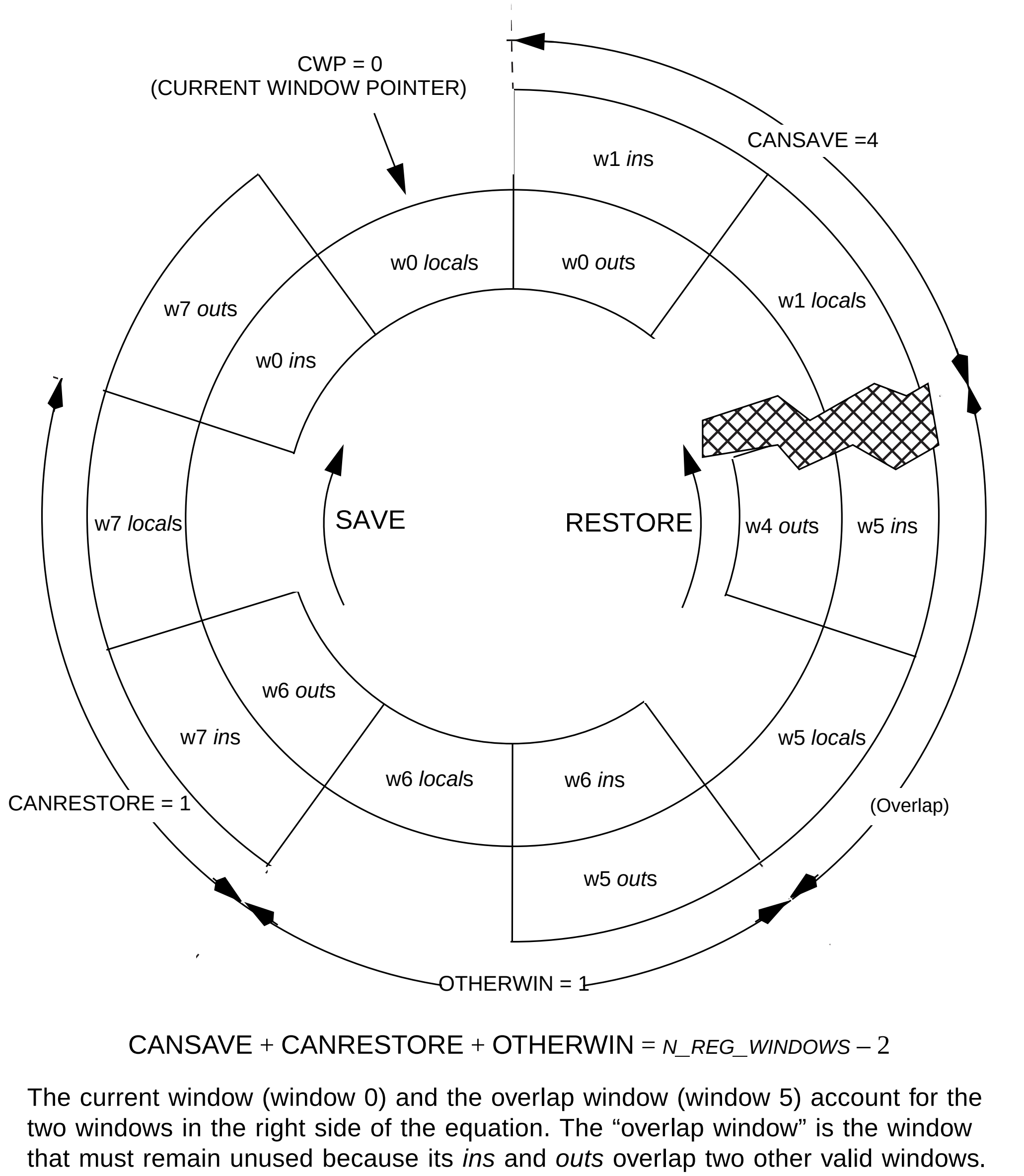


FIGURE 5-3 Windowed R Registers for *N_REG_WINDOWS* = 8

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

5.2.3 Special R Registers

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

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

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

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

An attempt to execute 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 A1

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

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

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

TABLE 5-2 Floating-Point Registers, with Aliasing *(1 of 3)*

Single Precision (32-bit)		Double Precision (64-bit)			Quad Precision (128-bit)		
Register	Assembly Language	Bits	Register	Assembly Language	Bits	Register	Assembly Language
F_S[0]	`%f0`	63:32	F_D[0]	`%d0`	127:64	F_Q[0]	`%q0`
F_S[1]	`%f1`	31:0					
F_S[2]	`%f2`	63:32	F_D[2]	`%d2`	63:0		
F_S[3]	`%f3`	31:0					
F_S[4]	`%f4`	63:32	F_D[4]	`%d4`	127:64	F_Q[4]	`%q4`
F_S[5]	`%f5`	31:0					
F_S[6]	`%f6`	63:32	F_D[6]	`%d6`	63:0		
F_S[7]	`%f7`	31:0					
F_S[8]]	`%f8`	63:32	F_D[8]	`%d8`	127:64	F_Q[8]	`%q8`
F_S[9]	`%f9`	31:0					
F_S[10]	`%f10`	63:32	F_D[10]	`%d10`	63:0		
F_S[11]	`%f11`	31:0					

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

Single Precision (32-bit)		Double Precision (64-bit)			Quad Precision (128-bit)		
Register	Assembly Language	Bits	Register	Assembly Language	Bits	Register	Assembly Language
$F_S[12]$	`%f12`	63:32	$F_D[12]$	`%d12`	127:64	$F_Q[12]$	`%q12`
$F_S[13]$	`%f13`	31:0					
$F_S[14]$	`%f14`	63:32	$F_D[14]$	`%d14`	63:0		
$F_S[15]$	`%f15`	31:0					
$F_S[16]$	`%f16`	63:32	$F_D[16]$	`%d16`	127:64	$F_Q[16]$	`%q16`
$F_S[17]$	`%f17`	31:0					
$F_S[18]$	`%f18`	63:32	$F_D[18]$	`%d18`	63:0		
$F_S[19]$	`%f19`	31:0					
$F_S[20]$	`%f20`	63:32	$F_D[20]$	`%d20`	127:64	$F_Q[20]$	`%q20`
$F_S[21]$	`%f21`	31:0					
$F_S[22]$	`%f22`	63:32	$F_D[22]$	`%d22`	63:0		
$F_S[23]$	`%f23`	31:0					
$F_S[24]$	`%f24`	63:32	$F_D[24]$	`%d24`	127:64	$F_Q[24]$	`%q24`
$F_S[25]$	`%f25`	31:0					
$F_S[26]$	`%f26`	63:32	$F_D[26]$	`%d26`	63:0		
$F_S[27]$	`%f27`	31:0					
$F_S[28]$	`%f28`	63:32	$F_D[28]$	`%d28`	127:64	$F_Q[28]$	`%q28`
$F_S[29]$	`%f29`	31:0					
$F_S[30]$	`%f30`	63:32	$F_D[30]$	`%d30`	63:0		
$F_S[31]$	`%f31`	31:0					
		63:32	$F_D[32]$	`%d32`	127:64	$F_Q[32]$	`%q32`
		31:0					
		63:32	$F_D[34]$	`%d34`	63:0		
		31:0					
		63:32	$F_D[36]$	`%d36`	127:64	$F_Q[36]$	`%q36`
		31:0					
		63:32	$F_D[38]$	`%d38`	63:0		
		31:0					
		63:32	$F_D[40]$	`%d40`	127:64	$F_Q[40]$	`%q40`
		31:0					
		63:32	$F_D[42]$	`%d42`	63:0		
		31:0					
		63:32	$F_D[44]$	`%d44`	127:64	$F_Q[44]$	`%q44`
		31:0					
		63:32	$F_D[46]$	`%d46`	63:0		
		31:0					

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

Single Precision (32-bit)		Double Precision (64-bit)			Quad Precision (128-bit)		
Register	Assembly Language	Bits	Register	Assembly Language	Bits	Register	Assembly Language
		63:32	$F_D[48]$	%d48	127:64	$F_Q[48]$	%q48
		31:0					
		63:32	$F_D[50]$	%d50	63:0		
		31:0					
		63:32	$F_D[52]$	%d52	127:64	$F_Q[52]$	%q52
		31:0					
		63:32	$F_D[54]$	%d54	63:0		
		31:0					
		63:32	$F_D[56]$	%d56	127:64	$F_Q[56]$	%q56
		31:0					
		63:32	$F_D[58]$	%d58	63:0		
		31:0					
		63:32	$F_D[60]$	%d60	127:64	$F_Q[60]$	%q60
		31:0					
		63:32	$F_D[62]$	%d62	63:0		
		31:0					

5.3.1 Floating-Point Register Number Encoding

Register numbers for single, double, and quad registers are encoded differently in the 5-bit register number field of a floating-point instruction. If the bits in a register number field are labelled b{4} … b{0} (where b{4} is the most significant bit of the register number), the encoding of floating-point register numbers into 5-bit instruction fields is as given in TABLE 5-3.

TABLE 5-3 Floating-Point Register Number Encoding

Register Operand Type	Full 6-bit Register Number						Encoding in a 5-bit Register Field in an Instruction				
Single	0	b{4}	b{3}	b{2}	b{1}	b{0}	b{4}	b{3}	b{2}	b{1}	b{0}
Double	b{5}	b{4}	b{3}	b{2}	b{1}	0	b{4}	b{3}	b{2}	b{1}	b{5}
Quad	b{5}	b{4}	b{3}	b{2}	0	0	b{4}	b{3}	b{2}	0	b{5}

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

5.3.2 Double and Quad Floating-Point Operands

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

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

1. Whenever possible, align floating-point data in memory on proper address boundaries. If access to a datum is required to be atomic, the datum *must* be properly aligned.
2. If a double- or quad-precision datum is not properly aligned in memory or is still aligned on a 4-byte boundary, and access to the datum in memory is not required to be atomic, then software should attempt to allocate a register for it in the lower half of the floating-point register file so that the datum can be loaded with multiple LDF[A] instructions.
3. If the only available registers for such a datum are located in the upper half of the floating-point register file and access to the datum in memory is not required to be atomic, the word-aligned datum can be loaded into them by one of two methods:
   - Load the datum into an upper register by using multiple LDF[A] instructions to first load it into a double- or quad-precision register in the lower half of the floating-point register file, then copy that register to the desired destination register in the upper half.

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

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

Implementation Note UltraSPARC Architecture 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) (A1)

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

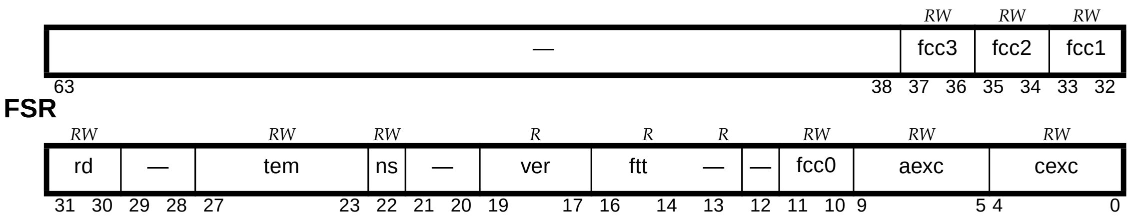


FIGURE 5-4 FSR Fields

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

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

The subsections on pages 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 (fcc*n* refers to any of the floating-point condition code fields).

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

In TABLE 5-4, f_{rs1} and f_{rs2} correspond to the single, double, or quad values in the floating-point registers specified by a floating-point compare instruction’s rs1 and rs2 fields. The question mark (?) indicates an unordered relation, which is true if either f_{rs1} or f_{rs2} is a signalling NaN or a quiet NaN. If FCMP or FCMPE generates an *fp_exception_ieee_754* exception, then fcc*n* is unchanged.

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

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

5.4.2 Rounding Direction (rd)

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

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

rd	Round Toward
0	Nearest (even, if tie)
1	0
2	$+\infty$
3	$-\infty$

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 UltraSPARC Architecture processor elects to implement floating-point nonstandard mode, see *Floating-Point Nonstandard Mode* on page 293.

5.4.5 FPU Version (ver)

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

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

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

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

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.

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

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

FSR.ftt encodes the primary condition ("floating-point trap type") that caused the generation of an *fp_exception_other* or *fp_exception_ieee_754* exception. It is possible for more than one such condition to occur simultaneously; in such a case, only the highest-priority condition will be encoded in FSR.ftt. The conditions leading to *fp_exception_other* and *fp_exception_ieee_754* exceptions, their relative priorities, and the corresponding FSR.ftt values are listed in TABLE 5-6. 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

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

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

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

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

The foregoing describes the result seen by a user trap handler if an IEEE exception is signalled, either immediately from an *fp_exception_ieee_754* exception or after recovery from an unfinished_FPop. In either case, cexc as seen by the trap handler reflects the exception causing the trap.

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

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

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

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

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

ftt = 3 (Reserved).

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

ftt = 4 (Reserved).

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

ftt = 5 (Reserved).

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

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

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

5.4.7 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 **and**ed together. If the result is nonzero, aexc is left unchanged and an *fp_exception_ieee_754* trap is generated; otherwise, the new cexc field is **or**ed into the aexc field and no trap is generated. Thus, while (and only while) traps are masked, exceptions are accumulated in the aexc field.

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

5.4.8 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 *fp_exception_ieee_754* trap *does* occur.

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

TABLE 5-7 Setting of FSR.cexc Bits

Conditions						Results			
Exception(s) Detected in F.p. operation			Trap Enable Mask bits (in FSR.tem)			*fp_exception_ieee_754* Trap Occurs?	Current Exception bits (in FSR.cexc)		
of	uf	nx	ofm	ufm	nxm		ofc	ufc	nxc
-	-	-	x	x	x	no	0	0	0
-	-	✔	x	x	0	no	0	0	1
-	✔[1]	✔[1]	x	0	0	no	0	1	1
✔[2]	-	✔[2]	0	x	0	no	1	0	1
-	-	✔	x	x	1	yes	0	0	1
-	✔[1]	✔[1]	x	0	1	yes	0	0	1
-	✔	-	x	1	x	yes	0	1	0
-	✔	✔	x	1	x	yes	0	1	0
✔[2]	-	✔[2]	1	x	x	yes	1	0	0
✔[2]	-	✔[2]	0	x	1	yes	0	0	1

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

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

5.4.9 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):

FSR.tem	RW	RW	RW	RW	RW
	nvm	ofm	ufm	dzm	nxm
	27	26	25	24	23

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

FSR.aexc	RW	RW	RW	RW	RW
	nva	ofa	ufa	dza	nxa
	9	8	7	6	5

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

FSR.cexc	RW	RW	RW	RW	RW
	nvc	ofc	ufc	dzc	nxc
	4	3	2	1	0

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

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

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

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

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

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

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

The SPARC V9 architecture allows tininess to be detected either before or after rounding. However, in all cases and regardless of the setting of FSR.tem.ufm, an UltraSPARC Architecture strand detects tininess before rounding (impl. dep. #55-V8-Cs10). See *Trapped Underflow Definition (ufm = 1)* on page 293 and *Untrapped Underflow Definition (ufm = 0)* on page 293 for additional details.

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

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

5.4.10 FSR Conformance

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

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

5.5 Ancillary State Registers

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

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

The SPARC V9 architecture left ASRs numbered 16–31 available for implementation-dependent uses. UltraSPARC Architecture virtual processors implement the ASRs summarized in TABLE 5-8 and defined in the following subsections.

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

TABLE 5-8 ASR Register Summary

ASR number	ASR name	Register	Read by Instruction(s)	Written by Instruction(s)
0	Y^{D}	Y register (deprecated)	RDY^{D}	WRY^{D}
1	—	*Reserved*	—	—
2	CCR	Condition Codes register	RDCCR	WRCCR
3	ASI	ASI register	RDASI	WRASI
4	TICK$^{P_{npt}}$	TICK register	RDTICK$^{P_{npt}}$, RDPR^{P} (TICK)	WRPR^{P} (TICK)
5	PC	Program Counter (PC)	RDPC	(all instructions)
6	FPRS	Floating-Point Registers Status register	RDFPRS	WRFPRS
7–14 (7-0E$_{16}$)	—	*Reserved*	—	—
15 (0F$_{16}$)	—	*Reserved*	—	—
16–31 (10$_{16}$-1F$_{16}$)		non-SPARC V9 ASRs	—	—
16-18 (10$_{16}$- 12$_{16}$)	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—
19 (13$_{16}$)	GSR	General Status register (GSR)	RDGSR, FALIGNDATA, many VIS and floating-point instructions	WRGSR, BMASK, SIAM
20 (14$_{16}$)	SOFTINT_SET^{P}	(pseudo-register, for "Write 1s Set" to SOFTINT register, ASR 22)	—	WRSOFTINT_SET^{P}
21 (15$_{16}$)	SOFTINT_CLR^{P}	(pseudo-register, for "Write 1s Clear" to SOFTINT register, ASR 22)	—	WRSOFTINT_CLR^{P}
22 (16$_{16}$)	SOFTINT^{P}	per-virtual processor Soft Interrupt register	RDSOFTINT^{P}	WRSOFTINT^{P}
23 (17$_{16}$)	TICK_CMPR^{P} N–	Tick Compare register	RDTICK_CMPR^{P} N–	WRTICK_CMPR^{P} N–
24 (18$_{16}$)	STICK$^{P_{npt}}$	System Tick register	RDSTICK$^{P_{npt}}$	—
25 (19$_{16}$)	STICK_CMPR^{P}	System Tick Compare register	RDSTICK_CMPR^{P}	WRSTICK_CMPR^{P}
26 (1A$_{16}$)	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—
27 (1B$_{16}$)	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—
28–29 (1C$_{16}$-1D$_{16}$)	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—
30 (1E$_{16}$)	—	*Reserved*	—	—
31 (1F$_{16}$)	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—

5.5.1 32-bit Multiply/Divide Register (Y) (ASR 0) (D2)

> The Y register is deprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC V9 software. It is recommended that all instructions that reference the Y register (that is, SMUL, SMULcc, UMUL, UMULcc, MULScc, SDIV, SDIVcc, UDIV, UDIVcc, RDY, and WRY) be avoided. For suitable substitute instructions, see the following pages: for the multiply instructions, see pages 246 and page 283; for the multiply step instruction, see page 209; for division instructions, see pages 240 and 281; for the read instruction, see page 226; and for the write instruction, see page 286.

The low-order 32 bits of the Y register, illustrated in FIGURE 5-9, contain the more significant word of the 64-bit product of an integer multiplication, as a result of either a 32-bit integer multiply (SMUL, SMULcc, UMUL, UMULcc) instruction or an integer multiply step (MULScc) instruction. The Y register also holds the more significant word of the 64-bit dividend for a 32-bit integer divide (SDIV, SDIVcc, UDIV, UDIVcc) instruction.

	R	*RW*
Y	0	product{63:32} or dividend{63:32}
	63 — 32	31 — 0

FIGURE 5-9 Y Register

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

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

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

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

	RW	*RW*
CCR	xcc	icc
	7 — 4	3 — 0

FIGURE 5-10 Condition Codes Register

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

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

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

	RW	*RW*	*RW*	*RW*
	n	z	v	c
xcc:	7	6	5	4
icc:	3	2	1	0

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

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

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

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

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

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

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

Both fields of CCR (xcc and icc) are modified by arithmetic and logical instructions, the names of which end with the letters "cc" (for example, ANDcc), and by the WRCCR instruction. They can be modified by a DONE or RETRY instruction, which replaces these bits with the contents of TSTATE.ccr. The behavior of the following instructions are conditioned by the contents of CCR.icc or CCR.xcc:

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

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

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

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

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

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

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

FIGURE 5-12 Address Space Identifier Register

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

FIGURE 5-13 illustrates the TICK register.

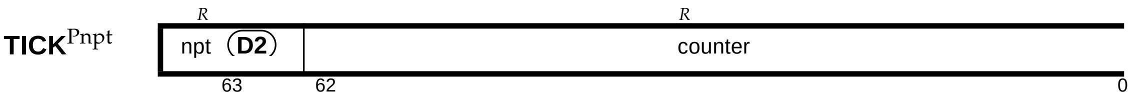


FIGURE 5-13 TICK Register

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

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

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

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

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

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

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

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

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

IMPL. DEP. #105-V9: (a) If an accurate count cannot always be returned when TICK is read, any inaccuracy should be small, bounded, and documented.
(b) An implementation may implement fewer than 63 bits in TICK.counter; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any upper bits not implemented must read as zero.

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

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

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

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

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

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

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

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

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

	RW	*RW*	*RW*
FPRS	fef	du	dl
	2	1	0

FIGURE 5-14 Floating-Point Registers State Register

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

Enable FPU (fef). Bit 2, fef, determines whether the FPU is enabled. If it is disabled, executing a floating-point instruction causes an *fp_disabled* trap. If this bit is set (FPRS.fef = 1) but the PSTATE.pef bit is not set (PSTATE.pef = 0), then executing a floating-point instruction causes an *fp_disabled* exception; that is, both FPRS.fef and PSTATE.pef must be set to 1 to enable floating-point operations.

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

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

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

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

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

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

The General Status Register[1] (GSR) is a nonprivileged read/write register that is implicitly referenced by many VIS instructions. The GSR can be read by the RDGSR instruction (see *Read Ancillary State Register* on page 225) and written by the WRGSR instruction (see *Write Ancillary State Register* on page 285).

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

The GSR is illustrated in FIGURE 5-15 and described in TABLE 5-10.

GSR^P

	RW		RW	RW		RW	RW
	mask	—	im	irnd	—	scale	align
Bits	63–32	31–28	27	26–25	24–8	7–3	2–0

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

TABLE 5-10 GSR Bit Description

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

5.5.8 SOFTINT^P Register (ASRs 20 A2 , 21 A2 , 22 A1)

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

SOFTINT A1 can be read with a RDSOFTINT instruction (see *Read Ancillary State Register* on page 225) and written with a WRSOFTINT, WRSOFTINT_SET, or WRSOFTINT_CLR instruction (see *Write Ancillary State Register* on page 285). An attempt to access to this register in nonprivileged mode causes a *privileged_opcode* exception.

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

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

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

		RW	RW	RW
SOFTINT^P	—	sm	int_level	tm
	63 … 17	16	15 … 1	0

FIGURE 5-16 SOFTINT Register (ASR 22)

TABLE 5-11 SOFTINT Bit Description

Bit	Field	Description
16	sm	When the STICK_CMPR (ASR 25) register's int_dis (interrupt disable) field is 0 (that is, System Tick Compare is enabled) and its stick_cmpr field matches the value in the STICK register, then SOFTINT.sm ("STICK match") is set to 1 and a level 14 interrupt (*interrupt_level_14*) is generated. See *System Tick Compare (stick_cmprP) Register (ASR 25)* on page 57 for details. SOFTINT.sm can also be directly written to 1 by software.
15:1	int_level	When SOFTINT.int_level{*n–1*} (SOFTINT{*n*}) is set to 1, an *interrupt_level_n* exception is generated. **Notes:** A level-14 interrupt (*interrupt_level_14*) can be triggered by SOFTINT.sm, SOFTINT.tm, or a write to SOFTINT.int_level{13} (SOFTINT{14}). 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).
0	tm N2	When the TICK_CMPR (ASR 23) register's int_dis (interrupt disable) field is 0 (that is, Tick Compare is enabled) and its tick_cmpr field matches the value in the TICK register, then the tm ("TICK match") field in SOFTINT is set to 1 and a level-14 interrupt (*interrupt_level_14*) is generated. See *Tick Compare (tick_cmprP) Register (ASR 23)* on page 56 for details. SOFTINT.tm can also be directly written to 1 by software.

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

See *Software Interrupt Register (softint)* on page 366 for additional information regarding the SOFTINT register.

5.5.8.1 SOFTINT_SET^P Pseudo-Register (ASR 20) A2

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

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

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

FIGURE 5-17 illustrates the SOFTINT_SET pseudo-register.

SOFTINT_SET^P

63 — 17	16 — 0
	W1s
—	ASR 22 bits to be set

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

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 54). That is, bits 16:0 of the write data are inverted and **and**ed into SOFTINT; any '1' bit in the write data causes the corresponding bit of SOFTINT to be set to 0. Bits 63:17 of the write data are ignored.

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

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

FIGURE 5-18 illustrates the SOFTINT_CLR pseudo-register.

SOFTINT_CLR^P

63 — 17	16 — 0
	W1c
—	ASR 22 bits to be cleared

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

5.5.9 Tick Compare (TICK_CMPR^P) Register (ASR 23) (D2)

The privileged TICK_CMPR register allows system software to cause a trap when the TICK register reaches a specified value. Nonprivileged accesses to this register cause a *privileged_opcode* exception (see *Exception and Interrupt Descriptions* on page 358).

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

TICK_CMPR^P

63	62 — 0
RW	*RW*
int_dis	tick_cmpr

FIGURE 5-19 TICK_CMPR Register

TABLE 5-12 TICK_CMPR Register Description

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

5.5.10 System Tick (STICK) Register (ASR 24) A1

The System Tick (STICK) register provides a counter that is synchronized across a system, useful for timestamping. The counter field of the STICK register is a 63-bit counter that increments at a rate determined by a clock signal external to the processor.

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

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

STICKPnpt

	R	R
STICKPnpt	npt	counter
	63	62 … 0

FIGURE 5-20 STICK Register

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

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

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

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

IMPL. DEP. #442-S10: (a) If an accurate count cannot always be returned when STICK is read, any inaccuracy should be small, bounded, and documented.
(b) An implementation may implement fewer than 63 bits in STICK.counter; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any upper bits not implemented must read as zero.

5.5.11 System Tick Compare (STICK_CMPRP) Register (ASR 25) A2

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

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

	RW	RW
STICK_CMPRP	int_dis	stick_cmpr
	63	62 … 0

FIGURE 5-21 STICK_CMPR Register

TABLE 5-13 STICK_CMPR Register Description

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

5.6 Register-Window PR State Registers

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

IMPL. DEP. #126-V9-Ms10: Privileged registers CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN contain values in the range 0 to *N_REG_WINDOWS* – 1. An attempt to write a value greater than *N_REG_WINDOWS* – 1 to any of these registers causes an implementation-dependent value between 0 and *N_REG_WINDOWS* – 1 (inclusive) to be written to the register. Furthermore, an attempt to write a value greater than *N_REG_WINDOWS* – 2 violates the register window state definition in *Register Window State Definition* on page 60.
Although the width of each of these five registers is architecturally 5 bits, the width is implementation dependent and shall be between $\lceil \log_2(N\_REG\_WINDOWS) \rceil$ and 5 bits, inclusive. If fewer than 5 bits are implemented, the unimplemented upper bits shall read as 0 and writes to them shall have no effect. All five registers should have the same width.
For UltraSPARC Architecture 2007 processors, *N_REG_WINDOWS* = 8. Therefore, each register window state register is implemented with 3 bits, the maximum value for CWP and CLEANWIN is 7, and the maximum value for CANSAVE, CANRESTORE, and OTHERWIN is 6. When these registers are written by the WRPR instruction, bits 63:3 of the data written are ignored.

For details of how the window-management registers are used, see *Register Window Management Instructions* on page 83.

Programming Note CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN must never be set to a value greater than *N_REG_WINDOWS* – 2 on an UltraSPARC Architecture virtual processor. Setting any of these to a value greater than *N_REG_WINDOWS* – 2 violates the register window state definition in *Register Window State Definition* on page 60. Hardware is not required to enforce this restriction; it is up to system software to keep the window state consistent.

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

5.6.1 Current Window Pointer (CWP^P) Register (PR 9) A1

The privileged CWP register, shown in FIGURE 5-22, is a counter that identifies the current window into the array of integer registers. See *Register Window Management Instructions* on page 83 and Chapter 12, *Traps*, for information on how hardware manipulates the CWP register.

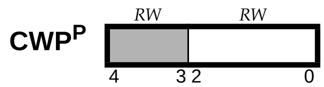


FIGURE 5-22 Current Window Pointer Register

5.6.2 Savable Windows (CANSAVE^P) Register (PR 10) A1

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

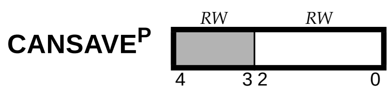


FIGURE 5-23 CANSAVE Register, Figure 5-24, page 88

5.6.3 Restorable Windows (CANRESTORE^P) Register (PR 11) A1

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

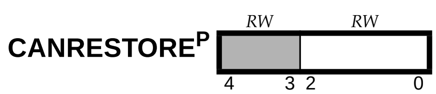


FIGURE 5-24 CANRESTORE Register

5.6.4 Clean Windows (CLEANWIN^P) Register (PR 12) 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.

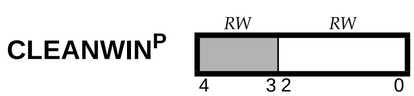


FIGURE 5-25 CLEANWIN Register

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

5.6.5 Other Windows (OTHERWIN^{P}) Register (PR 13) 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.

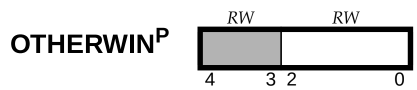


FIGURE 5-26 OTHERWIN Register

5.6.6 Window State (WSTATE^{P}) Register (PR 14) 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.

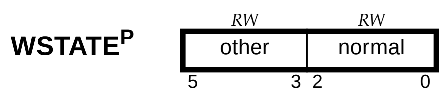


FIGURE 5-27 WSTATE Register

5.6.7 Register Window Management

The state of the register windows is determined by the contents of the set of privileged registers described in *Register-Window PR State Registers* on page 58. Those registers are affected by the instructions described in *Register Window Management Instructions* on page 83. Privileged software can read/write these state registers directly by using RDPR/WRPR instructions.

5.6.7.1 Register Window State Definition

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

CANSAVE + CANRESTORE + OTHERWIN = *N_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_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,

CLEANWIN ≥ CANRESTORE

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

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

Programming Note | System software is responsible for keeping the state of the register windows consistent at all times. Failure to do so will cause undefined behavior. For example, CANSAVE, CANRESTORE, and OTHERWIN must never be greater than or equal to *N_REG_WINDOWS* – 1.

5.6.7.2 Register Window Traps

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

See *Register Window Traps* on page 362 for a detailed description of how fill, spill, and *clean_window* traps support register windowing.

5.7 Non-Register-Window PR State Registers

The registers described in this section are visible only to software running in privileged mode (that is, when 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 *MAXPTL* instances of the TPC, but only one is accessible at any time. The current value in the TL register determines which instance of the TPC[TL] register is accessible. An attempt to read or write the TPC register when TL = 0 causes an *illegal_instruction* exception.

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

TABLE 5-14 lists the events that cause TPC to be read or written.

	RW	R
$TPC_1{}^P$	pc_high62 (PC{63:2} from trap while TL = 0)	00
$TPC_2{}^P$	pc_high62 (PC{63:2} from trap while TL = 1)	00
$TPC_3{}^P$	pc_high62 (PC{63:2} from trap while TL = 2)	00
:	:	:
$TPC_{MAXPTL}{}^P$	pc_high62 (PC{63:2} from trap while TL = ***MAXPTL*** – 1)	00
	63 2	1 0

FIGURE 5-28 Trap Program Counter Register Stack

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

Event	Effect
Trap	TPC[$n + 1$] ← PC
RETRY instruction	PC ← TPC[n]
RDPR (TPC)	R[rd] ← TPC[n]
WRPR (TPC)	TPC[n] ← *value*

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 *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.

	RW	R
$TNPC_1{}^P$	npc_high62 (NPC{63:2} from trap while TL = 0)	00
$TNPC_2{}^P$	npc_high62 (NPC{63:2} from trap while TL = 1)	00
$TNPC_3{}^P$	npc_high62 (NPC{63:2} from trap while TL = 2)	00
:	:	:
$TNPC_{MAXPTL}{}^P$	npc_high62 (NPC{63:2} from trap while TL = ***MAXPTL*** – 1)	00
	63 2	1 0

FIGURE 5-29 Trap Next Program Counter Register Stack

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

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

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

Event	Effect
Trap	TNPC[$n + 1$] ← NPC
DONE instruction	PC ← TNPC[n]; NPC ← TNPC[n] + 4
RETRY instruction	NPC ← TNPC[n]
RDPR (TNPC)	R[rd] ← TNPC[n]
WRPR (TNPC)	TNPC[n] ← *value*

5.7.3 Trap State (TSTATE[P]) Register (PR 2) (A1)

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

	RW	*RW*	*RW*	*R*	*RW*	*R*	*RW*
$TSTATE_1$ P	gl (GL from TL = 0)	ccr (CCR from TL = 0)	asi (ASI from TL = 0)	—	pstate (PSTATE from TL = 0)	—	cwp (CWP from TL = 0)
$TSTATE_2$ P	gl (GL from TL = 1)	ccr (CCR from TL = 1)	asi (ASI from TL = 1	—	pstate (PSTATE from TL = 1)	—	cwp (CWP from TL = 1)
$TSTATE_3$ P	gl (GL from TL = 2)	ccr (CCR from TL = 2)	asi (ASI from TL = 2	—	pstate (PSTATE from TL = 2)	—	cwp (CWP from TL = 2)
: P	:	:	:	:	:	:	:
$TSTATE_{MAXPTL}$ P	gl (GL from TL = *MAXPTL* − 1)	ccr (CCR from TL = *MAXPTL* − 1)	asi (ASI from TL = *MAXPTL* − 1)	—	pstate (PSTATE from TL = *MAXPTL* − 1)	—	cwp (CWP from TL = *MAXPTL* − 1)
	42 40	39 32	31 24	23 21	20 8	7 5	4 0

FIGURE 5-30 Trap State (TSTATE) Register Stack

During normal operation the value of TSTATE[n], when n is greater than the current trap level ($n >$ TL), is undefined.

V9 Compatibility Note Because there are more bits in the UltraSPARC Architecture's PSTATE register than in a SPARC V9 PSTATE register, a 13-bit PSTATE value is stored in TSTATE instead of the 10-bit value specified in the SPARC V9 architecture.

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

TABLE 5-16 Events That Involve TSTATE, When Executing with TL = n

Event	Effect
Trap	TSTATE[$n + 1$] ← (registers)
DONE instruction	(registers) ← TSTATE[n]
RETRY instruction	(registers) ← TSTATE[n]
RDPR (TSTATE)	R[rd] ← TSTATE[n]
WRPR (TSTATE)	TSTATE[n] ← *value*

5.7.4 Trap Type (TT^P) Register (PR 3) (A1)

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.

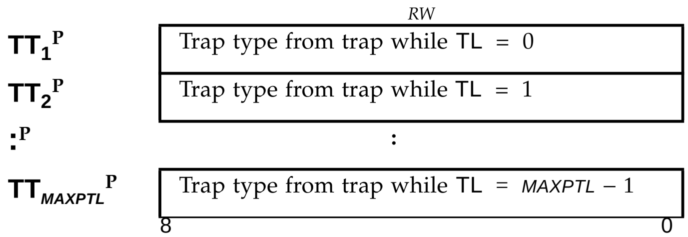


FIGURE 5-31 Trap Type Register Stack

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

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

TABLE 5-17 Events that involve TT, when executing with TL = *n*.

Event	Effect
Trap	TT[*n* + 1] ← (trap type)
RDPR (TT)	R[rd] ← TT[*n*]
WRPR (TT)	TT[*n*] ← *value*

5.7.5 Trap Base Address (TBA^P) Register (PR 5) (A1)

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

FIGURE 5-32 Trap Base Address Register

Details on how the full address for a trap vector is generated, using TBA and other state, are provided in *Trap-Table Entry Address to Privileged Mode* on page 348.

5.7.6 Processor State (PSTATE^P) Register (PR 6) (A1)

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

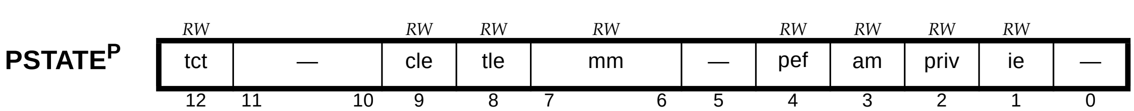


FIGURE 5-33 PSTATE Field

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

The following subsections describe the fields of the PSTATE register.

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

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

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

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

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

For the purposes of the *control_transfer_instruction* exception, a discontinuity in instruction-fetch addresses caused by a WRPR to PSTATE that changes the value of PSTATE.am (and thus, potentially the more-significant 32 bits of the address of the next instruction; see page 67) is *not* considered a control transfer. Only explicit CTIs can generate a *control_transfer_instruction* exception.

Current Little Endian (cle). This bit affects the endianness of data accesses performed using an implicit ASI. When PSTATE.cle = 1, all data accesses using an implicit ASI are performed in little-endian byte order. When PSTATE.cle = 0, all data accesses using an implicit ASI are performed in big-endian byte order. Specific ASIs used are shown in TABLE 6-3 on page 76. Note that the endianness of a data access may be further affected by TTE.ie used by the MMU.

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

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

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

Memory Model (mm). This 2-bit field determines the memory model in use by the virtual processor. The defined values for an UltraSPARC Architecture virtual processor are listed in TABLE 5-18.

TABLE 5-18 PSTATE.mm Encodings

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

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

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

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

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

SPARC V9 Compatibility Notes

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

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

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

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

Address Mask (am). The PSTATE.am bit is provided to allow 32-bit SPARC software to run correctly on a 64-bit SPARC processor. When PSTATE.am = 1, bits 63:32 of virtual addresses are masked out (treated as 0). PSTATE.am does not affect real addresses.

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

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

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

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

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

Instances in which the more-significant 32 bits of a virtual address **are 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 virtual address is sent to a watchpoint comparator

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

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

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

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

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

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

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

  Programming Note | Since writes to PSTATE are nondelayed (see page 65), a change to PSTATE.am can affect which instruction is executed immediately after the write to PSTATE.am. Specifically, if a WRPR to the PSTATE register changes the value of PSTATE.am from '0' to '1', and NPC{63:32} when the WRPR began execution was nonzero, then the next instruction executed after the WRPR will be from the address indicated in NPC{31:0} (with the more-significant 32 address bits set to zero).

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

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

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 116 and *Edge Handling Instructions (no CC)* on page 118.

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 346.

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

5.7.7 Trap Level Register (TL^P) (PR 7) (A1)

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

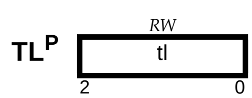


FIGURE 5-34 Trap Level Register

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

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

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

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

The effect of writing to TL with a WRPR instruction is summarized in TABLE 5-19.

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

Value x Written with WRPR	Privilege Level when Executing WRPR		
	Nonprivileged	**Privileged**	
$x \leq$ *MAXPTL*	*privileged_opcode* exception	TL $\leftarrow x$	
$x >$ *MAXPTL*		TL $\leftarrow$ *MAXPTL* (no exception generated)	

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

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

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

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

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

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

The privileged Processor Interrupt Level register (PIL; see FIGURE 5-35) specifies the interrupt level above which the virtual processor will accept an *interrupt_level_n* interrupt. Interrupt priorities are mapped so that interrupt level 2 has greater priority than interrupt level 1, and so on. See TABLE 12-4 on page 351 for a list of exception and interrupt priorities.

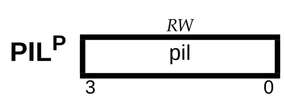


FIGURE 5-35 Processor Interrupt Level Register

V9 Compatibility Note On SPARC V8 processors, the level 15 interrupt is considered to be nonmaskable, so it has different semantics from other interrupt levels. SPARC V9 processors do not treat a level 15 interrupt differently from other interrupt levels.

5.7.9 Global Level Register (GL^{P}) (PR 16) A1

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

FIGURE 5-36 illustrates the Global Level register.

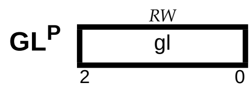


FIGURE 5-36 Global Level Register, GL

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

The valid range of values that the GL register may contain is 0 to *MAXPGL*, where *MAXPGL* is one fewer than the number of global register sets available to the virtual processor.

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

In all UltraSPARC Architecture 2007 implementations, *MAXPGL* = 2 (impl. dep. #401-S10).

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

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

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

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

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

TABLE 5-20 Effect of WRPR to Register GL

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

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

CHAPTER 6

Instruction Set Overview

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

6.1 Instruction Execution

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

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

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

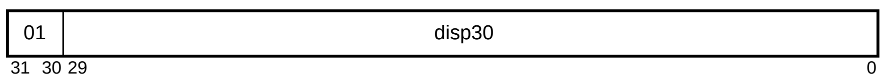
For each instruction access and each normal data access, an 8-bit address space identifier (ASI) is appended to the 64-bit memory address. Load/store alternate instructions (see *Address Space Identifiers (ASIs)* on page 76) can provide an arbitrary ASI with their data addresses or can use the ASI value currently contained in the ASI register.

6.2 Instruction Formats

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

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

op = 01_2: *CALL*



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

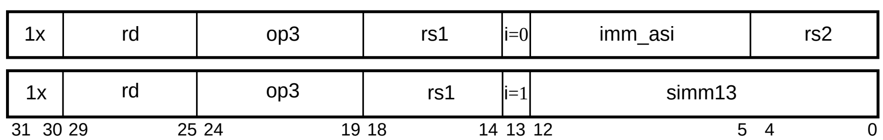


FIGURE 6-1 Summary of Instruction Formats

6.3 Instruction Categories

UltraSPARC Architecture instructions can be grouped into the following categories:

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

These categories are described in the following subsections.

6.3.1 Memory Access Instructions

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

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

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

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

Programming Note | For some instructions, by use of simm13, any location in the lowest or highest 4 Kbytes of an address space can be accessed without the use of a register to hold part of the address.

6.3.1.1 Memory Alignment Restrictions

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

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

A floating-point quadword access (LDQF, LDQFA, STQF, STQFA) should be aligned on a 16-byte boundary, but is only required to be aligned on a word (4-byte) boundary. A floating-point quadword access to an address that is 4-byte or 8-byte aligned but not 16-byte aligned may result in less efficient and nonatomic access (causes a trap and is emulated in software (impl. dep. #111-V9-Cs10)), so 16-byte alignment is recommended.

An improperly aligned address in a load, store, or load-store instruction causes a *mem_address_not_aligned* exception to occur, with these exceptions:

- An LDDF or LDDFA instruction accessing an address that is word aligned but not doubleword aligned may cause an *LDDF_mem_address_not_aligned* exception (impl. dep. #109-V9-Cs10).
- An STDF or STDFA instruction accessing an address that is word aligned but not doubleword aligned may cause an *STDF_mem_address_not_aligned* exception (impl. dep. #110-V9-Cs10).

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

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

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

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

6.3.1.2 Addressing Conventions

An UltraSPARC Architecture virtual processor uses big-endian byte order for all instruction accesses and, by default, for data accesses. It is possible to access data in little-endian format by use of selected ASIs. It is also possible to change the default byte order for implicit data accesses. See *Processor State (pstateP) Register (PR 6)* on page 64 for more information.[1]

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

TABLE 6-1 Big-endian Addressing Conventions

Term	Definition
byte	A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.
halfword	For a load/store halfword instruction, two bytes are accessed. The most significant byte (bits 15–8) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 1.
word	For a load/store word instruction, four bytes are accessed. The most significant byte (bits 31–24) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 3.
doubleword or extended word	For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The most significant byte (bits 63:56) is accessed at the address specified in the instruction; the least significant byte (bits 7:0) is accessed at the address + 7. For the deprecated integer load/store 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.
quadword	For a load/store quadword instruction, 16 bytes are accessed. The most significant byte (bits 127–120) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 15.

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.

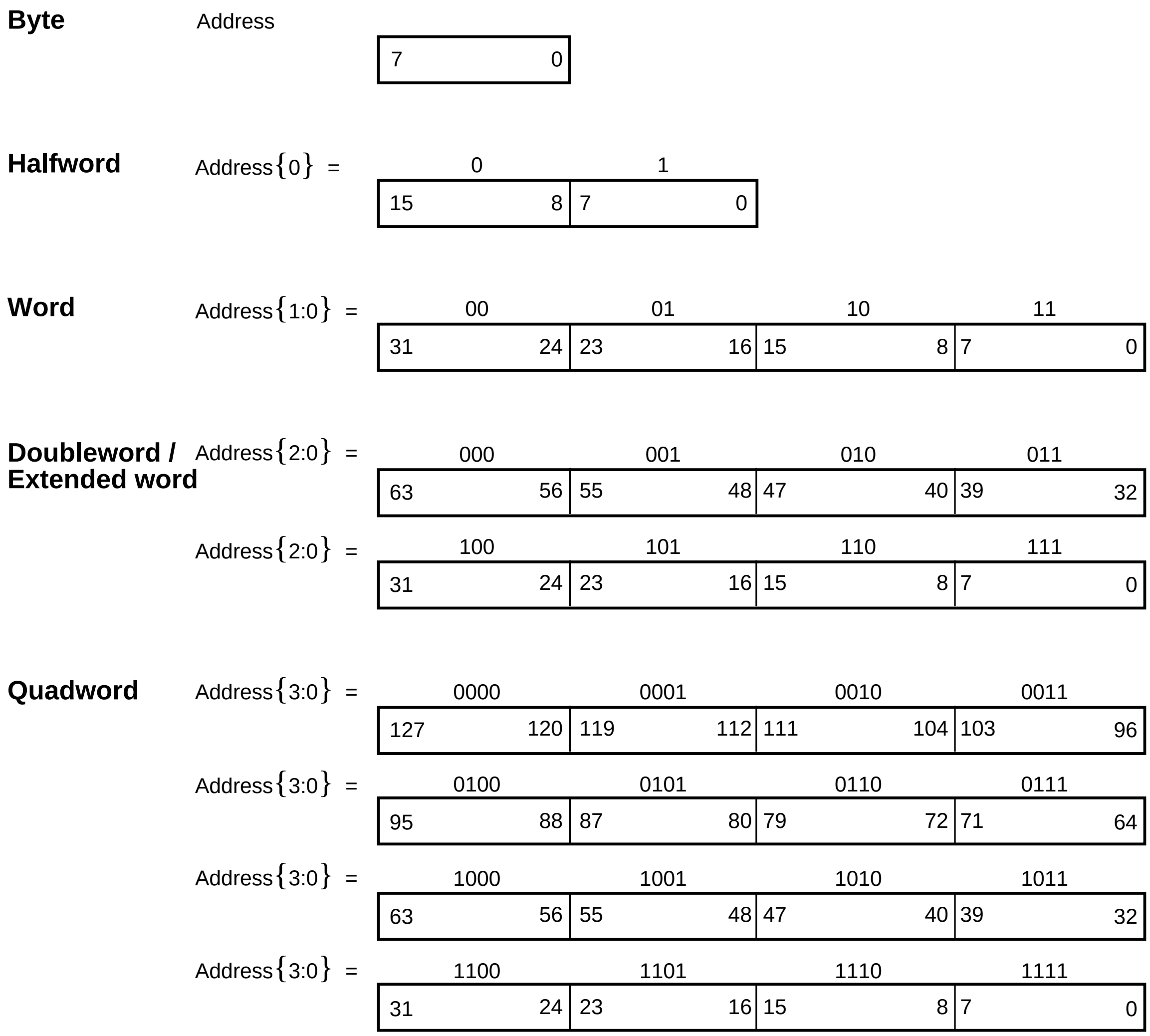


FIGURE 6-2 Big-endian Addressing Conventions

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

TABLE 6-2 Little-endian Addressing Convention

Term	Definition
byte	A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.
halfword	For a load/store halfword instruction, two bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 15–8) is accessed at the address + 1.
word	For a load/store word instruction, four bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 31–24) is accessed at the address + 3.
doubleword or extended word	For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The least significant byte (bits 7–0) is accessed at the address specified in the instruction; the most significant byte (bits 63–56) is accessed at the address + 7. 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 imm_asi field, and when i = 1, it is provided in the ASI register.

Non-alternate-space load, store, and load-store instructions use an *implicit* ASI value that depends on the current trap level (TL) and the value of PSTATE.cle. Instruction fetches use an implicit ASI that depends only on the current trap level. The cases are enumerated in TABLE 6-3.

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

Access Type	TL	PSTATE.cle	ASI Used
Instruction Fetch	= 0	any	`ASI_PRIMARY`
	> 0	any	`ASI_NUCLEUS*`

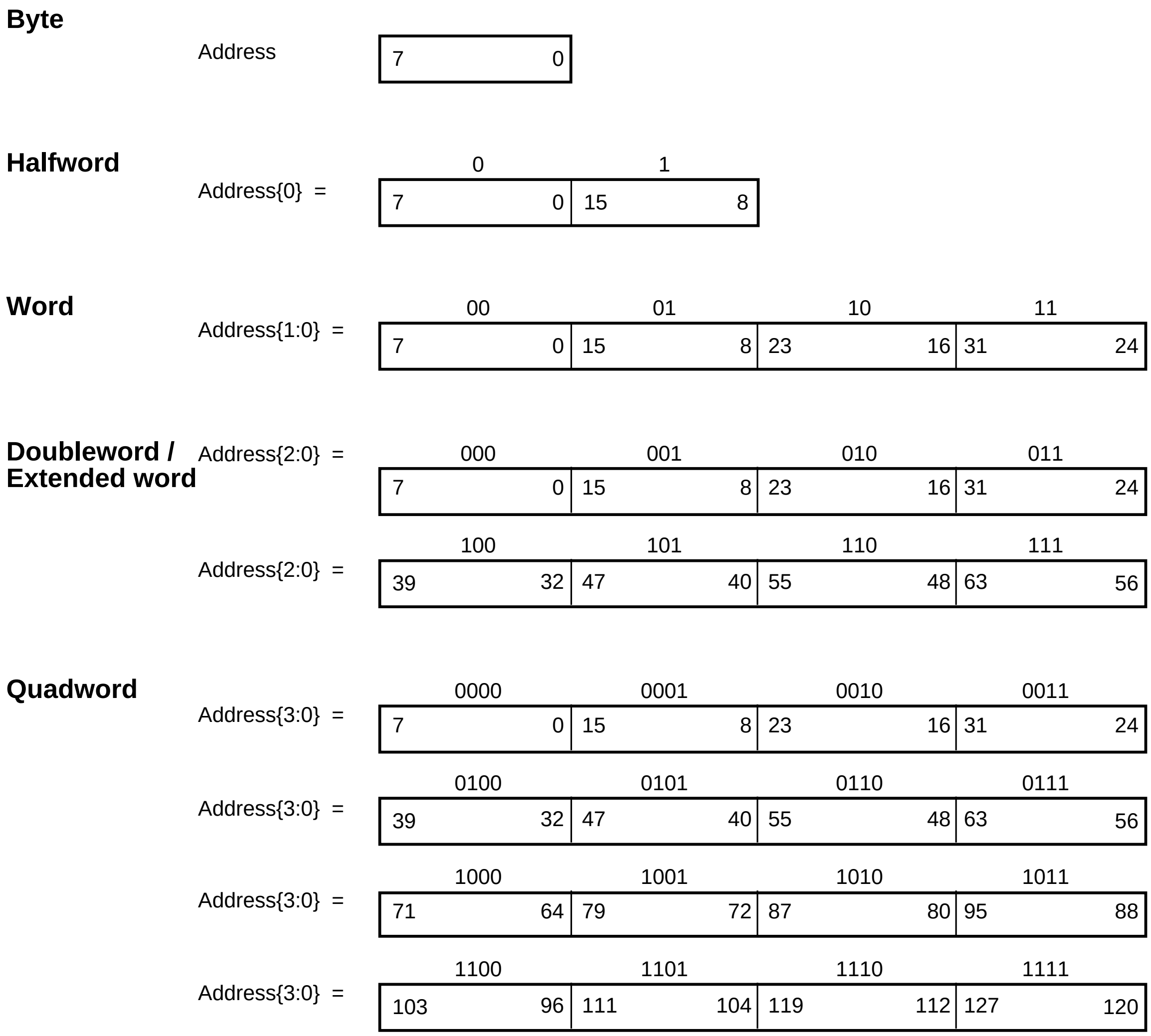


FIGURE 6-3 Little-endian Addressing Conventions

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

Access Type	TL	PSTATE.cle	ASI Used
Non-alternate-space Load, Store, or Load-Store	= 0	0	`ASI_PRIMARY`
		1	`ASI_PRIMARY_LITTLE`
	> 0	0	`ASI_NUCLEUS`*
		1	`ASI_NUCLEUS_LITTLE`**
Alternate-space Load, Store, or Load-Store	any	any	ASI explicitly specified in the instruction (subject to privilege-level restrictions)

*On some early SPARC V9 implementations, `ASI_PRIMARY` may have been used for this case.
**On some early SPARC V9 implementations, `ASI_PRIMARY_LITTLE` may have been used for this case.

See also *Memory Addressing and Alternate Address Spaces* on page 308.

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

TABLE 6-4 Allowed Accesses to ASIs

Value	Access Type	Processor Mode (PSTATE.priv)	Result of ASI Access
00_{16}–$7F_{16}$	Restricted	Nonprivileged (0)	*privileged_action* exception
		Privileged (1)	Valid access
80_{16}–FF_{16}	Unrestricted	Nonprivileged (0)	Valid access
		Privileged (1)	Valid access

IMPL. DEP. #29-V8: Some UltraSPARC Architecture 2007 ASIs are implementation dependent. See TABLE 10-1 on page 323 for details.

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

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

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

6.3.1.4 Separate Instruction Memory

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

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

6.3.2 Memory Synchronization Instructions

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

6.3.3 Integer Arithmetic and Logical Instructions

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

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

6.3.3.1 Setting Condition Codes

Most integer arithmetic instructions have two versions: one sets the integer condition codes (icc and xcc) as a side effect; the other does not affect the condition codes. A special comparison instruction for integer values is not needed since it is easily synthesized with the "subtract and set condition codes" (SUBcc) instruction. See *Synthetic Instructions* on page 414 for details.

6.3.3.2 Shift Instructions

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

6.3.3.3 Set High 22 Bits of Low Word

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

6.3.3.4 Integer Multiply/Divide

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

6.3.3.5 Tagged Add/Subtract

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

The trapping versions (TADDccTV, TSUBccTV) of these instructions are deprecated. See *Tagged Add* on page 274 and *Tagged Subtract* on page 279 for details.

6.3.4 Control-Transfer Instructions (CTIs)

The basic control-transfer instruction types are as follows:

- Conditional branch (Bicc, BPcc, BPr, FBfcc, FBPfcc)
- Unconditional branch
- Call and link (CALL)
- Jump and link (JMPL, RETURN)

- Return from trap (DONE, RETRY)
- Trap (Tcc)
-

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

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

Programming Note The annul bit increases the likelihood that a compiler can find a useful instruction to fill the delay slot after a branch, thereby reducing the number of instructions executed by a program. For example, the annul bit can be used to move an instruction from within a loop to fill the delay slot of the branch that closes the loop.

Likewise, the annul bit can be used to move an instruction from either the "else" or "then" branch of an "if-then-else" program block to the delay slot of the branch that selects between them. Since a full set of conditions is provided, a compiler can arrange the code (possibly reversing the sense of the condition) so that an instruction from either the "else" branch or the "then" branch can be moved to the delay slot. Use of annulled branches provided some benefit in older, single-issue SPARC implementations. On an UltraSPARC Architecture implementation, the only benefit of annulled branches might be a slight reduction in code size. Therefore, the use of annulled branch instructions is no longer encouraged.

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

TABLE 6-5 Control-Transfer Characteristics *(1 of 2)*

Instruction Group	Address Form	Delayed?	Taken?	Annul Bit?	New PC	New NPC
Non-CTIs	—	—	—	—	NPC	NPC + 4
Bcc	PC-relative	Yes	Yes	0	NPC	EA
Bcc	PC-relative	Yes	No	0	NPC	NPC + 4
Bcc	PC-relative	Yes	Yes	1	NPC	EA
Bcc	PC-relative	Yes	No	1	NPC + 4	NPC + 8
BA	PC-relative	Yes	Yes	0	NPC	EA
BA	PC-relative	No	Yes	1	EA	EA + 4
BN	PC-relative	Yes	No	0	NPC	NPC + 4
BN	PC-relative	Yes	No	1	NPC + 4	NPC + 8
CALL	PC-relative	Yes	—	—	NPC	EA

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

Instruction Group	Address Form	Delayed?	Taken?	Annul Bit?	New PC	New NPC
JMPL, RETURN	Register-indirect	Yes	—	—	NPC	EA
DONE	Trap state	No	—	—	TNPC[TL]	TNPC[TL] + 4
RETRY	Trap state	No	—	—	TPC[TL]	TNPC[TL]
Tcc	Trap vector	No	Yes	—	EA	EA + 4
Tcc	Trap vector	No	No	—	NPC	NPC + 4

The effective address, “EA” in TABLE 6-5, specifies the target of the control-transfer instruction. The effective address is computed in different ways, depending on the particular instruction.

- **PC-relative effective address —** A PC-relative effective address is computed by sign extending the instruction’s immediate field to 64-bits, left-shifting the word displacement by 2 bits to create a byte displacement, and adding the result to the contents of the PC.
- **Register-indirect effective address —** 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] + **sign_ext**(*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] + *imm_trap#* if i = 1. Whether 7 or 8 bits are used depends on the privilege level — 7 bits are used in nonprivileged mode and 8 bits are used in privileged mode. The trap level, TL, is incremented. The hardware trap type is computed as 256 + the software trap number and stored in TT[TL]. The effective address is generated by combining the contents of the TBA register with the trap type and other data; see *Trap Processing* on page 356 for details.
- **Trap state effective address —** A trap state effective address is not computed but is taken directly from either TPC[TL] or TNPC[TL].

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

6.3.4.1 Conditional Branches

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

Note The 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 (E2)

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

SPARC V8 and SPARC V9 Compatibility Note | The SPARC V8 architecture left behavior undefined for a DCTI couple. The SPARC V9 architecture defined behavior in that case, but as of UltraSPARC Architecture 2005, *use of DCTI couples was deprecated*.

6.3.5 Conditional Move Instructions

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

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

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

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

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

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

can be coded as

```
cmp             %i0, %i2     ! (A > B)
or              %g0, 0, %i3  ! set X  = 0
movg            %xcc, 1, %i3 ! overwrite X with 1 if A > B
```

to eliminate the need for a branch.

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

TABLE 6-6 MOVr and FMOVr Test Conditions

Condition	Description
NZ	Nonzero
Z	Zero
GEZ	Greater than or equal to zero
LZ	Less than zero
LEZ	Less than or equal to zero
GZ	Greater than zero

Any of the integer registers (treated as a signed value) may be tested for one of the conditions, and the result used to control the move. For example,

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

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

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

6.3.6 Register Window Management Instructions

This subsection describes the instructions that manage register windows in the UltraSPARC Architecture. The privileged registers affected by these instructions are described in *Register-Window PR State Registers* on page 58.

6.3.6.1 SAVE Instruction

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

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

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

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

6.3.6.2 RESTORE Instruction

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

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

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

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

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

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

The SAVE and RESTORE instructions can be used to atomically establish a new memory stack pointer in an R register and switch to a new or previous register window.

6.3.6.3 SAVED Instruction

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

See *SAVED* on page 239 for details.

6.3.6.4 RESTORED Instruction

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

See *RESTORED* on page 232 for details.

6.3.6.5 Flush Windows Instruction

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

N_REG_WINDOWS – 2 – CANSAVE

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

6.3.7 Ancillary State Register (ASR) Access

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

The supported RDasr and WRasr instructions are described in *Ancillary State Registers* on page 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 fcc*n* fields of the FSR.

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

The FMOVcc instructions function for the floating-point registers as the MOVcc instructions do for the integer registers. See *MOVcc and FMOVcc Instructions* on page 83.

The FMOVr instructions function for the floating-point registers as the MOVr instructions do for the integer registers. See *MOVr and FMOVr Instructions* on page 83.

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

All FPop instructions clear the ftt field and set the cexc field unless they generate an exception. Floating-point compare instructions also write one of the fcc*n* fields. All FPop instructions that can generate IEEE exceptions set the cexc and aexc fields unless they generate an exception. FABS<s|d|q>, FMOV<s|d|q>, FMOVcc<s|d|q>, FMOVr<s|d|q>, and FNEG<s|d|q> cannot generate IEEE exceptions, so they clear cexc and leave aexc unchanged.

IMPL. DEP. #3-V8: An implementation may indicate that a floating-point instruction did not produce a correct IEEE Std 754-1985 result by generating an *fp_exception_other* exception with FSR.ftt = unfinished_FPop. In this case, software running in a mode with greater privileges must emulate any functionality not present in the hardware.

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

6.3.10 Implementation-Dependent Instructions

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

In the UltraSPARC Architecture, the IMPDEP1 opcode space is used by many VIS instructions. The remaining opcodes in IMPDEP1 and IMPDEP2 are now marked as reserved opcodes.

6.3.11 Reserved Opcodes and Instruction Fields

If a conforming UltraSPARC Architecture 2007 implementation attempts to execute an instruction bit pattern that is not specifically defined in this specification, it behaves as follows:

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

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

Programming Note | For software portability, software (such as assemblers, static compilers, and dynamic compilers) that generates SPARC instructions must always generate zeroes in instruction fields marked “reserved” (“—”).

CHAPTER 7

Instructions

UltraSPARC Architecture 2007 extends the standard SPARC V9 instruction set with additional classes of instructions:

- Enhanced functionality:
  - Instructions for alignment (*Align Address* on page 98)
  - Array handling (*Three-Dimensional Array Addressing* on page 101)
  - Byte-permutation instructions (*Byte Mask and Shuffle* on page 106)
  - Edge handling (*Edge Handling Instructions* on pages 116 and 118)
  - Logical operations on floating-point registers (*f Register Logical Operate (1 operand)* on page 163)
  - Partitioned arithmetic (*Fixed-point Partitioned Add* on page 158*Fixed-point Partitioned Subtract (64-bit)* on page 161)
  - Pixel manipulation (*FEXPAND* on page 131, *FPACK* on page 153, and *FPMERGE* on page 160)
  -
- Efficient memory access
  - Partial store (*Store Partial Floating-Point* on page 260)
  - Short floating-point loads and stores (*Store Short Floating-Point* on page 263)
  - Block load and store (*Block Load* on page 178 and *Block Store* on page 250)
- Efficient interval arithmetic: SIAM (*Set Interval Arithmetic Mode* on page 243) and all instructions that reference GSR.im
- Floating-point Multiply-Add and Multiply-Subtract (FMA) instructions (*Floating-Point Multiply-Add and Multiply-Subtract (fused)* on page 137

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

TABLE 7-3 summarizes the instruction set, listed within functional categories.

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

TABLE 7-1 Instruction Superscripts

Superscript	Meaning
D	Deprecated instruction (do not use in new software)
N	Nonportable instruction
P	Privileged instruction
P_{ASI}	Privileged action if bit 7 of the referenced ASI is 0
P_{ASR}	Privileged instruction if the referenced ASR register is privileged
P_{npt}	Privileged action if in nonprivileged mode (PSTATE.priv = 0) and nonprivileged access is disabled

TABLE 7-2 *UltraSPARC Architecture 2007Instruction Set - Alphabetical (1 of 3)*

Page	Instruction				
97	ADD (ADDcc)	151	FdMULq	161	FPSUB<16,32>[S]
97	ADDC (ADDCcc)	131	FEXPAND		
		132	FiTO<s|d|q>		
98	ALIGNADDRESS[_LITTLE]			151	FsMULd
99	ALLCLEAN	133	FLUSH	166	FSQRT<s|d|q>
100	AND (ANDcc)	136	FLUSHW	164	FSRC<1|2>[s]
101	ARRAY<8|16|32>	137	FMADD(s,d)	170	FSUB<s|d|q>
104	Bicc				
106	BMASK	139	FMOV<s|d|q>	165	FXNOR[s]
107	BPcc	140	FMOV<s|d|q>cc	165	FXOR[s]
109	BPr	144	FMOV<s|d|q>R	171	FxTO<s|d|q>
106	BSHUFFLE	137	FMSUB(s,d)	163	FZERO[s]
111	CALL	151	FMUL<s|d|q>		
112	CASA[PASI]	146	FMUL8[SU|UL]x16	172	ILLTRAP
112	CASXA[PASI]	146	FMUL8x16	173	INVALW
		146	FMUL8x16[AU|AL]	174	JMPL
		146	FMULD8[SU|UL]x16		
				178	LDBLOCKF
114	DONE[P]	165	FNAND[s]	181	LDDF
116	EDGE<8|16|32>[L]cc	152	FNEG<s|d|q>	183	LDDFA[PASI]
118	EDGE<8|16|32>[L]N			181	LDF
168	F<s|d|q>TO<s|d|q>	137	FNMADD	183	LDFA[PASI]
167	F<s|d|q>TOi	137	FNMSUB	186	LDFSR[D]
167	F<s|d|q>TOx			181	LDQF
119	FABS<s|d|q>	165	FNOR[s]	183	LDQFA[PASI]
120	FADD<s|d|q>	164	FNOT<1|2>[s]	175	LDSB
121	FALIGNDATA			176	LDSBA[PASI]
165	FANDNOT<1|2>[s]	163	FONE[s]	175	LDSH
165	FAND[s]	165	FORNOT<1|2>[s]	176	LDSHA[PASI]
122	FBfcc[D]	165	FOR[s]	188	LDSHORTF
124	FBPfcc	153	FPACK<16|32|FIX>	190	LDSTUB
				191	LDSTUBA[PASI]
128	FCMP<s|d|q>	158	FPADD<16,32>[S]	175	LDSW
126	FCMP*<16,32>			176	LDSWA[PASI]
128	FCMPE<s|d|q>	160	FPMERGE	197	LDTXA[N]
130	FDIV<s|d|q>				

TABLE 7-2 *UltraSPARC Architecture 2007Instruction Set - Alphabetical (2 of 3)*

Page	Instruction				
192	LDTW^{D}			258	STFSR^{D}
194	LDTWA$^{D,\ P_{ASI}}$	225	RDPC	247	STH
190	LDUB			248	STHA$^{P_{ASI}}$
176	LDUBA$^{P_{ASI}}$			260	STPARTIALF
175	LDUH	228	RDPR^{P}	253	STQF
176	LDUHA$^{P_{ASI}}$	225	RDSOFTINT^{P}	255	STQFA$^{P_{ASI}}$
175	LDUW	225	RDSTICK_CMPR^{P}	263	STSHORTF
176	LDUWA$^{P_{ASI}}$	225	RDSTICK$^{P_{npt}}$	265	STTW^{D}
175	LDX	225	RDTICK_CMPR^{P}	267	STTWA$^{D,\ P_{ASI}}$
176	LDXA$^{P_{ASI}}$	225	RDTICK$^{P_{npt}}$	247	STW
		232	RESTORED^{P}	248	STWA$^{P_{ASI}}$
199	LDXFSR	230	RESTORE^{P}	247	STX
		233	RETRY^{P}	248	STXA$^{P_{ASI}}$
201	MEMBAR	235	RETURN	269	STXFSR
204	MOVcc	239	SAVED^{P}	270	SUB (SUBcc)
		237	SAVE^{P}	270	SUBC (SUBCcc)
		240	SDIV^{D} (SDIVcc^{D})	272	SWAPA$^{D,\ P_{ASI}}$
207	MOVr	211	SDIVX	271	SWAP^{D}
209	MULScc^{D}	242	SETHI	274	TADDcc
211	MULX			275	TADDccTV^{D}
212	NOP	243	SIAM	276	Tcc
213	NORMALW			279	TSUBcc
214	OR (ORcc)	244	SLL	280	TSUBccTV^{D}
214	ORN (ORNcc)	244	SLLX	281	UDIV^{D} (UDIVcc^{D})
215	OTHERW	246	SMUL^{D} (SMULcc^{D})	211	UDIVX
		244	SRA	283	UMUL^{D} (UMULcc^{D})
216	PDIST	244	SRAX		
		244	SRL	285	WRASI
217	POPC	244	SRLX	285	WRasr$^{P_{ASR}}$
219	PREFETCH	247	STB	285	WRCCR
219	PREFETCHA$^{P_{ASI}}$	248	STBA$^{P_{ASI}}$	285	WRFPRS
				285	WRGSR
225	RDASI	250	STBLOCKF		
225	RDasr$^{P_{ASR}}$	253	STDF		
225	RDCCR	255	STDFA$^{P_{ASI}}$		
225	RDFPRS	253	STF	288	WRPR^{P}
225	RDGSR	255	STFA$^{P_{ASI}}$		

TABLE 7-2 *UltraSPARC Architecture 2007Instruction Set - Alphabetical (3 of 3)*

Page	Instruction	Page	Instruction	Page	Instruction
285	WRSOFTINT_CLR^{P}	285	WRSTICK^{P}	290	XNOR (XNORcc)
285	WRSOFTINT_SET^{P}	285	WRTICK_CMPR^{P}	290	XOR (XORcc)
285	WRSOFTINT^{P}	285	WRY^{D}		
285	WRSTICK_CMPR^{P}				

TABLE 7-3 Instruction Set - by Functional Category *(1 of 5)*

Instruction	Category and Function	Page	Ext. to V9?
Data Movement Operations, Between R Registers			
MOVcc	Move integer register if condition is satisfied	204	
MOVr	Move integer register on contents of integer register	207	
Data Movement Operations, Between F Registers			
FMOV<s|d|q>	Floating-point move	139	
FMOV<s|d|q>cc	Move floating-point register if condition is satisfied	140	
FMOV<s|d|q>R	Move f-p reg. if integer reg. contents satisfy condition	144	
FSRC<1|2>[s]	Copy source	164	**VIS 1**
Data Conversion Instructions			
FiTO<s|d|q>	Convert 32-bit integer to floating-point	132	
F<s|d|q>TOi	Convert floating point to integer	167	
F<s|d|q>TOx	Convert floating point to 64-bit integer	167	
F<s|d|q>TO<s|d|q>	Convert between floating-point formats	168	
FxTO<s|d|q>	Convert 64-bit integer to floating-point	171	
Logical Operations on R Registers			
AND (ANDcc)	Logical **and** (and modify condition codes)	100	
OR (ORcc)	Inclusive-**or** (and modify condition codes)	214	
ORN (ORNcc)	Inclusive-**or not** (and modify condition codes)	214	
XNOR (XNORcc)	Exclusive-**nor** (and modify condition codes)	290	
XOR (XORcc)	Exclusive-**or** (and modify condition codes)	290	
Logical Operations on F Registers			
FAND[s]	Logical **and** operation	165	**VIS 1**
FANDNOT<1|2>[s]	Logical **and** operation with one inverted source	165	**VIS 1**
FNAND[s]	Logical **nand** operation	165	**VIS 1**
FNOR[s]	Logical **nor** operation	165	**VIS 1**
FNOT<1|2>[s]	Copy negated source	164	**VIS 1**
FONE[s]	One fill	163	**VIS 1**
FOR[s]	Logical **or** operation	165	**VIS 1**
FORNOT<1|2>[s]	Logical **or** operation with one inverted source	165	**VIS 1**
FXNOR[s]	Logical **xnor** operation	165	**VIS 1**
FXOR[s]	Logical **xor** operation	165	**VIS 1**
FZERO[s]	Zero fill	163	**VIS 1**
Shift Operations on R Registers			
SLL	Shift left logical	244	
SLLX	Shift left logical, extended	244	
SRA	Shift right arithmetic	244	
SRAX	Shift right arithmetic, extended	244	
SRL	Shift right logical	244	
SRLX	Shift right logical, extended	244	
Special Addressing Operations			
ALIGNADDRESS[_LITTLE]	Calculate address for misaligned data	98	**VIS 1**
ARRAY<8|16|32>	3-D array addressing instructions	101	**VIS 1**
FALIGNDATA	Perform data alignment for misaligned data	121	**VIS 1**
Control Transfers			
Bicc	Branch on integer condition codes	104	
BPcc	Branch on integer condition codes with prediction	107	

TABLE 7-3 Instruction Set - by Functional Category *(2 of 5)*

Instruction	Category and Function	Page	Ext. to V9?
BPr	Branch on contents of integer register with prediction	109	
CALL	Call and link	111	
DONE^{P}	Return from trap	114	
FBfcc^{D}	Branch on floating-point condition codes	122	
FBPfcc	Branch on floating-point condition codes with prediction	124	
ILLTRAP	Illegal instruction	172	
JMPL	Jump and link	174	
RETRY^{P}	Return from trap and retry	233	
RETURN	Return	235	
Tcc	Trap on integer condition codes	276	
Byte Permutation			
BMASK	Set the GSR.mask field	106	**VIS 2**
BSHUFFLE	Permute bytes as specified by GSR.mask	106	**VIS 2**
Data Formatting Operations on F Registers			
FEXPAND	Pixel expansion	131	**VIS 1**
FPACK<16 | 32 | FIX>	Pixel packing	153	**VIS 1**
FPMERGE	Pixel merge	160	**VIS 1**
Memory Operations to/from F Registers			
LDBLOCKF	Block loads	178	**VIS 1**
STBLOCKF	Block stores	250	**VIS 1**
LDDF	Load double floating-point	181	
LDDFA$^{P_{ASI}}$	Load double floating-point from alternate space	183	
LDF	Load floating-point	181	
LDFA$^{P_{ASI}}$	Load floating-point from alternate space	183	
LDQF	Load quad floating-point	181	
LDQFA$^{P_{ASI}}$	Load quad floating-point from alternate space	183	
LDSHORTF	Short floating-point loads	188	**VIS 1**
STDF	Store double floating-point	253	
STDFA$^{P_{ASI}}$	Store double floating-point into alternate space	255	
STF	Store floating-point	253	
STFA$^{P_{ASI}}$	Store floating-point into alternate space	255	
STPARTIALF	Partial Store instructions	260	**VIS 1**
STQF	Store quad floating point	253	
STQFA$^{P_{ASI}}$	Store quad floating-point into alternate space	255	
STSHORTF	Short floating-point stores	263	**VIS 1**
Memory Operations — Miscellaneous			
LDFSR^{D}	Load floating-point state register (lower)	186	
LDXFSR	Load floating-point state register	199	
MEMBAR	Memory barrier	201	
PREFETCH	Prefetch data	219	
PREFETCHA$^{P_{ASI}}$	Prefetch data from alternate space	219	
STFSR^{D}	Store floating-point state register (lower)	258	
STXFSR	Store floating-point state register	269	
Atomic (Load-Store) Memory Operations to/from R Registers			
CASA$^{P_{ASI}}$	Compare and swap word in alternate space	112	
CASXA$^{P_{ASI}}$	Compare and swap doubleword in alternate space	112	

TABLE 7-3 Instruction Set - by Functional Category *(3 of 5)*

Instruction	Category and Function	Page	Ext. to V9?
LDSTUB	Load-store unsigned byte	190	
LDSTUBA$^{\mathrm{P_{ASI}}}$	Load-store unsigned byte in alternate space	191	
SWAP$^{\mathrm{D}}$	Swap integer register with memory	271	
SWAPA$^{\mathrm{D,\ P_{ASI}}}$	Swap integer register with memory in alternate space	272	
	Memory Operations to/from R Registers		
LDSB	Load signed byte	175	
LDSBA$^{\mathrm{P_{ASI}}}$	Load signed byte from alternate space	176	
LDSH	Load signed halfword	175	
LDSHA$^{\mathrm{P_{ASI}}}$	Load signed halfword from alternate space	176	
LDSW	Load signed word	175	
LDSWA$^{\mathrm{P_{ASI}}}$	Load signed word from alternate space	176	
LDTXA$^{\mathrm{N}}$	Load integer twin extended word from alternate space	197	VIS 2+
LDTW$^{\mathrm{D,\ P_{ASI}}}$	Load integer twin word	192	
LDTWA$^{\mathrm{D,\ P_{ASI}}}$	Load integer twin word from alternate space	194	
LDUB	Load unsigned byte	190	
LDUBA$^{\mathrm{P_{ASI}}}$	Load unsigned byte from alternate space	176	
LDUH	Load unsigned halfword	175	
LDUHA$^{\mathrm{P_{ASI}}}$	Load unsigned halfword from alternate space	176	
LDUW	Load unsigned word	175	
LDUWA$^{\mathrm{P_{ASI}}}$	Load unsigned word from alternate space	176	
LDX	Load extended	175	
LDXA$^{\mathrm{P_{ASI}}}$	Load extended from alternate space	176	
STB	Store byte	247	
STBA$^{\mathrm{P_{ASI}}}$	Store byte into alternate space	248	
STTW$^{\mathrm{D}}$	Store twin word	265	
STTWA$^{\mathrm{D,\ P_{ASI}}}$	Store twin word into alternate space	267	
STH	Store halfword	247	
STHA$^{\mathrm{P_{ASI}}}$	Store halfword into alternate space	248	
STW	Store word	247	
STWA$^{\mathrm{P_{ASI}}}$	Store word into alternate space	248	
STX	Store extended	247	
STXA$^{\mathrm{P_{ASI}}}$	Store extended into alternate space	248	
	Floating-Point Arithmetic Operations		
FABS<s|d|q>	Floating-point absolute value	119	
FADD<s|d|q>	Floating-point add	120	
FDIV<s|d|q>	Floating-point divide	130	
FdMULq	Floating-point multiply double to quad	151	
FMADD(s,d)	Floating-point multiply-add single/double (fused)	137	
FMSUB(s,d)	Floating-point multiply-subtract single/double (fused)	137	
FMUL<s|d|q>	Floating-point multiply	151	
FNMADD(s,d)	Floating-point negative multiply-add single/double (fused)	137	
FNEG<s|d|q>	Floating-point negate	152	
FNMSUB(s,d)	Floating-point negative multiply-subtract single/double (fused)	137	
FsMULd	Floating-point multiply single to double	151	
FSQRT<s|d|q>	Floating-point square root	166	
FSUB<s|d|q>	Floating-point subtract	170	

TABLE 7-3 Instruction Set - by Functional Category *(4 of 5)*

Instruction	Category and Function	Page	Ext. to V9?
	Floating-Point Comparison Operations		
FCMP*<16,32>	Compare four 16-bit signed values or two 32-bit signed values	126	**VIS 1**
FCMP<s|d|q>	Floating-point compare	128	
FCMPE<s|d|q>	Floating-point compare (exception if unordered)	128	
	Register-Window Control Operations		
ALLCLEAN	Mark all register window sets as “clean”	99	
INVALW	Mark all register window sets as “invalid”	173	
FLUSHW	Flush register windows	136	
NORMALW	“Other” register windows become “normal” register windows	213	
OTHERW	“Normal” register windows become “other” register windows	215	
RESTORE	Restore caller’s window	230	
RESTORED[P]	Window has been restored	232	
SAVE	Save caller’s window	237	
SAVED[P]	Window has been saved	239	
	Miscellaneous Operations		
FLUSH	Flush instruction memory	133	
NOP	No operation	212	
	Integer SIMD Operations on F Registers		
FPADD<16,32>[S]	Fixed-point partitioned add	158	**VIS 1**
FPSUB<16,32>[S]	Fixed-point partitioned subtract	161	**VIS 1**
	Integer Arithmetic Operations on R Registers		
ADD (ADDcc)	Add (and modify condition codes)	97	
ADDC (ADDCcc)	Add with carry (and modify condition codes)	97	
MULScc[D]	Multiply step (and modify condition codes)	209	
MULX	Multiply 64-bit integers	211	
SDIV[D] (SDIVcc[D])	32-bit signed integer divide (and modify condition codes)	240	
SDIVX	64-bit signed integer divide	211	
SMUL[D] (SMULcc[D])	Signed integer multiply (and modify condition codes)	246	
SUB (SUBcc)	Subtract (and modify condition codes)	270	
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)	270	
TADDcc	Tagged add and modify condition codes (trap on overflow)	274	
TADDccTV[D]	Tagged add and modify condition codes (trap on overflow)	275	
TSUBcc	Tagged subtract and modify condition codes (trap on overflow)	279	
TSUBccTV[D]	Tagged subtract and modify condition codes (trap on overflow)	280	
UDIV[D] (UDIVcc[D])	Unsigned integer divide (and modify condition codes)	281	
UDIVX	64-bit unsigned integer divide	211	
UMUL[D] (UMULcc[D])	Unsigned integer multiply (and modify condition codes)	283	
	Integer Arithmetic Operations on F Registers		
FMUL8x16	8x16 partitioned product	146	**VIS 1**
FMUL8x16[AU|AL]	8x16 upper/lower α partitioned product	146	**VIS 1**
FMUL8[SU|UL]x16	8x16 upper/lower partitioned product	146	**VIS 1**
FMULD8[SU|UL]x16	8x16 upper/lower partitioned product	146	**VIS 1**
	Miscellaneous Operations on R Registers		
POPC	Population count	217	
SETHI	Set high 22 bits of low word of integer register	242	
	Miscellaneous Operations on F Registers		

TABLE 7-3 Instruction Set - by Functional Category *(5 of 5)*

Instruction	Category and Function	Page	Ext. to V9?
EDGE<8|16|32>[L]cc	Edge handling instructions (and modify condition codes)	116	VIS 1
EDGE<8|16|32>[L]N	Edge handling instructions	118	VIS 2
PDIST	Pixel component distance	216	VIS 1
	Control and Status Register Access		
RDASI	Read ASI register	225	
RDasr$^{P_{ASR}}$	Read ancillary state register	225	
RDCCR	Read Condition Codes register (CCR)	225	
RDFPRS	Read Floating-Point Registers State register (FPRS)	225	
RDGSR	Read General Status register (GSR)	225	
RDPC	Read Program Counter register (PC)	225	
RDPR^P	Read privileged register	228	
RDSOFTINT^P	Read per-virtual processor Soft Interrupt register (SOFTINT)	225	
RDSTICK$^{P_{npt}}$	Read System Tick register (STICK)	225	
RDSTICK_CMPR^P	Read System Tick Compare register (STICK_CMPR)	225	
RDTICK$^{P_{npt}}$	Read Tick register (TICK)	225	
RDTICK_CMPR^P	Read Tick Compare register (TICK_CMPR)	225	
SIAM	Set interval arithmetic mode	243	VIS 2
WRASI	Write ASI register	285	
WRasr$^{P_{ASR}}$	Write ancillary state register	285	
WRCCR	Write Condition Codes register (CCR)	285	
WRFPRS	Write Floating-Point Registers State register (FPRS)	285	
WRGSR	Write General Status register (GSR)	285	
WRPR^P	Write privileged register	288	
WRSOFTINT^P	Write per-virtual processor Soft Interrupt register (SOFTINT)	285	
WRSOFTINT_CLR^P	Clear bits of per-virtual processor Soft Interrupt register (SOFTINT)	285	
WRSOFTINT_SET^P	Set bits of per-virtual processor Soft Interrupt register (SOFTINT)	285	
WRTICK_CMPR^P	Write Tick Compare register (TICK_CMPR)	285	
WRSTICK^P	Write System Tick register (STICK)	285	
WRSTICK_CMPR^P	Write System Tick Compare register (STICK_CMPR)	285	
WRY^D	Write Y register	285	

In the remainder of this chapter, related instructions are grouped into subsections. Each subsection consists of the following sets of information:

(1) Instruction Table. This lists the instructions that are defined in the subsection, including the values of the field(s) that uniquely identify the instruction(s), assembly language syntax, and software and implementation classifications for the instructions. *(description of the Software Classes [letters] and Implementation Classes [digits] will be provided in a later update to this specification)*

> **Note** Instruction classes will be defined in a later draft of this document and in the meantime are subject to change.

(2) Illustration of Instruction Format(s). These illustrations show how the instruction is encoded in a 32-bit word in memory. In them, a dash (—) indicates that the field is *reserved* for future versions of the architecture and must be 0 in any instance of the instruction. If a conforming UltraSPARC Architecture implementation encounters nonzero values in these fields, its behavior is as defined in *Reserved Opcodes and Instruction Fields* on page 86.

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

(4) Exceptions. The exceptions that can occur as a consequence of attempting to execute the instruction(s). Exceptions due to an *IAE_**,, and interrupts are not listed because they can occur on any instruction. An instruction not implemented in hardware generates an *illegal_instruction* exception and therefore will not generate any of the other exceptions listed. Exceptions are listed in order of trap priority (see *Trap Priorities* on page 356), from highest to lowest priority.

(5) See Also. A list of related instructions (on selected pages).

> **Note** This specification does not contain any timing information (in either cycles or elapsed time), since timing is always implementation dependent.

7.1 Add

Instruction	op3	Operation	Assembly Language Syntax		Class
ADD	00 0000	Add	`add`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ADDcc	01 0000	Add and modify cc's	`addcc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ADDC	00 1000	Add with 32-bit Carry	`addc`	reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ADDCcc	01 1000	Add with 32-bit Carry and modify cc's	`addccc`	reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	

Description If i = 0, ADD and ADDcc compute "R[rs1] + R[rs2]". If i = 1, they compute "R[rs1] + **sign_ext**(simm13)". In either case, the sum is written to R[rd].

ADDC and ADDCcc ("ADD with carry") also add the CCR register's 32-bit carry (icc.c) bit. That is, if i = 0, they compute "R[rs1] + R[rs2] + icc.c" and if i = 1, they compute "R[rs1] + **sign_ext**(simm13) + icc.c". In either case, the sum is written to R[rd].

ADDcc and ADDCcc modify the integer condition codes (CCR.icc and CCR.xcc). Overflow occurs on addition if both operands have the same sign and the sign of the sum is different from that of the operands.

Programming Note ADDC and ADDCcc read the 32-bit condition codes' carry bit (CCR.icc.c), not the 64-bit condition codes' carry bit (CCR.xcc.c).

SPARC V8 Compatibility Note ADDC and ADDCcc were previously named ADDX and ADDXcc, respectively, in SPARC V8.

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

Exceptions *illegal_instruction*

7.2 Align Address VIS 1

Instruction	opf	Operation	Assembly Language Syntax	Class	Added
ALIGNADDRESS	0 0001 1000	Calculate address for misaligned data access	`alignaddr` reg_{rs1}, reg_{rs2}, reg_{rd}	**A1**	UA 2005
ALIGNADDRESS_ LITTLE	0 0001 1010	Calculate address for misaligned data access, little-endian	`alignaddrl` reg_{rs1}, reg_{rs2}, reg_{rd}	**A1**	UA 2005

31 30	29 25	24 19	18 14	13 5	4 0
10	rd	110110	rs1	opf	rs2

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

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

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

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

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

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

Exceptions *fp_disabled*

See Also Align Data on page 121

7.3 Mark All Register Window Sets "Clean"

Instruction	Operation	Assembly Language Syntax	Class	Added
ALLCLEAN[P]	Mark all register window sets as "clean"	`allclean`	**A1**	UA 2005

10	fcn = 0 0010	11 0001	—
31 30	29 25	24 19	18 0

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

CLEANWIN ← (*N_REG_WINDOWS* – 1)

Programming Note ALLCLEAN is used to indicate that all register windows are "clean"; that is, do not contain data belonging to other address spaces. It is needed because the value of *N_REG_WINDOWS* is not known to privileged software.

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 173
NORMALW on page 213
OTHERW on page 215
RESTORED on page 232
SAVED on page 239

7.4 AND Logical Operation

Instruction	op3	Operation	Assembly Language Syntax	Class
AND	00 0001	**and**	`and` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ANDcc	01 0001	**and** and modify cc's	`andcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ANDN	00 0101	**and not**	`andn` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ANDNcc	01 0101	**and not** and modify cc's	`andncc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	

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

ANDcc and ANDNcc modify the integer condition codes (icc and xcc). They set the condition codes as follows:

- icc.v, icc.c, xcc.v, and xcc.c are set to 0
- icc.n is copied from bit 31 of the result
- xcc.n is copied from bit 63 of the result
- icc.z is set to 1 if bits 31:0 of the result are zero (otherwise to 0)
- xcc.z is set to 1 if all 64 bits of the result are zero (otherwise to 0)

ANDN and ANDNcc logically negate their second operand before applying the main (**and**) operation.

An attempt to execute an AND, ANDcc, ANDN or ANDNcc instruction when i = 0 and reserved instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

Exceptions *illegal_instruction*

7.5 Three-Dimensional Array Addressing VIS 1

Instruction	opf	Operation	Assembly Language Syntax	Class	Added
ARRAY8	0 0001 0000	Convert 8-bit 3D address to blocked byte address	`array8` reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**	UA 2005
ARRAY16	0 0001 0010	Convert 16-bit 3D address to blocked byte address	`array16` reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**	UA 2005
ARRAY32	0 0001 0100	Convert 32-bit 3D address to blocked byte address	`array32` reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**	UA 2005

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description These instructions convert three-dimensional (3D) fixed-point addresses contained in R[rs1] to a blocked-byte address; they store the result in R[rd]. Fixed-point addresses typically are used for address interpolation for planar reformatting operations. Blocking is performed at the 64-byte level to maximize external cache block reuse, and at the 64-Kbyte level to maximize TLB entry reuse, regardless of the orientation of the address interpolation. These instructions specify an element size of 8 bits (ARRAY8), 16 bits (ARRAY16), or 32 bits (ARRAY32).

The second operand, R[rs2], specifies the power-of-2 size of the X and Y dimensions of a 3D image array. The legal values for R[rs2] and their meanings are shown in TABLE 7-4. Illegal values produce undefined results in the destination register, R[rd].

TABLE 7-4 3D R[rs2] Array X and Y Dimensions

R[rs2] Value (*n*)	Number of Elements
0	64
1	128
2	256
3	512
4	1024
5	2048

Implementation Note Architecturally, an illegal R[rs2] value (>5) causes the array instructions to produce undefined results. For historic reference, past implementations of these instructions have ignored R[rs2]{63:3} and have treated R[rs2] values of 6 and 7 as if they were 5.

The array instructions facilitate 3D texture mapping and volume rendering by computing a memory address for data lookup based on fixed-point x, y, and z coordinates. The data are laid out in a blocked fashion, so that points which are near one another have their data stored in nearby memory locations.

If the texture data were laid out in the obvious fashion (the z = 0 plane, followed by the z = 1 plane, etc.), then even small changes in z would result in references to distant pages in memory. The resulting lack of locality would tend to result in TLB misses and poor performance. The three versions of the array instruction, ARRAY8, ARRAY16, and ARRAY32, differ only in the scaling of the computed memory offsets. ARRAY16 shifts its result left by one position and ARRAY32 shifts left by two in order to handle 16- and 32-bit texture data.

When using the array instructions, a "blocked-byte" data formatting structure is imposed. The N × N × M volume, where $N = 2^n \times 64$, $M = m \times 32$, $0 \le n \le 5$, $1 \le m \le 16$ should be composed of 64 × 64 × 32 smaller volumes, which in turn should be composed of 4 × 4 × 2 volumes. This data structure is optimal for 16-bit data. For 16-bit data, the 4 × 4 × 2 volume has 64 bytes of data, which is ideal for reducing cache-line misses; the 64 × 64 × 32 volume will have 256 Kbytes of data, which is good for improving the TLB hit rate. FIGURE 7-1 illustrates how the data has to be organized, where the origin

(0,0,0) is assumed to be at the lower-left front corner and the x coordinate varies faster than y than z. That is, when traversing the volume from the origin to the upper right back, you go from left to right, front to back, bottom to top.

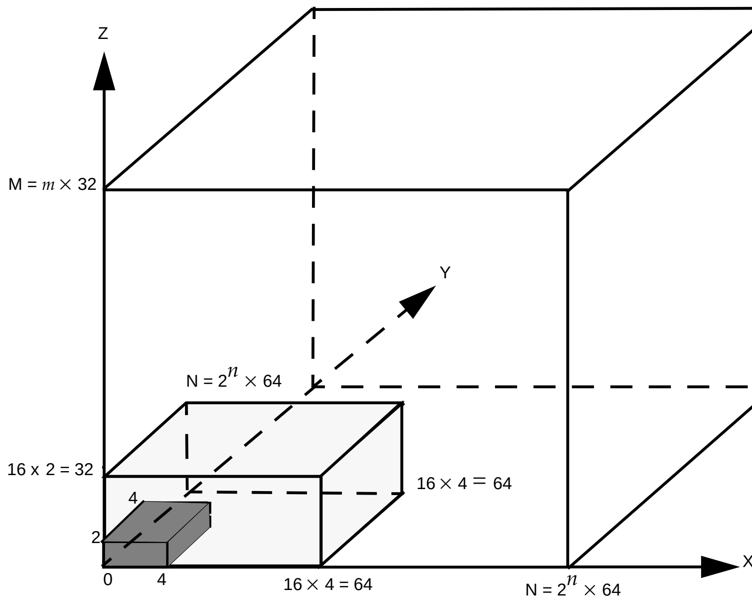


FIGURE 7-1 Blocked-Byte Data Formatting Structure

The array instructions have 2 inputs:

The (x,y,z) coordinates are input via a single 64-bit integer organized in R[rs1] as shown in FIGURE 7-2.

Z integer	Z fraction	Y integer	Y fraction	X integer	X fraction
63 55	54 44	43 33	32 22	21 11	10 0

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 × 512 × 32 or a 512 × 512 × 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.

UPPER			MIDDLE			LOWER		
Z	Y	X	Z	Y	X	Z	Y	X

20 + $2n$ 17 +$2n$ 17 + n 17   13   9   5   4   2   0

FIGURE 7-3 Three-Dimensional Array Blocked-Address Format (ARRAY8)

UPPER			MIDDLE			LOWER			0
Z	Y	X	Z	Y	X	Z	Y	X	

21 +2n 18 +2n 18 +n 18 14 10 6 5 3 1 0

FIGURE 7-4 Three-Dimensional Array Blocked-Address Format (ARRAY16)

UPPER			MIDDLE			LOWER			00
Z	Y	X	Z	Y	X	Z	Y	X	

22 +2n 19 +2n 19 +n 19 15 11 7 6 5 4 3 2 1 0

FIGURE 7-5 Three Dimensional Array Blocked-Address Format (ARRAY32)

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

TABLE 7-5 ARRAY8 Description

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

In the above description, if $n = 0$, there are 64 elements, so X_integer{6} and Y_integer{6} are not defined. That is, result{20:17} equals Z_integer{8:5}.

Note To maximize reuse of external cache and TLB data, software should block array references of a large image to the 64-Kbyte level. This means processing elements within a 32 × 32 × 64 block.

The code fragment below shows assembly of components along an interpolated line at the rate of one component per clock.

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

Exceptions None

7.6 Branch on Integer Condition Codes (Bicc)

Opcode	cond	Operation	*icc* Test	Assembly Language Syntax		Class
BA	1000	Branch Always	1	`ba{,a}`	*label*	**A1**
BN	0000	Branch Never	0	`bn{,a}`	*label*	**A1**
BNE	1001	Branch on Not Equal	**not** Z	`bne`†`{,a}`	*label*	**A1**
BE	0001	Branch on Equal	Z	`be`‡`{,a}`	*label*	**A1**
BG	1010	Branch on Greater	**not** (Z **or** (N **xor** V))	`bg{,a}`	*label*	**A1**
BLE	0010	Branch on Less or Equal	Z **or** (N **xor** V)	`ble{,a}`	*label*	**A1**
BGE	1011	Branch on Greater or Equal	**not** (N **xor** V)	`bge{,a}`	*label*	**A1**
BL	0011	Branch on Less	N **xor** V	`bl{,a}`	*label*	**A1**
BGU	1100	Branch on Greater Unsigned	**not** (C **or** Z)	`bgu{,a}`	*label*	**A1**
BLEU	0100	Branch on Less or Equal Unsigned	C **or** Z	`bleu{,a}`	*label*	**A1**
BCC	1101	Branch on Carry Clear (Greater Than or Equal, Unsigned)	**not** C	`bcc`◊`{,a}`	*label*	**A1**
BCS	0101	Branch on Carry Set (Less Than, Unsigned)	C	`bcs`∇`{,a}`	*label*	**A1**
BPOS	1110	Branch on Positive	**not** N	`bpos{,a}`	*label*	**A1**
BNEG	0110	Branch on Negative	N	`bneg{,a}`	*label*	**A1**
BVC	1111	Branch on Overflow Clear	**not** V	`bvc{,a}`	*label*	**A1**
BVS	0111	Branch on Overflow Set	V	`bvs{,a}`	*label*	**A1**

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

00	a	cond	010	disp22
31 30	29	28 25	24 22	21 0

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

Unconditional branches and icc-conditional branches are described below:

- **Unconditional branches (**BA, BN) — If its annul bit is 0 (a = 0), a BN (Branch Never) instruction is treated as a NOP. If its annul bit is 1 (a = 1), the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

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

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

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

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

Annulment, delay instructions, and delayed control transfers are described further in Chapter 6, *Instruction Set Overview*.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the Bicc instruction will cause a transfer of control (BA or taken conditional branch), then Bicc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the Bicc instruction) is stored in TPC[TL] and the value of NPC from before the Bicc was executed is stored in TNPC[TL].

Note that BN never causes a *control_transfer_instruction* exception.

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

7.7 Byte Mask and Shuffle VIS 2

Instruction	opf	Operation	Assembly Language Syntax	Class	Added
BMASK	0 0001 1001	Set the GSR.mask field in preparation for a subsequent BSHUFFLE instruction	`bmask` reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**	UA 2007
BSHUFFLE	0 0100 1100	Permute 16 bytes as specified by GSR.mask	`bshuffle` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**	UA 2007

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description BMASK adds two integer registers, R[rs1] and R[rs2], and stores the result in the integer register R[rd]. The least significant 32 bits of the result are stored in the GSR.mask field.

BSHUFFLE concatenates the two 64-bit floating-point registers F_D[rs1] (more significant half) and F_D[rs2] (less significant half) to form a 128-bit (16-byte) value. Bytes in the concatenated value are numbered from most significant to least significant, with the most significant byte being byte 0. BSHUFFLE extracts 8 of those 16 bytes and stores the result in the 64-bit floating-point register F_D[rd]. Bytes in F_D[rd] are also numbered from most to least significant, with the most significant being byte 0. The following table indicates which source byte is extracted from the concatenated value to generate each byte in the destination register, F_D[rd].

Destination Byte (in F[rd])	Source Byte
0 (most significant)	(F_D[rs1] :: F_D[[rs2]) {GSR.mask{31:28}}
1	(F_D[[rs1] :: F_D[[rs2]) {GSR.mask{27:24}}
2	(F_D[[rs1] :: F_D[[rs2]) {GSR.mask{23:20}}
3	(F_D[[rs1] :: F_D[[rs2]) {GSR.mask{19:16}}
4	(F_D[[rs1] :: F_D[[rs2]) {GSR.mask{15:12}}
5	(F_D[[rs1] :: F_D[[rs2]) {GSR.mask{11:8}}
6	(F_D[[rs1] :: F_D[[rs2]) {GSR.mask{7:4}}
7 (least significant)	(F_D[[rs1] :: F_D[[rs2]) {GSR.mask{3:0}}

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

Exceptions *fp_disabled*

7.8 Branch on Integer Condition Codes with Prediction (BPcc)

Instruction	cond	Operation	cc Test	Assembly Language Syntax	Class
BPA	1000	Branch Always	1	ba{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPN	0000	Branch Never	0	bn{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPNE	1001	Branch on Not Equal	**not** Z	bne†{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPE	0001	Branch on Equal	Z	be‡{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPG	1010	Branch on Greater	**not** (Z **or** (N **xor** V))	bg{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPLE	0010	Branch on Less or Equal	Z **or** (N **xor** V)	ble{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPGE	1011	Branch on Greater or Equal	**not** (N **xor** V)	bge{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPL	0011	Branch on Less	N **xor** V	bl{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPGU	1100	Branch on Greater Unsigned	**not** (C **or** Z)	bgu{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPLEU	0100	Branch on Less or Equal Unsigned	C **or** Z	bleu{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	**not** C	bcc◊{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPCS	0101	Branch on Carry Set (Less than, Unsigned)	C	bcs∇{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPPOS	1110	Branch on Positive	**not** N	bpos{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPNEG	0110	Branch on Negative	N	bneg{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPVC	1111	Branch on Overflow Clear	**not** V	bvc{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**
BPVS	0111	Branch on Overflow Set	V	bvs{,a}{,pt|,pn} *i_or_x_cc, label*	**A1**

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

00	a	cond	001	cc1	cc0	p	disp19
31 30	29	28 25	24 22	21	20	19	18

cc1	cc0	Condition Code
0	0	icc
0	1	—
1	0	xcc
1	1	—

Programming Note To set the annul (a) bit for BPcc instructions, append "`,a`" to the opcode mnemonic. For example, use `bgu,a %icc`, *label*. Braces in the preceding table signify that the "`,a`" is optional. To set the branch prediction bit, append to an opcode mnemonic either "`,pt`" for predict taken or "`,pn`" for predict not taken. If neither "`,pt`" nor "`,pn`" is specified, the assembler defaults to ",`pt`". To select the appropriate integer condition code, include "`%icc`" or "`%xcc`" before the label.

Description Unconditional branches and conditional branches are described below.

- **Unconditional branches (BPA, BPN)** — A BPN (Branch Never with Prediction) instruction for this branch type (op2 = 1) may be used in the SPARC V9 architecture as an instruction prefetch; that is, the effective address (PC + (4 × **sign_ext**(disp19))) specifies an address of an instruction that is expected to be executed soon. If the Branch Never's annul bit is 1 (a = 1), then the following (delay) instruction is annulled (not executed). If the annul bit is 0 (a = 0), then the following instruction is executed. In no case does a Branch Never cause a transfer of control to take place.

  BPA (Branch Always with Prediction) causes an unconditional PC-relative, delayed control transfer to the address "PC + (4 × **sign_ext**(disp19))". If the annul bit of the branch instruction is 1 (a = 1), then the delay instruction is annulled (not executed). If the annul bit is 0 (a = 0), then the delay instruction is executed.

- **Conditional branches** — Conditional BPcc instructions (except BPA and BPN) evaluate one of the two integer condition codes (icc or xcc), as selected by cc0 and cc1, according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken; that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 × **sign_ext**(disp19))". If FALSE, the branch is not taken.

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

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

  The predict bit (p) is used to give the hardware a hint about whether the branch is expected to be taken. A 1 in the p bit indicates that the branch is expected to be taken; a 0 indicates that the branch is expected not to be taken.

  Annulment, delay instructions, prediction, and delayed control transfers are described further in Chapter 6, *Instruction Set Overview*.

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

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the BPcc instruction will cause a transfer of control (BPA or taken conditional branch), then BPcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the BPcc) is stored in TPC[TL] and the value of NPC from before the BPcc was executed is stored in TNPC[TL].

Note that BPN never causes a *control_transfer_instruction* exception.

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

See Also Branch on Integer Register with Prediction (BPr) on page 109

7.9 Branch on Integer Register with Prediction (BPr)

Instruction	rcond	Operation	Register Contents Test	Assembly Language Syntax	Class
—	000	*Reserved*	—		—
BRZ	001	Branch on Register Zero	R[rs1] = 0	`brz` {,a}{,pt|,pn} *reg*$_{rs1}$, *label*	**A1**
BRLEZ	010	Branch on Register Less Than or Equal to Zero	R[rs1] ≤ 0	`brlez` {,a}{,pt|,pn} *reg*$_{rs1}$, *label*	**A1**
BRLZ	011	Branch on Register Less Than Zero	R[rs1] < 0	`brlz` {,a}{,pt|,pn} *reg*$_{rs1}$, *label*	**A1**
—	100	*Reserved*	—		—
BRNZ	101	Branch on Register Not Zero	R[rs1] ≠ 0	`brnz` {,a}{,pt|,pn} *reg*$_{rs1}$, *label*	**A1**
BRGZ	110	Branch on Register Greater Than Zero	R[rs1] > 0	`brgz` {,a}{,pt|,pn} *reg*$_{rs1}$, *label*	**A1**
BRGEZ	111	Branch on Register Greater Than or Equal to Zero	R[rs1] ≥ 0	`brgez` {,a}{,pt|,pn} *reg*$_{rs1}$, *label*	**A1**

00	a	0*	rcond	011	d16hi	p	rs1	d16lo
31 30	29	28	27 25	24 22	21 20	19	18 14	13 0

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

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

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

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

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

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

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

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the BPr instruction will cause a transfer of control (taken conditional branch), then BPr generates a *control_transfer_instruction* exception instead of causing a control transfer.

Annulment, delay instructions, prediction, and delayed control transfers are described further in Chapter 6, *Instruction Set Overview*.

Implementation Note If this instruction is implemented by tagging each register value with an N (negative) bit and Z (zero) bit, the table below can be used to determine if rcond is TRUE:

Branch	Test
BRNZ	**not** Z
BRZ	Z
BRGEZ	**not** N
BRLZ	N
BRLEZ	N **or** Z
BRGZ	**not** (N **or** Z)

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

See Also Branch on Integer Condition Codes with Prediction (BPcc) on page 107

7.10 Call and Link

Instruction	op	Operation	Assembly Language Syntax	Class
CALL	01	Call and Link	`call` *label*	**A1**

01	disp30
31 30	29 … 0

Description The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address PC + (4 × **sign_ext**(disp30)). Since the word displacement (disp30) field is 30 bits wide, the target address lies within a range of -2^{31} to $+2^{31} - 4$ bytes. The PC-relative displacement is formed by sign-extending the 30-bit word displacement field to 62 bits and appending two low-order zeroes to obtain a 64-bit byte displacement.

The CALL instruction also writes the value of PC, which contains the address of the CALL, into R[15] (*out* register 7).

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system and in the address written into R[15]. (closed impl. dep. #125-V9-Cs10)

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

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

See Also JMPL on page 174

7.11 Compare and Swap

Instruction	op3	Operation	Assembly Language Syntax	Class
CASA[PASI]	11 1100	Compare and Swap Word from Alternate Space	`casa` [reg_{rs1}] *imm_asi*, reg_{rs2}, reg_{rd} `casa` [reg_{rs1}] `%asi`, reg_{rs2}, reg_{rd}	**A1**
CASXA[PASI]	11 1110	Compare and Swap Extended from Alternate Space	`casxa` [reg_{rs1}] *imm_asi*, reg_{rs2}, reg_{rd} `casxa` [reg_{rs1}] `%asi`, reg_{rs2}, reg_{rd}	**A1**

11	rd	op3	rs1	i=0	imm_asi	rs2

11	rd	op3	rs1	i=1	—	rs2
31 30	29 25	24 19	18 14	13	12 5	4 0

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.

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

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

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

Compatibility Note | An implementation might cause an exception because of an error during the store memory access, even though there was no error during the load memory access.

Programming Note | Compare and Swap (CAS) and Compare and Swap Extended (CASX) synthetic instructions are available for "big endian" memory accesses. Compare and Swap Little (CASL) and Compare and Swap Extended Little (CASXL) synthetic instructions are available for "little endian" memory accesses. See *Synthetic Instructions* on page 536 for the syntax of these synthetic instructions.

The compare-and-swap instructions do not affect the condition codes.

The compare-and-swap instructions can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

ASIs valid for CASA and CASXA instructions

ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE

Exceptions

illegal_instruction
mem_address_not_aligned
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nc_page (attempted access to noncacheable page)
DAE_nfo_page (attempted access to non-faulting-only page)

7.12 DONE

Instruction	op3	Operation	Assembly Language Syntax	Class
DONE[P]	11 1110	Return from Trap (skip trapped instruction)	done	**A1**

10	fcn =0 0000	11 1110	—
31 30	29 25	24 19	18 0

Description The DONE instruction restores the saved state from TSTATE[TL] (GL, CCR, ASI, PSTATE, and CWP), sets PC and NPC, and decrements TL. DONE sets PC←TNPC[TL] and NPC←TNPC[TL]+4 (normally, the value of NPC saved at the time of the original trap and address of the instruction immediately after the one referenced by the NPC).

Programming Notes The DONE and RETRY instructions are used to return from privileged trap handlers.

Unlike RETRY, DONE ignores the contents of TPC[TL].

If the saved TNPC[TL] was not altered by trap handler software, DONE causes execution to resume immediately *after* the instruction that originally caused the trap (as if that instruction was "done" executing).

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

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

Note that since PSTATE.tct is automatically set to 0 during entry to a trap handler, execution of a DONE instruction at the end of a trap handler will not cause a *control_transfer_instruction* exception unless trap handler software has explicitly set PSTATE.tct to 1. During execution of the DONE instruction, the value of PSTATE.tct is restored from TSTATE.

Programming Notes If *control_transfer_instruction* traps are to be re-enabled (PSTATE.tct ← 1, restored from TSTATE[TL].pstate.tct) when trap handler software for the *control_transfer_instruction* trap returns, the trap handler must
(1) emulate the trapped CTI, setting TPC[TL] and TNPC[TL] appropriately, remembering to compensate for annul bits) and
(2) use a DONE (not RETRY) instruction to return.

If the CTI that caused the *control_transfer_instruction* trap was a DONE (RETRY) instruction, the trap handler must carefully emulate the trapped DONE (RETRY) (decrementing TL may suffice) before the trap handler returns using its own DONE (RETRY) instruction.

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system.

IMPL. DEP. #417-S10: If *(1)* TSTATE[TL].pstate.am = 1 and *(2)* a DONE instruction is executed (which sets PSTATE.am to '1' by restoring the value from TSTATE[TL].pstate.am to PSTATE.am), it is implementation dependent whether the DONE instruction masks (zeroes) the more-significant 32 bits of the values it places into PC and NPC.

Exceptions. In privileged mode (PSTATE.priv = 1), an attempt to execute DONE while TL = 0 causes an *illegal_instruction* exception. An attempt to execute DONE (in any mode) with instruction bits 18:0 nonzero causes an *illegal_instruction* exception.

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

Implementation Note | In nonprivileged mode, *illegal_instruction* exception due to TL = 0 does not occur. The *privileged_opcode* exception occurs instead, regardless of the current trap level (TL).

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

Exceptions

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

See Also

RETRY on page 233

7.13 Edge Handling Instructions VIS 1

Instruction	opf	Operation	Assembly Language Syntax †		Class
EDGE8cc	0 0000 0000	Eight 8-bit edge boundary processing	`edge8cc`	*reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$	**B1**
EDGE8Lcc	0 0000 0010	Eight 8-bit edge boundary processing, little-endian	`edge8lcc`	*reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$	**B1**
EDGE16cc	0 0000 0100	Four 16-bit edge boundary processing	`edge16cc`	*reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$	**B1**
EDGE16Lcc	0 0000 0110	Four 16-bit edge boundary processing, little-endian	`edge16lcc`	*reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$	**B1**
EDGE32cc	0 0000 1000	Two 32-bit edge boundary processing	`edge32cc`	*reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$	**B1**
EDGE32Lcc	0 0000 1010	Two 32-bit edge boundary processing, little-endian	`edge32lcc`	reg$_{rs1}$, reg$_{rs2}$, reg$_{rd}$	**B1**

† The original assembly language mnemonics for these instructions did not include the "`cc`" suffix, as appears in the names of all other instructions that set the integer condition codes. The old, non-"`cc`" mnemonics are deprecated. Over time, assemblers will support the new mnemonics for these instructions. In the meantime, some older assemblers may recognize only the mnemonics, without "`cc`".

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description These instructions handle the boundary conditions for parallel pixel scan line loops, where R[rs1] is the address of the next pixel to render and R[rs2] is the address of the last pixel in the scan line.

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

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

1. The left edge mask is computed from the 3 least significant bits of R[rs1] and the right edge mask is computed from the 3 least significant bits of R[rs2], according to TABLE 7-6.
2. If 32-bit address masking is disabled (PSTATE.am = 0) so 64-bit addressing is in use, and the most significant 61 bits of R[rs1] are equal to the corresponding bits in R[rs2], R[rd] is set to the right edge mask **and**ed with the left edge mask.
3. If 32-bit address masking is enabled (PSTATE.am = 1) so 32-bit addressing is in use, and bits 31:3 of R[rs1] match bits 31:3 of R[rs2], R[rd] is set to the right edge mask **and**ed with the left edge mask.
4. Otherwise, R[rd] is set to the left edge mask.

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

TABLE 7-6 lists edge mask specifications.

TABLE 7-6 Edge Mask Specification

Edge Size	R[rs*n*] {2:0}	Big Endian		Little Endian	
		Left Edge	Right Edge	Left Edge	Right Edge
8	000	1111 1111	1000 0000	1111 1111	0000 0001
8	001	0111 1111	1100 0000	1111 1110	0000 0011
8	010	0011 1111	1110 0000	1111 1100	0000 0111
8	011	0001 1111	1111 0000	1111 1000	0000 1111

TABLE 7-6 Edge Mask Specification *(Continued)*

Edge Size	R[rs*n*] {2:0}	Big Endian		Little Endian	
		Left Edge	Right Edge	Left Edge	Right Edge
8	100	0000 1111	1111 1000	1111 0000	0001 1111
8	101	0000 0111	1111 1100	1110 0000	0011 1111
8	110	0000 0011	1111 1110	1100 0000	0111 1111
8	111	0000 0001	1111 1111	1000 0000	1111 1111
16	00x	1111	1000	1111	0001
16	01x	0111	1100	1110	0011
16	10x	0011	1110	1100	0111
16	11x	0001	1111	1000	1111
32	0xx	11	10	11	01
32	1xx	01	11	10	11

Exceptions None

See Also EDGE<8|16|32>[L]N on page 118

7.14 Edge Handling Instructions (no CC) VIS 2

Instruction	opf	Operation	Assembly Language Syntax		Class
EDGE8N	0 0000 0001	Eight 8-bit edge boundary processing, no CC	`edge8n`	reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**
EDGE8LN	0 0000 0011	Eight 8-bit edge boundary processing, little-endian, no CC	`edge8ln`	reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**
EDGE16N	0 0000 0101	Four 16-bit edge boundary processing, no CC	`edge16n`	reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**
EDGE16LN	0 0000 0111	Four 16-bit edge boundary processing, little-endian, no CC	`edge16ln`	reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**
EDGE32N	0 0000 1001	Two 32-bit edge boundary processing, no CC	`edge32n`	reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**
EDGE32LN	0 0000 1011	Two 32-bit edge boundary processing, little-endian, no CC	`edge32ln`	reg_{rs1}, reg_{rs2}, reg_{rd}	**B1**

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description EDGE8[L]N, EDGE16[L]N, and EDGE32[L]N operate identically to EDGE8[L]cc, EDGE16[L]cc, and EDGE32[L]cc, respectively, but do not set the integer condition codes.

See *Edge Handling Instructions* on page 116 for details.

Exceptions None

See Also EDGE<8,16,32>[L]cc on page 116

7.15 Floating-Point Absolute Value

Instruction	op3	opf	Operation	Assembly Language Syntax	Class
FABSs	11 0100	0 0000 1001	Absolute Value Single	`fabss` $freg_{rs2}$, $freg_{rd}$	**A1**
FABSd	11 0100	0 0000 1010	Absolute Value Double	`fabsd` $freg_{rs2}$, $freg_{rd}$	**A1**
FABSq	11 0100	0 0000 1011	Absolute Value Quad	`fabsq` $freg_{rs2}$, $freg_{rd}$	**C3**

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description FABS copies the source floating-point register(s) to the destination floating-point register(s), with the sign bit cleared (set to 0).

FABSs operates on single-precision (32-bit) floating-point registers, FABSd operates on double-precision (64-bit) floating-point register pairs, and FABSq operates on quad-precision (128-bit) floating-point register quadruples.

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

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FABSq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

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

An attempt to execute an FABSq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FABSq only))

7.16 Floating-Point Add

Instruction	op3	opf	Operation	Assembly Language Syntax	Class
FADDs	11 0100	0 0100 0001	Add Single	`fadds` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FADDd	11 0100	0 0100 0010	Add Double	`faddd` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FADDq	11 0100	0 0100 0011	Add Quad	`faddq` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**C3**

10	rd	op3	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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

Rounding is performed as specified by FSR.rd.

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FADDq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

An attempt to execute an FADDq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) or (rd{1:0} ≠ 0) causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

> **Note** An *fp_exception_other* with FSR.ftt = unfinished_FPop can occur if the operation detects unusual, implementation-specific conditions.

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

Exceptions
illegal_instruction
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FADDq only))
fp_exception_other (FSR.ftt = unfinished_FPop)
fp_exception_ieee_754 (OF, UF, NX, NV)

See Also FMAf on page 137

7.17 Align Data VIS 1

Instruction	opf	Operation	Assembly Language Syntax	Class
FALIGNDATA	0 0100 1000	Perform data alignment for misaligned data	`faligndata` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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

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

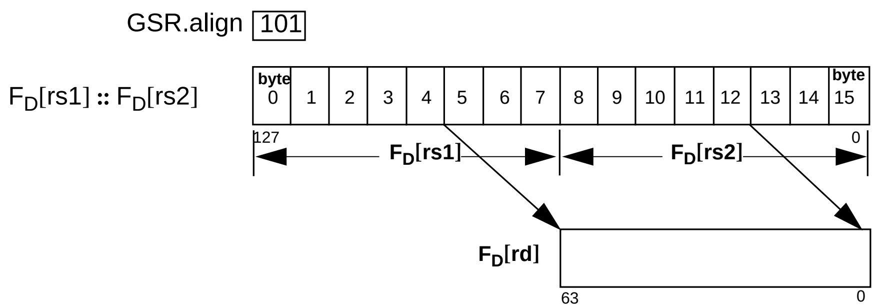


FIGURE 7-6 FALIGNDATA

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

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

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

Exceptions *fp_disabled*

See Also Align Address on page 98

7.18 Branch on Floating-Point Condition Codes (FBfcc)

Opcode	cond	Operation	fcc Test	Assembly Language Syntax		Class
FBA[D]	1000	Branch Always	1	`fba{,a}`	*label*	**A1**
FBN[D]	0000	Branch Never	0	`fbn{,a}`	*label*	**A1**
FBU[D]	0111	Branch on Unordered	U	`fbu{,a}`	*label*	**A1**
FBG[D]	0110	Branch on Greater	G	`fbg{,a}`	*label*	**A1**
FBUG[D]	0101	Branch on Unordered or Greater	G **or** U	`fbug{,a}`	*label*	**A1**
FBL[D]	0100	Branch on Less	L	`fbl{,a}`	*label*	**A1**
FBUL[D]	0011	Branch on Unordered or Less	L **or** U	`fbul{,a}`	*label*	**A1**
FBLG[D]	0010	Branch on Less or Greater	L **or** G	`fblg{,a}`	*label*	**A1**
FBNE[D]	0001	Branch on Not Equal	L **or** G **or** U	`fbne`†`{,a}`	*label*	**A1**
FBE[D]	1001	Branch on Equal	E	`fbe`‡`{,a}`	*label*	**A1**
FBUE[D]	1010	Branch on Unordered or Equal	E **or** U	`fbue{,a}`	*label*	**A1**
FBGE[D]	1011	Branch on Greater or Equal	E **or** G	`fbge{,a}`	*label*	**A1**
FBUGE[D]	1100	Branch on Unordered or Greater or Equal	E **or** G **or** U	`fbuge{,a}`	*label*	**A1**
FBLE[D]	1101	Branch on Less or Equal	E **or** L	`fble{,a}`	*label*	**A1**
FBULE[D]	1110	Branch on Unordered or Less or Equal	E **or** L **or** U	`fbule{,a}`	*label*	**A1**
FBO[D]	1111	Branch on Ordered	E **or** L **or** G	`fbo{,a}`	*label*	**A1**

† *synonym:* `fbnz` ‡ *synonym:* `fbz`

00	a	cond	110	disp22
31 30	29	28 25	24 22	21 0

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

Description Unconditional and Fcc branches are described below:

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

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

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

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

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

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

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

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the FBfcc instruction will cause a transfer of control (FBA or taken conditional branch), then FBfcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the FBfcc instruction) is stored in TPC[TL] and the value of NPC from before the FBfcc was executed is stored in TNPC[TL]. Note that FBN never causes a *control_transfer_instruction* exception.

Exceptions

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

7.19 Branch on Floating-Point Condition Codes with Prediction (FBPfcc)

Instruction	cond	Operation	fcc Test	Assembly Language Syntax		Class
FBPA	1000	Branch Always	1	`fba{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPN	0000	Branch Never	0	`fbn{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPU	0111	Branch on Unordered	U	`fbu{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPG	0110	Branch on Greater	G	`fbg{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPUG	0101	Branch on Unordered or Greater	G **or** U	`fbug{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPL	0100	Branch on Less	L	`fbl{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPUL	0011	Branch on Unordered or Less	L **or** U	`fbul{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPLG	0010	Branch on Less or Greater	L **or** G	`fblg{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPNE	0001	Branch on Not Equal	L **or** G **or** U	`fbne`† `{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPE	1001	Branch on Equal	E	`fbe`‡ `{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPUE	1010	Branch on Unordered or Equal	E **or** U	`fbue{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPGE	1011	Branch on Greater or Equal	E **or** G	`fbge{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPUGE	1100	Branch on Unordered or Greater or Equal	E **or** G **or** U	`fbuge{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPLE	1101	Branch on Less or Equal	E **or** L	`fble{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPULE	1110	Branch on Unordered or Less or Equal	E **or** L **or** U	`fbule{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**
FBPO	1111	Branch on Ordered	E **or** L **or** G	`fbo{,a}{,pt|,pn}`	`%fcc`*n*, *label*	**A1**

† *synonym:* `fbnz` ‡ *synonym:* `fbz`

00	a	cond	101	cc1	cc0	p	disp19
31 30	29	28 25	24 22	21	20	19	18 0

cc1	cc0	Condition Code
0	0	`fcc0`
0	1	`fcc1`
1	0	`fcc2`
1	1	`fcc3`

Programming Note To set the annul (`a`) bit for FBPfcc instructions, append "`,a`" to the opcode mnemonic. For example, use "`fbl,a %fcc3,` *label*". In the preceding table, braces signify that the "`,a`" is optional. To set the branch prediction bit, append either "`,pt`" (for predict taken) or "`pn`" (for predict not taken) to the opcode mnemonic. If neither "`,pt`" nor "`,pn`" is specified, the assembler defaults to "`,pt`". To select the appropriate floating-point condition code, include "`%fcc0`", "`%fcc1`", "`%fcc2`", or "`%fcc3`" before the label.

Description Unconditional branches and Fcc-conditional branches are described below.

- **Unconditional branches (FBPA, FBPN)** — If its annul field is 0, an FBPN (Floating-Point Branch Never with Prediction) instruction acts like a NOP. If the Branch Never's annul field is 0, the following (delay) instruction is executed; if the annul (a) bit is 1, the following instruction is annulled (not executed). In no case does an FBPN cause a transfer of control to take place.

  FBPA (Floating-Point Branch Always with Prediction) causes an unconditional PC-relative, delayed control transfer to the address "PC + (4 × **sign_ext**(disp19))". If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul (a) bit is 0, the delay instruction is executed.

- **Fcc-conditional branches** — Conditional FBPfcc instructions (except FBPA and FBPN) evaluate one of the four floating-point condition codes (fcc0, fcc1, fcc2, fcc3) as selected by cc0 and cc1, according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 × **sign_ext**(disp19))". If FALSE, the branch is not taken.

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

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

  The predict bit (p) gives the hardware a hint about whether the branch is expected to be taken. A 1 in the p bit indicates that the branch is expected to be taken. A 0 indicates that the branch is expected not to be taken.

Annulment, delay instructions, and delayed control transfers are described further in Chapter 6, *Instruction Set Overview*.

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

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20), PSTATE.tct = 1, and the FBPfcc instruction will cause a transfer of control (FBPA or taken conditional branch), then FBPfcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the FBPfcc instruction) is stored in TPC[TL] and the value of NPC from before the FBPfcc was executed is stored in TNPC[TL]. Note that FBPN never causes a *control_transfer_instruction* exception.

Exceptions

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

7.20 SIMD Signed Compare VIS 1

Instruction	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FCMPLE16	0 0010 0000	Four 16-bit compare; set R[rd] if *src1* ≤ *src2*	f64	f64	i64	`fcmple16` $freg_{rs1}$, $freg_{rs2}$, reg_{rd}	B1
FCMPNE16	0 0010 0010	Four 16-bit compare; set R[rd] if *src1* ≠ *src2*	f64	f64	i64	`fcmpne16` $freg_{rs1}$, $freg_{rs2}$, reg_{rd}	B1
FCMPLE32	0 0010 0100	Two 32-bit compare; set R[rd] if *src1* ≤ *src2*	f64	f64	i64	`fcmple32` $freg_{rs1}$, $freg_{rs2}$, reg_{rd}	B1
FCMPNE32	0 0010 0110	Two 32-bit compare; set R[rd] if *src1* ≠ *src2*	f64	f64	i64	`fcmpne32` $freg_{rs1}$, $freg_{rs2}$, reg_{rd}	B1
FCMPGT16	0 0010 1000	Four 16-bit compare; set R[rd] if *src1* > *src2*	f64	f64	i64	`fcmpgt16` $freg_{rs1}$, $freg_{rs2}$, reg_{rd}	B1
FCMPEQ16	0 0010 1010	Four 16-bit compare; set R[rd] if *src1* = *src2*	f64	f64	i64	`fcmpeq16` $freg_{rs1}$, $freg_{rs2}$, reg_{rd}	B1
FCMPGT32	0 0010 1100	Two 32-bit compare; set R[rd] if *src1* > *src2*	f64	f64	i64	`fcmpgt32` $freg_{rs1}$, $freg_{rs2}$, reg_{rd}	B1
FCMPEQ32	0 0010 1110	Two 32-bit compare; set R[rd] if *src1* = *src2*	f64	f64	i64	`fcmpeq32` $freg_{rs1}$, $freg_{rs2}$, reg_{rd}	B1

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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

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

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

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

For FCMPEQ{16,32}, each bit in the result is set to 1 if the corresponding signed value in F_D[rs1] is equal to the signed value in F_D[rs2].

For FCMPNE{16,32}, each bit in the result is set to 1 if the corresponding signed value in F_D[rs1] is not equal to the signed value in F_D[rs2].

FIGURE 7-7 and FIGURE 7-8 illustrate 16-bit and 32-bit pixel comparison operations, respectively.

FCMP*<16|32> (SIMD)

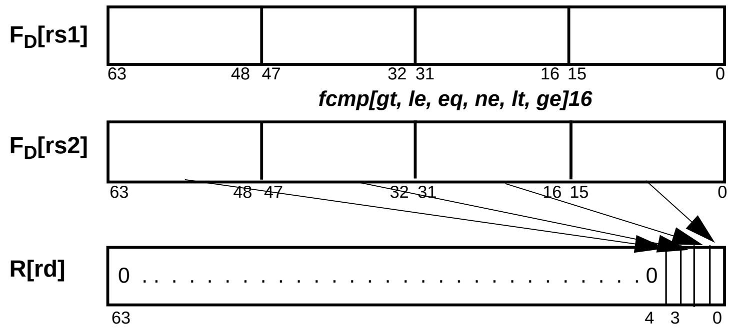


FIGURE 7-7 Four 16-bit Signed Fixed-point SIMD Comparison Operations

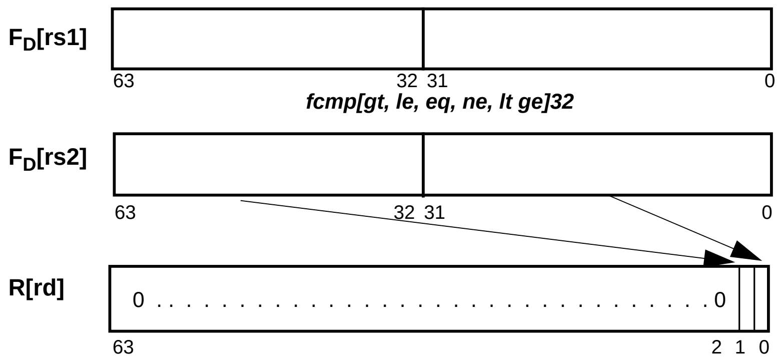


FIGURE 7-8 Two 32-bit Signed Fixed-point SIMD Comparison Operation

In all comparisons, if a compare condition is not true, the corresponding bit in the result is set to 0.

Programming Note The results of a SIMD signed compare operation can be used directly by both integer operations (for example, partial stores) and partitioned conditional moves.

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

Exception *fp_disabled*

See Also Floating-Point Compare on page 128
STPARTIALF on page 260

7.21 Floating-Point Compare

Instruction	opf	Operation	Assembly Language Syntax	Class
FCMPs	0 0101 0001	Compare Single	`fcmps` %fcc*n*, *freg*$_{rs1}$, *freg*$_{rs2}$	**A1**
FCMPd	0 0101 0010	Compare Double	`fcmpd` %fcc*n*, *freg*$_{rs1}$, *freg*$_{rs2}$	**A1**
FCMPq	0 0101 0011	Compare Quad	`fcmpq` %fcc*n*, *freg*$_{rs1}$, *freg*$_{rs2}$	**C3**
FCMPEs	0 0101 0101	Compare Single and Exception if Unordered	`fcmpes` %fcc*n*, *freg*$_{rs1}$, *freg*$_{rs2}$	**A1**
FCMPEd	0 0101 0110	Compare Double and Exception if Unordered	`fcmped` %fcc*n*, *freg*$_{rs1}$, *freg*$_{rs2}$	**A1**
FCMPEq	0 0101 0111	Compare Quad and Exception if Unordered	`fcmpeq` %fcc*n*, *freg*$_{rs1}$, *freg*$_{rs2}$	**C3**

10	—	cc1	cc0	11 0101	rs1	opf	rs2
31 30	29 27	26	25	24 19	18 14	13 5	4 0

cc1	cc0	Condition Code
0	0	`fcc0`
0	1	`fcc1`
1	0	`fcc2`
1	1	`fcc3`

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

Relation	Resulting fcc value
freg$_{rs1}$ = *freg*$_{rs2}$	0
freg$_{rs1}$ < *freg*$_{rs2}$	1
freg$_{rs1}$ > *freg*$_{rs2}$	2
freg$_{rs1}$? *freg*$_{rs2}$ (unordered)	3

The “?” in the preceding table means that the compared values are unordered. The unordered condition occurs when one or both of the operands to the comparison is a signalling or quiet NaN

The “compare and cause exception if unordered” (FCMPEs, FCMPEd, and FCMPEq) instructions cause an invalid (NV) exception if either operand is a NaN.

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

V8 Compatibility Note Unlike the SPARC V8 architecture, SPARC V9 and the UltraSPARC Architecture do not require an instruction between a floating-point compare operation and a floating-point branch (FBfcc, FBPfcc).

SPARC V8 floating-point compare instructions are required to have rd = 0. In SPARC V9 and the UltraSPARC Architecture, bits 26 and 25 of the instruction (rd{1:0}) specify the floating-point condition code to be set. Legal SPARC V8 code will work on SPARC V9 and the UltraSPARC Architecture because the zeroes in the R[rd] field are interpreted as `fcc0` and the FBfcc instruction branches based on the value of `fcc0`.

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

Note UltraSPARC Architecture 2007 processors do not implement in hardware the instructions that refer to quad-precision floating-point registers. An attempt to execute FCMPq or FCMPEq generates an *illegal_instruction* exception, which causes a trap, allowing privileged software to emulate the instruction.

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

An attempt to execute an FCMPq or FCMPEq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

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

Exceptions

illegal_instruction
fp_disabled
fp_exception_ieee_754 (NV)
fp_exception_other (FSR.ftt = invalid_fp_register (FCMPq, FCMPEq only))

See Also

SIMD Signed Compare on page 126

7.22 Floating-Point Divide

Instruction	op3	opf	Operation	Assembly Language Syntax	Class
FDIVs	11 0100	0 0100 1101	Divide Single	`fdivs` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**A1**
FDIVd	11 0100	0 0100 1110	Divide Double	`fdivd` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**A1**
FDIVq	11 0100	0 0100 1111	Divide Quad	`fdivq` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**C3**

10	rd	op3	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description The floating-point divide instructions divide the contents of the floating-point register(s) specified by the rs1 field by the contents of the floating-point register(s) specified by the rs2 field. The instructions then write the quotient into the floating-point register(s) specified by the rd field.

Rounding is performed as specified by FSR.rd.

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware the instructions that refer to quad-precision floating-point registers. An attempt to execute an FDIVq instruction generates an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

An attempt to execute an FADDq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

> **Note** For FDIVs and FDIVd, an *fp_exception_other* with FSR.ftt = unfinished_FPop can occur if the divide unit detects unusual, implementation-specific conditions.

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

Exceptions
illegal_instruction
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FDIVq only)
fp_exception_other (FSR.ftt = unfinished_FPop (FDIVs, FDIV))
fp_exception_ieee_754 (OF, UF, DZ, NV, NX)

7.23 FEXPAND VIS 1

Instruction	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FEXPAND	0 0100 1101	Four 16-bit expands	—	f32	f64	`fexpand` *freg*$_{rs2}$, *freg*$_{rd}$	**B1**

10	rd	110110	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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

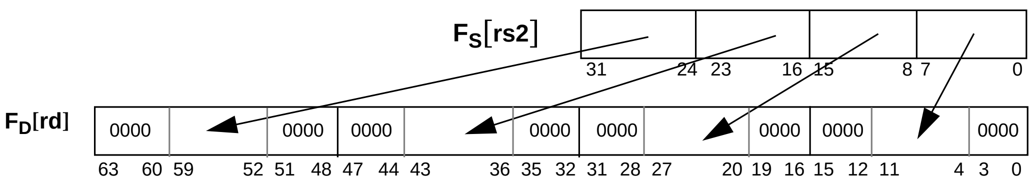


FIGURE 7-9 FEXPAND Operation

This operation is carried out as follows:

1. Left-shift each 8-bit value by 4 and zero-extend each result to a 16-bit fixed value.
2. Store the result in the destination register, F$_D$[rd].

Programming Note | FEXPAND performs the inverse of the FPACK16 operation.

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

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

Exceptions *illegal_instruction*
fp_disabled

See Also FPMERGE on page 160
FPACK on page 153

7.24 Convert 32-bit Integer to Floating Point

Instruction	op3	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FiTOs	11 0100	0 1100 0100	Convert 32-bit Integer to Single	—	f32	f32	`fitos` *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FiTOd	11 0100	0 1100 1000	Convert 32-bit Integer to Double	—	f32	f64	`fitod` *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FiTOq	11 0100	0 1100 1100	Convert 32-bit Integer to Quad	—	f32	f128	`fitoq` *freg*$_{rs2}$, *freg*$_{rd}$	**C3**

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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

The value of FSR.rd determines how rounding is performed by FiTOs.

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FiTOq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FiTO<s|d|q> instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FiTO<s|d|q> instruction causes an *fp_disabled* exception.

An attempt to execute an FiTOq instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

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

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FiTOq))
fp_exception_ieee_754 (NX (FiTOs only))

7.25 Flush Instruction Memory

Instruction	op3	Operation	Assembly Language Syntax†	Class
FLUSH	11 1011	Flush Instruction Memory	`flush [address]`	**A1**

† The original assembly language syntax for a FLUSH instruction ("`flush` *address*") has been deprecated because of inconsistency with other SPARC assembly language syntax. Over time, assemblers will support the new syntax for this instruction. In the meantime, some existing assemblers may only recognize the original syntax.

10	—	op3	rs1	i=0	—	rs2
10	—	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

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

The SPARC V9 instruction set architecture does not guarantee consistency between instruction memory and data memory. When software writes[1] to a memory location that may be executed as an instruction (self-modifying code[2]), a potential memory consistency problem arises, which is addressed by the `FLUSH` instruction. Use of FLUSH after instruction memory has been modified ensures that instruction and data memory are synchronized for the processor that issues the FLUSH instruction.

The virtual processor waits until all previous (cacheable) stores have completed before issuing a FLUSH instruction. For the purpose of memory ordering, a FLUSH instruction behaves like a store instruction.

In the following discussion P_{FLUSH} refers to the virtual processor that executed the FLUSH instruction.

FLUSH causes a synchronization within a virtual processor which ensures that instruction fetches from the specified effective address by P_{FLUSH} appear to execute after any loads, stores, and atomic load-stores to that address issued by P_{FLUSH} prior to the FLUSH. In a multiprocessor system, FLUSH also ensures that these values will eventually become visible to the instruction fetches of all other virtual processors in the system. With respect to MEMBAR-induced orderings, FLUSH behaves as if it is a store operation (see *Memory Barrier* on page 201).

Given any store S_A to address A, that precedes in memory order a FLUSH F_A to address A, that in turn precedes in memory order a store S_B to address B; if any instruction I_B fetched from address B executes the instruction created by store S_B, then any instruction I_A that fetched from address A and that follows I_B in program order cannot execute any version of the instruction from address A that existed prior to the store S_A.

The preceeding statement defines an ordering requirement to which UltraSPARC Architecture processors comply. By using a FLUSH instruction between two stores that modify instructions, atomicity between the two stores is guaranteed such that any virtual processor executing the instruction modified by the later store will never fetch and/or execute the instruction before it was modified by the earlier store.

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

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

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

FLUSH

See implementation-specific documentation for details on specific implementations of the FLUSH instruction.

On an UltraSPARC Architecture processor:

- A FLUSH instruction causes a synchronization within the virtual processor on which the FLUSH is executed, which flushes its instruction pipeline to ensure that no instruction already fetched has subsequently been modified in memory. Any other virtual processors on the same physical processor are unaffected by a FLUSH.
- Coherency between instruction and data memories may or may not be maintained by hardware.

IMPL. DEP. #409-S10: The implementation of the FLUSH instruction is implementation dependent. If the implementation automatically maintains consistency between instruction and data memory,
(1) the FLUSH address is ignored and
(2) the FLUSH instruction cannot cause any data access exceptions, because
its effective address operand is not translated or used by the MMU.
On the other hand, if the implementation does *not* maintain consistency between instruction and data memory, the FLUSH address is used to access the MMU and the FLUSH instruction can cause data access exceptions.

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

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

Programming Notes |

1.Typically, FLUSH is used in self-modifying code.
The use of self-modifying code is discouraged.

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

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

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

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

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

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

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

Implementation Note In a multiprocessor configuration, FLUSH requires all processors that may be referencing the addressed doubleword to flush their instruction caches, which is a potentially disruptive activity.

V9 Compatibility Note The effect of a FLUSH instruction as observed from the virtual processor on which FLUSH executes is immediate. Other virtual processors in a multiprocessor system eventually will see the effect of the FLUSH, but the latency is implementation dependent.

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

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

Exceptions

illegal_instruction
DAE_nfo_page

7.26 Flush Register Windows

Instruction	op3	Operation	Assembly Language Syntax	Class
FLUSHW	10 1011	Flush Register Windows	`flushw`	**A1**

31 30	29 … 25	24 … 19	18 … 14	13	12 … 0
10	—	op3	—	i=0	—

Description FLUSHW causes all active register windows except the current window to be flushed to memory at locations determined by privileged software. FLUSHW behaves as a NOP if there are no active windows other than the current window. At the completion of the FLUSHW instruction, the only active register window is the current one.

Programming Note The FLUSHW instruction can be used by application software to flush register windows to memory so that it can switch memory stacks or examine register contents from previous stack frames.

FLUSHW acts as a NOP if CANSAVE = *N_REG_WINDOWS* − 2. Otherwise, there is more than one active window, so FLUSHW causes a spill exception. The trap vector for the spill exception is based on the contents of OTHERWIN and WSTATE. The spill trap handler is invoked with the CWP set to the window to be spilled (that is, (CWP + CANSAVE + 2) **mod** *N_REG_WINDOWS*). See *Register Window Management Instructions* on page 83.

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

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

Exceptions *illegal_instruction*
spill_n_normal
spill_n_other

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

Instruction	op5	Operation	Assembly Language Syntax		Class	Added
FMADDs	00 01	Multiply-Add Single	`fmadds`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**C3**	UA 2007
FMADDd	00 10	Multiply-Add Double	`fmaddd`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**C3**	UA 2007
FMSUBs	01 01	Multiply-Subtract Single	`fmsubs`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**C3**	UA 2007
FMSUBd	01 10	Multiply-Subtract Double	`fmsubd`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**C3**	UA 2007
						UA 2007
FNMSUBs	10 01	Negative Multiply-Subtract Single	`fnmsubs`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**C3**	UA 2007
FNMSUBd	10 10	Negative Multiply-Subtract Double	`fnmsubd`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**C3**	UA 2007
FNMADDs	11 01	Negative Multiply-Add Single	`fnmadds`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**C3**	UA 2007
FNMADDd	11 10	Negative Multiply-Add Double	`fnmaddd`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**C3**	UA 2007

10	rd	110111	rs1	rs3	op5	rs2
31 30	29 25	24 19	18 14	13 9	8 5	4 0

Instruction	Implementation
Multiply-Add (fused)	F[rd] ← (F[rs1] x F[rs2]) + F[rs3]
Multiply-Subtract (fused)	F[rd] ← (F[rs1] x F[rs2]) − F[rs3]
Negative Multiply-Add (fused)	F[rd] ← − ((F[rs1] x F[rs2]) + F[rs3])
Negative Multiply-Subtract (fused)	F[rd] ← − ((F[rs1] x F[rs2]) − F[rs3])

Description

The fused floating-point multiply-add instructions, FMADD<s|d>, multiply the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, add that product to the register(s) specified by rs3, round the result, and write the result into the floating-point register(s) specified by rd.

The fused floating-point multiply-subtract instructions, FMSUB<s|d>, multiply the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, subtract from that product the register(s) specified by rs3, round the result, and write the result into the floating-point register(s) specified by rd.

The fused floating-point negative multiply-add instructions, FNMADD<s|d>, multiply the floating-point register(s) specified by rs1 and the floating-point register(s) specified by rs2, add to the product the register(s) specified by rs3, negate the result, round the result, and write the result into the floating-point register(s) specified by rd.

The fused floating-point negative multiply-subtract instructions, FNMSUB<s|d>, multiply the floating-point register(s) specified by the rs1 field and the floating-point register(s) specified by the rs2 field, subtract from the product the register(s) specified by the rs3 field, negate the result, round the result, and write the result into the floating-point register(s) specified by the rd field.

All of the above instructions are "fused" operations; no rounding is performed between the multiplication operation and the subsequent addition (or subtraction). Therefore, at most one rounding step occurs.

The negative fused multiply-add/subtract instructions (FNM*) treat NaN values as follows:

- A source QNaN propagates with its sign bit unchanged
- A generated (default response) QNaN result has a sign bit of zero
- A source SNaN that is converted to a QNaN result retains the sign bit of the source SNaN

Exceptions. If an FMAf instruction is not implemented in hardware, it generates an *illegal_instruction* exception, so that privileged software can emulate the instruction.

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

Overflow, underflow, and inexact exception bits within FSR.cexc and FSR.aexc are updated based on the final result of the operation and not on the intermediate result of the multiplication. The invalid operation exception bits within FSR.cexc and FSR.aexc are updated as if the multiplication and the addition/subtraction were performed using two individual instructions. An invalid operation exception is detected when any of the following conditions are true:

- A source operand (F[rs1], F[rs2], or F[rs3]) is a SNaN
- $\infty \times 0$
- $\infty - \infty$

If the instruction generates an IEEE-754 exception or exceptions for which the corresponding trap enable mask (FSR.tem) bits are set, an *fp_exception_ieee_754* exception and subsequent trap is generated.

If either the multiply or the add/subtract operation detects an unfinished_FPop condition (for example, due to a subnormal operand or final result), the Multiply-Add/Subtract instruction generates an *fp_exception_other* exception with FSR.ftt = unfinished_FPop. An *fp_exception_other* exception with FSR.ftt = unfinished_FPop always takes precedence over an *fp_exception_ieee_754* exception. That is, if an *fp_exception_other* exception occurs due to an unfinished_FPop condition, the FSR.cexc and FSR.aexc fields remain unchanged even if a floating point IEEE 754 exception occurs during the multiply operation (regardless whether traps are enabled, via FSR.tem, for the IEEE exception) and the unfinished_FPop condition occurs during the subsequent add/subtract operation.

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

Semantic Definitions

FMADD:

```
(1) tmp ← F[rs1] x F[rs2]
(2) tmp ← tmp + F[rs3]

(3) F[rd] ← round(tmp)
```

FNMADD:

```
(1) tmp ← F[rs1] × F[rs2]
(2) tmp ← tmp + F[rs3]
(3) tmp ← − tmp
(4) F[rd] ← round(tmp)
```

FMSUB:

```
(1) tmp ← F[rs1] x F[rs2]
(2) tmp ← tmp − F[rs3]

(3) F[rd] ← round(tmp)
```

FNMSUB:

```
(1) tmp ← F[rs1] × F[rs2]
(2) tmp ← tmp − F[rs3]
(3) tmp ← − tmp
(4) F[rd] ← round(tmp)
```

Exceptions

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

See Also

FMUL on page 151
FADD on page 120
FSUB on page 161

7.28 Floating-Point Move

Instruction	op3	opf	Operation	Assembly Language Syntax	Class
FMOVs	11 0100	0 0000 0001	Move (copy) Single	`fmovs` *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FMOVd	11 0100	0 0000 0010	Move (copy) Double	`fmovd` *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FMOVq	11 0100	0 0000 0011	Move (copy) Quad	`fmovq` *freg*$_{rs2}$, *freg*$_{rd}$	**C3**

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description FMOV copies the source floating-point register(s) to the destination floating-point register(s), unaltered.

FMOVs, FMOVd, and FMOVq perform 32-bit, 64-bit, and 128-bit operations, respectively.

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

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMOVq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

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

An attempt to execute an FMOVq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

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

Exceptions *illegal_instruction*
fp_disabled *fp_exception_other* (FSR.ftt = invalid_fp_register (FMOVq only))

See Also *f Register Logical Operate (2 operand)* on page 164

7.29 Move Floating-Point Register on Condition (FMOVcc)

Instruction	opf_low	Operation	Assembly Language Syntax	Class
FMOVSicc	00 0001	Move Floating-Point Single, based on 32-bit integer condition codes	`fmovs`*icc* `%icc`, $freg_{rs2}$, $freg_{rd}$	**A1**
FMOVDicc	00 0010	Move Floating-Point Double, based on 32-bit integer condition codes	`fmovd`*icc* `%icc`, $freg_{rs2}$, $freg_{rd}$	**A1**
FMOVQicc	00 0011	Move Floating-Point Quad, based on 32-bit integer condition codes	`fmovq`*icc* `%icc`, $freg_{rs2}$, $freg_{rd}$	**C3**
FMOVSxcc	00 0001	Move Floating-Point Single, based on 64-bit integer condition codes	`fmovs`*xcc* `%xcc`, $freg_{rs2}$, $freg_{rd}$	**A1**
FMOVDxcc	00 0010	Move Floating-Point Double, based on 64-bit integer condition codes	`fmovd`*xcc* `%xcc`, $freg_{rs2}$, $freg_{rd}$	**A1**
FMOVQxcc	00 0011	Move Floating-Point Quad, based on 64-bit integer condition codes	`fmovq`*xcc* `%xcc`, $freg_{rs2}$, $freg_{rd}$	**C3**
FMOVSfcc	00 0001	Move Floating-Point Single, based on floating-point condition codes	`fmovs`*fcc* `%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	**A1**
FMOVDfcc	00 0010	Move Floating-Point Double, based on floating-point condition codes	`fmovd`*fcc* `%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	**A1**
FMOVQfcc	00 0011	Move Floating-Point Quad, based on floating-point condition codes	`fmovq`*fcc* `%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	**C3**

10	rd	110101	—	cond	opf_cc	opf_low	rs2
31 30	29 25	24 19	18	17 14	13 11	10 5	4 0

FMOVcc

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

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

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

cond	Operation	fcc*n* Test	*fcc* name(s) in Assembly Language Mnemonics
1000	Move Always	1	`a`
0000	Move Never	0	`n`
0111	Move if Unordered	U	`u`
0110	Move if Greater	G	`g`
0101	Move if Unordered or Greater	G **or** U	`ug`
0100	Move if Less	L	`l`
0011	Move if Unordered or Less	L **or** U	`ul`
0010	Move if Less or Greater	L **or** G	`lg`
0001	Move if Not Equal	L **or** G **or** U	`ne` (or `nz`)
1001	Move if Equal	E	`e` (or `z`
1010	Move if Unordered or Equal	E **or** U	`ue`
1011	Move if Greater or Equal	E **or** G	`ge`
1100	Move if Unordered or Greater or Equal	E **or** G **or** U	`uge`
1101	Move if Less or Equal	E **or** L	`le`
1110	Move if Unordered or Less or Equal	E **or** L **or** U	`ule`
1111	Move if Ordered	E **or** L **or** G	`o`

Encoding of opf_cc *Field (also see* TABLE E-10 *on page 484)*

opf_cc	Instruction	Condition Code to be Tested
100_2	FMOV<s | d | q>icc	icc
110_2	FMOV<s | d | q>xcc	xcc
000_2	FMOV<s | d | q>fcc	fcc0
001_2		fcc1
010_2		fcc2
011_2		fcc3
101_2	(*illegal_instruction* exception)	
111_2		

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

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

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

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMOVQicc, FMOVQxcc, or FMOVQfcc instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FMOVcc instruction when instruction bit 18 is nonzero or opf_cc = 101_2 or 111_2 causes an *illegal_instruction* exception.

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

An attempt to execute an FMOVQicc, FMOVQxcc, or FMOVQfcc instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

FMOVcc

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

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

can be coded as

```
! assume A is in %f0; B is in %f2; %xx points to
! constant area
       ldd     [%xx+C_1.03],%f4    ! X  = 1.03
       fcmpd   %fcc3,%f0,%f2       ! A > B
       fble,a  %fcc3,label
       ! following instructiononly executed if the
       ! preceding branch was taken
       fsubd   %f4,%f4,%f4         ! X  = 0.0
   label:...
```

This code takes four instructions including a branch.

With FMOVcc, this could be coded as

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

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

Exceptions
illegal_instruction
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FMOVQ instructions))

7.30 Move Floating-Point Register on Integer Register Condition (FMOVR)

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

10	rd	110101	rs1	—	rcond	opf_low	rs2
31 30	29 25	24 19	18 14	13	12 10	9 5	4 0

Assembly Language Syntax		
`fmovr{s,d,q}z`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	(*synonym:* `fmovr{s,d,q}e`)
`fmovr{s,d,q}lez`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	
`fmovr{s,d,q}lz`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	
`fmovr{s,d,q}nz`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	(*synonym:* `fmovr{s,d,q}ne`)
`fmovr{s,d,q}gz`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	
`fmovr{s,d,q}gez`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	

FMOVR

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

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

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

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMOVRq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FMOVR instruction when instruction bit 13 is nonzero or rcond = 000_2 or 100_2 causes an *illegal_instruction* exception.

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

An attempt to execute an FMOVRq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

> **Implementation Note** If this instruction is implemented by tagging each register value with an N (negative) and a Z (zero) condition bit, use the following table to determine whether rcond is TRUE:
>
> | Branch | Test |
> |---|---|
> | FMOVRNZ | **not** Z |
> | FMOVRZ | Z |
> | FMOVRGEZ | **not** N |
> | FMOVRLZ | N |
> | FMOVRLEZ | N **or** Z |
> | FMOVRGZ | N n**or** Z |

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

7.31 Partitioned Multiply Instructions VIS 1

Instruction	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FMUL8x16	0 0011 0001	Unsigned 8-bit by signed 16-bit partitioned product	f32	f64	f64	`fmul8x16` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**
FMUL8x16AU	0 0011 0011	Unsigned 8-bit by signed 16-bit upper α partitioned product	f32	f32	f64	`fmul8x16au` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**
FMUL8x16AL	0 0011 0101	Unsigned 8-bit by signed 16-bit lower α partitioned product	f32	f32	f64	`fmul8x16al` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**
FMUL8SUx16	0 0011 0110	Signed upper 8-bit by signed 16-bit partitioned product	f64	f64	f64	`fmul8sux16` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**
FMUL8ULx16	0 0011 0111	Unsigned lower 8-bit by signed 16-bit partitioned product	f64	f64	f64	`fmul8ulx16` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**
FMULD8SUx16	0 0011 1000	Signed upper 8-bit by signed 16-bit partitioned product	f32	f32	f64	`fmuld8sux16` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**
FMULD8ULx16	0 0011 1001	Unsigned lower 8-bit by signed 16-bit partitioned product	f32	f32	f64	`fmuld8ulx16` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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

Description The following sections describe the versions of partitioned multiplies.

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

Exceptions *fp_disabled*

7.31.1 FMUL8x16 Instruction

FMUL8x16 multiplies each unsigned 8-bit value (for example, a pixel component) in the 32-bit floating-point register F_S[rs1] by the corresponding (signed) 16-bit fixed-point integer in the 64-bit floating-point register F_D[rs2]. It rounds the 24-bit product (assuming binary point between bits 7 and 8) and stores the most significant 16 bits of the result into the corresponding 16-bit field in the 64-bit floating-point destination register F_D[rd]. FIGURE 7-10 illustrates the operation.

Note | This instruction treats the pixel component values as fixed-point with the binary point to the left of the most significant bit. Typically, this operation is used with filter coefficients as the fixed-point rs2 value and image data as the rs1 pixel value. Appropriate scaling of the coefficient allows various fixed-point scaling to be realized.

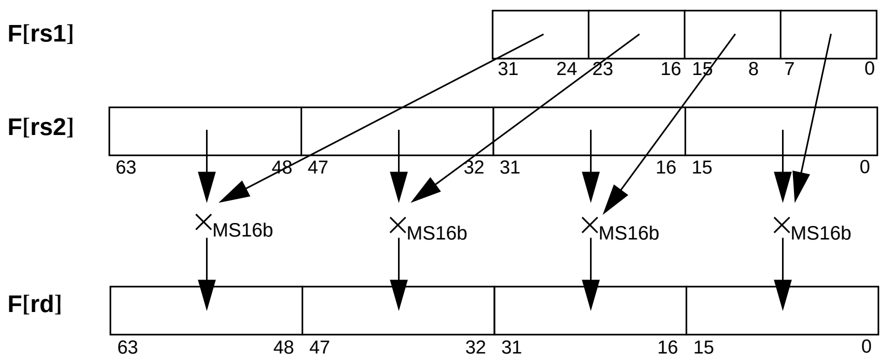


FIGURE 7-10 FMUL8x16 Operation

7.31.2 FMUL8x16AU Instruction

FMUL8x16AU is the same as FMUL8x16, except that one 16-bit fixed-point value is used as the multiplier for all four multiplies. This multiplier is the most significant ("upper") 16 bits of the 32-bit register F_S[rs2] (typically an α pixel component value). FIGURE 7-11 illustrates the operation.

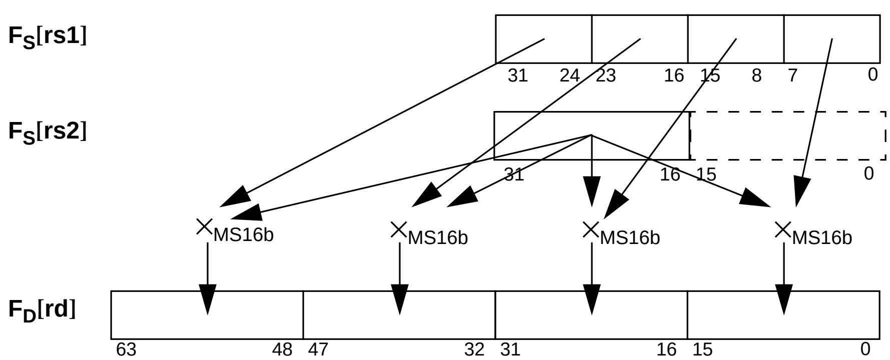


FIGURE 7-11 FMUL8x16AU Operation

7.31.3 FMUL8x16AL Instruction

FMUL8x16AL is the same as FMUL8x16AU, except that the least significant ("lower") 16 bits of the 32-bit register F_S[rs2] register are used as a multiplier. FIGURE 7-12 illustrates the operation.

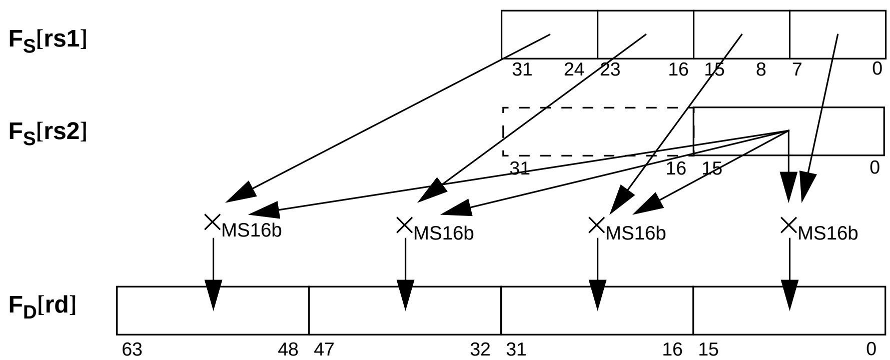


FIGURE 7-12 FMUL8x16AL Operation

7.31.4 FMUL8SUx16 Instruction

FMUL8SUx16 multiplies the most significant ("upper") 8 bits of each 16-bit signed value in the 64-bit floating-point register F_D[rs1] by the corresponding signed, 16-bit, fixed-point, signed integer in the 64-bit floating-point register F_D[rs2]. It rounds the 24-bit product toward the nearest representable value and then stores the most significant 16 bits of the result into the corresponding 16-bit field of the 64-bit floating-point destination register F_D[rd]. If the product is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE 7-13 illustrates the operation.

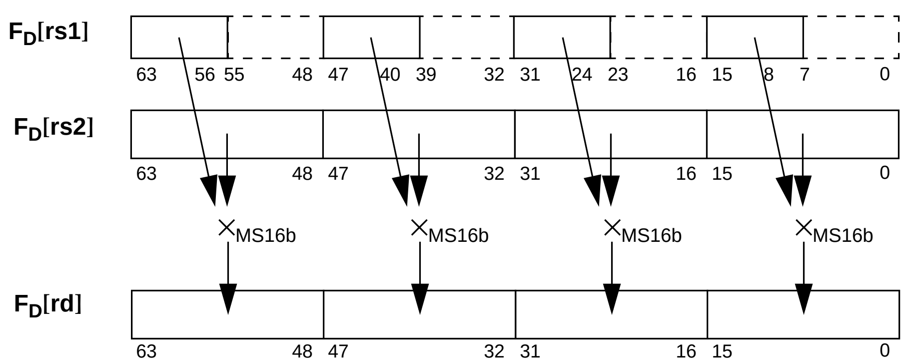


FIGURE 7-13 FMUL8SUx16 Operation

7.31.5 FMUL8ULx16 Instruction

FMUL8ULx16 multiplies the unsigned least significant ("lower") 8 bits of each 16-bit value in the 64-bit floating-point register F_D[rs1] by the corresponding fixed-point signed 16-bit integer in the 64-bit floating-point register F_D[rs2]. Each 24-bit product is sign-extended to 32 bits. The most significant ("upper") 16 bits of the sign-extended value are rounded to nearest and then stored in the corresponding 16-bit field of the 64-bit floating-point destination register F_D[rd]. If the result is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE 7-14 illustrates the operation; CODE EXAMPLE 7-1 exemplifies the operation.

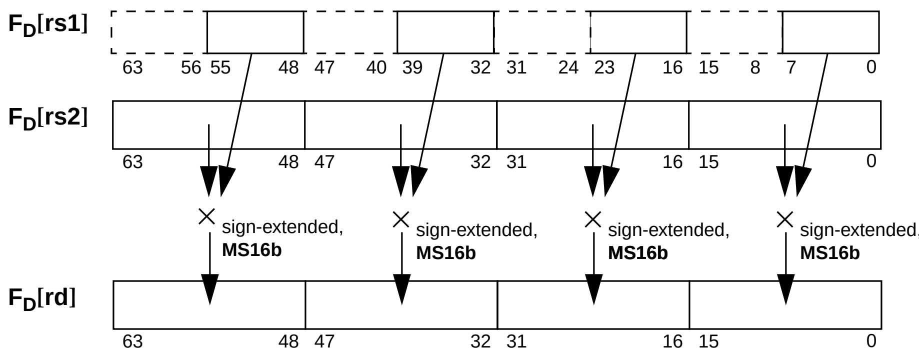


FIGURE 7-14 FMUL8ULx16 Operation

CODE EXAMPLE 7-1 16-bit × 16-bit 16-bit Multiply

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

7.31.6 FMULD8SUx16 Instruction

FMULD8SUx16 multiplies the most significant (“upper”) 8 bits of each 16-bit signed value in F[rs1] by the corresponding signed 16-bit fixed-point value in F[rs2]. Each 24-bit product is shifted left by 8 bits to generate a 32-bit result, which is then stored in the 64-bit floating-point register specified by rd. FIGURE 7-15 illustrates the operation.

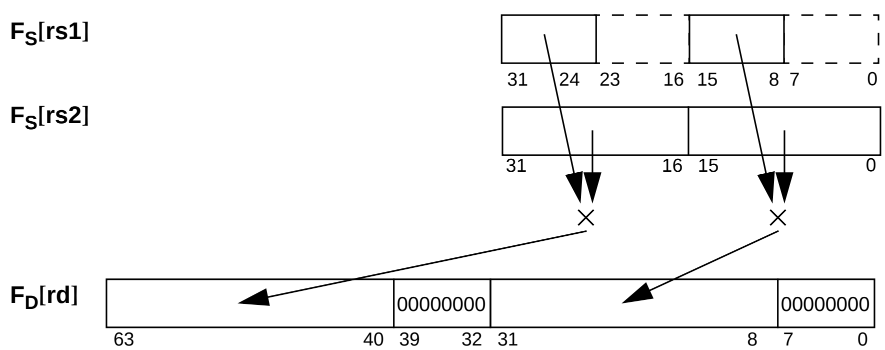


FIGURE 7-15 FMULD8SUx16 Operation

7.31.7 FMULD8ULx16 Instruction

FMULD8ULx16 multiplies the unsigned least significant ("lower") 8 bits of each 16-bit value in F[rs1] by the corresponding 16-bit fixed-point signed integer in F[rs2]. Each 24-bit product is sign-extended to 32 bits and stored in the corresponding half of the 64-bit floating-point register specified by rd. FIGURE 7-16 illustrates the operation; CODE EXAMPLE 7-2 exemplifies the operation.

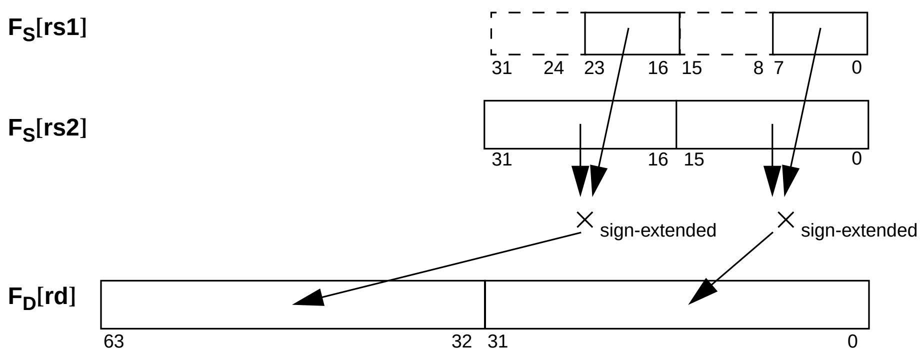


FIGURE 7-16 FMULD8ULx16 Operation

CODE EXAMPLE 7-2 16-bit x 16-bit 32-bit Multiply

```
fmuld8sux16  %f0, %f1, %f2
fmuld8ulx16  %f0, %f1, %f3
fpadd32      %f2, %f3, %f4
```

7.32 Floating-Point Multiply

Instruction	op3	opf	Operation	Assembly Language Syntax	Class
FMULs	11 0100	0 0100 1001	Multiply Single	`fmuls` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**A1**
FMULd	11 0100	0 0100 1010	Multiply Double	`fmuld` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**A1**
FMULq	11 0100	0 0100 1011	Multiply Quad	`fmulq` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**C3**
FsMULd	11 0100	0 0110 1001	Multiply Single to Double	`fsmuld` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**A1**
FdMULq	11 0100	0 0110 1110	Multiply Double to Quad	`fdmulq` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**C3**

10	rd	op3	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description The floating-point multiply instructions multiply the contents of the floating-point register(s) specified by the rs1 field by the contents of the floating-point register(s) specified by the rs2 field. The instructions then write the product into the floating-point register(s) specified by the rd field.

The FsMULd instruction provides the exact double-precision product of two single-precision operands, without underflow, overflow, or rounding error. Similarly, FdMULq provides the exact quad-precision product of two double-precision operands.

Rounding is performed as specified by FSR.rd.

Note UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FMULq or FdMULq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

An attempt to execute an FMULq instruction when rs1{1} ≠ 0 or rs2{1} ≠ 0 or rd{1:0} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

An attempt to execute an FdMULq instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

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

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FMULq and FdMULq only))
fp_exception_other (FSR.ftt = unfinished_FPop)
fp_exception_ieee_754 (any: NV; FMUL<s|d|q> only: OF, UF, NX)

See Also FMAf on page 137

7.33 Floating-Point Negate

Instruction	op3	opf	Operation	Assembly Language Syntax	Class
FNEGs	11 0100	0 0000 0101	Negate Single	`fnegs` *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FNEGd	11 0100	0 0000 0110	Negate Double	`fnegd` *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FNEGq	11 0100	0 0000 0111	Negate Quad	`fnegq` *freg*$_{rs2}$, *freg*$_{rd}$	**C3**

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description FNEG copies the source floating-point register(s) to the destination floating-point register(s), with the sign bit complemented.

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

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FNEGq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

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

An attempt to execute an FNEGq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FNEGq only))

7.34 FPACK VIS 1

Instruction	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FPACK16	0 0011 1011	Four 16-bit packs into 8 unsigned bits	—	f64	f32	`fpack16` $freg_{rs2}$, $freg_{rd}$	**B1**
FPACK32	0 0011 1010	Two 32-bit packs into 8 unsigned bits	f64	f64	f64	`fpack32` $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	**B1**
FPACKFIX	0 0011 1101	Four 16-bit packs into 16 signed bits	—	f64	f32	`fpackfix` $freg_{rs2}$, $freg_{rd}$	**B1**

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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 131
FPMERGE on page 160

7.34.1 FPACK16

FPACK16 takes four 16-bit fixed values from the 64-bit floating-point register F_D[rs2], scales, truncates, and clips them into four 8-bit unsigned integers, and stores the results in the 32-bit destination register, F_S[rd]. FIGURE 7-17 illustrates the FPACK16 operation.

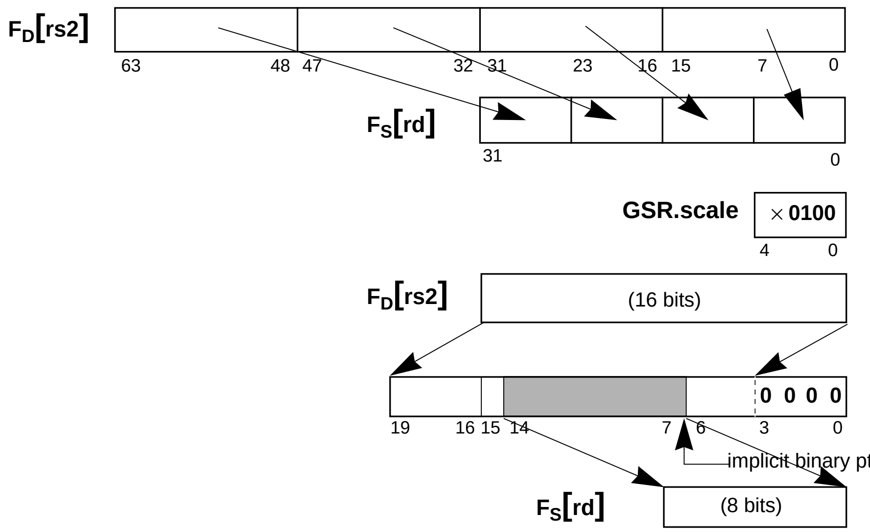


FIGURE 7-17 FPACK16 Operation

> **Note** FPACK16 ignores the most significant bit of GSR.scale (GSR.scale{4}).

This operation is carried out as follows:

1. Left-shift the value from F_D[rs2] by the number of bits specified in GSR.scale while maintaining clipping information.
2. Truncate and clip to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 7 and 6 for each 16-bit word). Truncation converts the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative (that is, its most significant bit is set), 0 is returned as the clipped value. If the value is greater than 255, then 255 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.
3. Store the result in the corresponding byte in the 32-bit destination register, F_S[rd].

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

```
tmp ← source_operand{15:0} << GSR.scale;
// Pick off the bits from bit position 15+GSR.scale to
// bit position 7 from the shifted result
trunc_signed_value ← tmp{(15+GSR.scale):7};
If (trunc_signed_value < 0)
    unsigned_8bit_result ← 0;
else if (trunc_signed_value > 255)
    unsigned_8bit_result ← 255;
else
    unsigned_8bit_result ← trunc_signed_value{14:7};
```

7.34.2 FPACK32

FPACK32 takes two 32-bit fixed values from the second source operand (64-bit floating-point register F_D[rs2]) and scales, truncates, and clips them into two 8-bit unsigned integers. The two 8-bit integers are merged at the corresponding least significant byte positions of each 32-bit word in the 64-bit floating-point register F_D[rs1], left-shifted by 8 bits. The 64-bit result is stored in F_D[rd]. Thus, successive FPACK32 instructions can assemble two pixels by using three or four pairs of 32-bit fixed values. FIGURE 7-18 illustrates the FPACK32 operation.

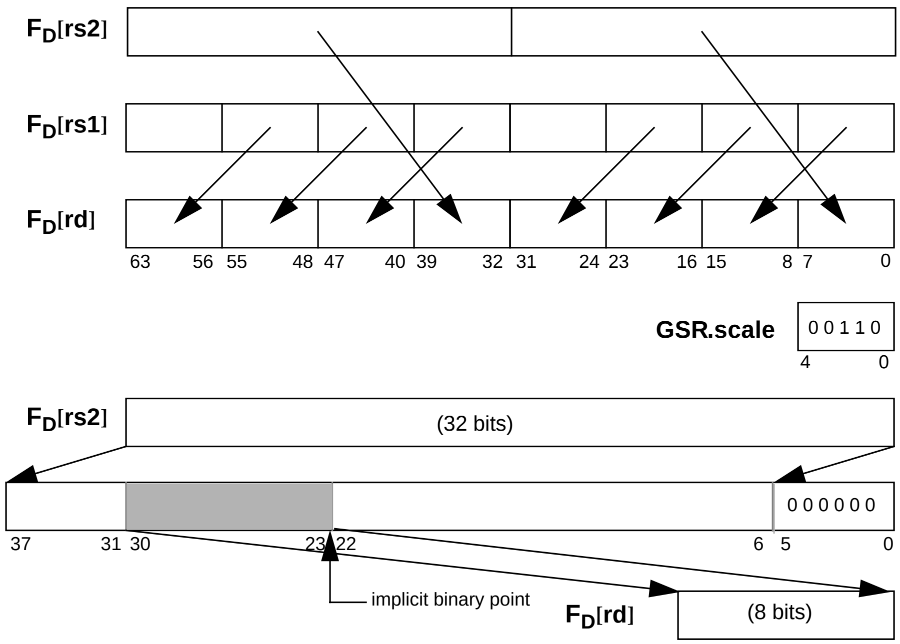


FIGURE 7-18 FPACK32 Operation

This operation, illustrated in FIGURE 7-18, is carried out as follows:

1. Left-shift each 32-bit value in F_D[rs2] by the number of bits specified in GSR.scale, while maintaining clipping information.
2. For each 32-bit value, truncate and clip to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 23 and 22 for each 32-bit word). Truncation is performed to convert the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative (that is, the most significant bit is 1), then 0 is returned as the clipped value. If the value is greater than 255, then 255 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.
3. Left-shift each 32-bit value from F_D[rs1] by 8 bits.
4. Merge the two clipped 8-bit unsigned values into the corresponding least significant byte positions in the left-shifted F_D[rs2] value.
5. Store the result in the 64-bit destination register F_D[rd].

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

```
tmp ← source_operand2{31:0} << GSR.scale;
// Pick off the bits from bit position 31+GSR.scale to
// bit position 23 from the shifted result
trunc_signed_value ← tmp{(31+GSR.scale):23};
if (trunc_signed_value < 0)
    unsigned_8bit_value ← 0;
```

```
else if (trunc_signed_value > 255)
   unsigned_8bit_value ← 255;
else
   unsigned_8bit_value ← trunc_signed_value{30:23};
Final_32bit_Result ← (source_operand1{31:0} << 8) |
         (unsigned_8bit_value{7:0});
```

7.34.3 FPACKFIX

FPACKFIX takes two 32-bit fixed values from the 64-bit floating-point register F$_D$[rs2], scales, truncates, and clips them into two 16-bit unsigned integers, and then stores the result in the 32-bit destination register F$_S$[rd]. FIGURE 7-19 illustrates the FPACKFIX operation.

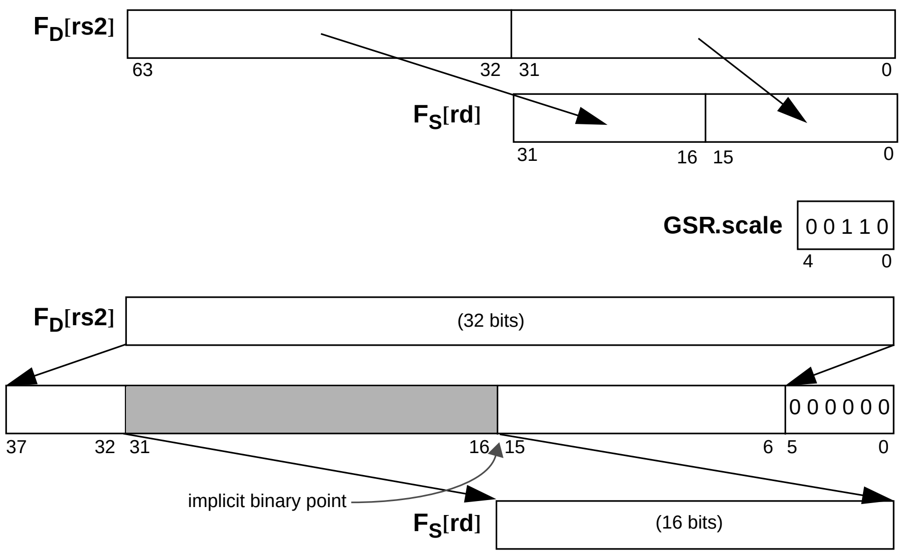


FIGURE 7-19 FPACKFIX Operation

This operation is carried out as follows:

1. Left-shift each 32-bit value from F$_D$[rs2]) by the number of bits specified in GSR.scale, while maintaining clipping information.
2. For each 32-bit value, truncate and clip to a 16-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (that is, between bits 16 and 15 for each 32-bit word). Truncation is performed to convert the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is less than –32768, then –32768 is returned as the clipped value. If the value is greater than 32767, then 32767 is delivered as the clipped value. Otherwise, the scaled value is returned as the result.
3. Store the result in the 32-bit destination register F$_S$[rd].

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

```
tmp ← source_operand{31:0} << GSR.scale;
// Pick off the bits from bit position 31+GSR.scale to
// bit position 16 from the shifted result
trunc_signed_value ← tmp{(31+GSR.scale):16};
if (trunc_signed_value < -32768)
   signed_16bit_result ← -32768;
else if (trunc_signed_value > 32767)
   signed_16bit_result ← 32767;
```

```
else
    signed_16bit_result ← trunc_signed_value{31:16};
```

7.35 Fixed-point Partitioned Add VIS 1

Instruction	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FPADD16	0 0101 0000	Four 16-bit adds	f64	f64	f64	`fpadd16` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FPADD16S	0 0101 0001	Two 16-bit adds	f32	f32	f32	`fpadd16s` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FPADD32	0 0101 0010	Two 32-bit adds	f64	f64	f64	`fpadd32` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FPADD32S	0 0101 0011	One 32-bit add	f32	f32	f32	`fpadd32s` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description FPADD16 (FPADD32) performs four 16-bit (two 32-bit) partitioned additions between the corresponding fixed-point values contained in the source operands (F_D[rs1], F_D[rs2]). The result is placed in the destination register, F_D[rd].

The 32-bit versions of these instructions (FPADD16S and FPADD32S) perform two 16-bit or one 32-bit partitioned additions.

Any carry out from each addition is discarded and a 2′s-complement arithmetic result is produced.

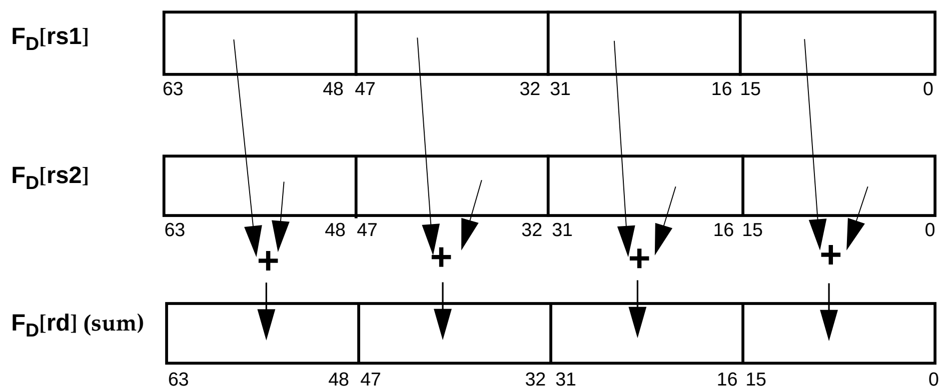


FIGURE 7-20 FPADD16 Operation

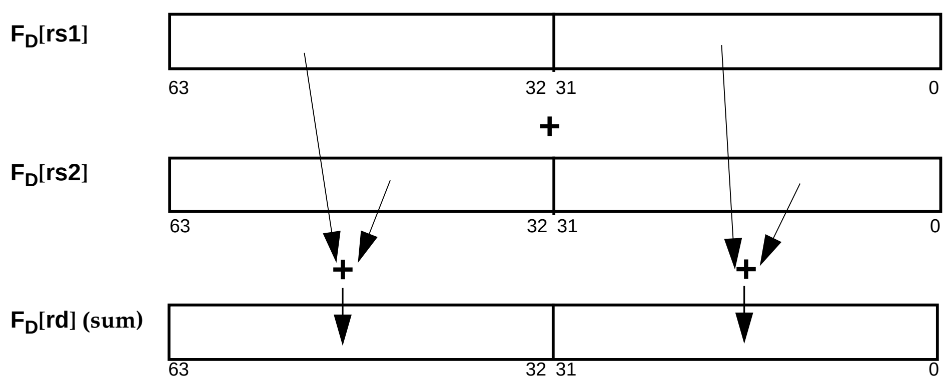


FIGURE 7-21 FPADD32 Operation

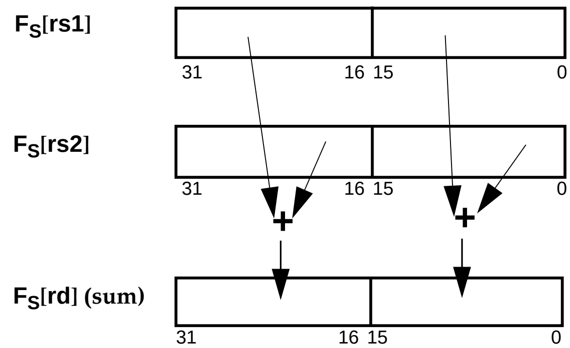


FIGURE 7-22 FPADD16S Operation

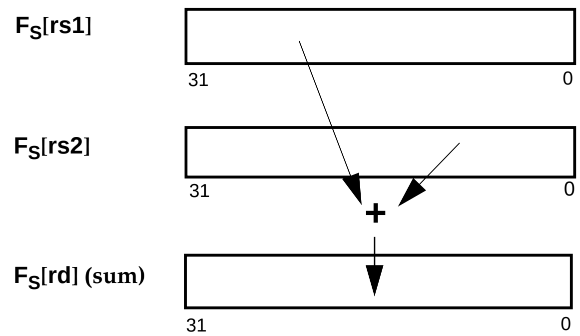


FIGURE 7-23 FPADD32S Operation

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

Exceptions *fp_disabled*

7.36 FPMERGE VIS 1

Instruction	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FPMERGE	0 0100 1011	Two 32-bit merges	f32	f32	f64	`fpmerge` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**B1**

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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

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

FIGURE 7-24 illustrates the operation.

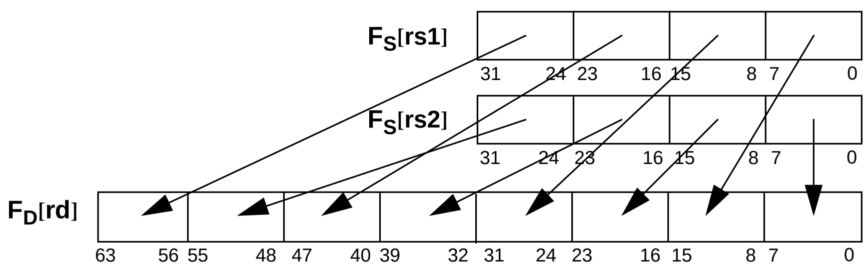


FIGURE 7-24 FPMERGE Operation

```
                           %d0    R1 G1 B1 A1 R2 G2 B2 A2
                           %d2    R3 G3 B3 A3 R4 G4 B4 A4 } packed representation
fpmerge %f0, %f2, %d4  !r1 R3 G1 G3 B1 B3 A1 A3
fpmerge %f1, %f3, %d6  !r2 R4 G2 G4 B2 B4 A2 A4 } intermediate
fpmerge %f4, %f6, %d0  !r1 R2 R3 R4 G1 G2 G3 G4
fpmerge %f5, %f7, %d2  !B1 B2 B3 B4 A1 A2 A3 A4 } planar representation
fpmerge %f0, %f2, %d4  !r1 B1 R2 B2 R3 B3 R4 B4
fpmerge %f1, %f3, %d6  !G1 A1 G2 A2 G3 A3 G4 A4 } intermediate
fpmerge %f4, %f6, %d0  !R1 G1 B1 A1 R2 G2 B2 A2
fpmerge %f5, %f7, %d2  !R3 G3 B3 A3 R4 G4 B4 A4 } packed representation
```

CODE EXAMPLE 7-3 FPMERGE

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

Exceptions *fp_disabled*

See Also FPACK on page 153
FEXPAND on page 131

7.37 Fixed-point Partitioned Subtract (64-bit) VIS 1

Instruction	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FPSUB16	0 0101 0100	Four 16-bit subtracts	f64	f64	f64	`fpsub16` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FPSUB16S	0 0101 0101	Two 16-bit subtracts	f32	f32	f32	`fpsub16s` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FPSUB32	0 0101 0110	Two 32-bit subtracts	f64	f64	f64	`fpsub32` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FPSUB32S	0 0101 0111	One 32-bit subtract	f32	f32	f32	`fpsub32s` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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

The 32-bit versions of these instructions (FPSUB16S and FPSUB32S) perform two 16-bit or one 32-bit partitioned subtractions.

Any carry out from each subtraction is discarded and a 2's-complement arithmetic result is produced.

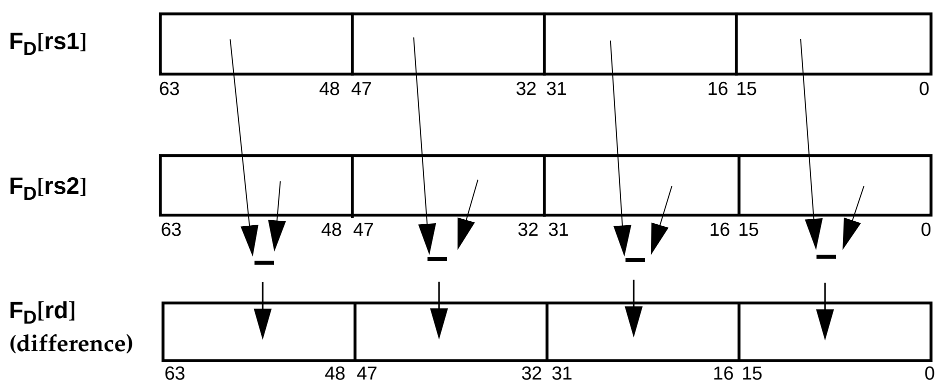


FIGURE 7-25 FPSUB16 Operation

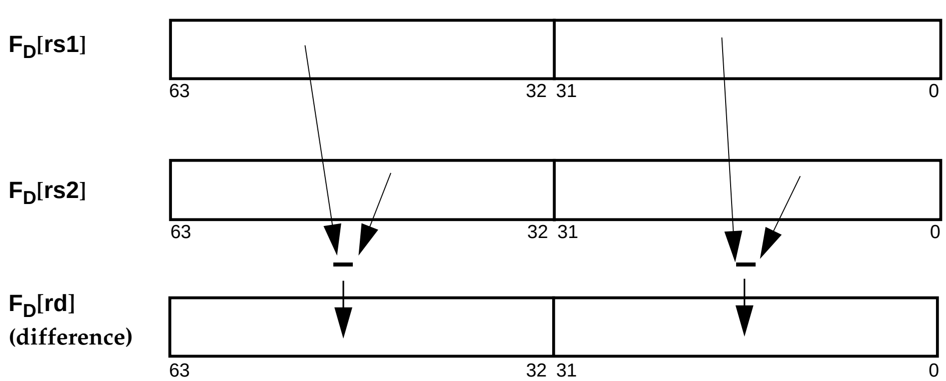


FIGURE 7-26 FPSUB32 Operation

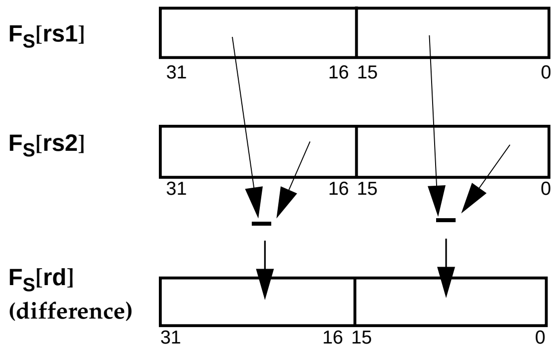


FIGURE 7-27 FPSUB16S Operation

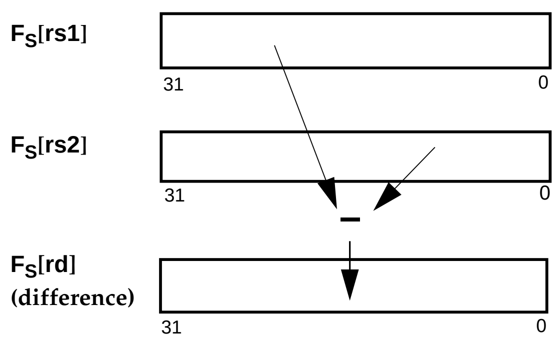


FIGURE 7-28 FPSUB32S Operation

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

Exceptions *fp_disabled*

7.38 F Register Logical Operate (1 operand) VIS 1

Instruction	opf	Operation	Assembly Language Syntax		Class
FZEROd	0 0110 0000	Zero fill	`fzero`	$freg_{rd}$	**A1**
FZEROs	0 0110 0001	Zero fill, 32-bit	`fzeros`	$freg_{rd}$	**A1**
FONEd	0 0111 1110	One fill	`fone`	$freg_{rd}$	**A1**
FONEs	0 0111 1111	One fill, 32-bit	`fones`	$freg_{rd}$	**A1**

10	rd	110110	—	opf	—
31 30	29 25	24 19	18 14	13 5	4 0

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

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

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

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

Exceptions *illegal_instruction*
fp_disabled

See Also F Register 2-operand Logical Operations on page 164
F Register 3-operand Logical Operations on page 165

7.39 F Register Logical Operate (2 operand) VIS 1

Instruction	opf	Operation	Assembly Language Syntax		Class
FSRC1d	0 0111 0100	Copy F_D[rs1] to F_D[rd]	`fsrc1`	*freg*$_{rs1}$, *freg*$_{rd}$	**A1**
FSRC1s	0 0111 0101	Copy F_S[rs1] to F_S[rd], 32-bit	`fsrc1s`	*freg*$_{rs1}$, *freg*$_{rd}$	**A1**
FSRC2d	0 0111 1000	Copy F_D[rs2] to F_D[rd]	`fsrc2`	*freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FSRC2s	0 0111 1001	Copy F_S[rs2] to F_S[rd], 32-bit	`fsrc2s`	*freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FNOT1d	0 0110 1010	Negate (1's complement) F_D[rs1]	`fnot1`	*freg*$_{rs1}$, *freg*$_{rd}$	**A1**
FNOT1s	0 0110 1011	Negate (1's complement) F_S[rs1], 32-bit	`fnot1s`	*freg*$_{rs1}$, *freg*$_{rd}$	**A1**
FNOT2d	0 0110 0110	Negate (1's complement) F_D[rs2]	`fnot2`	*freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FNOT2s	0 0110 0111	Negate (1's complement) F_S[rs2], 32-bit	`fnot2s`	*freg*$_{rs2}$, *freg*$_{rd}$	**A1**

31 30	29 25	24 19	18 14	13 5	4 0
10	rd	110110	rs1	opf	—
10	rd	110110	—	opf	rs2

Description The standard 64-bit versions of these instructions perform one of four 64-bit logical operations on the data from the 64-bit floating-point source register F_D[rs1] (or F_D[rs2]) and store the result in the 64-bit floating-point destination register F_D[rd].

The 32-bit (single-precision) versions of these instructions perform 32-bit logical operations on F_S[rs1] (or F_S[rs2]) and store the result in F_S[rd].

An attempt to execute an FSRC1(s) or FNOT1(s) instruction when instruction bits 4:0 are nonzero causes an *illegal_instruction* exception. An attempt to execute an FSRC2(s) or FNOT2(s) instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an FSRC1[s], FNOT1[s], FSRC1[s], or FNOT1[s] instruction causes an *fp_disabled* exception.

Programming Note FSRC1s (FSRC1) functions similarly to FMOVs (FMOVd), except that FSRC1s (FSRC1) does not modify the FSR register while FMOVs (FMOVd) update some fields of FSR (see *Floating-Point Move* on page 139). Programmers are encouraged to use FMOVs (FMOVd) instead of FSRC1s (FSRC1) whenever practical.

Exceptions *illegal_instruction*
fp_disabled

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

7.40 F Register Logical Operate (3 operand) VIS 1

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

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description The standard 64-bit versions of these instructions perform one of ten 64-bit logical operations between the 64-bit floating-point registers F_D[rs1] and F_D[rs2]. The result is stored in the 64-bit floating-point destination register F_D[rd].

The 32-bit (single-precision) versions of these instructions perform 32-bit logical operations between F_S[rs1] and F_S[rs2], storing the result in F_S[rd].

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

Exceptions *fp_disabled*

See Also F Register 1-operand Logical Operations on page 163
F Register 2-operand Logical Operations on page 164

7.41 Floating-Point Square Root

Instruction	op3	opf	Operation	Assembly Language Syntax	Class
FSQRTs	11 0100	0 0010 1001	Square Root Single	`fsqrts` $freg_{rs2}$, $freg_{rd}$	**A1**
FSQRTd	11 0100	0 0010 1010	Square Root Double	`fsqrtd` $freg_{rs2}$, $freg_{rd}$	**A1**
FSQRTq	11 0100	0 0010 1011	Square Root Quad	`fsqrtq` $freg_{rs2}$, $freg_{rd}$	**C3**

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description These SPARC V9 instructions generate the square root of the floating-point operand in the floating-point register(s) specified by the rs2 field and place the result in the destination floating-point register(s) specified by the rd field. Rounding is performed as specified by FSR.rd.

Note UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute an FSQRTq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

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

An attempt to execute an FSQRTq instruction when rs2{1} ≠ 0 or rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

An *fp_exception_other* (with FSR.ftt = unfinished_FPop) can occur if the operand to the square root is positive and subnormal. See *FSR_floating-point_trap_type (ftt)* on page 55 for additional details.

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

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FSQRTq only))
fp_exception_other (FSR.ftt = unfinished_FPop)
fp_exception_ieee_754 (IEEE_754_exception (NV, NX))

7.42 Convert Floating-Point to Integer

Instruction	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FsTOx	0 1000 0001	Convert Single to 64-bit Integer	—	f32	f64	`fstox` $freg_{rs2}$, $freg_{rd}$	**A1**
FdTOx	0 1000 0010	Convert Double to 64-bit Integer	—	f64	f64	`fdtox` $freg_{rs2}$, $freg_{rd}$	**A1**
FqTOx	0 1000 0011	Convert Quad to 64-bit Integer	—	f128	f64	`fqtox` $freg_{rs2}$, $freg_{rd}$	**C3**
FsTOi	0 1101 0001	Convert Single to 32-bit Integer	—	f32	f32	`fstoi` $freg_{rs2}$, $freg_{rd}$	**A1**
FdTOi	0 1101 0010	Convert Double to 32-bit Integer	—	f64	f32	`fdtoi` $freg_{rs2}$, $freg_{rd}$	**A1**
FqTOi	0 1101 0011	Convert Quad to 32-bit Integer	—	f128	f32	`fqtoi` $freg_{rs2}$, $freg_{rd}$	**C3**

10	rd	op3 = 11 0100	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description FsTOx, FdTOx, and FqTOx convert the floating-point operand in the floating-point register(s) specified by rs2 to a 64-bit integer in the floating-point register F_D[rd].

FsTOi, FdTOi, and FqTOi convert the floating-point operand in the floating-point register(s) specified by rs2 to a 32-bit integer in the floating-point register F_S[rd].

The result is always rounded toward zero; that is, the rounding direction (rd) field of the FSR register is ignored.

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FqTOx or FqTOi instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an F<s|d|q>TO<i|x> instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an F<s|d|q>TO<i|x> instruction causes an *fp_disabled* exception.

An attempt to execute an FqTOi or FqTOx instruction when rs2{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

If the floating-point operand's value is too large to be converted to an integer of the specified size or is a NaN or infinity, then an *fp_exception_ieee_754* "invalid" exception occurs. The value written into the floating-point register(s) specified by rd in these cases is as defined in *Integer Overflow Definition* on page 293.

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

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FqTOx and FqTOi only))
fp_exception_other (FSR.ftt = unfinished_FPop)
fp_exception_ieee_754 (NV, NX)

7.43 Convert Between Floating-Point Formats

Instruction	op3	opf	Operation	s1	s2	d	Assembly Language Syntax		Class
FsTOd	11 0100	0 1100 1001	Convert Single to Double	—	f32	f64	`fstod`	$freg_{rs2}$, $freg_{rd}$	A1
FsTOq	11 0100	0 1100 1101	Convert Single to Quad	—	f32	f128	`fstoq`	$freg_{rs2}$, $freg_{rd}$	C3
FdTOs	11 0100	0 1100 0110	Convert Double to Single	—	f64	f32	`fdtos`	$freg_{rs2}$, $freg_{rd}$	A1
FdTOq	11 0100	0 1100 1110	Convert Double to Quad	—	f64	f128	`fdtoq`	$freg_{rs2}$, $freg_{rd}$	C3
FqTOs	11 0100	0 1100 0111	Convert Quad to Single	—	f128	f32	`fqtos`	$freg_{rs2}$, $freg_{rd}$	C3
FqTOd	11 0100	0 1100 1011	Convert Quad to Double	—	f128	f64	`fqtod`	$freg_{rs2}$, $freg_{rd}$	C3

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description These instructions convert the floating-point operand in the floating-point register(s) specified by rs2 to a floating-point number in the destination format. They write the result into the floating-point register(s) specified by rd.

The value of FSR.rd determines how rounding is performed by these instructions.

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FsTOq, FdTOq, FqTOs, or FqTOd instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an F<s|d|q>TO<s|d|q> instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if no FPU is present, an attempt to execute an F<s|d|q>TO<s|d|q> instruction causes an *fp_disabled* exception.

An attempt to execute an FsTOq or FdTOq instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception. An attempt to execute an FqTOs orFqTOd instruction when rs2{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

FqTOd, FqTOs, and FdTOs (the "narrowing" conversion instructions) can cause *fp_exception_ieee_754* OF, UF, and NX exceptions. FdTOq, FsTOq, and FsTOd (the "widening" conversion instructions) cannot.

Any of these six instructions can trigger an *fp_exception_ieee_754* NV exception if the source operand is a signalling NaN.

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

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

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FsTOq, FqTOs, FdTOq, and FqTOd only))
fp_exception_other (FSR.ftt = unfinished_FPop)

fp_exception_ieee_754 (NV)
fp_exception_ieee_754 (OF, UF, NX (FqTOd, FqTOs, and FdTOs))

7.44 Floating-Point Subtract

Instruction	op3	opf	Operation	Assembly Language Syntax	Class
FSUBs	11 0100	0 0100 0101	Subtract Single	`fsubs` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FSUBd	11 0100	0 0100 0110	Subtract Double	`fsubd` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FSUBq	11 0100	0 0100 0111	Subtract Quad	`fsubq` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**C3**

10	rd	op3	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description The floating-point subtract instructions subtract the floating-point register(s) specified by the rs2 field from the floating-point register(s) specified by the rs1 field. The instructions then write the difference into the floating-point register(s) specified by the rd field.

Rounding is performed as specified by FSR.rd.

> **Note** UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FSUBq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

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

An attempt to execute an FSUBq instruction when (rs1{1} ≠ 0) or (rs2{1} ≠ 0) or (rd{1:0} ≠ 0) causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

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

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

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FSUBq only))
fp_exception_other (FSR.ftt = unfinished_FPop)
fp_exception_ieee_754 (OF, UF, NX, NV)

See Also FMAf on page 137

7.45 Convert 64-bit Integer to Floating Point

Instruction	op3	opf	Operation	s1	s2	d	Assembly Language Syntax	Class
FxTOs	11 0100	0 1000 0100	Convert 64-bit Integer to Single	—	i64	f32	`fxtos` *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FxTOd	11 0100	0 1000 1000	Convert 64-bit Integer to Double	—	i64	f64	`fxtod` *freg*$_{rs2}$, *freg*$_{rd}$	**A1**
FxTOq	11 0100	0 1000 1100	Convert 64-bit Integer to Quad	—	i64	f128	`fxtoq` *freg*$_{rs2}$, *freg*$_{rd}$	**C3**

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description FxTOs, FxTOd, and FxTOq convert the 64-bit signed integer operand in the floating-point register F_D[rs2] into a floating-point number in the destination format.

All write their result into the floating-point register(s) specified by rd.

The value of FSR.rd determines how rounding is performed by FxTOs and FxTOd.

Note UltraSPARC Architecture 2007 processors do not implement in hardware instructions that refer to quad-precision floating-point registers. An attempt to execute a FxTOq instruction causes an *illegal_instruction* exception, allowing privileged software to emulate the instruction.

An attempt to execute an FxTO<s|d|q> instruction when instruction bits 18:14 are nonzero causes an *illegal_instruction* exception.

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

An attempt to execute an FxTOq instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

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

Exceptions *illegal_instruction*
fp_disabled
fp_exception_other (FSR.ftt = invalid_fp_register (FxTOq))
fp_exception_ieee_754 (NX (FxTOs and FxTOd only))

7.46 Illegal Instruction Trap

Instruction	op	op2	Operation	Assembly Language Syntax	Class
ILLTRAP	00	000	*illegal_instruction* trap	`illtrap` *const22*	**A1**

00	—	000	const22
31 30	29 25	24 22	21 0

Description The ILLTRAP instruction causes an *illegal_instruction* exception. The const22 value in the instruction is ignored by the virtual processor; specifically, this field is *not* reserved by the architecture for any future use.

V9 Compatibility Note | Except for its name, this instruction is identical to the SPARC V8 UNIMP instruction.

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

Exceptions *illegal_instruction*

7.47 Mark Register Window Sets as "Invalid"

Instruction	Operation	Assembly Language Syntax	Class
INVALW[P]	Mark all register window sets as "invalid"	`invalw`	**A1**

31 30	29 25	24 19	18 0
10	fcn = 0 0101	11 0001	—

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

CANSAVE ← (*N_REG_WINDOWS* – 2)
CANRESTORE ← 0
OTHERWIN ← 0

Programming Notes INVALW marks all windows as invalid; after executing INVALW, *N_REG_WINDOWS*-2 SAVEs can be performed without generating a spill trap.

An attempt to execute an INVALW instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute an INVALW instruction in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_opcode* exception.

Exceptions *illegal_instruction*
privileged_opcode

See Also ALLCLEAN on page 99
NORMALW on page 213
OTHERW on page 215
RESTORED on page 232
SAVED on page 239

7.48 Jump and Link

Instruction	op3	Operation	Assembly Language Syntax	Class
JMPL	11 1000	Jump and Link	`jmpl` *address*, *reg*$_{rd}$	**A1**

10	rd	op3	rs1	i=0	—	rs2

10	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 0

Description The JMPL instruction causes a register-indirect delayed control transfer to the address given by "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

The JMPL instruction copies the PC, which contains the address of the JMPL instruction, into register R[rd].

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

If either of the low-order two bits of the jump address is nonzero, a *mem_address_not_aligned* exception occurs.

Programming Notes A JMPL instruction with rd = 15 functions as a register-indirect call using the standard link register.

JMPL with rd = 0 can be used to return from a subroutine. The typical return address is "r[31] + 8" if a nonleaf routine (one that uses the SAVE instruction) is entered by a CALL instruction, or "R[15] + 8" if a leaf routine (one that does not use the SAVE instruction) is entered by a CALL instruction or by a JMPL instruction with rd = 15.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then JMPL generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the JMPL instruction) is stored in TPC[TL] and the value of NPC from before the JMPL was executed is stored in TNPC[TL].

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system or being written into R[rd] (or, if a *control_transfer_instruction* trap occurs, into TPC[TL]). (closed impl. dep. #125-V9-Cs10)

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

See Also CALL on page 111
Bicc on page 104
BPcc on page 109

7.49 Load Integer

Instruction	op3	Operation	Assembly Language Syntax		Class
LDSB	00 1001	Load Signed Byte	`ldsb`	[*address*], *reg*$_{rd}$	**A1**
LDSH	00 1010	Load Signed Halfword	`ldsh`	[*address*], *reg*$_{rd}$	**A1**
LDSW	00 1000	Load Signed Word	`ldsw`	[*address*], *reg*$_{rd}$	**A1**
LDUB	00 0001	Load Unsigned Byte	`ldub`	[*address*], *reg*$_{rd}$	**A1**
LDUH	00 0010	Load Unsigned Halfword	`lduh`	[*address*], *reg*$_{rd}$	**A1**
LDUW	00 0000	Load Unsigned Word	`lduw`†	[*address*], *reg*$_{rd}$	**A1**
LDX	00 1011	Load Extended Word	`ldx`	[*address*], *reg*$_{rd}$	**A1**

† *synonym:* `ld`

31–30	29–25	24–19	18–14	13	12–5	4–0
11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	

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 76). The effective address is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

A successful load (notably, load extended) instruction operates atomically.

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

If the effective address is not halfword-aligned, an attempt to execute an LDUH or LDSH causes a *mem_address_not_aligned* exception. If the effective address is not word-aligned, an attempt to execute an LDUW or LDSW instruction causes a *mem_address_not_aligned* exception. If the effective address is not doubleword-aligned, an attempt to execute an LDX instruction causes a *mem_address_not_aligned* exception.

V8 Compatibility Note The SPARC V8 LD instruction was renamed LDUW in the SPARC V9 architecture. The LDSW instruction was new in the SPARC V9 architecture.

A load integer twin word (LDTW) instruction exists, but is deprecated; see *Load Integer Twin Word* on page 192 for details.

Exceptions *illegal_instruction*
mem_address_not_aligned (all except LDSB, LDUB)
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

7.50 Load Integer from Alternate Space

Instruction	op3	Operation	Assembly Language Syntax	Class
LDSBA$^{P_{ASI}}$	01 1001	Load Signed Byte from Alternate Space	`ldsba` [*regaddr*] *imm_asi*, reg_{rd} `ldsba` [*reg_plus_imm*] `%asi`, reg_{rd}	**A1**
LDSHA$^{P_{ASI}}$	01 1010	Load Signed Halfword from Alternate Space	`ldsha` [*regaddr*] *imm_asi*, reg_{rd} `ldsha` [*reg_plus_imm*] `%asi`, reg_{rd}	**A1**
LDSWA$^{P_{ASI}}$	01 1000	Load Signed Word from Alternate Space	`ldswa` [*regaddr*] *imm_asi*, reg_{rd} `ldswa` [*reg_plus_imm*] `%asi`, reg_{rd}	**A1**
LDUBA$^{P_{ASI}}$	01 0001	Load Unsigned Byte from Alternate Space	`lduba` [*regaddr*] *imm_asi*, reg_{rd} `lduba` [*reg_plus_imm*] `%asi`, reg_{rd}	**A1**
LDUHA$^{P_{ASI}}$	01 0010	Load Unsigned Halfword from Alternate Space	`lduha` [*regaddr*] *imm_asi*, reg_{rd} `lduha` [*reg_plus_imm*] `%asi`, reg_{rd}	**A1**
LDUWA$^{P_{ASI}}$	01 0000	Load Unsigned Word from Alternate Space	`lduwa`† [*regaddr*] *imm_asi*, reg_{rd} `lduwa` [*reg_plus_imm*] `%asi`, reg_{rd}	**A1**
LDXA$^{P_{ASI}}$	01 1011	Load Extended Word from Alternate Space	`ldxa` [*regaddr*] *imm_asi*, reg_{rd} `ldxa` [*reg_plus_imm*] `%asi`, reg_{rd}	**A1**

† *synonym:* `lda`

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	op3	rs1	i=0	imm_asi	rs2
11	rd	op3	rs1	i=1	simm13	

Description The load integer from alternate space instructions copy a byte, a halfword, a word, or an extended word from memory. All copy the fetched value into R[rd]. A fetched byte, halfword, or word is right-justified in the destination register R[rd]; it is either sign-extended or zero-filled on the left, depending on whether the opcode specifies a signed or unsigned operation, respectively.

The load integer from alternate space instructions contain the address space identifier (ASI) to be used for the load in the imm_asi field if i = 0, or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

A successful load (notably, load extended) instruction operates atomically.

A load integer twin word from alternate space (LDTWA) instruction exists, but is deprecated; see *Load Integer Twin Word from Alternate Space* on page 194 for details.

If the effective address is not halfword-aligned, an attempt to execute an LDUHA or LDSHA instruction causes a *mem_address_not_aligned* exception. If the effective address is not word-aligned, an attempt to execute an LDUWA or LDSWA instruction causes a *mem_address_not_aligned* exception. If the effective address is not doubleword-aligned, an attempt to execute an LDXA instruction causes a *mem_address_not_aligned* exception.

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

LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

ASIs valid for LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_REAL_IO	ASI_REAL_IO_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE
ASI_PRIMARY_NO_FAULT	ASI_PRIMARY_NO_FAULT_LITTLE
ASI_SECONDARY_NO_FAULT	ASI_SECONDARY_NO_FAULT_LITTLE

LDXA can be used with any ASI (including, but not limited to, the above list), unless it either (a) violates the privilege mode rules described for the *privileged_action* exception above or (b) is used with any of the following ASIs, which causes a *DAE_invalid_asi* exception.

ASIs invalid for LDXA	(cause *DAE_invalid_asi* exception)
22_{16} (ASI_TWINX_AIUP)	$2A_{16}$ (ASI_TWINX_AIUP_L)
23_{16} (ASI_TWINX_AIUS)	$2B_{16}$ (ASI_TWINX_AIUS_L)
26_{16} (ASI_TWINX_REAL)	$2E_{16}$ (ASI_TWINX_REAL_L)
27_{16} (ASI_TWINX_N)	$2F_{16}$ (ASI_TWINX_NL)
ASI_BLOCK_AS_IF_USER_PRIMARY	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE
ASI_BLOCK_AS_IF_USER_SECONDARY	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE
ASI_PST8_PRIMARY	ASI_PST8_PRIMARY_LITTLE
ASI_PST8_SECONDARY	ASI_PST8_SECONDARY_LITTLE
ASI_PST16_PRIMARY	ASI_PST16_PRIMARY_LITTLE
ASI_PST16_SECONDARY	ASI_PST16_SECONDARY_LITTLE
ASI_PST32_PRIMARY	ASI_PST32_PRIMARY_LITTLE
ASI_PST32_SECONDARY	ASI_PST32_SECONDARY_LITTLE
ASI_FL8_PRIMARY	ASI_FL8_PRIMARY_LITTLE
ASI_FL8_SECONDARY	ASI_FL8_SECONDARY_LITTLE
ASI_FL16_PRIMARY	ASI_FL16_PRIMARY_LITTLE
ASI_FL16_SECONDARY	ASI_FL16_SECONDARY_LITTLE
ASI_BLOCK_COMMIT_PRIMARY	ASI_BLOCK_COMMIT_SECONDARY
$E2_{16}$ (ASI_TWINX_P)	EA_{16} (ASI_TWINX_PL)
$E3_{16}$ (ASI_TWINX_S)	EB_{16} (ASI_TWINX_SL)
ASI_BLOCK_PRIMARY	ASI_BLOCK_PRIMARY_LITTLE
ASI_BLOCK_SECONDARY	ASI_BLOCK_SECONDARY_LITTLE

Exceptions

mem_address_not_aligned (all except LDSBA and LDUBA)
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nfo_page
DAE_side_effect_page

See Also

LD on page 175
STA on page 248

7.51 Block Load VIS 1

The LDBLOCKF instructions are deprecated and should not be used in new software. A sequence of LDX instructions should be used instead.

The LDBLOCKF instruction is intended to be a processor-specific instruction, which may or may not be implemented in future UltraSPARC Architecture implementations. Therefore, it should only be used in platform-specific dynamically-linked libraries or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

Instruc-tion	ASI Value	Operation	Assembly Language Syntax	Class
LDBLOCKF^{D}	16_{16}	64-byte block load from primary address space, user privilege	`ldda` [*regaddr*] `#ASI_BLK_AIUP`, $freg_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, $freg_{rd}$	**D2**
LDBLOCKF^{D}	17_{16}	64-byte block load from secondary address space, user privilege	`ldda` [*regaddr*] `#ASI_BLK_AIUS`, $freg_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, $freg_{rd}$	**D2**
LDBLOCKF^{D}	$1E_{16}$	64-byte block load from primary address space, little-endian, user privilege	`ldda` [*regaddr*] `#ASI_BLK_AIUPL`, $freg_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, $freg_{rd}$	**D2**
LDBLOCKF^{D}	$1F_{16}$	64-byte block load from secondary address space, little-endian, user privilege	`ldda` [*regaddr*] `#ASI_BLK_AIUSL`, $freg_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, $freg_{rd}$	**D2**
LDBLOCKF^{D}	$F0_{16}$	64-byte block load from primary address space	`ldda` [*regaddr*] `#ASI_BLK_P`, $freg_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, $freg_{rd}$	**D2**
LDBLOCKF^{D}	$F1_{16}$	64-byte block load from secondary address space	`ldda` [*regaddr*] `#ASI_BLK_S`, $freg_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, $freg_{rd}$	**D2**
LDBLOCKF^{D}	$F8_{16}$	64-byte block load from primary address space, little-endian	`ldda` [*regaddr*] `#ASI_BLK_PL`, $freg_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, $freg_{rd}$	**D2**
LDBLOCKF^{D}	$F9_{16}$	64-byte block load from secondary address space, little-endian	`ldda` [*regaddr*] `#ASI_BLK_SL`, $freg_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, $freg_{rd}$	**D2**

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	110011	rs1	i=0	imm_asi	rs2
11	rd	110011	rs1	i=1	simm_13	

Description A block load (LDBLOCKF) instruction uses one of several special block-transfer ASIs. Block transfer ASIs allow block loads to be performed accessing the same address space as normal loads. Little-endian ASIs (those with an 'L' suffix) access data in little-endian format; otherwise, the access is assumed to be big-endian. Byte swapping is performed separately for each of the eight 64-bit (double-precision) F registers used by the instruction.

A block load instruction loads 64 bytes of data from a 64-byte aligned memory area into the eight double-precision floating-point registers specified by rd. The lowest-addressed eight bytes in memory are loaded into the lowest-numbered 64-bit (double-precision) destination F register.

A block load only guarantees atomicity for each 64-bit (8-byte) portion of the 64 bytes it accesses.

The block load instruction is intended to support fast block-copy operations.

LDBLOCKF

Programming Note | LDBLOCKF is intended to be a processor-specific instruction (see the warning at the top of page 178). If LDBLOCKF *must* be used in software intended to be portable across current and previous processor implementations, then it must be coded to work in the face of any implementation variation that is permitted by implementation dependency #410-S10, described below.

IMPL. DEP. #410-S10: The following aspects of the behavior of block load (LDBLOCKF) instructions are implementation dependent:

- What memory ordering model is used by LDBLOCKF (LDBLOCKF is not required to follow TSO memory ordering)
- Whether LDBLOCKF follows memory ordering with respect to stores (including block stores), including whether the virtual processor detects read-after-write and write-after-read hazards to overlapping addresses
- Whether LDBLOCKF appears to execute out of order, or follow LoadLoad ordering (with respect to older loads, younger loads, and other LDBLOCKFs)
- Whether LDBLOCKF follows register-dependency interlocks, as do ordinary load instructions
- Whether *VA_watchpoint* exceptions are recognized on accesses to all 64 bytes of a LDBLOCKF (the recommended behavior), or only on the first eight bytes
- Whether the MMU ignores the side-effect bit (TTE.e) for LDBLOCKF accesses

Programming Note | If ordering with respect to earlier stores is important (for example, a block load that overlaps a previous store) and read-after-write hazards are not detected, there must be a MEMBAR `#StoreLoad` instruction between earlier stores and a block load.

If ordering with respect to later stores is important, there must be a MEMBAR `#LoadStore` instruction between a block load and subsequent stores.

If LoadLoad ordering with respect to older or younger loads or other block load instructions is important and is not provided by an implementation, an intervening MEMBAR `#LoadLoad` is required.

For further restrictions on the behavior of the block load instruction, see implementation-specific processor documentation.

Implementation Note | In all UltraSPARC Architecture implementations, the MMU ignores the side-effect bit (TTE.e) for LDBLOCKF accesses (impl. dep. #410-S10).

Exceptions. An *illegal_instruction* exception occurs if LDBLOCKF's floating-point destination registers are not aligned on an eight-double-precision register boundary.

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

If the least significant 6 bits of the effective memory address in an LDBLOCKF instruction are nonzero, a *mem_address_not_aligned* exception occurs.

In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0 (ASIs 16_{16}, 17_{16}, $1E_{16}$, and $1F_{16}$), LDBLOCKF causes a *privileged_action* exception.

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

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

LDBLOCKF

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

Exceptions

illegal_instruction
fp_disabled
mem_address_not_aligned
privileged_action
VA_watchpoint (impl. dep. #410-S10)
DAE_privilege_violation
DAE_nc_page
DAE_nfo_page (attempted access to Non-Faulting-Only page of memory)

See Also

STBLOCKF on page 250

7.52 Load Floating-Point Register

Instruction	op3	rd	Operation	Assembly Language Syntax		Class
LDF	10 0000	0–31	Load Floating-Point Register	`ld`	[*address*], $freg_{rd}$	**A1**
LDDF	10 0011	‡	Load Double Floating-Point Register	`ldd`	[*address*], $freg_{rd}$	**A1**
LDQF	10 0010	‡	Load Quad Floating-Point Register	`ldq`	[*address*], $freg_{rd}$	**C3**

‡ Encoded floating-point register value, as described on page 51.

31 30	29 … 25	24 … 19	18 … 14	13	12 … 5	4 … 0
11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	

Description The load single floating-point instruction (LDF) copies a word from memory into 32-bit floating-point destination register F_S[rd].

The load doubleword floating-point instruction (LDDF) copies a word-aligned doubleword from memory into a 64-bit floating-point destination register, F_D[rd]. The unit of atomicity for LDDF is 4 bytes (one word).

The load quad floating-point instruction (LDQF) copies a word-aligned quadword from memory into a 128-bit floating-point destination register, F_Q[rd]. The unit of atomicity for LDQF is 4 bytes (one word).

These load floating-point instructions access memory using the implicit ASI (see page 76).

If i = 0, the effective address for these instructions is "R[rs1] + R[rs2]" and if i = 0, the effective address is "R[rs1] + **sign_ext**(simm13)".

Exceptions. An attempt to execute an LDF, LDDF, or LDQF instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

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

If the effective address is not word-aligned, an attempt to execute an LDF instruction causes a *mem_address_not_aligned* exception.

LDDF requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute an LDDF instruction causes an *LDDF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the LDDF instruction and return (impl. dep. #109-V9-Cs10(a)).

LDQF requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an LDQF instruction causes an *LDQF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the LDQF instruction and return (impl. dep. #111-V9-Cs10(a)).

Programming Note Some compilers issued sequences of single-precision loads for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9 processors, since emulation of misaligned loads is expected to be fast, compilers should issue sets of single-precision loads only when they can determine that doubleword or quadword operands are *not* properly aligned.

An attempt to execute an LDQF instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

Implementation Note Since UltraSPARC Architecture 2007 processors do not implement in hardware instructions (including LDQF) that refer to quad-precision floating-point registers, the *LDQF_mem_address_not_aligned* and *fp_exception_other* (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an *illegal_instruction* exception and subsequent trap.

Destination Register(s) when Exception Occurs. If a load floating-point instruction generates an exception that causes a *precise* trap, the destination floating-point register(s) remain unchanged.

IMPL. DEP. #44-V8-Cs10(a)(1): If a load floating-point instruction generates an exception that causes a *non-precise* trap, the contents of the destination floating-point register(s) remain unchanged or are undefined.

Exceptions

illegal_instruction
fp_disabled
LDDF_mem_address_not_aligned
LDQF_mem_address_not_aligned (not used in UltraSPARC Architecture 2007)
mem_address_not_aligned
fp_exception_other (FSR.ftt = invalid_fp_register (LDQF only))
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

See Also

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

7.53 Load Floating-Point from Alternate Space

Instruction	op3	rd	Operation	Assembly Language Syntax	Class
LDFA$^{P_{ASI}}$	11 0000	0–31	Load Floating-Point Register from Alternate Space	`lda` [*regaddr*] *imm_asi*, *freg*$_{rd}$ `lda` [*reg_plus_imm*] `%asi`, *freg*$_{rd}$	**A1**
LDDFA$^{P_{ASI}}$	11 0011	‡	Load Double Floating-Point Register from Alternate Space	`ldda` [*regaddr*] *imm_asi*, *freg*$_{rd}$ `ldda` [*reg_plus_imm*] `%asi`, *freg*$_{rd}$	**A1**
LDQFA$^{P_{ASI}}$	11 0010	‡	Load Quad Floating-Point Register from Alternate Space	`ldqa` [*regaddr*] *imm_asi*, *freg*$_{rd}$ `ldqa` [*reg_plus_imm*] `%asi`, *freg*$_{rd}$	**C3**

‡ Encoded floating-point register value, as described in *Floating-Point Register Number Encoding* on page 51.

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	op3	rs1	i=0	imm_asi	rs2
11	rd	op3	rs1	i=1	simm13	

Description The load single floating-point from alternate space instruction (LDFA) copies a word from memory into 32-bit floating-point destination register F_S[rd].

The load double floating-point from alternate space instruction (LDDFA) copies a word-aligned doubleword from memory into a 64-bit floating-point destination register, F_D[rd]. The unit of atomicity for LDDFA is 4 bytes (one word).

The load quad floating-point from alternate space instruction (LDQFA) copies a word-aligned quadword from memory into a 128-bit floating-point destination register, F_Q[rd]. The unit of atomicity for LDQFA is 4 bytes (one word).

If i = 0, these instructions contain the address space identifier (ASI) to be used for the load in the imm_asi field and the effective address for the instruction is "R[rs1] + R[rs2]". If i = 1, the ASI to be used is contained in the ASI register and the effective address for the instruction is "R[rs1] + **sign_ext**(simm13)".

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

LDFA causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

V9 Compatibility Note | LDFA, LDDFA, and LDQFA cause a *privileged_action* exception if PSTATE.priv = 0 and bit 7 of the ASI is 0.

LDDFA requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, LDDFA causes an *LDDF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the LDDFA instruction and return (impl. dep. #109-V9-Cs10(b)).

LDQFA requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, LDQFA causes an *LDQF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the LDQFA instruction and return (impl. dep. #111-V9-Cs10(b)).

An attempt to execute an LDQFA instruction when rd{1} ≠ 0 causes an *fp_exception_other* (with FSR.ftt = invalid_fp_register) exception.

Implementation Note | Since UltraSPARC Architecture 2007 processors do not implement in hardware instructions (including LDQFA) that refer to quad-precision floating-point registers, the *LDQF_mem_address_not_aligned* and *fp_exception_other* (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an *illegal_instruction* exception and subsequent trap.

Programming Note | Some compilers issued sequences of single-precision loads for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9 processors, since emulation of misaligned loads is expected to be fast, compilers should issue sets of single-precision loads only when they can determine that doubleword or quadword operands are *not* properly aligned.

In nonprivileged mode (PSTATE.priv = 0), 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.

ASIs valid for LDFA and LDQFA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_REAL_IO	ASI_REAL_IO_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE
ASI_PRIMARY_NO_FAULT	ASI_PRIMARY_NO_FAULT_LITTLE
ASI_SECONDARY_NO_FAULT	ASI_SECONDARY_NO_FAULT_LITTLE

LDDFA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with the LDDFA instruction causes a *DAE_invalid_asi* exception.

ASIs valid for LDDFA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_REAL_IO	ASI_REAL_IO_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE
ASI_PRIMARY_NO_FAULT	ASI_PRIMARY_NO_FAULT_LITTLE
ASI_SECONDARY_NO_FAULT	ASI_SECONDARY_NO_FAULT_LITTLE

Behavior with Block-Store-with-Commit ASIs. ASIs $E0_{16}$ and $E1_{16}$ are only defined for use in Block Store with Commit operations (see page 250). Neither ASI $E0_{16}$ nor $E1_{16}$ should be used with LDDFA; however, if it *is* used, the LDDFA behaves as follows:

1. If an LDDFA opcode is used with an ASI of $E0_{16}$ or $E1_{16}$ and a destination register number rd is specified which is not a multiple of 8 ("misaligned" rd), an UltraSPARC Architecture 2007 virtual processor generates an *illegal_instruction* exception (impl. dep. #255-U3-Cs10).

2. **IMPL. DEP. #256-U3:** If an LDDFA opcode is used with an ASI of $E0_{16}$ or $E1_{16}$ and a memory address is specified with less than 64-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the exception generated is *DAE_invalid_asi*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned*.

3. If both rd and the memory address are correctly aligned, a *DAE_invalid_asi* exception occurs.

Behavior with Partial Store ASIs. ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–CD_{16} are only defined for use in Partial Store operations (see page 260). None of them should be used with LDDFA; however, if any of those ASIs *is* used with LDDFA, the LDDFA behaves as follows:

1. **IMPL. DEP. #257-U3:** If an LDDFA opcode is used with an ASI of $C0_{16}$–$C5_{16}$ or $C8_{16}$–CD_{16} (Partial Store ASIs, which are an illegal combination with LDDFA) and a memory address is specified with less than 8-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the generated exception is a *DAE_invalid_asi*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned* exception.

2. If the memory address is correctly aligned, the virtual processor generates a *DAE_invalid_asi*.

Destination Register(s) when Exception Occurs. If a load floating-point alternate instruction generates an exception that causes a precise trap, the destination floating-point register(s) remain unchanged.

IMPL. DEP. #44-V8-Cs10(b): If a load floating-point alternate instruction generates an exception that causes a non-precise trap, it is implementation dependent whether the contents of the destination floating-point register(s) are undefined or are guaranteed to remain unchanged.

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

Exceptions

illegal_instruction
fp_disabled
LDDF_mem_address_not_aligned
LDQF_mem_address_not_aligned (not generated in UltraSPARC Architecture 2007)
mem_address_not_aligned
fp_exception_other (FSR.ftt = invalid_fp_register (LDQFA only))
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nfo_page
DAE_side_effect_page

See Also

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

7.54 Load Floating-Point State Register (Lower)

> The LDFSR instruction is deprecated and should not be used in new software. The LDXFSR instruction should be used instead.

Opcode	op3	rd	Operation	Assembly Language Syntax	Class
LDFSR[D]	10 0001	0	Load Floating-Point State Register (Lower)	`ld [`*address*`], %fsr`	**D2**
	10 0001	1-31	(see page 199)		

31–30	29–25	24–19	18–14	13	12–5	4–0
11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	

Description The Load Floating-point State Register (Lower) instruction (LDFSR) waits for all FPop instructions that have not finished execution to complete and then loads a word from memory into the less significant 32 bits of the FSR. The more-significant 32 bits of FSR are unaffected by LDFSR. LDFSR does not alter the ver, ftt, qne, reserved, or unimplemented (for example, ns) fields of FSR (see page 42).

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 76).

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

If the FPU is not enabled (FPRS.fef = 0 or PSTATE.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 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 181
Load Floating-Point State Register on page 199
Store Floating-Point on page 253

7.55 Short Floating-Point Load VIS 1

Instruction	ASI Value	Operation	Assembly Language Syntax	Class
LDSHORTF	$D0_{16}$	8-bit load from primary address space	`ldda` [*regaddr*] `#ASI_FL8_P`, freg_{rd} `ldda` [*reg_plus_imm*] `%asi`, freg_{rd}	**B1**
LDSHORTF	$D1_{16}$	8-bit load from secondary address space	`ldda` [*regaddr*] `#ASI_FL8_S`, freg_{rd} `ldda` [*reg_plus_imm*] `%asi`, freg_{rd}	**B1**
LDSHORTF	$D8_{16}$	8-bit load from primary address space, little-endian	`ldda` [*regaddr*] `#ASI_FL8_PL`, freg_{rd} `ldda` [*reg_plus_imm*] `%asi`, freg_{rd}	**B1**
LDSHORTF	$D9_{16}$	8-bit load from secondary address space, little-endian	`ldda` [*regaddr*] `#ASI_FL8_SL`, freg_{rd} `ldda` [*reg_plus_imm*] `%asi`, freg_{rd}	**B1**
LDSHORTF	$D2_{16}$	16-bit load from primary address space	`ldda` [*regaddr*] `#ASI_FL16_P`, freg_{rd} `ldda` [*reg_plus_imm*] `%asi`, freg_{rd}	**B1**
LDSHORTF	$D3_{16}$	16-bit load from secondary address space	`ldda` [*regaddr*] `#ASI_FL16_S`, freg_{rd} `ldda` [*reg_plus_imm*] `%asi`, freg_{rd}	**B1**
LDSHORTF	DA_{16}	16-bit load from primary address space, little-endian	`ldda` [*regaddr*] `#ASI_FL16_PL`, freg_{rd} `ldda` [*reg_plus_imm*] `%asi`, freg_{rd}	**B1**
LDSHORTF	DB_{16}	16-bit load from secondary address space, little-endian	`ldda` [*regaddr*] `#ASI_FL16_SL`, freg_{rd} `ldda` [*reg_plus_imm*] `%asi`, freg_{rd}	**B1**

11	rd	110011	rs1	i=0	imm_asi	rs2

11	rd	110011	rs1	i=1	simm_13
31 30	29 25	24 19	18 14	13	5 4 0

Description Short floating-point load instructions allow an 8- or 16-bit value to be loaded from memory into a 64-bit floating-point register.

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

An 8-bit load places the loaded value in the least significant byte of F_D[rd] and zeroes in the most-significant three bytes of F_D[rd]. An 8-bit LDSHORTF can be performed from an arbitrary byte address.

A 16-bit load places the loaded value in the least significant halfword of F_D[rd] and zeroes in the more-significant halfword of F_D[rd]. A 16-bit LDSHORTF from an address that is not halfword-aligned (an odd address) causes a *mem_address_not_aligned* exception.

Little-endian ASIs transfer data in little-endian format from memory; otherwise, memory is assumed to be in big-endian byte order.

Programming Note LDSHORTF is typically used with the FALIGNDATA instruction (see *Align Address* on page 98) to assemble or store 64 bits from noncontiguous components.

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

Exceptions *fp_disabled*
mem_address_not_aligned
VA_watchpoint

DAE_privilege_violation
DAE_nfo_page

7.56 Load-Store Unsigned Byte

Instruction	op3	Operation	Assembly Language Syntax	Class
LDSTUB	00 1101	Load-Store Unsigned Byte	`ldstub` [*address*], reg_{rd}	**A1**

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	

Description The load-store unsigned byte instruction copies a byte from memory into R[rd], then rewrites the addressed byte in memory to all 1's. The fetched byte is right-justified in the destination register R[rd] and zero-filled on the left.

The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing LDSTUB, LDSTUBA, CASA, CASXA, SWAP, or SWAPA instructions addressing all or parts of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

LDSTUB accesses memory using the implicit ASI (see page 76). The effective address for this instruction is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

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

Exceptions *illegal_instruction*
VA_watchpoint
DAE_privilege_violation
DAE_nc_page
DAE_nfo_page

7.57 Load-Store Unsigned Byte to Alternate Space

Instruction	op3	Operation	Assembly Language Syntax	Class
LDSTUBA$^{\text{P}_{\text{ASI}}}$	01 1101	Load-Store Unsigned Byte into Alternate Space	`ldstuba` [*regaddr*] *imm_asi*, reg_{rd} `ldstuba` [*reg_plus_imm*] `%asi`, reg_{rd}	**A1**

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	op3	rs1	i=0	imm_asi	rs2
11	rd	op3	rs1	i=1	simm13	

Description The load-store unsigned byte into alternate space instruction copies a byte from memory into R[rd], then rewrites the addressed byte in memory to all 1's. The fetched byte is right-justified in the destination register R[rd] and zero-filled on the left.

The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing LDSTUB, LDSTUBA, CASA, CASXA, SWAP, or SWAPA instructions addressing all or parts of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

If i = 0, LDSTUBA contains the address space identifier (ASI) to be used for the load in the imm_asi field. If i = 1, the ASI is found in the ASI register. In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0, this instruction causes a *privileged_action* exception. In privileged mode (PSTATE.priv = 1), if the ASI is in the range 30_{16} to $7F_{16}$, this instruction causes a *privileged_action* exception.

LDSTUBA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *DAE_invalid_asi* exception.

ASIs valid for LDSTUBA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE

Exceptions
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nc_page
DAE_nfo_page

7.58 Load Integer Twin Word

> The LDTW instruction is deprecated and should not be used in new software. It is provided only for compatibility with previous versions of the architecture.The LDX instruction should be used instead.

Instruction	op3	Operation	Assembly Language Syntax †	Class
LDTW[D]	00 0011	Load Integer Twin Word	`ldtw` [*address*], *reg*$_{rd}$	D2

† The original assembly language syntax for this instruction used an "`ldd`" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "`ldtw`" mnemonic for this instruction. In the mean-time, some existing assemblers may only recognize the original "`ldd`" mnemonic.

11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

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 76). If i = 0, the effective address for these instructions is "R[rs1] + R[rs2]" and if i = 0, the effective address is "R[rs1] + **sign_ext**(simm13)".

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.

A successful LDTW instruction operates atomically.

Exceptions *unimplemented_LDTW* (not used in UltraSPARC Architecture 2007)
illegal_instruction
mem_address_not_aligned
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

See Also LDW/LDX on page 175
STTW on page 265

7.59 Load Integer Twin Word from Alternate Space

The LDTWA instruction is deprecated and should not be used in new software. The LDXA instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
LDTWA$^{D, P_{ASI}}$	01 0011	Load Integer Twin Word from Alternate Space	`ldtwa` [*regaddr*] *imm_asi*, *reg$_{rd}$* `ldtwa` [*reg_plus_imm*] `%asi`, *reg$_{rd}$*	**D2, Y3‡**

† The original assembly language syntax for this instruction used an "`ldda`" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "`ldtwa`" mnemonic for this instruction. In the meantime, some assemblers may only recognize the original "`ldda`" mnemonic.

‡ **Y3** for restricted ASIs (00_{16}-$7F_{16}$); **D2** for unrestricted ASIs (80_{16}-FF_{16})

31–30	29–25	24–19	18–14	13	12–5	4–0
11	rd	op3	rs1	i=0	imm_asi	rs2

31–30	29–25	24–19	18–14	13	12–0
11	rd	op3	rs1	i=1	simm13

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 Note LDTWA is provided for compatibility with existing SPARC V8 software. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties.

If LDTWA is emulated in software, an LDXA instruction instruction should be used for the memory access in the emulation code in order to preserve atomicity.

SPARC V9 Compatibility Note LDTWA was (inaccurately) named LDDA in the SPARC V8 and SPARC V9 specifications.

The least significant bit of the rd field in an LDTWA instruction is unused and should always be set to 0 by software. An attempt to execute an LDTWA instruction that refers to a misaligned (odd-numbered) destination register causes an *illegal_instruction* exception.

If the effective address is not doubleword-aligned, an attempt to execute an LDTWA instruction causes a *mem_address_not_aligned* exception.

A successful LDTWA instruction operates atomically.

LDTWA causes a *mem_address_not_aligned* exception if the address is not doubleword-aligned.

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

LDTWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *DAE_invalid_asi* exception (impl. dep. #300-U4-Cs10).

ASIs valid for LDTWA

ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_REAL_IO	ASI_REAL_IO_LITTLE
22_{16}‡ (ASI_TWINX_AIUP)	$2A_{16}$‡ (ASI_TWINX_AIUP_L)
23_{16}‡ (ASI_TWINX_AIUS)	$2B_{16}$‡ (ASI_TWINX_AIUS_L)
26_{16}‡ (ASI_TWINX_REAL)	$2E_{16}$‡ (ASI_TWINX_REAL_L)
27_{16}‡ (ASI_TWINX_N)	$2F_{16}$‡ (ASI_TWINX_NL)
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE
ASI_PRIMARY_NO_FAULT	ASI_PRIMARY_NO_FAULT_LITTLE
ASI_SECONDARY_NO_FAULT	ASI_SECONDARY_NO_FAULT_LITTLE
$E2_{16}$‡ (ASI_TWINX_P)	EA_{16}‡ (ASI_TWINX_PL)
$E3_{16}$‡ (ASI_TWINX_S)	EB_{16}‡ (ASI_TWINX_SL)

‡ If this ASI is used with the opcode for LDTWA and i = 0, the LDTXA instruction is executed instead of LDTWA. For behavior of LDTXA, see *Load Integer Twin Extended Word from Alternate Space* on page 197. 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 321) should only be accessed using LDXA (not LDTWA) instructions. If an LDTWA referencing a nontranslating ASI is executed, per the above table, it generates a *DAE_invalid_asi* exception (impl. dep. #300-U4-Cs10).

Implementation Note The deprecated instruction LDTWA shares an opcode with LDTXA. LDTXA is *not* deprecated and has different address alignment requirements than LDTWA. See *Load Integer Twin Extended Word from Alternate Space* on page 197.

Exceptions

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

DAE_invalid_asi
DAE_privilege_violation
DAE_nfo_page
DAE_side_effect_page

See Also LDWA/LDXA on page 176
LDTXA on page 197
STTWA on page 267

7.60 Load Integer Twin Extended Word from Alternate Space VIS 2+

> The LDTXA instructions are not guaranteed to be implemented on all UltraSPARC Architecture implementations. Therefore, they should only be used in platform-specific dynamically-linked libraries or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

Instruction	ASI Value	Operation	Assembly Language Syntax †	Class
LDTXA[N]	22_{16}	Load Integer Twin Extended Word, as if user (nonprivileged), Primary address space	`ldtxa` [*regaddr*] `#ASI_TWINX_AIUP,` *reg*$_{rd}$	**N**–
	23_{16}	Load Integer Twin Extended Word, as if user (nonprivileged), Secondary address space	`ldtxa` [*regaddr*] `#ASI_TWINX_AIUS,` *reg*$_{rd}$	**N**–
	26_{16}	Load Integer Twin Extended Word, real address	`ldtxa` [*regaddr*] `#ASI_TWINX_REAL,` *reg*$_{rd}$	**N**–
	27_{16}	Load Integer Twin Extended Word, nucleus context	`ldtxa` [*regaddr*] `#ASI_TWINX_N,` *reg*$_{rd}$	**N**–
	$2A_{16}$	Load Integer Twin Extended Word, as if user (nonprivileged), Primary address space, little endian	`ldtxa` [*regaddr*] `#ASI_TWINX_AIUP_L,` *reg*$_{rd}$	**N**–
	$2B_{16}$	Load Integer Twin Extended Word, as if user (nonprivileged), Secondary address space, little endian	`ldtxa` [*regaddr*] `#ASI_TWINX_AIUS_L,` *reg*$_{rd}$	**N**–
	$2E_{16}$	Load Integer Twin Extended Word, real address, little endian	`ldtxa` [*regaddr*] `#ASI_TWINX_REAL_L,` *reg*$_{rd}$	**N**–
	$2F_{16}$	Load Integer Twin Extended Word, nucleus context, little-endian	`ldtxa` [*regaddr*] `#ASI_TWINX_NL,` *reg*$_{rd}$	**N**–
LDTXA[N]	$E2_{16}$	Load Integer Twin Extended Word, Primary address space	`ldtxa` [*regaddr*] `#ASI_TWINX_P,` *reg*$_{rd}$	**N**–
	$E3_{16}$	Load Integer Twin Extended Word, Secondary address space	`ldtxa` [*regaddr*] `#ASI_TWINX_S,` *reg*$_{rd}$	**N**–
	EA_{16}	Load Integer Twin Extended Word, Primary address space, little endian	`ldtxa` [*regaddr*] `#ASI_TWINX_PL,` *reg*$_{rd}$	**N**–
	EB_{16}	Load Integer Twin Extended Word, Secondary address space, little-endian	`ldtxa` [*regaddr*] `#ASI_TWINX_SL,` *reg*$_{rd}$	**N**–

† 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.

11	rd	01 0011	rs1	i=0	imm_asi	rs2
31 30	29 25	24 19	18 14	13	12 5	4 0

LDTXA

Description ASIs 26_{16}, $2E_{16}$, $E2_{16}$, $E3_{16}$, $F0_{16}$, and $F1_{16}$ are used with the LDTXA instruction to atomically read a 128-bit data item into a pair of 64-bit R registers (a "twin extended word"). The data are placed in an even/odd pair of 64-bit registers. The lowest-address 64 bits are placed in the even-numbered register; the highest-address 64 bits are placed in the odd-numbered register.

> **Note** Execution of an LDTXA instruction with rd = 0 modifies only R[1].

ASIs $E2_{16}$, $E3_{16}$, $F0_{16}$, and $F1_{16}$ perform an access using a virtual address, while ASIs 26_{16} and $2E_{16}$ use a real address.

An LDTXA instruction that performs a little-endian access behaves as if it comprises two 64-bit loads (performed atomically), each of which is byte-swapped independently before being written into its respective destination register.

Exceptions. An attempt to execute an LDTXA instruction with an odd-numbered destination register (rd{0} = 1) causes an *illegal_instruction* exception.

An attempt to execute an LDTXA instruction with an effective memory address that is not aligned on a 16-byte boundary causes a *mem_address_not_aligned* exception.

IMPL. DEP. #413-S10: It is implementation dependent whether *VA_watchpoint*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 26_{16} and $2E_{16}$; the effective address provided with either of those ASIs is interpreted directly as a real address.

> **Compatibility Note** ASIs 27_{16}, $2F_{16}$, 26_{16}, and $2E_{16}$ are now standard ASIs that replace (respectively) ASIs 24_{16}, $2C_{16}$, 34_{16}, and $3C_{16}$ that were supported in some previous UltraSPARC implementations.

A *mem_address_not_aligned* trap is taken if the access is not aligned on a 128-byte boundary.

> **Implementation Note** LDTXA shares an opcode with the "i = 0" variant of the (deprecated) LDTWA instruction; they are differentiated by the combination of the value of "i" and the ASI used in the instruction. See *Load Integer Twin Word from Alternate Space* on page 194.

Exceptions

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

See Also LDTWA on page 194

7.61 Load Floating-Point State Register

Instruction	op3	rd	Operation	Assembly Language Syntax	Class
	10 0001	0	(see page 186)		
LDXFSR	10 0001	1	Load Floating-Point State Register	`ldx` [*address*], `%fsr`	**A1**
—	10 0001	2–31	*Reserved*		

31–30	29–25	24–19	18–14	13	12–5	4–0
11	rd	op3	rs1	i=0	—	rs2

31–30	29–25	24–19	18–14	13	12–0
11	rd	op3	rs1	i=1	simm13

Description A load floating-point state register instruction (LDXFSR) waits for all FPop instructions that have not finished execution to complete and then loads a doubleword from memory into the FSR.

LDXFSR does not alter the ver, ftt, qne, reserved, or unimplemented (for example, ns) fields of FSR (see page 42).

Programming Note For future compatibility, software should only issue an LDXFSR instruction with a zero value (or a value previously read from the same field) written into any reserved field of FSR.

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

If i = 0, the effective address for these instructions is “R[rs1] + R[rs2]” and if i = 0, the effective address is “R[rs1] + **sign_ext**(simm13)”.

Exceptions. An attempt to execute an instruction encoded as op = 2 and op3 = 21_{16} when any of the following conditions exist causes an *illegal_instruction* exception:

- i = 0 and instruction bits 12:5 are nonzero
- (rd > 1)

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

If the effective address is not doubleword-aligned, an attempt to execute an LDXFSR instruction causes a *mem_address_not_aligned* exception.

Destination Register(s) when Exception Occurs. If a load floating-point state register instruction generates an exception that causes a *precise* trap, the destination register (FSR) remains unchanged.

IMPL. DEP. #44-V8-Cs10(a)(2): If an LDXFSR instruction generates an exception that causes a *non-precise* trap, it is implementation dependent whether the contents of the destination register (FSR) is undefined or is guaranteed to remain unchanged.

Implementation Note LDXFSR shares an opcode with the (deprecated) LDFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = 11_2, op3 = $10\ 0001_2$ opcode with an invalid rd value causes an *illegal_instruction* exception.

LDTXA

Exceptions *illegal_instruction*
fp_disabled
mem_address_not_aligned
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

See Also *Load Floating-Point Register on page 181*
Load Floating-Point State Register (Lower) on page 186
Store Floating-Point State Register on page 269

7.62 Memory Barrier

Instruction	op3	Operation	Assembly Language Syntax	Class
MEMBAR	10 1000	Memory Barrier	`membar` *membar_mask*	**A1**

10	0	op3	0 1111	i=1	—	cmask	mmask
31 30	29 25	24 19	18 14	13	12 7	6 4	3 0

Description The memory barrier instruction, MEMBAR, has two complementary functions: to express order constraints between memory references and to provide explicit control of memory-reference completion. The *membar_mask* field in the suggested assembly language is the concatenation of the cmask and mmask instruction fields.

MEMBAR introduces an order constraint between classes of memory references appearing before the MEMBAR and memory references following it in a program. The particular classes of memory references are specified by the mmask field. Memory references are classified as loads (including load instructions LDSTUB[A], SWAP[A], CASA, and CASX[A] and stores (including store instructions LDSTUB[A], SWAP[A], CASA, CASXA, and FLUSH). The mmask field specifies the classes of memory references subject to ordering, as described below. MEMBAR applies to all memory operations in all address spaces referenced by the issuing virtual processor, but it has no effect on memory references by other virtual processors. When the cmask field is nonzero, completion as well as order constraints are imposed, and the order imposed can be more stringent than that specifiable by the mmask field alone.

A load has been performed when the value loaded has been transmitted from memory and cannot be modified by another virtual processor. A store has been performed when the value stored has become visible, that is, when the previous value can no longer be read by any virtual processor. In specifying the effect of MEMBAR, instructions are considered to be executed as if they were processed in a strictly sequential fashion, with each instruction completed before the next has begun.

The mmask field is encoded in bits 3 through 0 of the instruction. TABLE 7-7 specifies the order constraint that each bit of mmask (selected when set to 1) imposes on memory references appearing before and after the MEMBAR. From zero to four mask bits may be selected in the mmask field.

TABLE 7-7 MEMBAR mmask Encodings

Mask Bit	Assembly Language Name	Description
mmask{3}	`#StoreStore`	The effects of all stores appearing prior to the MEMBAR instruction must be visible to all virtual processors before the effect of any stores following the MEMBAR.
mmask{2}	`#LoadStore`	All loads appearing prior to the MEMBAR instruction must have been performed before the effects of any stores following the MEMBAR are visible to any other virtual processor.
mmask{1}	`#StoreLoad`	The effects of all stores appearing prior to the MEMBAR instruction must be visible to all virtual processors before loads following the MEMBAR may be performed.
mmask{0}	`#LoadLoad`	All loads appearing prior to the MEMBAR instruction must have been performed before any loads following the MEMBAR may be performed.

The cmask field is encoded in bits 6 through 4 of the instruction. Bits in the cmask field, described in TABLE 7-8, specify additional constraints on the order of memory references and the processing of instructions. If cmask is zero, then MEMBAR enforces the partial ordering specified by the mmask field; if cmask is nonzero, then completion and partial order constraints are applied.

TABLE 7-8 MEMBAR cmask Encodings

Mask Bit	Function	Assembly Language Name	Description
cmask{2}	Synchronization barrier	`#Sync`	All operations (including nonmemory reference operations) appearing prior to the MEMBAR must have been performed and the effects of any exceptions be visible before any instruction after the MEMBAR may be initiated.
cmask{1}	Memory issue barrier	`#MemIssue`	All memory reference operations appearing prior to the MEMBAR must have been performed before any memory operation after the MEMBAR may be initiated.
cmask{0}	Lookaside barrier	`#Lookaside`	A store appearing prior to the MEMBAR must complete before any load following the MEMBAR referencing the same address can be initiated.

A MEMBAR instruction with both mmask = 0 and cmask = 0 is functionally a NOP.

For information on the use of MEMBAR, see *Memory Ordering and Synchronization* on page 316 and *Programming with the Memory Models* contained in the separate volume *UltraSPARC Architecture Application Notes*. For additional information about the memory models themselves, see Chapter 9, *Memory*.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

V9 Compatibility Note MEMBAR with mmask = 8_{16} and cmask = 0_{16} (MEMBAR `#StoreStore`) is identical in function to the SPARC V8 STBAR instruction, which is deprecated.

An attempt to execute a MEMBAR instruction when instruction bits 12:7 are nonzero causes an *illegal_instruction* exception.

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

7.62.1 Memory Synchronization

The UltraSPARC Architecture provides some level of software control over memory synchronization, through use of the MEMBAR and FLUSH instructions for explicit control of memory ordering in program execution.

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

Implementation Note For an UltraSPARC Architecture virtual processor that only provides TSO memory ordering semantics, three of the ordering MEMBARs would normally be implemented as NOPs. TABLE 7-9 shows an acceptable implementation of MEMBAR for a TSO-only UltraSPARC Architecture implementation.

TABLE 7-9 MEMBAR Semantics for TSO-only implementation

MEMBAR variant	Preferred Implementation
`#StoreStore`	NOP
`#LoadStore`	NOP
`#StoreLoad`	`#Sync`
`#LoadLoad`	NOP
`#Sync`	`#Sync`
`#MemIssue`	`#Sync`
`#Lookaside`	`#Sync`

If an UltraSPARC Architecture implementation provides a less restrictive memory model than TSO (for example, RMO), the implementation of the MEMBAR variants may be different. See implementation-specific documentation for details.

7.62.2 Synchronization of the Virtual Processor

Synchronization of a virtual processor forces all outstanding instructions to be completed and any associated hardware errors to be detected and reported before any instruction after the synchronizing instruction is issued.

Synchronization can be explicitly caused by executing a synchronizing MEMBAR instruction (MEMBAR `#Sync`) or by executing an LDXA/STXA/LDDFA/STDFA instruction with an ASI that forces synchronization.

Programming Note Completion of a MEMBAR `#Sync` instruction does *not* guarantee that data previously stored has been written all the way out to external memory. Software cannot rely on that behavior. There is no mechanism in the UltraSPARC Architecture that allows software to wait for all previous stores to be written to external memory.

7.62.3 TSO Ordering Rules affecting Use of MEMBAR

For detailed rules on use of MEMBAR to enable software to adhere to the ordering rules on a virtual processor running with the TSO memory model, refer to *TSO Ordering Rules* on page 315.

Exceptions *illegal_instruction*

7.63 Move Integer Register on Condition (MOVcc)

For Integer Condition Codes

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

† *synonym:* `movnz` ‡ *synonym:* `movz` ◊ *synonym:* `movgeu` ∇ *synonym:* `movlu`

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

MOVcc

For Floating-Point Condition Codes

Instruction	op3	cond	Operation	fcc Test	Assembly Language Syntax		Class
MOVFA	10 1100	1000	Move Always	1	`mova`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFN	10 1100	0000	Move Never	0	`movn`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFU	10 1100	0111	Move if Unordered	U	`movu`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFG	10 1100	0110	Move if Greater	G	`movg`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFUG	10 1100	0101	Move if Unordered or Greater	G **or** U	`movug`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFL	10 1100	0100	Move if Less	L	`movl`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFUL	10 1100	0011	Move if Unordered or Less	L **or** U	`movul`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFLG	10 1100	0010	Move if Less or Greater	L **or** G	`movlg`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFNE	10 1100	0001	Move if Not Equal	L **or** G **or** U	`movne`†	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFE	10 1100	1001	Move if Equal	E	`move`‡	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFUE	10 1100	1010	Move if Unordered or Equal	E **or** U	`movue`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFGE	10 1100	1011	Move if Greater or Equal	E **or** G	`movge`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFUGE	10 1100	1100	Move if Unordered or Greater or Equal	E **or** G **or** U	`movuge`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFLE	10 1100	1101	Move if Less or Equal	E **or** L	`movle`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFULE	10 1100	1110	Move if Unordered or Less or Equal	E **or** L **or** U	`movule`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**
MOVFO	10 1100	1111	Move if Ordered	E **or** L **or** G	`movo`	`%fcc`*n*, *reg_or_imm11*, *reg*$_{rd}$	**A1**

† *synonym:* `movnz` ‡ *synonym:* `movz`

Programming Note In assembly language, to select the appropriate condition code, include `%fcc0`, `%fcc1`, `%fcc2`, or `%fcc3` before the *reg_or_imm11* field.

31 30	29 25	24 19	18	17 14	13	12	11	10 5	4 0
10	rd	op3	cc2	cond	i=0	cc1	cc0	—	rs2
10	rd	op3	cc2	cond	i=1	cc1	cc0	simm11	

cc2	cc1	cc0	Condition Code
0	0	0	fcc0
0	0	1	fcc1
0	1	0	fcc2
0	1	1	fcc3
1	0	0	icc
1	0	1	*Reserved* (*illegal_instruction*)
1	1	0	xcc
1	1	1	*Reserved* (*illegal_instruction*)

MOVcc

Description These instructions test to see if cond is TRUE for the selected condition codes. If so, they copy the value in R[rs2] if i field = 0, or "**sign_ext**(simm11)" if i = 1 into R[rd]. The condition code used is specified by the cc2, cc1, and cc0 fields of the instruction. If the condition is FALSE, then R[rd] is not changed.

These instructions copy an integer register to another integer register if the condition is TRUE. The condition code that is used to determine whether the move will occur can be either integer condition code (icc or xcc) or any floating-point condition code (fcc0, fcc1, fcc2, or fcc3).

These instructions do not modify any condition codes.

Programming Note Branches cause the performance of many implementations to degrade significantly. Frequently, the MOVcc and FMOVcc instructions can be used to avoid branches. For example, the C language if-then-else statement

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

can be coded as

```
        cmp    %i0,%i2
        bg,a   %xcc,label
        or     %g0,1,%i3! X  = 1
        or     %g0,0,%i3! X  = 0
label:...
```

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

```
cmp    %i0,%i2
        or     %g0,1,%i3! assume X = 1
        movle  %xcc,0,%i3! overwrite with X = 0
```

This approach takes only three instructions and no branches and may boost performance significantly. Use MOVcc and FMOVcc instead of branches wherever these instructions would increase performance.

An attempt to execute a MOVcc instruction when either instruction bits 10:5 are nonzero or (cc2 :: cc1 :: cc0) = 101_2 or 111_2 causes an *illegal_instruction* exception.

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

Exceptions *illegal_instruction*
fp_disabled

7.64 Move Integer Register on Register Condition (MOVr)

Instruction	op3	rcond	Operation	Test	Assembly Language Syntax	Class
—	10 1111	000	*Reserved* (*illegal_instruction*)			—
MOVRZ	10 1111	001	Move if Register Zero	R[rs1] = 0	`movrz`† *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$	**A1**
MOVRLEZ	10 1111	010	Move if Register Less Than or Equal to Zero	R[rs1] ≤ 0	`movrlez` *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$	**A1**
MOVRLZ	10 1111	011	Move if Register Less Than Zero	R[rs1] < 0	`movrlz` *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$	**A1**
—	10 1111	100	*Reserved* (*illegal_instruction*)			—
MOVRNZ	10 1111	101	Move if Register Not Zero	R[rs1] ≠ 0	`movrnz`‡ *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$	**A1**
MOVRGZ	10 1111	110	Move if Register Greater Than Zero	R[rs1] > 0	`movrgz` *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$	**A1**
MOVRGEZ	10 1111	111	Move if Register Greater Than or Equal to Zero	R[rs1] ≥ 0	`movrgez` *reg*$_{rs1}$, *reg_or_imm10*, *reg*$_{rd}$	**A1**

† *synonym:* `movre` ‡ *synonym:* `movrne`

31 30	29 25	24 19	18 14	13	12 10	9 5	4 0
10	rd	op3	rs1	i=0	rcond	—	rs2
10	rd	op3	rs1	i=1	rcond	simm10	

Description If the contents of integer register R[rs1] satisfy the condition specified in the rcond field, these instructions copy their second operand (if i = 0, R[rs2]; if i = 1, **sign_ext**(simm10)) into R[rd]. If the contents of R[rs1] do not satisfy the condition, then R[rd] is not modified.

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

Programming Note The MOVr instructions are "64-bit-only" instructions; there is no version of these instructions that operates on just the less-significant 32 bits of their source operands.

Implementation Note If this instruction is implemented by tagging each register value with an n (negative) and a z (zero) bit, use the table below to determine if rcond is TRUE.

Move	Test
MOVRNZ	**not** Z
MOVRZ	Z
MOVRGEZ	**not** N
MOVRLZ	N
MOVRLEZ	N **or** Z
MOVRGZ	N **nor** Z

An attempt to execute a MOVr instruction when either instruction bits 9:5 are nonzero or rcond = 000_2 or 100_2 causes an *illegal_instruction* exception.

Exceptions *illegal_instruction*

7.65 Multiply Step

> The MULScc instruction is deprecated and should not be used in new software. The MULX instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
MULScc[D]	10 0100	Multiply Step and modify cc's	`mulscc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**Y3**

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	

Description MULScc treats the less-significant 32 bits of R[rs1] and the less-significant 32 bits of the Y register as a single 64-bit, right-shiftable doubleword register. The least significant bit of R[rs1] is treated as if it were adjacent to bit 31 of the Y register. The MULScc instruction performs an addition operation, based on the least significant bit of Y.

Multiplication assumes that the Y register initially contains the multiplier, R[rs1] contains the most significant bits of the product, and R[rs2] contains the multiplicand. Upon completion of the multiplication, the Y register contains the least significant bits of the product.

Note | In a standard MULScc instruction, rs1 = rd.

MULScc operates as follows:

1. If i = 0, the multiplicand is R[rs2]; if i = 1, the multiplicand is **sign_ext***(*simm13).
2. A 32-bit value is computed by shifting the value from R[rs1] right by one bit with "CCR.icc.n **xor** CCR.icc.v" replacing bit 31 of R[rs1]. (This is the proper sign for the previous partial product.)
3. If the least significant bit of Y = 1, the shifted value from step (2) and the multiplicand are added. If the least significant bit of the Y = 0, then 0 is added to the shifted value from step (2).
4. MULScc writes the following result values:

Register field	Value written by MULScc
CCR.icc	updated according to the result of the addition in step (3) above
R[rd]{63:33}	0
R[rd]{32}	CCR.icc.c
R[rd]{31:0}	the least-significant 32 bits of the sum from step (3) above
Y	the previous value of the Y register, shifted right by one bit, with Y{31} replaced by the value of R[rs1]{0} prior to shifting in step (2)
CCR.xcc.n	0
CCR.xcc.v	0
CCR.xcc.c	0
CCR.xcc.z	**if** (R[rd]{63:0} = 0) **then** 1 **else** 0

SPARC V9 Compatibility Note | In SPARC V9, MULScc's effect on R[rd]{63:32} and CCR.xcc were explicitly left undefined.

5. The Y register is shifted right by one bit, with the least significant bit of the unshifted R[rs1] replacing bit 31 of Y.

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

Exceptions *illegal_instruction*

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

7.66 Multiply and Divide (64-bit)

Instruction	op3	Operation	Assembly Language		Class
MULX	00 1001	Multiply (signed or unsigned)	`mulx`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
SDIVX	10 1101	Signed Divide	`sdivx`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
UDIVX	00 1101	Unsigned Divide	`udivx`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**

10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

Description MULX computes “R[rs1] × R[rs2]” if i = 0 or “R[rs1] × **sign_ext**(simm13)” if i = 1, and writes the 64-bit product into R[rd]. MULX can be used to calculate the 64-bit product for signed or unsigned operands (the product is the same).

SDIVX and UDIVX compute “R[rs1] ÷ R[rs2]” if i = 0 or “R[rs1] ÷ **sign_ext**(simm13)” if i = 1, and write the 64-bit result into R[rd]. SDIVX operates on the operands as signed integers and produces a corresponding signed result. UDIVX operates on the operands as unsigned integers and produces a corresponding unsigned result.

For SDIVX, if the largest negative number is divided by −1, the result should be the largest negative number. That is:

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

These instructions do not modify any condition codes.

An attempt to execute a MULX, SDIVX, or UDIVX instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

Exceptions *illegal_instruction*
division_by_zero

7.67 No Operation

Instruction	op2	Operation	Assembly Language Syntax	Class
NOP	100	No Operation	nop	**A1**

00	rd = 0 0 0 0 0	op2	imm22 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
31 30	29 25	24 22	21 0

Description The NOP instruction changes no program-visible state (except that of the PC register).

NOP is a special case of the SETHI instruction, with imm22 = 0 and rd = 0.

Programming Note There are many other opcodes that may execute as NOPs; however, this dedicated NOP instruction is the only one guaranteed to be implemented efficiently across all implementations.

Exceptions None

7.68 NORMALW

Instruction	Operation	Assembly Language Syntax	Class
NORMALW^P	"Other" register windows become "normal" register windows	`normalw`	**A1**

31 30	29 25	24 19	18 0
10	fcn = 0 0100	11 0001	—

Description NORMALW^P is a privileged instruction that copies the value of the OTHERWIN register to the CANRESTORE register, then sets the OTHERWIN register to zero.

Programming Notes The NORMALW instruction is used when changing address spaces. NORMALW indicates the current "other" windows are now "normal" windows and should use the *spill_n_normal* and *fill_n_normal* traps when they generate a trap due to window spill or fill exceptions. The window state may become inconsistent if NORMALW is used when CANRESTORE is nonzero.

An attempt to execute a NORMALW instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute an NORMALW instruction in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_opcode* exception.

Exceptions *illegal_instruction*
privileged_opcode

See Also ALLCLEAN on page 99
INVALW on page 173
OTHERW on page 215
RESTORED on page 232
SAVED on page 239

7.69 OR Logical Operation

Instruction	op3	Operation	Assembly Language Syntax		Class
OR	00 0010	Inclusive **or**	`or`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ORcc	01 0010	Inclusive **or** and modify cc's	`orcc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ORN	00 0110	Inclusive **or not**	`orn`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
ORNcc	01 0110	Inclusive **or not** and modify cc's	`orncc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**

10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

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

ORcc and ORNcc modify the integer condition codes (icc and xcc). They set the condition codes as follows:

- icc.v, icc.c, xcc.v, and xcc.c are set to 0
- icc.n is copied from bit 31 of the result
- xcc.n is copied from bit 63 of the result
- icc.z is set to 1 if bits 31:0 of the result are zero (otherwise to 0)
- xcc.z is set to 1 if all 64 bits of the result are zero (otherwise to 0)

ORN and ORNcc logically negate their second operand before applying the main (**or**) operation.

An attempt to execute an OR[N][cc] instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

Exceptions *illegal_instruction*

7.70 OTHERW

Instruction	Operation	Assembly Language Syntax	Class
OTHERW^{P}	“Normal” register windows become “other” register windows	`otherw`	**A1**

10	fcn = 0 0011	11 0001	—
31 30	29 25	24 19	18 0

Description OTHERW^{P} is a privileged instruction that copies the value of the CANRESTORE register to the OTHERWIN register, then sets the CANRESTORE register to zero.

Programming Notes The OTHERW instruction is used when changing address spaces. OTHERW indicates the current "normal" register windows are now "other" register windows and should use the *spill_n_other* and *fill_n_other* traps when they generate a trap due to window spill or fill exceptions. The window state may become inconsistent if OTHERW is used when OTHERWIN is nonzero.

An attempt to execute an OTHERW instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute an OTHERW instruction in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_opcode* exception.

Exceptions *illegal_instruction*
privileged_opcode

See Also ALLCLEAN on page 99
INVALW on page 173
NORMALW on page 213
RESTORED on page 232
SAVED on page 239

7.71 Pixel Component Distance (with Accumulation) VIS 1

Instruction	opf	Operation	Assembly Language Syntax	Class
PDIST	0 0011 1110	Distance between eight 8-bit components, with accumulation	`pdist` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**C2**

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Description Eight unsigned 8-bit values are contained in the 64-bit floating-point source registers F_D[rs1] and F_D[rs2]. The corresponding 8-bit values in the source registers are subtracted (that is, each byte in F_D[rs2] is subtracted from the corresponding byte in F_D[rs1]). The sum of the absolute value of each difference is added to the integer in F_D[rd] and the resulting integer sum is stored in the destination register, F_D[rd].

Programming Notes PDIST uses F_D[rd] as both a source and a destination register.

Typically, PDIST is used for motion estimation in video compression algorithms.

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

Exceptions *fp_disabled*

7.72 Population Count

Instruction	op3	Operation	Assembly Language Syntax	Class
POPC	10 1110	Population Count	`popc` *reg_or_imm*, *reg*$_{rd}$	**C2**

10	rd	op3	0 0000	i=0	—	rs2

10	rd	op3	0 0000	i=1	simm13
31 30	29 25	24 19	18 14	13	12 0

Description POPC counts the number of ‘1’ bits in R[rs2] if i = 0, or the number of ‘1’ bits in **sign_ext(**simm13) if i = 1, and stores the count in R[rd]. This instruction does not modify the condition codes.

V9 Compatibility Note Instruction bits 18 through 14 must be zero for POPC. Other encodings of this field (rs1) may be used in future versions of the SPARC architecture for other instructions.

Programming Note POPC can be used to “find first bit set” in a register. A ‘C’-language program illustrating how POPC can be used for this purpose follows:

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

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

```
neg     %IN, %NEG_IN        ! –zz(2’s complement)
xnor    %IN, %NEG_IN,%TEMP! ^ ~ –zz (exclusive nor)
popc    %TEMP,%RESULT       ! result  = popc(zz ^ ~ –zz)
movrz   %IN,%g0,%RESULT     ! %RESULT should be 0 for %IN=0
```

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

Example computation:

```
           IN = ...00101000 !1st ‘1’ bit from right is
          –IN = ...11011000 !          bit 3 (4th bit)
        ~ –IN = ...00100111
     IN ^ ~ –IN = ...00001111
popc(IN ^ ~ –IN) = 4
```

Programming Note POPC can be used to “centrifuge” all the ‘1’ bits in a register to the least significant end of a destination register. Assembly-language code illustrating how POPC can be used for this purpose follows:

```
popc    %IN, %DEST
cmp     %IN, -1             ! Test for pattern of all 1’s
mov     -1, %TEMP           ! Constant -1 -> temp register
sllx    %TEMP,%DEST,%DEST   ! (shift count of 64 same as 0)
not     %DEST               !
movcc   %xcc, -1, %DEST     ! If src was -1, result is -1
```

where *IN*, *TEMP*, and *DEST* are integer registers.

Programming Note POPC is a “64-bit-only” instruction; there is no version of this instruction that operates on just the less-significant 32 bits of its source operand.

An attempt to execute a POPC instruction when either instruction bits 18:14 are nonzero, or i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

Exceptions *illegal_instruction*

7.73 Prefetch

Instruction	op3	Operation	Assembly Language Syntax	Class
PREFETCH	10 1101	Prefetch Data	`prefetch` [*address*], *prefetch_fcn*	**A1**
PREFETCHA$^{P_{ASI}}$	11 1101	Prefetch Data from Alternate Space	`prefetcha` [*regaddr*] *imm_asi*, *prefetch_fcn* `prefetcha` [*reg_plus_imm*] `%asi`, *prefetch_fcn*	**A1**

PREFETCH

31–30	29–25	24–19	18–14	13	12–5	4–0
11	fcn	op3	rs1	i=0	—	rs2

31–30	29–25	24–19	18–14	13	12–0
11	fcn	op3	rs1	i=1	simm13

PREFETCHA

31–30	29–25	24–19	18–14	13	12–5	4–0
11	fcn	op3	rs1	i=0	imm_asi	rs2

31–30	29–25	24–19	18–14	13	12–0
11	fcn	op3	rs1	i=1	simm13

TABLE 7-10 Prefetch Variants, by Function Code

fcn	Prefetch Variant
0	(Weak) Prefetch for several reads
1	(Weak) Prefetch for one read
2	(Weak) Prefetch for several writes and possibly reads
3	(Weak) Prefetch for one write
4	Prefetch page
5–15 (05_{16}–$0F_{16}$)	*Reserved* (*illegal_instruction*)
16 (10_{16})	Implementation dependent (NOP if not implemented)
17 (11_{16})	Prefetch to nearest unified cache
18–19 (12_{16}–13_{16})	Implementation dependent (NOP if not implemented)
20 (14_{16})	Strong Prefetch for several reads
21 (15_{16})	Strong Prefetch for one read
22 (16_{16})	Strong Prefetch for several writes and possibly reads
23 (17_{16})	Strong Prefetch for one write
24-31 (18_{16}–$1F_{16}$)	Implementation dependent (NOP if not implemented)

Description

A PREFETCH[A] instruction provides a hint to the virtual processor that software expects to access a particular address in memory in the near future, so that the virtual processor may take action to reduce the latency of accesses near that address. Typically, execution of a prefetch instruction initiates movement of a block of data containing the addressed byte from memory toward the virtual processor or creates an address mapping.

Implementation Note

A PREFETCH[A] instruction may be used by software to:

- prefetch a cache line into a cache
- prefetch a valid address translation into a TLB
-

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

PREFETCH instructions access the primary address space (`ASI_PRIMARY[_LITTLE]`).

PREFETCHA instructions access an alternate address space. If i = 0, the address space identifier (ASI) to be used for the instruction is in the imm_asi field. If i = 1, the ASI is found in the ASI register.

A prefetch operates much the same as a regular load operation, but with certain important differences. In particular, a PREFETCH[A] instruction is non-blocking; subsequent instructions can continue to execute while the prefetch is in progress.

Implementation Note A PREFETCH[A] instruction is "released" by hardware after the TLB access, allowing subsequent instructions to continue to execute while the virtual processor performs the hardware tablewalk (in the case of a TLB miss for a Strong prefetch) or the cache access in the background.

When executed in nonprivileged or privileged mode, PREFETCH[A] has the same observable effect as a NOP. A prefetch instruction will not cause a trap if applied to an illegal or nonexistent memory address. (impl. dep. #103-V9-Ms10(e))

IMPL. DEP. #103-V9-Ms10(a): The size and alignment in memory of the data block prefetched is implementation dependent; the minimum size is 64 bytes and the minimum alignment is a 64-byte boundary.

Programming Note Software may prefetch 64 bytes beginning at an arbitrary address address by issuing the instructions

```
prefetch  [address], prefetch_fcn
prefetch  [address + 63], prefetch_fcn
```

Variants of the prefetch instruction can be used to prepare the memory system for different types of accesses.

IMPL. DEP. #103-V9-Ms10(b): An implementation may implement none, some, or all of the defined PREFETCH[A] variants. It is implementation-dependent whether each variant is (1) not implemented and executes as a NOP, (2) is implemented and supports the full semantics for that variant, or (3) is implemented and only supports the simple common-case prefetching semantics for that variant.

7.73.1 Exceptions

Prefetch instructions PREFETCH and PREFETCHA generate exceptions under the conditions detailed in TABLE 7-11. Only the implementation-dependent prefetch variants (see TABLE 7-10) may generate an exception under conditions not listed in this table; the predefined variants only generate the exceptions listed here.

TABLE 7-11 Behavior of PREFETCH[A] Instructions Under Exceptional Conditions *(1 of 2)*

fcn	Instruction	Condition	Result
any	PREFETCH	i = 0 and instruction bits 12:5 are nonzero	*illegal_instruction*
any	PREFETCHA	reference to an ASI in the range 0_{16}-$7F_{16}$, while in nonprivileged mode (*privileged_action* condition)	executes as NOP
any	PREFETCHA	reference to an ASI in range 30_{16}..$7F_{16}$, while in privileged mode (*privileged_action* condition)	executes as NOP
0-3 (weak)	PREFETCH[A]	condition detected for MMU miss	executes as NOP
0-4	PREFETCH[A]	variant unimplemented	executes as NOP

TABLE 7-11 Behavior of PREFETCH[A] Instructions Under Exceptional Conditions *(2 of 2)*

fcn	Instruction	Condition	Result
0-4	PREFETCHA	reference to an invalid ASI (ASI not listed in following table)	executes as NOP
0-4, 17, 20-23	PREFETCH[A]	condition detected for *DAE_invalid_asi* (see following table), *DAE_privilege_violation*, *DAE_nc_page* ((TTE.cp = 0) **or** ((fcn = 0) **and** TTE.cv = 0)), *DAE_nfo_page*, or *DAE_side_effect_page* (TTE.e = 1)	executes as NOP
4, 20-23 (strong)	PREFETCH[A]	prefetching the requested data would be a very time-consuming operation	executes as NOP
5–15 (05_{16}–$0F_{16}$)	PREFETCH[A]	(always)	*illegal_instruction*
16-31 (18_{16}–$1F_{16}$)	PREFETCH[A]	variant unimplemented	executes as NOP

ASIs valid for PREFETCHA (all others are invalid)	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE
ASI_PRIMARY_NO_FAULT	ASI_PRIMARY_NO_FAULT_LITTLE
ASI_SECONDARY_NO_FAULT	ASI_SECONDARY_NO_FAULT_LITTLE
ASI_REAL	ASI_REAL_LITTLE

7.73.2 Weak versus Strong Prefetches

Some prefetch variants are available in two versions, “Weak” and “Strong”.

From software’s perspective, the difference between the two is the degree of certainty that the data being prefetched will subsequently be accessed. That, in turn, affects the amount of effort (time) it’s willing for the underlying hardware to invest to perform the prefetch. If the prefetch is speculative (software believes the data will probably be needed, but isn’t sure), a Weak prefetch will initiate data movement if the operation can be performed quickly, but abort the prefetch and behave like a NOP if it turns out that performing the full prefetch will be time-consuming. If software has very high confidence that data being prefetched will subsequently be accessed, then a Strong prefetch will ensure that the prefetch operation will continue, even if the prefetch operation does become time-consuming.

From the virtual processor’s perspective, the difference between a Weak and a Strong prefetch is whether the prefetch is allowed to perform a time-consuming operation in order to complete. If a time-consuming operation is required, a Weak prefetch will abandon the operation and behave like a

NOP while a Strong prefetch will pay the cost of performing the time-consuming operation so it can finish initiating the requested data movement. Behavioral differences among loads, strong prefetches, and weak prefetches are compared in TABLE 7-12.

TABLE 7-12 Comparative Behavior of Load and Weak Prefetch Operations

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

7.73.3 Prefetch Variants

The prefetch variant is selected by the fcn field of the instruction. fcn values 5–15 are reserved for future extensions of the architecture, and PREFETCH fcn values of 16–19 and 24–31 are implementation dependent in UltraSPARC Architecture 2007.

Each prefetch variant reflects an intent on the part of the compiler or programmer, a “hint” to the underlying virtual processor. This is different from other instructions (except BPN), all of which cause specific actions to occur. An UltraSPARC Architecture implementation may implement a prefetch variant by any technique, as long as the intent of the variant is achieved (impl. dep. #103-V9-Ms10(b)).

The prefetch instruction is designed to treat common cases well. The variants are intended to provide scalability for future improvements in both hardware and compilers. If a variant is implemented, it should have the effects described below. In case some of the variants listed below are implemented and some are not, a recommended overloading of the unimplemented variants is provided in the SPARC V9 specification. An implementation must treat any unimplemented prefetch fcn values as NOPs (impl. dep. #103-V9-Ms10).

7.73.3.1 Prefetch for Several Reads (fcn = 0, 20(14_{16}))

The intent of these variants is to cause movement of data into the cache nearest the virtual processor.

There are Weak and Strong versions of this prefetch variant; fcn = 0 is Weak and fcn = 20 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

Programming Note | The intended use of this variant is for streaming relatively small amounts of data into the primary data cache of the virtual processor.

7.73.3.2 Prefetch for One Read (fcn = 1, 21(15_{16}))

The data to be read from the given address are expected to be read once and not reused (read or written) soon after that. Use of this PREFETCH variant indicates that, if possible, the data cache should be minimally disturbed by the data read from the given address.

There are Weak and Strong versions of this prefetch variant; fcn = 1 is Weak and fcn = 21 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

Programming Note | The intended use of this variant is in streaming medium amounts of data into the virtual processor without disturbing the data in the primary data cache memory.

7.73.3.3 Prefetch for Several Writes (and Possibly Reads) (fcn = 2, 22(16_{16}))

The intent of this variant is to cause movement of data in preparation for multiple writes.

There are Weak and Strong versions of this prefetch variant; fcn = 2 is Weak and fcn = 22 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

Programming Note | An example use of this variant is to initialize a cache line, in preparation for a partial write.

Implementation Note | On a multiprocessor system, this variant indicates that exclusive ownership of the addressed data is needed. Therefore, it may have the additional effect of obtaining exclusive ownership of the addressed cache line.

7.73.3.4 Prefetch for One Write (fcn = 3, 23(17_{16}))

The intent of this variant is to initiate movement of data in preparation for a single write. This variant indicates that, if possible, the data cache should be minimally disturbed by the data written to this address, because those data are not expected to be reused (read or written) soon after they have been written once.

There are Weak and Strong versions of this prefetch variant; fcn = 3 is Weak and fcn = 23 is Strong. The choice of Weak or Strong variant controls the degree of effort that the virtual processor may expend to obtain the data.

7.73.3.5 Prefetch Page (fcn = 4)

In a virtual memory system, the intended action of this variant is for hardware (or privileged or hyperprivileged software) to initiate asynchronous mapping of the referenced virtual address (assuming that it is legal to do so).

Programming Note | Prefetch Page is used is to avoid a later page fault for the given address, or at least to shorten the latency of a page fault.

In a non-virtual-memory system or if the addressed page is already mapped, this variant has no effect.

Implementation Note | The mapping required by Prefetch Page may be performed by privileged software, hyperprivileged software, or hardware.

7.73.3.6 Prefetch to Nearest Unified Cache (fcn = 17(11_{16}))

The intent of this variant is to cause movement of data into the nearest unified (combined instruction and data) cache. At the successful completion of this variant, the selected line from memory will be in the unified cache in the shared state, and in caches (if any) below it in the cache hierarchy.

Prefetch to Nearest Unified Cache is a Strong prefetch variant.

7.73.4 Implementation-Dependent Prefetch Variants (fcn = 16, 18, 19, and 24–31)

IMPL. DEP. #103-V9-Ms10(c): Whether and how PREFETCH fcns 16, 18, 19 and 24-31 are implemented are implementation dependent. If a variant is not implemented, it must execute as a NOP.

7.73.5 Additional Notes

Programming Note Prefetch instructions do have some "cost to execute". As long as the cost of executing a prefetch instruction is well less than the cost of a cache miss, use of prefetching provides a net gain in performance.

It does not appear that prefetching causes a significant number of useless fetches from memory, though it may increase the rate of *useful* fetches (and hence the bandwidth), because it more efficiently overlaps computing with fetching.

Programming Note A compiler that generates PREFETCH instructions should generate each of the variants where its use is most appropriate. That will help portable software be reasonably efficient across a range of hardware configurations.

Implementation Note Any effects of a data prefetch operation in privileged code should be reasonable (for example, no page prefetching is allowed within code that handles page faults). The benefits of prefetching should be available to most privileged code.

Implementation Note A prefetch from a nonprefetchable location has no effect. It is up to memory management hardware to determine how locations are identified as not prefetchable.

Exceptions *illegal_instruction*

7.74 Read Ancillary State Register

Instruction	rs1	Operation	Assembly Language Syntax	Class
RDY^{D}	0	Read Y register *(deprecated)*	`rd %y,` reg_{rd}	D2
—	1	*Reserved*		
RDCCR	2	Read Condition Codes register (CCR)	`rd %ccr,` reg_{rd}	A1
RDASI	3	Read ASI register	`rd %asi,` reg_{rd}	A1
RDTICK$^{P_{npt}}$	4	Read TICK register	`rd %tick,` reg_{rd}	A1
RDPC	5	Read Program Counter (PC)	`rd %pc,` reg_{rd}	A2
RDFPRS	6	Read Floating-Point Registers Status (FPRS) register	`rd %fprs,` reg_{rd}	A1
—	7 − 14 ($7\text{-}0E_{16}$)	*Reserved*		
See text	15 (F_{16})	MEMBAR or *Reserved*; see text		
—	16-18 (10_{16}-12_{16})	*Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
RDGSR	19 (13_{16})	Read General Status register (GSR)	`rd %gsr,` reg_{rd}	A1
—	20–21 (14_{16}-15_{16})	*Reserved* (impl. dep. #8-V8-Cs20, #9-V8-Cs20)		
RDSOFTINT^{P}	22 (16_{16})	Read per-virtual processor Soft Interrupt register (SOFTINT)	`rd %softint,` reg_{rd}	A2
RDTICK_CMPR^{P}	23 (17_{16})	Read Tick Compare register (TICK_CMPR)	`rd %tick_cmpr,` reg_{rd}	N–
RDSTICK$^{P_{npt}}$	24 (18_{16})	Read System Tick Register (STICK)	`rd %stick†,` reg_{rd}	A2
RDSTICK_CMPR^{P}	25 (19_{16})	Read System Tick Compare register (STICK_CMPR)	`rd %stick_cmpr†,` reg_{rd}	A2
—	26 (20_{16})	*Reserved* (impl. dep. #8-V8-Cs20, #9-V8-Cs20)		
—	27 ($1B_{16}$)	*Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	28 ($1C_{16}$)	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	29 ($1D_{16}$)	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	30 ($1E_{16}$)	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	31 ($1F_{16}$)	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		

† The original assembly language names for `%stick` and `%stick_cmpr` were, respectively, `%sys_tick` and `%sys_tick_cmpr`, which are now deprecated. Over time, assemblers will support the new `%stick` and `%stick_cmpr` names for these registers (which are consistent with `%tick` and `%tick_cmpr`). In the meantime, some existing assemblers may only recognize the original names.

10	rd	10 1000	rs1	i=0	—
31 30	29 25	24 19	18 14	13	12 0

RDasr

Description The Read Ancillary State Register (RDasr) instructions copy the contents of the state register specified by rs1 into R[rd].

An RDasr instruction with rs1 = 0 is a (deprecated) RDY instruction (which should not be used in new software).

> The RDY instruction is deprecated. It is recommended that all instructions that reference the Y register be avoided.

RDPC copies the contents of the PC register into R[rd]. If PSTATE.am = 0, the full 64-bit address is copied into R[rd]. If PSTATE.am = 1, only a 32-bit address is saved; PC{31:0} is copied to R[rd]{31:0} and R[rd]{63:32} is set to 0. (closed impl. dep. #125-V9-Cs10)

RDFPRS waits for all pending FPops and loads of floating-point registers to complete before reading the FPRS register.

The following values of rs1 are reserved for future versions of the architecture: 1, 7–14, 16-18, 20-21, and 26-27.

IMPL. DEP. #47-V8-Cs20: RDasr instructions with rd in the range 28–31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For an RDasr instruction with rs1 in the range 28–31, the following are implementation dependent:
- the interpretation of bits 13:0 and 29:25 in the instruction
- whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20), and
- whether an attempt to execute the instruction causes an *illegal_instruction* exception.

Implementation Note See the section “Read/Write Ancillary State Registers (ASRs)” in *Extending the UltraSPARC Architecture*, contained in the separate volume *UltraSPARC Architecture Application Notes*, for a discussion of extending the SPARC V9 instruction set using read/write ASR instructions.

Note Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers.

SPARC V8 Compatibility Note The SPARC V8 RDPSR, RDWIM, and RDTBR instructions do not exist in the UltraSPARC Architecture, since the PSR, WIM, and TBR registers do not exist.

See *Ancillary State Registers* on page 48 for more detailed information regarding ASR registers.

Exceptions. An attempt to execute a RDasr instruction when any of the following conditions are true causes an *illegal_instruction* exception:

- rs1 = 15 and rd ≠ 0 (reserved for future versions of the architecture)
- rs1 = 1, 7–14, 16-18, 20-21, or 26-27 (reserved for future versions of the architecture)
- instruction bits 13:0 are nonzero

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

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

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

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

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

In nonprivileged mode (PSTATE.priv = 0), the following cause a *privileged_action* exception:

- execution of RDTICK when nonprivileged access to TICK is disabled
- execution of RDSTICK when nonprivileged access to STICK is disabled

Implementation Note | RDasr shares an opcode with MEMBAR; it is distinguished by rs1 = 15 or rd = 0 or (i = 0, and bit 12 = 0).

Exceptions

illegal_instruction
privileged_opcode
fp_disabled
privileged_action

See Also

RDPR on page 228
WRasr on page 285

7.75 Read Privileged Register

Instruction	op3	Operation	rs1	Assembly Language Syntax	Class
RDPR[P]	10 1010	Read Privileged register			**A2?**
		TPC	0	`rdpr %tpc,` *reg*$_{rd}$	**A1?**
		TNPC	1	`rdpr %tnpc,` *reg*$_{rd}$	
		TSTATE	2	`rdpr %tstate,` *reg*$_{rd}$	
		TT	3	`rdpr %tt,` *reg*$_{rd}$	
		TICK	4	`rdpr %tick,` *reg*$_{rd}$	
		TBA	5	`rdpr %tba,` *reg*$_{rd}$	
		PSTATE	6	`rdpr %pstate,` *reg*$_{rd}$	
		TL	7	`rdpr %tl,` *reg*$_{rd}$	
		PIL	8	`rdpr %pil,` *reg*$_{rd}$	
		CWP	9	`rdpr %cwp,` *reg*$_{rd}$	
		CANSAVE	10	`rdpr %cansave,` *reg*$_{rd}$	
		CANRESTORE	11	`rdpr %canrestore,` *reg*$_{rd}$	
		CLEANWIN	12	`rdpr %cleanwin,` *reg*$_{rd}$	
		OTHERWIN	13	`rdpr %otherwin,` *reg*$_{rd}$	
		WSTATE	14	`rdpr %wstate,` *reg*$_{rd}$	
		Reserved	15		
		GL	16	`rdpr %gl,` *reg*$_{rd}$	
		Reserved	17–31		

31 30	29 … 25	24 … 19	18 … 14	13 … 0
10	rd	op3	rs1	—

Description The rs1 field in the instruction determines the privileged register that is read. There are *MAXPTL* copies of the TPC, TNPC, TT, and TSTATE registers. A read from one of these registers returns the value in the register indexed by the current value in the trap level register (TL). A read of TPC, TNPC, TT, or TSTATE when the trap level is zero (TL = 0) causes an *illegal_instruction* exception.

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

- instruction bits 13:0 are nonzero
- rs1 = 15, or 17 ≤ rs1 ≤ 31 (reserved rs1 values)
- 0 ≤ rs1 ≤ 3 (attempt to read TPC, TNPC,TSTATE, or TT register) while TL = 0 (current trap level is zero) and the virtual processor is in privileged mode.

Implementation Note In nonprivileged mode, *illegal_instruction* exception due to 0 ≤ rs1 ≤ 3 and TL = 0 does not occur; the *privileged_opcode* exception occurs instead.

An attempt to execute a RDPR instruction in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_opcode* exception.

Historical Note On some early SPARC implementations, floating-point exceptions could cause deferred traps. To ensure that execution could be correctly resumed after handling a deferred trap, hardware provided a floating-point queue (FQ), from which the address of the trapping instruction could be obtained by the trap handler. The front of the FQ was accessed by executing a RDPR instruction with rs1 = 15.

On UltraSPARC Architecture implementations, all floating-point traps are precise. When one occurs, the address of a trapping instruction can be found by the trap handler in the TPC[TL], so no floating-point queue (FQ) is needed or implemented (impl. dep. #25-V8) and RDPR with rs1 = 15 generates an *illegal_instruction* exception.

Exceptions *illegal_instruction*
privileged_opcode

See Also RDasr on page 225
WRPR on page 288

7.76 RESTORE

Instruction	op3	Operation	Assembly Language Syntax	Class
RESTORE	11 1101	Restore Caller's Window	`restore` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**

10	rd	11 1101	rs1	i=0	—	rs2
10	rd	11 1101	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

Description The RESTORE instruction restores the register window saved by the last SAVE instruction executed by the current process. The *in* registers of the old window become the *out* registers of the new window. The *in* and *local* registers in the new window contain the previous values.

Furthermore, if and only if a fill trap is not generated, RESTORE behaves like a normal ADD instruction, except that the source operands R[rs1] or R[rs2] are read from the *old* window (that is, the window addressed by the original CWP) and the sum is written into R[rd] of the *new* window (that is, the window addressed by the new CWP).

Note CWP arithmetic is performed modulo the number of implemented windows, *N_REG_WINDOWS*.

Programming Notes Typically, if a RESTORE instruction traps, the fill trap handler returns to the trapped instruction to reexecute it. So, although the ADD operation is not performed the first time (when the instruction traps), it is performed the second time the instruction executes. The same applies to changing the CWP.

There is a performance trade-off to consider between using SAVE/ RESTORE and saving and restoring selected registers explicitly.

Description (Effect on Privileged State)

If a RESTORE instruction does not trap, it decrements the CWP (**mod** *N_REG_WINDOWS*) to restore the register window that was in use prior to the last SAVE instruction executed by the current process. It also updates the state of the register windows by decrementing CANRESTORE and incrementing CANSAVE.

If the register window to be restored has been spilled (CANRESTORE = 0), then a fill trap is generated. The trap vector for the fill trap is based on the values of OTHERWIN and WSTATE, as described in *Trap Type for Spi ll/Fill Traps* on page 355. The fill trap handler is invoked with CWP set to point to the window to be filled, that is, old CWP – 1.

Programming Note The vectoring of fill traps can be controlled by setting the value of the OTHERWIN and WSTATE registers appropriately. For details, see the section "Splitting the Register Windows" in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes.*

The fill handler normally will end with a RESTORED instruction followed by a RETRY instruction.

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

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

See Also
SAVE on page 237

7.77 RESTORED

Instruction	Operation	Assembly Language Syntax	Class
RESTORED[P]	Window has been restored	`restored`	**A1**

10	fcn = 0 0001	11 0001	—
31 30	29 25	24 19	18 0

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

RESTORED increments CANRESTORE.

If CLEANWIN < (*N_REG_WINDOWS*–1), then RESTORED increments CLEANWIN.

If OTHERWIN = 0, RESTORED decrements CANSAVE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

Programming Notes Trap handler software for register window fills use the RESTORED instruction to indicate that a window has been filled successfully. For details, see the section “Example Code for Spill Handler” in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

Normal privileged software would probably not execute a RESTORED instruction from trap level zero (TL = 0). However, it is not illegal to do so and doing so does not cause a trap.

Executing a RESTORED instruction outside of a window fill trap handler is likely to create an inconsistent window state. Hardware will not signal an exception, however, since maintaining a consistent window state is the responsibility of privileged software.

If CANSAVE = 0 or CANRESTORE ≥ (*N_REG_WINDOWS* – 2) just prior to execution of a RESTORED instruction, the subsequent behavior of the processor is undefined. In neither of these cases can RESTORED generate a register window state that is both valid (see *Register Window State Definition* on page 60) and consistent with the state prior to the RESTORED.

An attempt to execute a RESTORED instruction when instruction bits 18:0 are nonzero causes an *illegal_instruction* exception.

An attempt to execute a RESTORED instruction in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_opcode* exception.

Exceptions *illegal_instruction*
privileged_opcode

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

7.78 RETRY

Instruction	op3	Operation	Assembly Language Syntax	Class
RETRY^{P}	11 1110	Return from Trap (retry trapped instruction)	`retry`	**A1**

10	fcn =0 0001	11 1110	—
31 30	29 25	24 19	18 0

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.

Programming Note | RETRY should *not* normally be used to return from the trap handler for the *control_transfer_instruction* exception itself.

See the DONE instruction on page 114 and *Trap on Control Transfer (tct)* on page 65.

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

See Also DONE on page 114

7.79 RETURN

Instruction	op3	Operation	Assembly Language Syntax	Class
RETURN	11 1001	Return	`return` *address*	**A1**

10	—	op3	rs1	i=0	—	rs2
10	—	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

Description The RETURN instruction causes a register-indirect delayed transfer of control to the target address and has the window semantics of a RESTORE instruction; that is, it restores the register window prior to the last SAVE instruction. The target address is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1. Registers R[rs1] and R[rs2] come from the *old* window.

Like other DCTIs, all effects of RETURN (including modification of CWP) are visible prior to execution of the delay slot instruction.

Programming Note To reexecute the trapped instruction when returning from a user trap handler, use the RETURN instruction in the delay slot of a JMPL instruction, for example:

```
jmpl   %l6,%g0    !Trapped PC supplied to user trap handler
return %l7        !Trapped NPC supplied to user trap handler
```

Programming Note A routine that uses a register window may be structured either as:

```
save   %sp,-framesize, %sp
. . .
ret          ! "ret" is shorthand for "jmpl %i7 + 8,%g0"
restore      ! A useful instruction in the delay slot, such as
             !     "restore %o2,%l2,%o0"
```

or as:

```
save   %sp, -framesize, %sp
. . .
return %i7 + 8
nop     ! Instead of "nop", could do some useful work in the
        !    caller's window, for example, "or %o1,%o2,%o0"
```

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

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

A RETURN instruction may cause a *window_fill* exception as part of its RESTORE semantics.

When PSTATE.am = 1, the more-significant 32 bits of the target instruction address are masked out (set to 0) before being sent to the memory system. However, if a *control_transfer_instruction* trap occurs, the full 64-bit (nonmasked) address of the RETURN instruction is written into TPC[TL].

A RETURN instruction causes a *mem_address_not_aligned* exception if either of the two least-significant bits of the target address is nonzero.

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then RETURN generates a *control_transfer_instruction* exception instead of causing a control transfer.

Exceptions

illegal_instruction
fill_n_normal (n = 0–7)
fill_n_other (n = 0–7)
mem_address_not_aligned
control_transfer_instruction (impl. dep. #450-S20)

7.80 SAVE

Instruction	op3	Operation	Assembly Language Syntax	Class
SAVE	11 1100	Save Caller's Window	save *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	A1

10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

Description The SAVE instruction provides the routine executing it with a new register window. The *out* registers from the old window become the *in* registers of the new window. The contents of the *out* and the *local* registers in the new window are zero or contain values from the executing process; that is, the process sees a clean window.

Furthermore, if and only if a spill trap is not generated, SAVE behaves like a normal ADD instruction, except that the source operands R[rs1] or R[rs2] are read from the *old* window (that is, the window addressed by the original CWP) and the sum is written into R[rd] of the *new* window (that is, the window addressed by the new CWP).

Note CWP arithmetic is performed modulo the number of implemented windows, *N_REG_WINDOWS*.

Programming Notes Typically, if a SAVE instruction traps, the spill trap handler returns to the trapped instruction to reexecute it. So, although the ADD operation is not performed the first time (when the instruction traps), it is performed the second time the instruction executes. The same applies to changing the CWP.

The SAVE instruction can be used to atomically allocate a new window in the register file and a new software stack frame in memory. For details, see the section "Leaf-Procedure Optimization" in Software Considerations, contained in the separate volume *UltraSPARC Architecture Application Notes*.

There is a performance trade-off to consider between using SAVE/RESTORE and saving and restoring selected registers explicitly.

Description (Effect on Privileged State)

If a SAVE instruction does not trap, it increments the CWP (**mod** *N_REG_WINDOWS*) to provide a new register window and updates the state of the register windows by decrementing CANSAVE and incrementing CANRESTORE.

If the new register window is occupied (that is, CANSAVE = 0), a spill trap is generated. The trap vector for the spill trap is based on the value of OTHERWIN and WSTATE. The spill trap handler is invoked with the CWP set to point to the window to be spilled (that is, old CWP + 2).

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

If CANSAVE ≠ 0, the SAVE instruction checks whether the new window needs to be cleaned. It causes a *clean_window* trap if the number of unused clean windows is zero, that is, (CLEANWIN – CANRESTORE) = 0. The *clean_window* trap handler is invoked with the CWP set to point to the window to be cleaned (that is, old CWP + 1).

Programming Note The vectoring of spill traps can be controlled by setting the value of the OTHERWIN and WSTATE registers appropriately. For details, see the section “Splitting the Register Windows” in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

The spill handler normally will end with a SAVED instruction followed by a RETRY instruction.

Exceptions

illegal_instruction
spill_n_normal (n = 0–7)
spill_n_other (n = 0–7)
clean_window

See Also

RESTORE on page 230

7.81 SAVED

Instruction	Operation	Assembly Language Syntax	Class
SAVED[P]	Window has been saved	`saved`	**A1**

10	fcn = 0 0000	11 0001	—
31 30	29 25	24 19	18 0

Description SAVED adjusts the state of the register-windows control registers.

SAVED increments CANSAVE. If OTHERWIN = 0, SAVED decrements CANRESTORE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

Programming Notes

Trap handler software for register window spills uses the SAVED instruction to indicate that a window has been spilled successfully. For details, see the section “Example Code for Spill Handler” in Software Considerations, contained in the separate volume *UltraSPARC Architecture Application Notes*.

Normal privileged software would probably not execute a SAVED instruction from trap level zero (TL = 0). However, it is not illegal to do so and doing so does not cause a trap.

Executing a SAVED instruction outside of a window spill trap handler is likely to create an inconsistent window state. Hardware will not signal an exception, however, since maintaining a consistent window state is the responsibility of privileged software.

If CANSAVE ≥ (*N_REG_WINDOWS* – 2) or CANRESTORE = 0 just prior to execution of a SAVED instruction, the subsequent behavior of the processor is undefined. In neither of these cases can SAVED generate a register window state that is both valid (see *Register Window State Definition* on page 60) 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 99
INVALW on page 173
NORMALW on page 213
OTHERW on page 215
RESTORED on page 232

7.82 Signed Divide (64-bit ÷ 32-bit)

> The SDIV and SDIVcc instructions are deprecated and should not be used in new software. The SDIVX instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax		Class
SDIV[D]	00 1111	Signed Integer Divide	`sdiv`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*	**D2**
SDIVcc[D]	01 1111	Signed Integer Divide and modify cc's	`sdivcc`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*	**D2**

10	rd	op3	rs1	i=0	—	rs2

10	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 5 4 0

Description The signed divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If i = 0, they compute "(Y :: R[rs1]{31:0}) ÷ R[rs2]{31:0}". Otherwise (that is, if i = 1), the divide instructions compute "(Y :: R[rs1]{31:0}) ÷ (**sign_ext**(simm13){31:0})". In either case, if overflow does not occur, the less significant 32 bits of the integer quotient are sign- or zero-extended to 64 bits and are written into R[rd].

The contents of the Y register are undefined after any 64-bit by 32-bit integer divide operation.

Signed Divide Signed divide (SDIV, SDIVcc) assumes a signed integer doubleword dividend (Y :: lower 32 bits of R[rs1]) and a signed integer word divisor (lower 32 bits of R[rs2] or lower 32 bits of **sign_ext**(simm13)) and computes a signed integer word quotient (R[rd]).

Signed division rounds an inexact quotient toward zero. For example, –7 ÷ 4 equals the rational quotient of –1.75, which rounds to –1 (not –2) when rounding toward zero.

The result of a signed divide can overflow the low-order 32 bits of the destination register R[rd] under certain conditions. When overflow occurs, the largest appropriate signed integer is returned as the quotient in R[rd]. The conditions under which overflow occurs and the value returned in R[rd] under those conditions are specified in TABLE 7-13.

TABLE 7-13 SDIV / SDIVcc Overflow Detection and Value Returned

Condition Under Which Overflow Occurs	Value Returned in R[rd]
Rational quotient $\geq 2^{31}$	$2^{31} - 1$ (0000 0000 7FFF FFFF$_{16}$)
Rational quotient $\leq -2^{31-1}$	-2^{31} (FFFF FFFF 8000 0000$_{16}$)

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

SDIV does not affect the condition code bits. SDIVcc writes the integer condition code bits as shown in the following table. Note that negative (N) and zero (Z) are set according to the value of R[rd] after it has been set to reflect overflow, if any.

Bit	Effect on bit of SDIVcc instruction
icc.n	Set to 1 if R[rd]{31} = 1; otherwise, set to 0
icc.z	Set to 1 if R[rd]{31:0} = 0; otherwise, set to 0
icc.v	Set to 1 if overflow (*per* TABLE 7-12); otherwise set to 0
icc.c	Set to 0
xcc.n	Set to 1 if R[rd]{63} = 1; otherwise, set to 0
xcc.z	Set to 1 if R[rd]{63:0} = 0; otherwise, set to 0
xcc.v	Set to 0
xcc.c	Set to 0

An attempt to execute an SDIV or SDIVcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

Exceptions

illegal_instruction
division_by_zero

See Also

MULScc on page 209
RDY on page 225
UDIV[cc] on page 281

7.83 SETHI

Instruction	op2	Operation	Assembly Language Syntax	Class
SETHI	100	Set High 22 Bits of Low Word	`sethi` *const22*, *reg*$_{rd}$ `sethi` `%hi` (*value*), *reg*$_{rd}$	**A1**

00	rd	op2	imm22
31 30	29 25	24 22	21 0

Description SETHI zeroes the least significant 10 bits and the most significant 32 bits of R[rd] and replaces bits 31 through 10 of R[rd] with the value from its imm22 field.

SETHI does not affect the condition codes.

Some SETHI instructions with rd = 0 have special uses:

- rd = 0 and imm22 = 0: defined to be a NOP instruction (described in *No Operation*)
- rd = 0 and imm22 ≠ 0 may be used to trigger hardware performance counters in some UltraSPARC Architecture implementations (for details, see implementation-specific documentation).

Programming Note The most common form of 64-bit constant generation is creating stack offsets whose magnitude is less than 2^{32}. The code below can be used to create the constant 0000 0000 ABCD 1234_{16}:

```
sethi   %hi(0xabcd1234),%o0
or      %o0, 0x234, %o0
```

The following code shows how to create a negative constant. **Note**: The immediate field of the xor instruction is sign extended and can be used to place 1's in all of the upper 32 bits. For example, to set the negative constant FFFF FFFF ABCD 1234_{16}:

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

Exceptions None

7.84 Set Interval Arithmetic Mode **VIS 2**

Instruction	opf	Operation	Assembly Language Syntax	Class
SIAM	0 1000 0001	Set the interval arithmetic mode fields in the GSR	`siam` *siam_mode*	**B1**

10	—	110110	—	opf	—	mode
31 30	29 25	24 19	18 14	13 5	4 3	2 0

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

GSR.im ← mode{2}

GSR.irnd ← mode{1:0}

> **Note** When GSR.im is set to 1, all subsequent floating-point instructions requiring round mode settings derive rounding-mode information from the General Status Register (GSR.irnd) instead of the Floating-Point State Register (FSR.rd).

> **Note** When GSR.im = 1, the processor operates in standard floating-point mode regardless of the setting of FSR.ns.

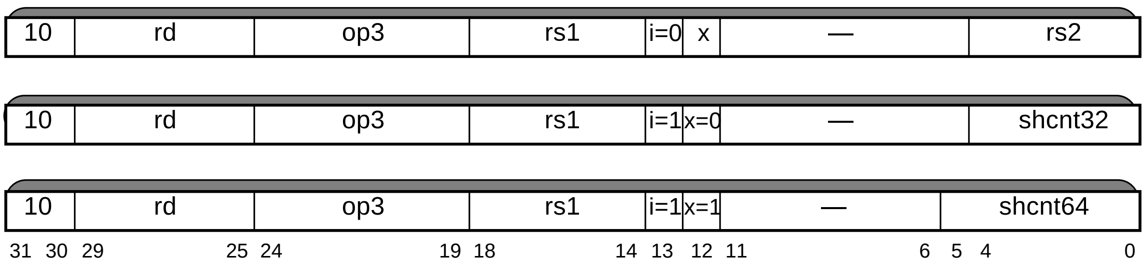
An attempt to execute a SIAM instruction when instruction bits 29:25, 18:14, or 4:3 are nonzero causes an *illegal_instruction* exception.

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

Exceptions *illegal_instruction*
fp_disabled

7.85 Shift

Instruction	op3	x	Operation	Assembly Language Syntax	Class
SLL	10 0101	0	Shift Left Logical – 32 bits	`sll` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$	**A1**
SRL	10 0110	0	Shift Right Logical – 32 bits	`srl` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$	**A1**
SRA	10 0111	0	Shift Right Arithmetic– 32 bits	`sra` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$	**A1**
SLLX	10 0101	1	Shift Left Logical – 64 bits	`sllx` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$	**A1**
SRLX	10 0110	1	Shift Right Logical – 64 bits	`srlx` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$	**A1**
SRAX	10 0111	1	Shift Right Arithmetic – 64 bits	`srax` *reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$	**A1**



Description These instructions perform logical or arithmetic shift operations.

When i = 0 and x = 0, the shift count is the least significant five bits of R[rs2].
When i = 0 and x = 1, the shift count is the least significant six bits of R[rs2]. When i = 1 and x = 0, the shift count is the immediate value specified in bits 0 through 4 of the instruction.
When i = 1 and x = 1, the shift count is the immediate value specified in bits 0 through 5 of the instruction.

TABLE 7-14 shows the shift count encodings for all values of i and x.

TABLE 7-14 Shift Count Encodings

i	x	Shift Count
0	0	bits 4–0 of R[rs2]
0	1	bits 5–0 of R[rs2]
1	0	bits 4–0 of instruction
1	1	bits 5–0 of instruction

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

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

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

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

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

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

These instructions do not modify the condition codes.

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

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

An attempt to execute a SLL, SRL, or SRA instruction when instruction bits 11:5 are nonzero causes an *illegal_instruction* exception.

An attempt to execute a SLLX, SRLX, or SRAX instruction when either of the following conditions exist causes an *illegal_instruction* exception:

- `i` = 0 or `x` = 0 and instruction bits 11:5 are nonzero
- `x` = 1 and instruction bits 11:6 are nonzero

Exceptions *illegal_instruction*

7.86 Signed Multiply (32-bit)

> The SMUL and SMULcc instructions are deprecated and should not be used in new software. The MULX instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
SMUL[D]	00 1011	Signed Integer Multiply	`smul` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**D2**
SMULcc[D]	01 1011	Signed Integer Multiply and modify cc's	`smulcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**D2**

10	rd	op3	rs1	i=0	—	rs2

10	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 5 4 0

Description The signed multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute "R[rs1]{31:0} × R[rs2]{31:0}" if i = 0, or "R[rs1]{31:0} × **sign_ext**(simm13){31:0}" if i = 1. They write the 32 most significant bits of the product into the Y register and all 64 bits of the product into R[rd].

Signed multiply instructions (SMUL, SMULcc) operate on signed integer word operands and compute a signed integer doubleword product.

SMUL does not affect the condition code bits. SMULcc writes the integer condition code bits, icc and xcc, as shown below.

Bit	Effect on bit by execution of SMULcc
icc.n	Set to 1 if product{31} = 1; otherwise, set to 0
icc.z	Set to 1 if product{31:0}= 0; otherwise, set to 0
icc.v	Set to 0
icc.c	Set to 0
xcc.n	Set to 1 if product{63} = 1; otherwise, set to 0
xcc.z	Set to 1 if product{63:0} = 0; otherwise, set to 0
xcc.v	Set to 0
xcc.c	Set to 0

Note 32-bit negative (icc.n) and zero (icc.z) condition codes are set according to the *less* significant word of the product, not according to the full 64-bit result.

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 283

7.87 Store Integer

Instruction	op3	Operation	Assembly Language Syntax	Class
STB	00 0101	Store Byte	`stb`† reg_{rd}, [*address*]	**A1**
STH	00 0110	Store Halfword	`sth`‡ reg_{rd}, [*address*]	**A1**
STW	00 0100	Store Word	`stw`◊ reg_{rd}, [*address*]	**A1**
STX	00 1110	Store Extended Word	`stx` reg_{rd}, [*address*]	**A1**

† *synonyms:* `stub`, `stsb` ‡ *synonyms:* `stuh`, `stsh` ◊ *synonyms:* `st`, `stuw`, `stsw`

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	

Description The store integer instructions (except store doubleword) copy the whole extended (64-bit) integer, the less significant word, the least significant halfword, or the least significant byte of R[rd] into memory.

These instructions access memory using the implicit ASI (see page 76). The effective address for these instructions is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + **sign_ext**(simm13)” if i = 1.

A successful store (notably, STX) integer instruction operates atomically.

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

STH causes a *mem_address_not_aligned* exception if the effective address is not halfword-aligned. STW causes a *mem_address_not_aligned* exception if the effective address is not word-aligned. STX causes a *mem_address_not_aligned* exception if the effective address is not doubleword-aligned.

Exceptions

illegal_instruction
mem_address_not_aligned
VA_watchpoint

See Also STTW on page 265

7.88 Store Integer into Alternate Space

Instruction	op3	Operation	Assembly Language Syntax	Class
STBA$^{\text{P}_{\text{ASI}}}$	01 0101	Store Byte into Alternate Space	`stba`† reg_{rd}, [*regaddr*] *imm_asi* `stba` reg_{rd}, [*reg_plus_imm*] `%asi`	**A1**
STHA$^{\text{P}_{\text{ASI}}}$	01 0110	Store Halfword into Alternate Space	`stha`‡ reg_{rd}, [*regaddr*] *imm_asi* `stha` reg_{rd}, [*reg_plus_imm*] `%asi`	**A1**
STWA$^{\text{P}_{\text{ASI}}}$	01 0100	Store Word into Alternate Space	`stwa`◊ reg_{rd}, [*regaddr*] *imm_asi* `stwa` reg_{rd}, [*reg_plus_imm*] `%asi`	**A1**
STXA$^{\text{P}_{\text{ASI}}}$	01 1110	Store Extended Word into Alternate Space	`stxa` reg_{rd}, [*regaddr*] *imm_asi* `stxa` reg_{rd}, [*reg_plus_imm*] `%asi`	**A1**

† *synonyms:* `stuba`, `stsba` ‡ *synonyms:* `stuha`, `stsha` ◊ *synonyms:* `sta`, `stuwa`, `stswa`

11	rd	op3	rs1	i=0	imm_asi	rs2

11	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 0

Description The store integer into alternate space instructions copy the whole extended (64-bit) integer, the less significant word, the least significant halfword, or the least significant byte of R[rd] into memory.

Store integer to alternate space instructions contain the address space identifier (ASI) to be used for the store in the imm_asi field if i = 0, or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1]+**sign_ext**(simm13)" if i = 1.

A successful store (notably, STXA) instruction operates atomically.

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

STHA causes a *mem_address_not_aligned* exception if the effective address is not halfword-aligned. STWA causes a *mem_address_not_aligned* exception if the effective address is not word-aligned. STXA causes a *mem_address_not_aligned* exception if the effective address is not doubleword-aligned.

STBA, STHA, and STWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

ASIs valid for STBA, STHA, and STWA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_REAL_IO	ASI_REAL_IO_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE

STXA can be used with any ASI (including, but not limited to, the above list), unless it either (a) violates the privilege mode rules described for the *privileged_action* exception above or (b) is used with any of the following ASIs, which causes a *DAE_invalid_asi* exception.

ASIs invalid for STXA	(cause *DAE_invalid_asi* exception)
ASI_BLOCK_AS_IF_USER_PRIMARY	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE
ASI_BLOCK_AS_IF_USER_SECONDARY	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE
ASI_BLOCK_AS_IF_USER_PRIMARY	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE
ASI_BLOCK_AS_IF_USER_SECONDARY	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE
ASI_PST8_PRIMARY	ASI_PST8_PRIMARY_LITTLE
ASI_PST8_SECONDARY	ASI_PST8_SECONDARY_LITTLE
ASI_PRIMARY_NO_FAULT	ASI_PRIMARY_NO_FAULT_LITTLE
ASI_SECONDARY_NO_FAULT	ASI_SECONDARY_NO_FAULT_LITTLE
ASI_PST16_PRIMARY	ASI_PST16_PRIMARY_LITTLE
ASI_PST16_SECONDARY	ASI_PST16_SECONDARY_LITTLE
ASI_PST32_PRIMARY	ASI_PST32_PRIMARY_LITTLE
ASI_PST32_SECONDARY	ASI_PST32_SECONDARY_LITTLE
ASI_FL8_PRIMARY	ASI_FL8_PRIMARY_LITTLE
ASI_FL8_SECONDARY	ASI_FL8_SECONDARY_LITTLE
ASI_FL16_PRIMARY	ASI_FL16_PRIMARY_LITTLE
ASI_FL16_SECONDARY	ASI_FL16_SECONDARY_LITTLE
ASI_BLOCK_COMMIT_PRIMARY	ASI_BLOCK_COMMIT_SECONDARY
ASI_BLOCK_PRIMARY	ASI_BLOCK_PRIMARY_LITTLE
ASI_BLOCK_SECONDARY	ASI_BLOCK_SECONDARY_LITTLE

V8 Compatibility Note | The SPARC V8 STA instruction was renamed STWA in the SPARC V9 architecture.

Exceptions

mem_address_not_aligned (all except STBA)
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nfo_page

See Also

LDA on page 176
STTWA on page 267

7.89 Block Store VIS 1

> The STBLOCKF instruction is intended to be a processor-specific instruction, which may or may not be implemented in future UltraSPARC Architecture implementations. Therefore, it should only be used in platform-specific dynamically-linked libraries or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

Instruction	ASI Value	Operation	Assembly Language Syntax	Class
STBLOCKF	16_{16}	64-byte block store to primary address space, user privilege	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_AIUP` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	A1 D2
STBLOCKF	17_{16}	64-byte block store to secondary address space, user privilege	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_AIUS` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	A1 D2
STBLOCKF	$1E_{16}$	64-byte block store to primary address space, little-endian, user privilege	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_AIUPL` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	A1 D2
STBLOCKF	$1F_{16}$	64-byte block store to secondary address space, little-endian, user privilege	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_AIUSL` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	A1 D2
STBLOCKF	$F0_{16}$	64-byte block store to primary address space	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_P` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	A1 D2
STBLOCKF	$F1_{16}$	64-byte block store to secondary address space	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_S` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	A1 D2
STBLOCKF	$F8_{16}$	64-byte block store to primary address space, little-endian	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_PL` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	A1 D2
STBLOCKF	$F9_{16}$	64-byte block store to secondary address space, little-endian	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_SL` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	A1 D2
STBLOCKF	$E0_{16}$	64-byte block commit store to primary address space	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_COMMIT_P` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	B1 D3
STBLOCKF	$E1_{16}$	64-byte block commit store to secondary address space	`stda` *freg*$_{rd}$, [*regaddr*] `#ASI_BLK_COMMIT_S` `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	B1 D3

11	rd	110111	rs1	i=0	imm_asi	rs2
11	rd	110111	rs1	i=1	simm_13	
31 30	29 25	24 19	18 14	13	5	4 0

Description A block store instruction references one of several special block-transfer ASIs. Block-transfer ASIs allow block stores to be performed accessing the same address space as normal stores. Little-endian ASIs (those with an ‘L’ suffix) access data in little-endian format; otherwise, the access is assumed to be big-endian. Byte swapping is performed separately for each of the eight double-precision registers accessed by the instruction.

Programming Note The block store instruction, STBLOCKF, and its companion, LDBLOCKF, were originally defined to provide a fast mechanism for block-copy operations.

STBLOCKF

STBLOCKF stores data from the eight double-precision floating-point registers specified by `rd` to a 64-byte-aligned memory area. The lowest-addressed eight bytes in memory are stored from the lowest-numbered double-precision `rd`.

While a STBLOCKF operation is in progress, any of the following values may be observed in a destination doubleword memory locations: (1) the old data value, (2) zero, or (3) the new data value. When the operation is complete, only the new data values will be seen.

> **Compatibility Note** Software written for older UltraSPARC implementations that reads data being written by STBLOCKF instructions may or may not allow for case (2) above. Such software should be checked to verify that either it always waits for STBLOCKF to complete before reading the values written, or that it will operate correctly if an intermediate value of zero (not the "old" or "new" data values) is observed while the STBLOCKF operation is in progress.

A Block Store only guarantees atomicity for each 64-bit (8-byte) portion of the 64 bytes that it stores.

A Block Store with Commit forces the data to be written to memory and invalidates copies in all caches present. As a result, a Block Store with Commit maintains coherency with the I-cache[1]. It does not, however, flush instructions that have already been fetched into the pipeline before executing the modified code. If a Block Store with Commit is used to write modified instructions, a FLUSH instruction must still be executed to guarantee that the instruction pipeline is flushed. (See *Synchronizing Instruction and Data Memory* on page 318 for more information.)

ASIs $E0_{16}$ and $E1_{16}$ are only used for block store-with-commit operations; they are not available for use by block load operations. See *Block Load and Store ASIs* on page 333 for more information.

Software should assume the following (where "load operation" includes load, load-store, and LDBLOCKF instructions and "store operation" includes store, load-store, and STBLOCKF instructions):

- A STBLOCKF does not follow memory ordering with respect to earlier or later load operations. If there is overlap between the addresses of destination memory locations of a STBLOCKF and the source address of a later load operation, the load operation may receive incorrect data. Therefore, if ordering with respect to later load operations is important, a MEMBAR `#StoreLoad` instruction must be executed between the STBLOCKF and subsequent load operations.
- A STBLOCKF does not follow memory ordering with respect to earlier or later store operations. Those instructions' data may commit to memory in a different order from the one in which those instructions were issued. Therefore, if ordering with respect to later store operations is important, a MEMBAR `#StoreStore` instruction must be executed between the STBLOCKF and subsequent store operations.
- STBLOCKFs do not follow register dependency interlocks, as do ordinary stores.

> **Programming Note** STBLOCKF is intended to be a processor-specific instruction (see the warning at the top of page 250). If STBLOCKF *must* be used in software intended to be portable across current and previous processor implementations, then it must be coded to work in the face of any implementation variation that is permitted by implementation dependency #411-S10, described below.

IMPL. DEP. #411-S10: The following aspects of the behavior of the block store (STBLOCKF) instruction are implementation dependent:

- The memory ordering model that STBLOCKF follows (other than as constrained by the rules outlined above).
- Whether *VA_watchpoint* exceptions are recognized on accesses to all 64 bytes of the STBLOCKF (the recommended behavior), or only on accesses to the first eight bytes.

[1] Even if all data stores on a given implementation coherently update the instruction cache (see page 389), stores (other than Block Store with Commit) on SPARC V9 implementations in general do *not* maintain coherency between instruction and data caches.

- Whether STBLOCKFs to non-cacheable (TTE.cp = 0) pages execute in strict program order or not. If not, a STBLOCKF to a non-cacheable page causes an *illegal_instruction* exception.
- Whether STBLOCKF follows register dependency interlocks (as ordinary stores do).
- Whether a non-Commit STBLOCKF forces the data to be written to memory and invalidates copies in all caches present (as the Commit variants of STBLOCKF do).
- Any other restrictions on the behavior of STBLOCKF, as described in implementation-specific documentation.

Exceptions. An *illegal_instruction* exception occurs if the source floating-point registers are not aligned on an eight-register boundary.

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

If the least significant 6 bits of the memory address are not all zero, a *mem_address_not_aligned* exception occurs.

In nonprivileged mode (PSTATE.priv = 0), if bit 7 of the ASI is 0 (ASIs 16_{16}, 17_{16}, $1E_{16}$, and $1F_{16}$), STBLOCKF causes a *privileged_action* exception.

An access caused by STBLOCKF may trigger a *VA_watchpoint* exception (impl. dep. #411-S10).

Implementation Note | STBLOCKF shares an opcode with the STDFA, STPARTIALF, and STSHORTF instructions; it is distinguished by the ASI used.

Exceptions

illegal_instruction
fp_disabled
mem_address_not_aligned
privileged_action
VA_watchpoint (impl. dep. #411-S10)
DAE_privilege_violation
DAE_nfo_page

See Also

LDBLOCKF on page 178

7.90 Store Floating-Point

Instruction	op3	rd	Operation	Assembly Language		Class
STF	10 0100	0–31	Store Floating-Point register	st	*freg*$_{rd}$, [*address*]	**A1**
STDF	10 0111	†	Store Double Floating-Point register	std	*freg*$_{rd}$, [*address*]	**A1**
STQF	10 0110	†	Store Quad Floating-Point register	stq	*freg*$_{rd}$, [*address*]	**C3**

† Encoded floating-point register value, as described on page 51.

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	

Description The store single floating-point instruction (STF) copies the contents of the 32-bit floating-point register F_S[rd] into memory.

The store double floating-point instruction (STDF) copies the contents of 64-bit floating-point register F_D[rd] into a word-aligned doubleword in memory. The unit of atomicity for STDF is 4 bytes (one word).

The store quad floating-point instruction (STQF) copies the contents of 128-bit floating-point register F_Q[rd] into a word-aligned quadword in memory. The unit of atomicity for STQF is 4 bytes (one word).

These instruction access memory using the implicit ASI (see page 76). The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

Exceptions. An attempt to execute a STF or STDF instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STF or STDF instruction causes an *fp_disabled* exception.

STF causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

STDF requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute an STDF instruction causes an *STDF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the STDF instruction and return (impl. dep. #110-V9-Cs10(a)).

STQF requires only word alignment in memory. If the effective address is word-aligned but not quadword-aligned, an attempt to execute an STQF instruction causes an *STQF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the STQF instruction and return (impl. dep. #112-V9-Cs10(a)).

Programming Note Some compilers issued sequences of single-precision stores for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned stores is expected to be fast, compilers should issue sets of single-precision stores only when they can determine that double- or quadword operands are *not* properly aligned.

STF / STDF / STQF / STXFSR

An attempt to execute an STQF instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

Implementation Note Since UltraSPARC Architecture 2007 processors do not implement in hardware instructions (including STQF) that refer to quad-precision floating-point registers, the *STQF_mem_address_not_aligned* and *fp_exception_other* (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an *illegal_instruction* exception and subsequent trap.

Exceptions

illegal_instruction
fp_disabled
STDF_mem_address_not_aligned
STQF_mem_address_not_aligned (not used in UltraSPARC Architecture 2007)
mem_address_not_aligned
fp_exception_other (FSR.ftt = invalid_fp_register (STQF only))
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

See Also

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

7.91 Store Floating-Point into Alternate Space

Instruction	op3	rd	Operation	Assembly Language Syntax	Class
STFA$^{P_{ASI}}$	11 0100	0–31	Store Floating-Point Register to Alternate Space	`sta` *freg*$_{rd}$, [*regaddr*] *imm_asi* `sta` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	**A1**
STDFA$^{P_{ASI}}$	11 0111	†	Store Double Floating-Point Register to Alternate Space	`stda` *freg*$_{rd}$, [*regaddr*] *imm_asi* `stda` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	**A1**
STQFA$^{P_{ASI}}$	11 0110	†	Store Quad Floating-Point Register to Alternate Space	`stqa` *freg*$_{rd}$, [*regaddr*] *imm_asi* `stqa` *freg*$_{rd}$, [*reg_plus_imm*] `%asi`	**C3**

† Encoded floating-point register value, as described on page 51.

31–30	29–25	24–19	18–14	13	12–5	4–0
11	rd	op3	rs1	i=0	imm_asi	rs2
11	rd	op3	rs1	i=1	simm13	

Description The store single floating-point into alternate space instruction (STFA) copies the contents of the 32-bit floating-point register F_S[rd] into memory.

The store double floating-point into alternate space instruction (STDFA) copies the contents of 64-bit floating-point register F_D[rd] into a word-aligned doubleword in memory. The unit of atomicity for STDFA is 4 bytes (one word).

The store quad floating-point into alternate space instruction (STQFA) copies the contents of 128-bit floating-point register F_Q[rd] into a word-aligned quadword in memory. The unit of atomicity for STQFA is 4 bytes (one word).

Store floating-point into alternate space instructions contain the address space identifier (ASI) to be used for the load in the imm_asi field if i = 0 or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

Programming Note Some compilers issued sequences of single-precision stores for SPARC V8 processor targets when the compiler could not determine whether doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned stores is expected to be fast, compilers should issue sets of single-precision stores only when they can determine that double- or quadword operands are *not* properly aligned.

Exceptions. STFA causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

STDFA requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute an STDFA instruction causes an *STDF_mem_address_not_aligned* exception. In this case, trap handler software must emulate the STDFA instruction and return (impl. dep. #110-V9-Cs10(b)).

STQFA requires only word alignment in memory. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an STQFA instruction may cause an *STQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STQFA instruction and return (impl. dep. #112-V9-Cs10(b)).

Implementation Note STDFA shares an opcode with the STBLOCKF, STPARTIALF, and STSHORTF instructions; it is distinguished by the ASI used.

An attempt to execute an STQFA instruction when rd{1} ≠ 0 causes an *fp_exception_other* (FSR.ftt = invalid_fp_register) exception.

Implementation Note Since UltraSPARC Architecture 2007 processors do not implement in hardware instructions (including STQFA) that refer to quad-precision floating-point registers, the *STQF_mem_address_not_aligned* and *fp_exception_other* (with FSR.ftt = invalid_fp_register) exceptions do not occur in hardware. However, their effects must be emulated by software when the instruction causes an *illegal_instruction* exception and subsequent trap.

In nonprivileged mode (PSTATE.priv = 0), 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.

STFA and STQFA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

ASIs valid for STFA and STQFA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_REAL_IO	ASI_REAL_IO_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE

STDFA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with the STDFA instruction causes a *DAE_invalid_asi* exception.

ASIs valid for STDFA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_REAL_IO	ASI_REAL_IO_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE
ASI_BLOCK_AS_IF_USER_PRIMARY †	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE †
ASI_BLOCK_AS_IF_USER_SECONDARY †	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE †
ASI_BLOCK_PRIMARY †	ASI_BLOCK_PRIMARY_LITTLE †
ASI_BLOCK_SECONDARY †	ASI_BLOCK_SECONDARY_LITTLE †
ASI_BLOCK_COMMIT_PRIMARY †	
ASI_BLOCK_COMMIT_SECONDARY †	
ASI_FL8_PRIMARY ‡	ASI_FL8_PRIMARY_LITTLE ‡
ASI_FL8_SECONDARY ‡	ASI_FL8_SECONDARY_LITTLE ‡
ASI_FL16_PRIMARY ‡	ASI_FL16_PRIMARY_LITTLE ‡
ASI_FL16_SECONDARY ‡	ASI_FL16_SECONDARY_LITTLE ‡
ASI_PST8_PRIMARY *	ASI_PST8_PRIMARY_LITTLE *
ASI_PST8_SECONDARY *	ASI_PST8_SECONDARY_LITTLE *
ASI_PST16_PRIMARY *	ASI_PST16_PRIMARY_LITTLE *
ASI_PST16_SECONDARY *	ASI_PST16_SECONDARY_LITTLE *
ASI_PST32_PRIMARY *	ASI_PST32_PRIMARY_LITTLE *

STFA / STDFA / STQFA

ASIs valid for STDFA	
`ASI_PST32_SECONDARY` *	`ASI_PST32_SECONDARY_LITTLE` *

† If this ASI is used with the opcode for STDFA, the STBLOCKF instruction is executed instead of STFA. For behavior of STBLOCKF, see *Block Store* on page 250.
‡ If this ASI is used with the opcode for STDFA, the STSHORTF instruction is executed instead of STDFA. For behavior of STSHORTF, see *Store Short Floating-Point* on page 263.
* If this ASI is used with the opcode for STDFA, the STPARTIALF instruction is executed instead of STDFA. For behavior of STPARTIALF, see *Store Partial Floating-Point* on page 260.

Exceptions

fp_disabled
STDF_mem_address_not_aligned
STQF_mem_address_not_aligned (STQFA only) (not used in UA-2007)
mem_address_not_aligned
fp_exception_other (FSR.ftt = invalid_fp_register (STQFA only))
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nfo_page

See Also

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

7.92 Store Floating-Point State Register (Lower)

> The STFSR instruction is deprecated and should not be used in new software. The STXFSR instruction should be used instead.

Opcode	op3	rd	Operation	Assembly Language Syntax	Class
STFSR[D]	10 0101	0	Store Floating-Point State Register (Lower)	`st %fsr,` [*address*]	**D2**
	10 0101	1-31	(see page 269)		

31 30	29 … 25	24 … 19	18 … 14	13	12 … 5	4 … 0
11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	

Description The Store Floating-point State Register (Lower) instruction (STFSR) waits for any currently executing FPop instructions to complete, and then it writes the less-significant 32 bits of FSR into memory.

After writing the FSR to memory, STFSR zeroes FSR.ftt

V9 Compatibility Note | FSR.ftt should not be zeroed until it is known that the store will not cause a precise trap.

STFSR accesses memory using the implicit ASI (see page 76). The effective address for this instruction is “R[rs1] + R[rs2]” if i = 0, or “R[rs1] + **sign_ext**(simm13)” if i = 1.

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

If the floating-point unit is not enabled (FPRS.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, UltraSPARC Architecture implementations continue to support it for compatibility with existing SPARC V8 software. The STFSR instruction is defined to store only the less-significant 32 bits of the FSR into memory, while STXFSR allows SPARC V9 software to store all 64 bits of the FSR.

Implementation Note | STFSR shares an opcode with the STXFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = 10_2, op3 = 10 0101_2 opcode with an invalid rd value causes an *illegal_instruction* exception.

Exceptions
illegal_instruction
fp_disabled
mem_address_not_aligned
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

STFSR (Deprecated)

See Also

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

7.93 Store Partial Floating-Point VIS 1

Instruction	ASI Value	Operation	Assembly Language Syntax †	Class
STPARTIALF	$C0_{16}$	Eight 8-bit conditional stores to primary address space	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST8_P	**B1**
STPARTIALF	$C1_{16}$	Eight 8-bit conditional stores to secondary address space	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST8_S	**B1**
STPARTIALF	$C8_{16}$	Eight 8-bit conditional stores to primary address space, little-endian	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST8_PL	**B1**
STPARTIALF	$C9_{16}$	Eight 8-bit conditional stores to secondary address space, little-endian	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST8_SL	**B1**
STPARTIALF	$C2_{16}$	Four 16-bit conditional stores to primary address space	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST16_P	**B1**
STPARTIALF	$C3_{16}$	Four 16-bit conditional stores to secondary address space	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST16_S	**B1**
STPARTIALF	CA_{16}	Four 16-bit conditional stores to primary address space, little-endian	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST16_PL	**B1**
STPARTIALF	CB_{16}	Four 16-bit conditional stores to secondary address space, little-endian	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST16_SL	**B1**
STPARTIALF	$C4_{16}$	Two 32-bit conditional stores to primary address space	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST32_P	**B1**
STPARTIALF	$C5_{16}$	Two 32-bit conditional stores to secondary address space	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST32_S	**B1**
STPARTIALF	CC_{16}	Two 32-bit conditional stores to primary address space, little-endian	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST32_PL	**B1**
STPARTIALF	CD_{16}	Two 32-bit conditional stores to secondary address space, little-endian	stda $freg_{rd}$, reg_{rs2}, [reg_{rs1}] #ASI_PST32_SL	**B1**

† The original assembly language syntax for a Partial Store instruction ("stda $freg_{rd}$, [reg_{rs1}] reg_{rs2}, imm_asi") has been deprecated because of inconsistency with the rest of the SPARC assembly language. Over time, assemblers will support the new syntax for this instruction. In the meantime, some existing assemblers may only recognize the original syntax.

11	rd	110111	rs1	i=0	imm_asi	rs2
31 30	29 25	24 19	18 14	13	5	4 0

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_D[rd] are conditionally stored at the address specified by R[rs1], using the mask specified in R[rs2]. STPARTIALF has the effect of merging selected data from its source register, F_D[rd], into the existing data at the corresponding destination locations.

The mask value in R[rs2] has the same format as the result specified by the pixel compare instructions (see *SIMD Signed Compare* on page 126). The most significant bit of the mask (not of the entire register) corresponds to the most significant part of F_D[rd]. The data is stored in little-endian form in memory if the ASI name has an "L" (or "_LITTLE") suffix; otherwise, it is stored in big-endian format.

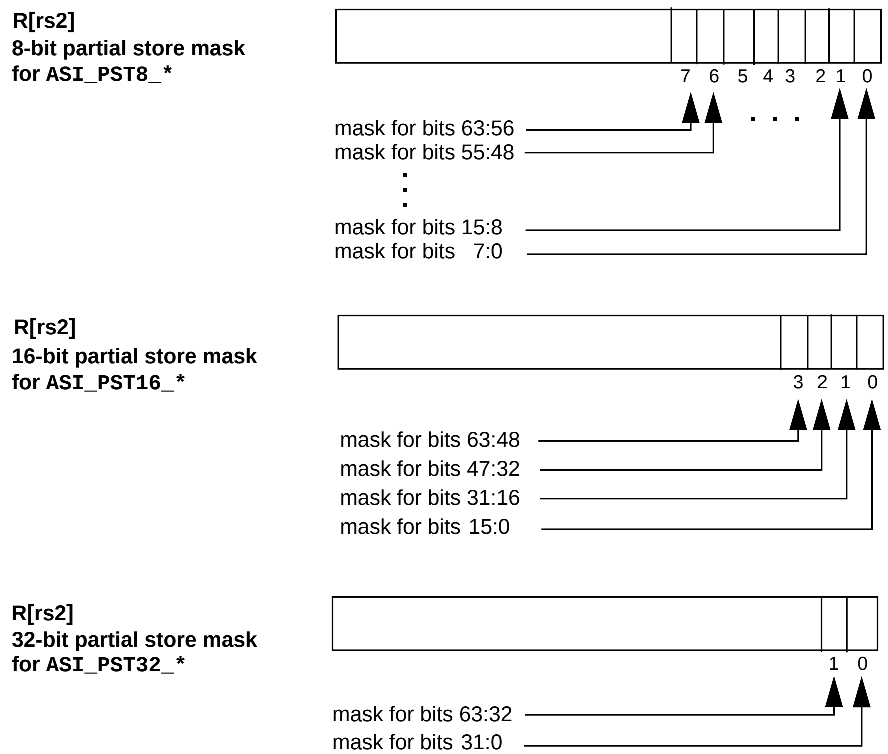


FIGURE 7-29 Mask Format for Partial Store

Exceptions. A Partial Store instruction can cause a virtual watchpoint exception when the following conditions are met:

- The virtual address in R[rs1] matches the address in the VA Data Watchpoint Register.
- The byte store mask in R[rs2] indicates that a byte, halfword or word is to be stored.
- The Virtual (Physical) Data Watchpoint Mask in `ASI_DCU_WATCHPOINT_CONTROL_REG` indicates that one or more of the bytes to be stored at the watched address is being watched.

For data watchpoints of partial stores in UltraSPARC Architecture 2007, the byte store mask (R[rs2]) in the Partial Store instruction is ignored, and a watchpoint exception can occur even if the mask is zero (that is, no store will take place). The `ASI_DCU_WATCHPOINT_CONTROL_REG` Data Watchpoint masks are only checked for nonzero value (watchpoint enabled) (impl. dep. #249).

An attempt to execute a STPARTIALF instruction when i = 1 causes an *illegal_instruction* exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STPARTIALF instruction causes an *fp_disabled* exception.

STPARTIALF causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned.

STPARTIALF requires only word alignment in memory for eight byte stores. If the effective address is word-aligned but not doubleword-aligned, it generates an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the STDFA instruction and return.

STPARTIALF

IMPL. DEP. #249-U3-Cs10: For an STPARTIAL instruction, the following aspects of data watchpoints are implementation dependent: (a) whether data watchpoint logic examines the byte store mask in R[rs2] or it conservatively behaves as if every Partial Store always stores all 8 bytes, and (b) whether data watchpoint logic examines individual bits in the Virtual (Physical) Data Watchpoint Mask in the LSU Control register DCUCR to determine which bytes are being watched or (when the Watchpoint Mask is nonzero) it conservatively behaves as if all 8 bytes are being watched.

ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–CD_{16} are only used for partial store operations. In particular, they should not be used with the LDDFA instruction; however, if any of them *is* used, the resulting behavior is specified in the LDDFA instruction description on page 185.

Implementation Note | STPARTIALF shares an opcode with the STBLOCKF, STDFA, and STSHORTF instructions; it is distinguished by the ASI used.

Exceptions

illegal_instruction
fp_disabled
mem_address_not_aligned
VA_watchpoint (see text)
DAE_privilege_violation
DAE_nc_page
DAE_nfo_page

7.94 Store Short Floating-Point VIS 1

Instruction	ASI Value	Operation	Assembly Language Syntax	Class
STSHORTF	$D0_{16}$	8-bit store to primary address space	`stda` $freg_{rd}$, [*regaddr*] `#ASI_FL8_P` `stda` $freg_{rd}$, [*reg_plus_imm*] `%asi`	**B1** **D2**
STSHORTF	$D1_{16}$	8-bit store to secondary address space	`stda` $freg_{rd}$, [*regaddr*] `#ASI_FL8_S` `stda` $freg_{rd}$, [*reg_plus_imm*] `%asi`	**B1** **D2**
STSHORTF	$D8_{16}$	8-bit store to primary address space, little-endian	`stda` $freg_{rd}$, [*regaddr*] `#ASI_FL8_PL` `stda` $freg_{rd}$, [*reg_plus_imm*] `%asi`	**B1** **D2**
STSHORTF	$D9_{16}$	8-bit store to secondary address space, little-endian	`stda` $freg_{rd}$, [*regaddr*] `#ASI_FL8_SL` `stda` $freg_{rd}$, [*reg_plus_imm*] `%asi`	**B1** **D2**
STSHORTF	$D2_{16}$	16-bit store to primary address space	`stda` $freg_{rd}$, [*regaddr*] `#ASI_FL16_P` `stda` $freg_{rd}$, [*reg_plus_imm*] `%asi`	**B1** **D2**
STSHORTF	$D3_{16}$	16-bit store to secondary address space	`stda` $freg_{rd}$, [*regaddr*] `#ASI_FL16_S` `stda` $freg_{rd}$, [*reg_plus_imm*] `%asi`	**B1** **D2**
STSHORTF	DA_{16}	16-bit store to primary address space, little-endian	`stda` $freg_{rd}$, [*regaddr*] `#ASI_FL16_PL` `stda` $freg_{rd}$, [*reg_plus_imm*] `%asi`	**B1** **D2**
STSHORTF	DB_{16}	16-bit store to secondary address space, little-endian	`stda` $freg_{rd}$, [*regaddr*] `#ASI_FL16_SL` `stda` $freg_{rd}$, [*reg_plus_imm*] `%asi`	**B1** **D2**

11	rd	110111	rs1	i=0	imm_asi	rs2

11	rd	110111	rs1	i=1	simm_13
31 30	29 25	24 19	18 14	13	5 4 0

Description The short floating-point store instruction allows 8- and 16-bit stores to be performed from the floating-point registers. Short stores access the low-order 8 or 16 bits of the register.

Little-endian ASIs transfer data in little-endian format from memory; otherwise, memory is assumed to be big-endian. Short stores are typically used with the FALIGNDATA instruction (see *Align Data* on page 121) to assemble or store 64 bits on noncontiguous components.

> **Implementation Note** | STSHORTF shares an opcode with the STBLOCKF, STDFA, and STPARTIALF instructions; it is distinguished by the ASI used.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STSHORTF instruction causes an *fp_disabled* exception.

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

An 8-bit STSHORTF (using ASI $D0_{16}$, $D1_{16}$, $D8_{16}$, or $D9_{16}$) can be performed to an arbitrary memory address (no alignment requirement).

A 16-bit STSHORTF (using ASI $D2_{16}$, $D3_{16}$, DA_{16}, or DB_{16}) to an address that is not halfword-aligned (an odd address) causes a *mem_address_not_aligned* exception.

Exceptions *fp_disabled*
mem_address_not_aligned
VA_watchpoint

DAE_privilege_violation
DAE_nfo_page

7.95 Store Integer Twin Word

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

Opcode	op3	Operation	Assembly Language Syntax †		Class
STTW[D]	00 0111	Store Integer Twin Word	`sttw`	reg_{rd}, [*address*]	**D2**

† The original assembly language syntax for this instruction used an "`std`" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "`sttw`" mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original "`std`" mnemonic.

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	

Description The store integer twin word instruction (STTW) copies two words from an R register pair into memory. The least significant 32 bits of the even-numbered R register are written into memory at the effective address, and the least significant 32 bits of the following odd-numbered R register are written into memory at the "effective address + 4".

The least significant bit of the rd field of a store twin word instruction is unused and should always be set to 0 by software.

STTW accesses memory using the implicit ASI (see page 76). The effective address for this instruction is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

A successful store twin word instruction operates atomically.

IMPL. DEP. #108-V9a: It is implementation dependent whether STTW is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_STTW* exception. (STTW is implemented in hardware in all UltraSPARC Architecture 2007 implementations.)

An attempt to execute an STTW instruction when either of the following conditions exist causes an *illegal_instruction* exception:

- destination register number rd is an odd number (is misaligned)
- i = 0 and instruction bits 12:5 are nonzero

STTW causes a *mem_address_not_aligned* exception if the effective address is not doubleword-aligned.

With respect to little-endian memory, an STTW instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.

Programming Notes STTW is provided for compatibility with SPARC V8. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties. Therefore, software should avoid using STTW.

If STTW is emulated in software, STX instruction should be used for the memory access in the emulation code to preserve atomicity.

STTW (Deprecated)

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

See Also STW/STX on page 247
STTWA on page 267

7.96 Store Integer Twin Word into Alternate Space

> The STTWA instruction is deprecated and should not be used in new software. The STXA instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
STTWA[D, PASI]	01 0111	Store Twin Word into Alternate Space	`sttwa` *reg*$_{rd}$ [*regaddr*] *imm_asi* `sttwa` *reg*$_{rd}$ [*reg_plus_imm*] `%asi`	**D2, Y3**‡

† The original assembly language syntax for this instruction used an "`stda`" instruction mnemonic, which is now deprecated. Over time, assemblers will support the new "`sttwa`" mnemonic for this instruction. In the meantime, some existing assemblers may only recognize the original "`stda`" mnemonic.

‡ **Y3** for restricted ASIs (00_{16}-$7F_{16}$); **D2** for unrestricted ASIs (80_{16}-FF_{16})

11	rd	op3	rs1	i=0	imm_asi	rs2

11	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 5 4 0

Description The store twin word integer into alternate space instruction (STTWA) copies two words from an R register pair into memory. The least significant 32 bits of the even-numbered R register are written into memory at the effective address, and the least significant 32 bits of the following odd-numbered R register are written into memory at the "effective address + 4".

The least significant bit of the rd field of an STTWA instruction is unused and should always be set to 0 by software.

Store integer twin word to alternate space instructions contain the address space identifier (ASI) to be used for the store in the imm_asi field if i = 0, or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1]+**sign_ext**(simm13)" if i = 1.

A successful store twin word instruction operates atomically.

With respect to little-endian memory, an STTWA instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into its respective destination memory word.

IMPL. DEP. #108-V9b: It is implementation dependent whether STTWA is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_STTW* exception. (STTWA is implemented in hardware in all UltraSPARC Architecture 2007 implementations.)

An attempt to execute an STTWA instruction with a misaligned (odd) destination register number rd causes an *illegal_instruction* exception.

STTWA causes a *mem_address_not_aligned* exception if the effective address is not doubleword-aligned.

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

STTWA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *DAE_invalid_asi* exception (impl. dep. #300-U4-Cs10).

ASIs valid for STTWA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_REAL_IO	ASI_REAL_IO_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE

Programming Note Nontranslating ASIs (see page 321) may only be accessed using STXA (not STTWA) instructions. If an STTWA referencing a nontranslating ASI is executed, per the above table, it generates a *DAE_invalid_asi* exception (impl. dep. #300-U4-Cs10).

Programming Note STTWA is provided for compatibility with existing SPARC V8 software. It may execute slowly on SPARC V9 machines because of data path and register-access difficulties. Therefore, software should avoid using STTWA.

If STTWA is emulated in software, the STXA instruction should be used for the memory access in the emulation code to preserve atomicity.

Exceptions

unimplemented_STTW
illegal_instruction
mem_address_not_aligned
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nfo_page

See Also

STWA/STXA on page 248
STTW on page 265

7.97 Store Floating-Point State Register

Instruction	op3	rd	Operation	Assembly Language		Class
	10 0101	0	(see page 258)			
STXFSR	10 0101	1	Store Floating-Point State register	`stx`	`%fsr, [`*address*`]`	**A1**
—	10 0101	2–31	*Reserved*			

11	rd	op3	rs1	i=0	—	rs2
11	rd	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

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 76). The effective address for this instruction is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

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

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a STXFSR instruction causes an *fp_disabled* exception.

If the effective address is not doubleword-aligned, an attempt to execute an STXFSRinstruction causes a *mem_address_not_aligned* exception.

> **Implementation Note** STXFSR shares an opcode with the (deprecated) STFSR instruction (and possibly with other implementation-dependent instructions); they are differentiated by the instruction rd field. An attempt to execute the op = 10_2, op3 = 10 0101_2 opcode with an invalid rd value causes an *illegal_instruction* exception.

Exceptions

illegal_instruction
fp_disabled
mem_address_not_aligned
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

See Also

Load Floating-Point State Register on page 199
Store Floating-Point on page 253
Store Floating-Point State Register (Lower) on page 258

7.98 Subtract

Instruction	op3	Operation	Assembly Language Syntax	Class
SUB	00 0100	Subtract	`sub` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
SUBcc	01 0100	Subtract and modify cc's	`subcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
SUBC	00 1100	Subtract with Carry	`subc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**
SUBCcc	01 1100	Subtract with Carry and modify cc's	`subccc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	

Description These instructions compute "R[rs1] – R[rs2]" if i = 0, or "R[rs1] – **sign_ext**(simm13)" if i = 1, and write the difference into R[rd].

SUBC and SUBCcc ("SUBtract with carry") also subtract the CCR register's 32-bit carry (icc.c) bit; that is, they compute "R[rs1] – R[rs2] – icc.c" or "R[rs1] – **sign_ext**(simm13) – icc.c" and write the difference into R[rd].

SUBcc and SUBCcc modify the integer condition codes (CCR.icc and CCR.xcc). A 32-bit overflow (CCR.icc.v) occurs on subtraction if bit 31 (the sign) of the operands differs and bit 31 (the sign) of the difference differs from R[rs1]{31}. A 64-bit overflow (CCR.xcc.v) occurs on subtraction if bit 63 (the sign) of the operands differs and bit 63 (the sign) of the difference differs from R[rs1]{63}.

Programming Notes A SUBcc instruction with rd = 0 can be used to effect a signed or unsigned integer comparison. See the `cmp` synthetic instruction in Appendix C, *Assembly Language Syntax*.

SUBC and SUBCcc read the 32-bit condition codes' carry bit (CCR.icc.c), not the 64-bit condition codes' carry bit (CCR.xcc.c).

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

Exceptions *illegal_instruction*

7.99 Swap Register with Memory

> The SWAP instruction is deprecated and should not be used in new software. The CASA or CASXA instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
SWAP[D]	00 1111	Swap Register with Memory	`swap` [*address*], *reg*$_{rd}$	**D2**

11	rd	op3	rs1	i=0	—	rs2

11	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 5 4 0

Description SWAP exchanges the less significant 32 bits of R[rd] with the contents of the word at the addressed memory location. The upper 32 bits of R[rd] are set to 0. The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA instructions addressing any or all of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

SWAP accesses memory using the implicit ASI (see page 76). The effective address for these instructions is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

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

If the effective address is not word-aligned, an attempt to execute a SWAP instruction causes a *mem_address_not_aligned* exception.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep. #120-V9).

Exceptions
illegal_instruction
mem_address_not_aligned
VA_watchpoint
DAE_privilege_violation
DAE_nfo_page

7.100 Swap Register with Alternate Space Memory

> The SWAPA instruction is deprecated and should not be used in new software. The CASXA instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
SWAPA[D, PASI]	01 1111	Swap register with Alternate Space Memory	`swapa` [*regaddr*] *imm_asi*, reg_{rd} `swapa` [*reg_plus_imm*] `%asi`, reg_{rd}	**D2, Y3‡**

‡ **Y3** for restricted ASIs (00_{16}-$7F_{16}$); **D2** for unrestricted ASIs (80_{16}-FF_{16})

11	rd	op3	rs1	i=0	imm_asi	rs2

11	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 5 4 0

Description SWAPA exchanges the less significant 32 bits of R[rd] with the contents of the word at the addressed memory location. The upper 32 bits of R[rd] are set to 0. The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more virtual processors executing CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA instructions addressing any or all of the same doubleword simultaneously are guaranteed to execute them in an undefined, but serial, order.

The SWAPA instruction contains the address space identifier (ASI) to be used for the load in the imm_asi field if i = 0, or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for this instruction is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

This instruction causes a *mem_address_not_aligned* exception if the effective address is not word-aligned. It causes a *privileged_action* exception if PSTATE.priv = 0 and bit 7 of the ASI is 0.

The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent (impl. dep #120-V9).

If the effective address is not word-aligned, an attempt to execute a SWAPA instruction causes a *mem_address_not_aligned* exception.

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

SWAPA can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception above. Use of any other ASI with this instruction causes a *DAE_invalid_asi* exception.

ASIs valid for SWAPA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE

Exceptions *mem_address_not_aligned*
privileged_action
VA_watchpoint
DAE_invalid_asi
DAE_privilege_violation
DAE_nc_page
DAE_nfo_page

7.101 Tagged Add

Instruction	op3	Operation	Assembly Language Syntax	Class
TADDcc	10 0000	Tagged Add and modify cc's	`taddcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**

10	rd	op3	rs1	i=0	—	rs2

10	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 0

Description This instruction computes a sum that is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

TADDcc modifies the integer condition codes (`icc` and `xcc`).

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the addition generates 32-bit arithmetic overflow (that is, both operands have the same value in bit 31 and bit 31 of the sum is different).

If a TADDcc causes a tag overflow, the 32-bit overflow bit (CCR.icc.v) is set to 1; if TADDcc does not cause a tag overflow, CCR.icc.v is set to 0.

In either case, the remaining integer condition codes (both the other CCR.icc bits and all the CCR.xcc bits) are also updated as they would be for a normal ADD instruction. In particular, the setting of the CCR.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). CCR.xcc.v is set based on the 64-bit arithmetic overflow condition, like a normal 64-bit add.

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

Exceptions *illegal_instruction*

See Also TADDccTV[D] on page 275
TSUBcc on page 279

7.102 Tagged Add and Trap on Overflow

> The TADDccTV instruction is deprecated and should not be used in new software. The TADDcc instruction followed by the BPVS instruction (with instructions to save the pre-TADDcc integer condition codes if necessary) should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
TADDccTV[D]	10 0010	Tagged Add and modify cc's or Trap on Overflow	`taddcctv` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**D2**

10	rd	op3	rs1	i=0	—	rs2

10	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 5 4 0

Description This instruction computes a sum that is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1.

TADDccTV modifies the integer condition codes if it does not trap.

An attempt to execute a TADDccTV instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the addition generates 32-bit arithmetic overflow (that is, both operands have the same value in bit 31 and bit 31 of the sum is different).

If TADDccTV causes a tag overflow, a *tag_overflow* exception is generated and R[rd] and the integer condition codes remain unchanged. If a TADDccTV does not cause a tag overflow, the sum is written into R[rd] and the integer condition codes are updated. CCR.icc.v is set to 0 to indicate no 32-bit overflow.

In either case, the remaining integer condition codes (both the other CCR.icc bits and all the CCR.xcc bits) are also updated as they would be for a normal ADD instruction. In particular, the setting of the CCR.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). CCR.xcc.v is set only on the basis of the normal 64-bit arithmetic overflow condition, like a normal 64-bit add.

SPARC V8 Compatibility Note | TADDccTV traps based on the 32-bit overflow condition, just as in the SPARC V8 architecture. Although the tagged add instructions set the 64-bit condition codes CCR.xcc, there is no form of the instruction that traps on the 64-bit overflow condition.

Exceptions *illegal_instruction*
tag_overflow

See Also TADDcc on page 274
TSUBccTV[D] on page 280

7.103 Trap on Integer Condition Codes (Tcc)

Instruction	op3	cond	Operation	cc Test	Assembly Language Syntax	Class
TA	11 1010	1000	Trap Always	1	`ta` *i_or_x_cc*, *software_trap_number*	**A1**
TN	11 1010	0000	Trap Never	0	`tn` *i_or_x_cc*, *software_trap_number*	**A1**
TNE	11 1010	1001	Trap on Not Equal	**not** Z	`tne`† *i_or_x_cc*, *software_trap_number*	**A1**
TE	11 1010	0001	Trap on Equal	Z	`te`‡ *i_or_x_cc*, *software_trap_number*	**A1**
TG	11 1010	1010	Trap on Greater	**not** (Z **or** (N **xor** V))	`tg` *i_or_x_cc*, *software_trap_number*	**A1**
TLE	11 1010	0010	Trap on Less or Equal	Z **or** (N **xor** V)	`tle` *i_or_x_cc*, *software_trap_number*	**A1**
TGE	11 1010	1011	Trap on Greater or Equal	**not** (N **xor** V)	`tge` *i_or_x_cc*, *software_trap_number*	**A1**
TL	11 1010	0011	Trap on Less	N **xor** V	`tl` *i_or_x_cc*, *software_trap_number*	**A1**
TGU	11 1010	1100	Trap on Greater, Unsigned	**not** (C **or** Z)	`tgu` *i_or_x_cc*, *software_trap_number*	**A1**
TLEU	11 1010	0100	Trap on Less or Equal, Unsigned	(C **or** Z)	`tleu` *i_or_x_cc*, *software_trap_number*	**A1**
TCC	11 1010	1101	Trap on Carry Clear (Greater than or Equal, Unsigned)	**not** C	`tcc`◊ *i_or_x_cc*, *software_trap_number*	**A1**
TCS	11 1010	0101	Trap on Carry Set (Less Than, Unsigned)	C	`tcs`∇ *i_or_x_cc*, *software_trap_number*	**A1**
TPOS	11 1010	1110	Trap on Positive or zero	**not** N	`tpos` *i_or_x_cc*, *software_trap_number*	**A1**
TNEG	11 1010	0110	Trap on Negative	N	`tneg` *i_or_x_cc*, *software_trap_number*	**A1**
TVC	11 1010	1111	Trap on Overflow Clear	**not** V	`tvc` *i_or_x_cc*, *software_trap_number*	**A1**
TVS	11 1010	0111	Trap on Overflow Set	V	`tvs` *i_or_x_cc*, *software_trap_number*	**A1**

† *synonym:* `tnz` ‡ *synonym:* `tz` ◊ *synonym:* `tgeu` ∇ *synonym:* `tlu`

31 30	29	28 – 25	24 – 19	18 – 14	13	12	11	10 – 8	7 – 5	4 – 0
10	—	cond	op3	rs1	i=0	cc1	cc0	—	—	rs2
10	—	cond	op3	rs1	i=1	cc1	cc0	—	imm_trap_#	imm_trap_#

cc1 :: cc0	Condition Codes Evaluated
00	CCR.icc
01	— (*illegal_instruction*)
10	CCR.xcc
11	— (*illegal_instruction*)

Tcc

Description

The Tcc instruction evaluates the selected integer condition codes (icc or xcc) according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE and no higher-priority exceptions or interrupt requests are pending, then a *trap_instruction* or *htrap_instruction* exception is generated. If FALSE, the *trap_instruction* (or *htrap_instruction*) exception does not occur and the instruction behaves like a NOP.

For brevity, in the remainder of this section the value of the "software trap number" used by Tcc will be referred to as "SWTN".

In nonprivileged mode, if i = 0 the SWTN is specified by the least significant seven bits of "R[rs1] + R[rs2]". If i = 1, the SWTN is provided by the least significant seven bits of "R[rs1] + imm_trap_#". Therefore, the valid range of values for SWTN in nonprivileged mode is 0 to 127. The most significant 57 bits of SWTN are unused and should be supplied as zeroes by software.

In privileged mode, if i = 0 the SWTN is specified by the least significant eight bits of "R[rs1] + R[rs2]". If i = 1, the SWTN is provided by the least significant eight bits of "R[rs1] + *imm_trap_#*". Therefore, the valid range of values for SWTN in privileged mode is 0 to 255. The most significant 56 bits of SWTN are unused an should be supplied as zeroes by software.

Generally, values of 0 ≤ SWTN ≤ 127 are used to trap to privileged-mode software and values of 128 ≤ SWTN ≤ 255 are used to trap to hyperprivileged-mode software. The behavior of Tcc, based on the privilege mode in effect when it is executed and the value of the supplied SWTN, is as follows:

	Behavior of Tcc instruction	
Privilege Mode in effect when Tcc is executed	**0 ≤ SWTN ≤ 127**	**128 ≤ SWTN ≤ 255**
Nonprivileged (PSTATE.priv = 0)	*trap_instruction* exception (to privileged mode) (256 ≤ TT ≤ 383)	— (not possible, because SWTN is a 7-bit value in nonprivileged mode)
Privileged (PSTATE.priv = 1)	*trap_instruction* exception (to privileged mode) (256 ≤ TT ≤ 383)	*htrap_instruction* exception (to hyperprivileged mode) (384 ≤ TT ≤ 511)

Programming Note Tcc can be used to implement breakpointing, tracing, and calls to privileged and hyperprivileged software. It can also be used for runtime checks, such as for out-of-range array indexes and integer overflow.

Exceptions. An attempt to execute a Tcc instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- instruction bit 29 is nonzero
- i = 0 and instruction bits 10:5 are nonzero
- i = 1 and instruction bits 10:8 are nonzero
- cc0 = 1

If the Trap on Control Transfer feature is implemented (impl. dep. #450-S20) and PSTATE.tct = 1, then Tcc generates a *control_transfer_instruction* exception instead of causing a control transfer. When a *control_transfer_instruction* trap occurs, PC (the address of the Tcc instruction) is stored in TPC[TL] and the value of NPC from before the Tcc was executed is stored in TNPC[TL]. The full 64-bit (nonmasked) PC and NPC values are stored in TPC[TL] and TNPC[TL], regardless of the value of PSTATE.am.

If a Tcc instruction causes a *trap_instruction* trap, 256 plus the SWTN value is written into TT[TL]. Then the trap is taken and the virtual processor performs the normal trap entry procedure, as described in *Trap Processing* on page 356.

Exceptions *illegal_instruction*
control_transfer_instruction (impl. dep. #450-S20)
trap_instruction (0 ≤ SWTN ≤ 127)
htrap_instruction (128 ≤ SWTN ≤ 255)

7.104 Tagged Subtract

Instruction	op3	Operation	Assembly Language Syntax	Class
TSUBcc	10 0001	Tagged Subtract and modify cc's	`tsubcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**A1**

10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

Description This instruction computes "R[rs1] – R[rs2]" if i = 0, or "R[rs1] – **sign_ext**(simm13)" if i = 1.

TSUBcc modifies the integer condition codes (icc and xcc).

A tag overflow condition occurs if bit 1 or bit 0 of either operand is nonzero or if the subtraction generates 32-bit arithmetic overflow; that is, the operands have different values in bit 31 (the 32-bit sign bit) and the sign of the 32-bit difference in bit 31 differs from bit 31 of R[rs1].

If a TSUBcc causes a tag overflow, the 32-bit overflow bit (CCR.icc.v) is set to 1; if TSUBcc does not cause a tag overflow, CCR.icc.v is set to 0.

In either case, the remaining integer condition codes (both the other CCR.icc bits and all the CCR.xcc bits) are also updated as they would be for a normal subtract instruction. In particular, the setting of the CCR.xcc.v bit is not determined by the tag overflow condition (tag overflow is used only to set the 32-bit overflow bit). ccr.xcc.v is set based on the 64-bit arithmetic overflow condition, like a normal 64-bit subtract.

An attempt to execute a TSUBcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

Exceptions *illegal_instruction*

See Also TADDcc on page 274
TSUBccTV^{D} on page 280

7.105 Tagged Subtract and Trap on Overflow

> The TSUBccTV instruction is deprecated and should not be used in new software. The TSUBcc instruction followed by BPVS instead (with instructions to save the pre-TSUBcc integer condition codes if necessary) should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
TSUBccTV[D]	10 0011	Tagged Subtract and modify cc's or Trap on Overflow	`tsubcctv` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**D2**

10	rd	op3	rs1	i=0	—	rs2

10	rd	op3	rs1	i=1	simm13
31 30	29 25	24 19	18 14	13	12 0

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.

SPARC V8 Compatibility Note TSUBccTV traps based on the 32-bit overflow condition, just as in the SPARC V8 architecture. Although the tagged add instructions set the 64-bit condition codes CCR.xcc, there is no form of the instruction that traps on the 64-bit overflow condition.

Exceptions *illegal_instruction*
tag_overflow

See Also TADDccTV[D] on page 275
TSUBcc on page 279

7.106 Unsigned Divide (64-bit ÷ 32-bit)

> The UDIV and UDIVcc instructions are deprecated and should not be used in new software. The UDIVX instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax		Class
UDIV^{D}	00 1110	Unsigned Integer Divide	`udiv`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*	**D2**
UDIVcc^{D}	01 1110	Unsigned Integer Divide and modify cc's	`udivcc`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*	**D2**

31 30	29 ... 25	24 ... 19	18 ... 14	13	12 ... 5	4 ... 0
10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	

Description The unsigned divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If i = 0, they compute "(Y :: R[rs1]{31:0}) ÷ R[rs2]{31:0}". Otherwise (that is, if i = 1), the divide instructions compute "(Y :: R[rs1]{31:0}) ÷ (**sign_ext**(simm13){31:0})". In either case, if overflow does not occur, the less significant 32 bits of the integer quotient are sign- or zero-extended to 64 bits and are written into R[rd].

The contents of the Y register are undefined after any 64-bit by 32-bit integer divide operation.

Unsigned Divide

Unsigned divide (UDIV, UDIVcc) assumes an unsigned integer doubleword dividend (Y :: R[rs1]{31:0}) and an unsigned integer word divisor R[rs2{31:0}] or (**sign_ext**(simm13){31:0}) and computes an unsigned integer word quotient (R[rd]). Immediate values in simm13 are in the ranges 0 to $2^{12} - 1$ and $2^{32} - 2^{12}$ to $2^{32} - 1$ for unsigned divide instructions.

Unsigned division rounds an inexact rational quotient toward zero.

Programming Note The *rational quotient* is the infinitely precise result quotient. It includes both the integer part and the fractional part of the result. For example, the rational quotient of 11/4 = 2.75 (integer part = 2, fractional part = .75).

The result of an unsigned divide instruction can overflow the less significant 32 bits of the destination register R[rd] under certain conditions. When overflow occurs, the largest appropriate unsigned integer is returned as the quotient *in* R[rd]. The condition under which overflow occurs and the value returned in R[rd] under this condition are specified in TABLE 7-15.

TABLE 7-15 UDIV / UDIVcc Overflow Detection and Value Returned

Condition Under Which Overflow Occurs	Value Returned in R[rd]
Rational quotient $\geq 2^{32}$	$2^{32} - 1$ (0000 0000 FFFF FFFF_{16})

When no overflow occurs, the 32-bit result is zero-extended to 64 bits and written into register R[rd].

UDIV does not affect the condition code bits. UDIVcc writes the integer condition code bits as shown in the following table. Note that negative (N) and zero (Z) are set according to the value of R[rd] after it has been set to reflect overflow, if any.

Bit	Effect on bit of UDIVcc instruction
icc.n	Set if R[rd]{31} = 1
icc.z	Set if R[rd]{31:0} = 0
icc.v	Set if overflow (*per* TABLE 7-15)
icc.c	Zero
xcc.n	Set if R[rd]{63} = 1
xcc.z	Set if R[rd]{63:0} = 0
xcc.v	Zero
xcc.c	Zero

An attempt to execute a UDIV or UDIVcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

Exceptions *illegal_instruction*
division_by_zero

See Also RDY on page 225
SDIV[cc] on page 240,
UMUL[cc] on page 283

7.107 Unsigned Multiply (32-bit)

> The UMUL and UMULcc instructions are deprecated and should not be used in new software. The MULX instruction should be used instead.

Opcode	op3	Operation	Assembly Language Syntax	Class
UMUL^{D}	00 1010	Unsigned Integer Multiply	`umul` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**D2**
UMULcc^{D}	01 1010	Unsigned Integer Multiply and modify cc's	`umulcc` *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	**D2**

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	

Description The unsigned multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute "R[rs1]{31:0} × R[rs2]{31:0}" if i = 0, or "R[rs1]{31:0} × **sign_ext(**simm13){31:0}" if i = 1. They write the 32 most significant bits of the product into the Y register and all 64 bits of the product into R[rd].

Unsigned multiply instructions (UMUL, UMULcc) operate on unsigned integer word operands and compute an unsigned integer doubleword product.

UMUL does not affect the condition code bits. UMULcc writes the integer condition code bits, icc and xcc, as shown below.

Bit	Effect on bit by execution of UMULcc
icc.n	Set to 1 if product{31} = 1; otherwise, set to 0
icc.z	Set to 1 if product{31:0}= 0; otherwise, set to 0
icc.v	Set to 0
icc.c	Set to 0
xcc.n	Set to 1 if product{63} = 1; otherwise, set to 0
xcc.z	Set to 1 if product{63:0} = 0; otherwise, set to 0
xcc.v	Set to 0
xcc.c	Set to 0

Note 32-bit negative (icc.n) and zero (icc.z) condition codes are set according to the *less* significant word of the product, not according to the full 64-bit result.

Programming Notes 32-bit overflow after UMUL or UMULcc is indicated by Y ≠ 0.

An attempt to execute a UMUL or UMULcc instruction when i = 0 and instruction bits 12:5 are nonzero causes an *illegal_instruction* exception.

Exceptions *illegal_instruction*

See Also

MULScc on page 209
RDY on page 225
SMUL[cc] on page 246,
UDIV[cc] on page 281

7.108 Write Ancillary State Register

Instruction	rd	Operation	Assembly Language Syntax	Class
WRY^{D}	0	Write Y register *(deprecated)*	`wr` reg_{rs1}, *reg_or_imm*, `%y`	**D2**
—	1	*Reserved*		
WRCCR	2	Write Condition Codes register	`wr` reg_{rs1}, *reg_or_imm*, `%ccr`	**A1**
WRASI	3	Write ASI register	`wr` reg_{rs1}, *reg_or_imm*, `%asi`	**A1**
—	4	*Reserved* (read-only ASR (TICK))		
—	5	*Reserved* (read-only ASR (PC))		
WRFPRS	6	Write Floating-Point Registers Status register	`wr` reg_{rs1}, *reg_or_imm*, `%fprs`	**A1**
—	7–14 ($7\text{-}0E_{16}$)	*Reserved*		
—	15 ($0F_{16}$)	*used at higher privilege level*		
—	16-18 ($10\text{-}12_{16}$)	*Reserved* (impl. dep. #8-V8-Cs20, #9-V8-Cs20)		
WRGSR	19 (13_{16})	Write General Status register (GSR)	`wr` reg_{rs1}, *reg_or_imm*, `%gsr`	**A1**
WRSOFTINT_SET^{P}	20 (14_{16})	Set bits of per-virtual processor Soft Interrupt register	`wr` reg_{rs1}, *reg_or_imm*, `%softint_set`	**N–**
WRSOFTINT_CLR^{P}	21 (15_{16})	Clear bits of per-virtual processor Soft Interrupt register	`wr` reg_{rs1}, *reg_or_imm*, `%softint_clr`	**N–**
WRSOFTINT^{P}	22 (16_{16})	Write per-virtual processor Soft Interrupt register	`wr` reg_{rs1}, *reg_or_imm*, `%softint`	**N–**
WRTICK_CMPR^{P}	23 (17_{16})	Write Tick Compare register	`wr` reg_{rs1}, *reg_or_imm*, `%tick_cmpr`	**N–**
—	24 (18_{16})	*used at higher privilege level*		
WRSTICK_CMPR^{P}	25 (19_{16})	Write System Tick Compare register	`wr` reg_{rs1}, *reg_or_imm*, `%stick_cmpr`†	**N–**
—	26 ($1A_{16}$)	*Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	26 ($1A_{16}$)	*Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	27 ($1B_{16}$)	*Reserved* (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	28 ($1C_{16}$)	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	29 ($1D_{16}$)	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		
—	30 ($1E_{16}$)	*Reserved*		
—	31 ($1F_{16}$)	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)		

† The original assembly language names for `%stick` and `%stick_cmpr` were, respectively, `%sys_tick` and `%sys_tick_cmpr`, which are now deprecated. Over time, assemblers will support the new `%stick` and `%stick_cmpr` names for these registers (which are consistent with `%tick` and `%tick_cmpr`). In the meantime, some existing assemblers may only recognize the original names.

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3 = 11 0000	rs1	i=0	—	rs2
10	rd	op3 = 11 0000	rs1	i=1	simm13	

WRasr

Description The WRasr instructions each store a value to the writable fields of the ancillary state register (ASR) specified by rd.

The value stored by these instructions (other than the implementation-dependent variants) is as follows: if i = 0, store the value "R[rs1] **xor** R[rs2]"; if i = 1, store "R[rs1] **xor sign_ext**(simm13)".

Note | The operation is **exclusive-or**.

The WRasr instruction with rd = 0 is a (deprecated) WRY instruction (which should not be used in new software). WRY is *not* a delayed-write instruction; the instruction immediately following a WRY observes the new value of the Y register.

> The WRY instruction is deprecated. It is recommended that all instructions that reference the Y register be avoided.

WRCCR, WRFPRS, and WRASI are *not* delayed-write instructions. The instruction immediately following a WRCCR, WRFPRS, or WRASI observes the new value of the CCR, FPRS, or ASI register.

WRFPRS waits for any pending floating-point operations to complete before writing the FPRS register.

IMPL. DEP. # 48-V8-Cs20: WRasr instructions with rd of 16-18, 28, 29, or 31 are available for implementation-dependent uses (impl. dep. #8-V8-Cs20). For a WRasr instruction using one of those rd values, the following are implementation dependent:

- the interpretation of bits 18:0 in the instruction
- the operation(s) performed (for example, **xor**) to generate the value written to the ASR
- whether the instruction is nonprivileged or privileged (impl. dep. #9-V8-Cs20), and
- whether an attempt to execute the instruction causes an *illegal_instruction* exception.

Note | See the section "Read/Write Ancillary State Registers (ASRs)" in *Extending the UltraSPARC Architecture*, contained in the separate volume *UltraSPARC Architecture Application Notes*, for a discussion of extending the SPARC V9 instruction set by means of read/write ASR instructions.

V9 Compatibility Notes | Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers.

The SPARC V8 WRIER, WRPSR, WRWIM, and WRTBR instructions do not exist in the UltraSPARC Architecture because the IER, PSR, TBR, and WIM registers do not exist in the UltraSPARC Architecture.

See *Ancillary State Registers* on page 48 for more detailed information regarding ASR registers.

Exceptions. An attempt to execute a WRasr instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- i = 0 **and** instruction bits 12:5 are nonzero
- rd = 1, 4, 5, 7–14, 18, or 26-31
- rd = 15 **and** ((rs1 ≠ 0) or (i = 0))

An attempt to execute a WRSOFTINT_SET, WRSOFTINT_CLR, WRSOFTINT, WRTICK_CMPR, or WRSTICK_CMPR instruction in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_opcode* exception.

If the floating-point unit is not enabled (FPRS.fef = 0 or PSTATE.pef = 0) or if the FPU is not present, then an attempt to execute a WRGSR instruction causes an *fp_disabled* exception.

Exceptions *illegal_instruction*
privileged_opcode
fp_disabled

See Also RDasr on page 225
WRPR on page 288

7.109 Write Privileged Register

Instruction	op3	Operation	rd	Assembly Language Syntax	Class
WRPR[P]	11 0010	Write Privileged register			**A1**
		TPC	0	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%tpc`	
		TNPC	1	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%tnpc`	
		TSTATE	2	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%tstate`	
		TT	3	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%tt`	
		(*illegal_instruction*)	4		
		TBA	5	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%tba`	
		PSTATE	6	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%pstate`	
		TL	7	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%tl`	
		PIL	8	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%pil`	
		CWP	9	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%cwp`	
		CANSAVE	10	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%cansave`	
		CANRESTORE	11	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%canrestore`	
		CLEANWIN	12	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%cleanwin`	
		OTHERWIN	13	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%otherwin`	
		WSTATE	14	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%wstate`	
		Reserved	15		
		GL	16	`wrpr` *reg*$_{rs1}$, *reg_or_imm*, `%gl`	
		Reserved	17–31		

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	

Description This instruction stores the value "R[rs1] **xor** R[rs2]" if i = 0, or "R[rs1] **xor sign_ext**(simm13)" if i = 1 to the writable fields of the specified privileged state register.

> **Note** | The operation is **exclusive-or**.

The rd field in the instruction determines the privileged register that is written. There are *MAXPTL* copies of the TPC, TNPC, TT, and TSTATE registers, one for each trap level. A write to one of these registers sets the register, indexed by the current value in the trap-level register (TL).

A WRPR to TL only stores a value to TL; it does not cause a trap, cause a return from a trap, or alter any machine state other than TL and state (such as PC, NPC, TICK, etc.) that is indirectly modified by every instruction.

> **Programming Note** | A WRPR of TL can be used to read the values of TPC, TNPC, and TSTATE for any trap level; however, software must take care that traps do not occur while the TL register is modified.

The WRPR instruction is a *non*-delayed-write instruction. The instruction immediately following the WRPR observes any changes made to virtual processor state made by the WRPR.

MAXPTL is the maximum value that may be written by a WRPR to TL; an attempt to write a larger value results in *MAXPTL* being written to TL. For details, see TABLE 5-19 on page 69.

MAXPGL is the maximum value that may be written by a WRPR to GL; an attempt to write a larger value results in *MAXPGL* being written to GL. For details, see TABLE 5-20 on page 70.

Exceptions. An attempt to execute a WRPR instruction in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_opcode* exception.

An attempt to execute a WRPR instruction when any of the following conditions exist causes an *illegal_instruction* exception:

- i = 0 and instruction bits 12:5 are nonzero
- rd = 4
- rd = 15, or 17-31 (reserved for future versions of the architecture)
- 0 ≤ rd ≤ 3 (attempt to write TPC, TNPC,TSTATE, or TT register) while TL = 0 (current trap level is zero) and the virtual processor is in privileged mode.

Implementation Note | In nonprivileged mode, *illegal_instruction* exception due to 0 ≤ rd ≤ 3 and TL = 0 does not occur; the *privileged_opcode* exception occurs instead.

Exceptions

privileged_opcode
illegal_instruction

See Also

RDPR on page 228
WRasr on page 285

7.110 XOR Logical Operation

Instruction	op3	Operation	Assembly Language Syntax		Class
XOR	00 0011	Exclusive **or**	`xor`	reg_{rs1}, *reg_or_imm*, reg_{rd}	**A1**
XORcc	01 0011	Exclusive **or** and modify cc's	`xorcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}	**A1**
XNOR	00 0111	Exclusive **nor**	`xnor`	reg_{rs1}, *reg_or_imm*, reg_{rd}	**A1**
XNORcc	01 0111	Exclusive **nor** and modify cc's	`xnorcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}	**A1**

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	

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*

CHAPTER **8**

IEEE Std 754-1985 Requirements for UltraSPARC Architecture 2007

The IEEE Std 754-1985 floating-point standard contains a number of implementation dependencies. This chapter specifies choices for these implementation dependencies, to ensure that SPARC V9 implementations are as consistent as possible.

The chapter contains these major sections:

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_exception*s; in that case, cexc contains a bit set to 1, corresponding to the exception that caused the trap. Only one bit shall be set in cexc.

Instructions causing an *fp_exception_other* trap because of unfinished FPops execute as if by hardware; that is, such a trap is undetectable by application software, except that timing may be affected.

Programming Note A user-mode trap handler invoked for an IEEE_754_exception, whether as a direct result of a hardware *fp_exception_ieee_754* trap or as an indirect result of privileged software handling of an *fp_exception_other* trap with FSR.ftt = unfinished_FPop, can rely on the following behavior:

- The address of the instruction that caused the exception will be available.
- The destination floating-point register(s) are unchanged from their state prior to that instruction's execution.
- The floating-point condition codes (`fcc0`, `fcc1`, `fcc2`, and `fcc3`) are unchanged.
- The FSR.aexc field is unchanged.
- The FSR.cexc field contains exactly one bit set to 1, corresponding to the exception that caused the trap.
- The FSR.ftt, FSR.qne, and reserved fields of FSR are zero.

~~8.2 Underflow Behavior~~

An UltraSPARC Architecture virtual processor detects tininess before rounding occurs. (impl. dep. #55-V8-Cs10)

TABLE 8-1 summarizes what happens when an exact *unrounded* value *u* satisfying

$$0 \le |u| \le \textit{smallest normalized number}$$

would round, if no trap intervened, to a *rounded* value *r* which might be zero, subnormal, or the smallest normalized value.

TABLE 8-1 Floating-Point Underflow Behavior (Tininess Detected Before Rounding)

	Underflow trap: Inexact trap:	**ufm = 1** **nxm = x**	**ufm = 0** **nxm = 1**	**ufm = 0** **nxm = 0**
$u = r$	*r* is minimum normal	None	None	None
	r is subnormal	UF	None	None
	r is zero	None	None	None
$u \neq r$	*r* is minimum normal	UF	NX	uf nx
	r is subnormal	UF	NX	uf nx
	r is zero	UF	NX	uf nx

UF = *fp_exception_ieee_754* trap with cexc.ufc = 1
NX = *fp_exception_ieee_754* trap with cexc.nxc = 1
uf = cexc.ufc = 1, aexc.ufa = 1, no *fp_exception_ieee_754* trap
nx = cexc.nxc = 1, aexc.nxa = 1, no *fp_exception_ieee_754* trap

8.2.1 Trapped Underflow Definition (ufm = 1)

Since tininess is detected before rounding, trapped underflow occurs when the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format.

Note The wrapped exponent results intended to be delivered on trapped underflows and overflows in IEEE 754 are irrelevant to the UltraSPARC Architecture at the hardware, and privileged software levels. If they are created at all, it would be by user software in a nonprivileged-mode trap handler.

8.2.2 Untrapped Underflow Definition (ufm = 0)

Untrapped underflow occurs when the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format *and* the correctly rounded result in the destination format is inexact.

8.3 Integer Overflow Definition

- **F<sdq>TOi** — When a NaN, infinity, large positive argument $\geq 2^{31}$ or large negative argument $\leq -(2^{31} + 1)$ is converted to an integer, the invalid_current (nvc) bit of FSR.cexc is set to 1, and if the floating-point invalid trap is enabled (FSR.tem.nvm = 1), the *fp_exception_IEEE_754* exception is raised. If the floating-point invalid trap is disabled (FSR.tem.nvm = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is $2^{31} - 1$; if the sign bit of the operand is 1, the result is -2^{31}.
- **F<sdq>TOx** — When a NaN, infinity, large positive argument $\geq 2^{63}$, or large negative argument $\leq -(2^{63} + 1)$ is converted to an extended integer, the invalid_current (nvc) bit of FSR.cexc is set to 1, and if the floating-point invalid trap is enabled (FSR.tem.nvm = 1), the *fp_exception_IEEE_754* exception is raised. If the floating-point invalid trap is disabled (FSR.tem.nvm = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is $2^{63} - 1$; if the sign bit of the operand is 1, the result is -2^{63}.

8.4 Floating-Point Nonstandard Mode

If implemented, floating-point nonstandard mode is enabled by setting FSR.ns = 1 (see *Nonstandard Floating-Point (ns)* on page 43).

An UltraSPARC Architecture 2007 processor may choose to implement nonstandard floating-point mode in order to obtain higher performance in certain circumstances. For example, when FSR.ns = 1 an implementation that processes fully normalized operands more efficiently than subnormal operands may convert a subnormal floating-point operand or result to zero.

Implementation Note UltraSPARC Architecture virtual processors are strongly discouraged from implementing a nonstandard floating-point mode.

Implementations are encouraged to support standard IEEE 754 floating-point arithmetic with reasonable performance in all cases, even if some cases are slower than others.

Assuming that nonstandard floating-point mode is implemented, the effects of FSR.ns = 1 are as follows:

- **IMPL. DEP. #18-V8-Ms10(a):** When FSR.ns = 1 and a floating-point *source operand* is subnormal, an implementation may treat the subnormal operand as if it were a floating-point zero value of the same sign.
  The cases in which this replacement is performed are implementation dependent. However, if it occurs,
  (1) it should *not* apply to FABS, FMOV, or FNEG instructions and
  (2) FADD, FSUB, and FCMP should give identical treatment to subnormal source operands.
  Treating a subnormal source operand as zero may generate an IEEE 754 floating-point “inexact”, “division by zero”, or “invalid” condition (see *Current Exception (cexc)* on page 46). Whether the generated condition(s) trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem.
- **IMPL. DEP. #18-V8-Ms10(b):** When a floating-point operation generates a subnormal *result* value, an UltraSPARC Architecture 2007 implementation may either write the result as a subnormal value or replace the subnormal result by a floating-point zero value of the same sign and generate IEEE 754 floating-point “inexact” and “underflow” conditions. Whether these generated conditions trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem.
- **IMPL. DEP. #18-V8-Ms10(c):** If an FPop generates an *intermediate* result value, the intermediate value is subnormal, and FSR.ns = 1, it is implementation dependent whether (1) the operation continues, using the subnormal value (possibly with some loss of accuracy), or (2) the virtual processor replaces the subnormal intermediate value with a floating-point zero value of the same sign, generates IEEE 754 floating-point “inexact” and “underflow” conditions, completes the instruction, and writes a final result (possibly with some loss of accuracy). Whether generated IEEE conditions trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem.

If GSR.im = 1, then the value of FSR.ns is ignored and the processor operates as if FSR.ns = 0 (see page 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:

- N*n* is a number in F[rs*n*], which may be normal or subnormal.
- QNaN*n* and SNaN*n* are Quiet and Signaling Not-a-Number values in F[rs*n*], respectively.

Notes on result types:

- R: (rounded) result of operation, which may be normal, subnormal, zero, or infinity. May also cause OF, UF, NX, unfinished.
- dQNaN is the generated default Quiet NaN (sign = 0, exponent = all 1s, fraction = all 1s). The sign of the default Quiet NaN is zero to distinguish it from storage initialized to all ones.
- QSNaN*n* is the Signalling NaN operand from F[rs*n*] with the Quiet bit asserted

8.5.1 Floating-Point Add (FADD)

TABLE 8-2 Floating-Point Add operation (F[rs1] + F[rs2])

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="8">F[rs2]</th></tr>
<tr><th>–∞</th><th>–N2</th><th>–0</th><th>+0</th><th>+N2</th><th>+∞</th><th>QNaN2</th><th>SNaN2</th></tr>
<tr><th rowspan="8">F[rs1]</th><th>–∞</th><td colspan="5">-∞</td><td>dQNaN, NV</td><td rowspan="7">QNaN2</td><td rowspan="8">QSNaN2, NV</td></tr>
<tr><th>–N1</th><td rowspan="4"></td><td>–R</td><td colspan="2">–N1</td><td>±R*</td><td rowspan="4"></td></tr>
<tr><th>–0</th><td rowspan="2">–N2</td><td>–0</td><td>±0**</td><td rowspan="2">+N2</td></tr>
<tr><th>+0</th><td>±0**</td><td>+0</td></tr>
<tr><th>+N1</th><td>±R*</td><td colspan="2">+N1</td><td>+R</td></tr>
<tr><th>+∞</th><td>dQNaN, NV</td><td colspan="5">+∞</td></tr>
<tr><th>QNaN1</th><td colspan="6">QNaN1</td></tr>
<tr><th>SNaN1</th><td colspan="7">QSNaN1, NV</td></tr>
</table>

* if N1 = -N2, then **

** result is +0 unless rounding mode is round to –∞, in which case the result is –0

For the FADD instructions, R may be any number; its generation may cause OF, UF, and/or NX.

Floating-point add is not commutative when both operands are NaN.

8.5.2 Floating-Point Subtract (FSUB)

TABLE 8-3 Floating-Point Subtract operation (F[rs1] – F[rs2])

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="8">F[rs2]</th></tr>
<tr><th>–∞</th><th>–N2</th><th>–0</th><th>+0</th><th>+N2</th><th>+∞</th><th>QNaN2</th><th>SNaN2</th></tr>
<tr><th rowspan="8">F[rs1]</th><th>–∞</th><td>dQNaN, NV</td><td colspan="5">-∞</td><td rowspan="7">QNaN2</td><td rowspan="8">QSNaN2, NV</td></tr>
<tr><th>–N1</th><td rowspan="4"></td><td>±R*</td><td colspan="2">–N1</td><td>–R</td><td rowspan="4"></td></tr>
<tr><th>–0</th><td rowspan="2">+N2</td><td>±0**</td><td>–0</td><td rowspan="2">–N2</td></tr>
<tr><th>+0</th><td>+0</td><td>±0**</td></tr>
<tr><th>+N1</th><td>+R</td><td colspan="2">+N1</td><td>±R*</td></tr>
<tr><th>+∞</th><td colspan="5">+∞</td><td>dQNaN, NV</td></tr>
<tr><th>QNaN1</th><td colspan="6">QNaN1</td></tr>
<tr><th>SNaN1</th><td colspan="7">QSNaN1, NV</td></tr>
</table>

* if N1 = N2, then **

** result is +0 unless rounding mode is round to –∞, in which case the result is –0

For the FSUB instructions, R may be any number; its generation may cause OF, UF, and/or NX.

Note that $-x \neq 0 - x$ when x is zero or NaN.

8.5.3 Floating-Point Multiply

TABLE 8-4 Floating-Point Multiply operation (F[rs1] × F[rs2])

<table>
<tr><td colspan="2" rowspan="2"></td><th colspan="8">F[rs2]</th></tr>
<tr><th>−∞</th><th>−N2</th><th>−0</th><th>+0</th><th>+N2</th><th>+∞</th><th>QNaN2</th><th>SNaN2</th></tr>
<tr><th rowspan="8">F[rs1]</th><th>-∞</th><td colspan="2">+∞</td><td colspan="2">dQNaN,
NV</td><td colspan="2">-∞</td><td rowspan="7">QNaN2</td><td rowspan="8">QSNaN2,
NV</td></tr>
<tr><th>−N1</th><td></td><td>+R</td><td></td><td></td><td>−R</td><td></td></tr>
<tr><th>− 0</th><td rowspan="2">dQNaN,
NV</td><td colspan="2">+ 0</td><td colspan="2">−0</td><td rowspan="2">dQNaN,
NV</td></tr>
<tr><th>+0</th><td colspan="2">−0</td><td colspan="2">+0</td></tr>
<tr><th>+ N1</th><td></td><td>−R</td><td></td><td></td><td>+R</td><td></td></tr>
<tr><th>+∞</th><td colspan="2">-∞</td><td colspan="2">dQNaN,
NV</td><td colspan="2">+∞</td></tr>
<tr><th>QNaN1</th><td colspan="6">QNaN1</td></tr>
<tr><th>SNaN1</th><td colspan="7">QSNaN1,
NV</td></tr>
</table>

R may be any number; its generation may cause OF, UF, and/or NX.

Floating-point multiply is not commutative when both operands are NaN.

FsMULd (FdMULq) never causes OF, UF, or NX.

A NaN input operand to FsMULd (FdMULq) must be widened to produce a double-precision (quad-precision) NaN output, by filling the least-significant bits of the NaN result with zeros.

8.5.4 Floating-Point Multiply-Add (FMADD

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 296) to select a row in the table below.

TABLE 8-5 Floating-Point Multiply-Add ((F[rs1] × F[rs2]) + F[rs3])

<table>
<tr><th colspan="2"></th><th colspan="8">F[rs3]</th></tr>
<tr><th colspan="2"></th><th>–∞</th><th>–N3</th><th>–0</th><th>+0</th><th>+N3</th><th>+∞</th><th>QNaN3</th><th>SNaN3</th></tr>
<tr><th rowspan="12">F[rs1] × F[rs2]</th><th>–∞</th><td colspan="5">–∞</td><td>dQNaN, NV</td><td rowspan="8">QNaN3</td><td rowspan="12">QSNaN3, NV</td></tr>
<tr><th>–N</th><td rowspan="4"></td><td>–R</td><td colspan="2">–N</td><td>±R*</td><td rowspan="4"></td></tr>
<tr><th>–0</th><td rowspan="2">–N3</td><td>–0</td><td>±0**</td><td rowspan="2">+N3</td></tr>
<tr><th>+0</th><td>±0**</td><td>+0</td></tr>
<tr><th>+N</th><td>±R*</td><td colspan="2">+N</td><td>+R</td></tr>
<tr><th>+∞</th><td>dQNaN, NV</td><td colspan="5">+∞</td></tr>
<tr><th>QNaN1</th><td colspan="6">QNaN1</td></tr>
<tr><th>QNaN2</th><td colspan="6">QNaN2</td></tr>
<tr><th>QNaN (±0 × ±∞)</th><td colspan="6">dQNaN, NV***</td><td>QNaN3, NV***</td></tr>
<tr><th>QSNaN1</th><td colspan="7">QSNaN1, NV***</td></tr>
<tr><th>QSNaN2</th><td colspan="7">QSNaN2, NV***</td></tr>
</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 FMADD FSR.nvc ← 1.

In the above table, R may be any number; its generation may cause OF, UF, and/or NX

The multiply operation in fused floating-point multiply-add (FMADD) instructions cannot cause inexact, underflow, or overflow exceptions.

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

8.5.5 Floating-Point Negative Multiply-Add (FNMADD)

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 296) to select a row in the table below.

TABLE 8-6 Floating-Point Negative Multiply-Add (–(F[rs1] × F[rs2])– F[rs3])

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="8">F[rs3]</th></tr>
<tr><th>–∞</th><th>–N3</th><th>–0</th><th>+0</th><th>+N3</th><th>+∞</th><th>QNaN3</th><th>SNaN3</th></tr>
<tr><th rowspan="11">F[rs1] × F[rs2]</th><th>–∞</th><td colspan="5">+∞</td><td>dQNaN, NV</td><td rowspan="7">QNaN3</td><td rowspan="11">QSNaN3, NV</td></tr>
<tr><th>–N</th><td rowspan="4"></td><td>+R</td><td colspan="2">+N</td><td>±R*</td><td rowspan="4"></td></tr>
<tr><th>–0</th><td rowspan="2">+N3</td><td>+0</td><td>±0**</td><td rowspan="2">–N3</td></tr>
<tr><th>+0</th><td>±0**</td><td>–0</td></tr>
<tr><th>+N</th><td>±R*</td><td colspan="2">–N</td><td>–R</td></tr>
<tr><th>+∞</th><td>dQNaN, NV</td><td colspan="5">–∞</td></tr>
<tr><th>QNaN1</th><td colspan="6">QNaN1</td></tr>
<tr><th>QNaN2</th><td colspan="6">QNaN2</td><td></td></tr>
<tr><th>QNaN (±0 × ±∞)</th><td colspan="6">dQNaN, NV***</td><td>QNaN3 NV***</td></tr>
<tr><th>QSNaN1</th><td colspan="7">QSNaN1, NV***</td></tr>
<tr><th>QSNaN2</th><td colspan="7">QSNaN2, NV***</td></tr>
</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 FMADD FSR.nvc ← 1.

R may be any number; its generation may cause OF, UF, and/or NX

The multiply operation in fused floating-point negative multiply-add (FNMADD) instructions cannot cause inexact, underflow, or overflow exceptions.

Note that rounding occurs after the negation. Thus, when the rounding mode is towards ±∞, FNMADD is not equivalent to FMADD followed by FNEG.

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

8.5.6 Floating-Point Multiply-Subtract (FMSUB)

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 296) to select a row in the table below.

TABLE 8-7 Floating-Point Multiply-Subtract ((F[rs1] × F[rs2])– F[rs3])

<table>
<tr><td colspan="2"></td><td colspan="8">F[rs3]</td></tr>
<tr><td colspan="2"></td><td>−∞</td><td>−N3</td><td>−0</td><td>+0</td><td>+N3</td><td>+∞</td><td>QNaN3</td><td>SNaN3</td></tr>
<tr><td rowspan="11">F[rs1] × F[rs2]</td><td>−∞</td><td>dQNaN, NV</td><td colspan="5">− ∞</td><td rowspan="8">QNaN3</td><td rowspan="11">QSNaN3, NV</td></tr>
<tr><td>−N</td><td rowspan="5">+∞</td><td>±R*</td><td colspan="2">−N</td><td>−R</td><td rowspan="4"></td></tr>
<tr><td>−0</td><td rowspan="2">+N3</td><td>±0**</td><td>−0</td><td rowspan="2">−N3</td></tr>
<tr><td>+0</td><td>+0</td><td>±0**</td></tr>
<tr><td>+N</td><td>+R</td><td colspan="2">+N</td><td>±R*</td></tr>
<tr><td>+∞</td><td colspan="4"></td><td>dQNaN, NV</td></tr>
<tr><td>QNaN1</td><td colspan="6">QNaN1</td></tr>
<tr><td>QNaN2</td><td colspan="6">QNaN2</td></tr>
<tr><td>QNaN (±0 × ±∞)</td><td colspan="6">dQNaN, NV***</td><td>QNaN3, NV***</td></tr>
<tr><td>QSNaN1</td><td colspan="7">QSNaN1, NV***</td></tr>
<tr><td>QSNaN2</td><td colspan="7">QSNaN2, NV***</td></tr>
</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 FMSUB FSR.nvc ← 1.

R may be any number; its generation may cause OF, UF, and/or NX.

The multiply operation in fused floating-point multiply-subtract (FMSUB) instructions cannot cause inexact, underflow, or overflow exceptions.

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

8.5.7 Floating-Point Negative Multiply-Subtract (FNMSUB)

First refer to the Floating-Point Multiply table (TABLE 8-4 on page 296) to select a row in the table below.

TABLE 8-8 Floating-Point Negative Multiply-Subtract (– (F[rs1] × F[rs2]) + F[rs3])

<table>
<tr><th></th><th></th><th colspan="8">F[rs3]</th></tr>
<tr><th></th><th></th><th>–∞</th><th>–N3</th><th>–0</th><th>+0</th><th>+N3</th><th>+∞</th><th>QNaN3</th><th>SNaN3</th></tr>
<tr><td rowspan="11">F[rs1] × F[rs2]</td><td>–∞</td><td>dQNaN, NV</td><td colspan="4"></td><td rowspan="5">+∞</td><td rowspan="8">QNaN3</td><td rowspan="11">QSNaN3, NV</td></tr>
<tr><td>–N</td><td rowspan="5">– ∞</td><td>±R*</td><td colspan="2">+N</td><td>+R</td></tr>
<tr><td>–0</td><td rowspan="2">–N3</td><td>±0**</td><td>+0</td><td rowspan="2">+N3</td></tr>
<tr><td>+0</td><td>–0</td><td>±0**</td></tr>
<tr><td>+N</td><td>–R</td><td colspan="2">–N</td><td>±R*</td></tr>
<tr><td>+∞</td><td colspan="4"></td><td>dQNaN, NV</td></tr>
<tr><td>QNaN1</td><td colspan="6">QNaN1</td></tr>
<tr><td>QNaN2</td><td colspan="6">QNaN2</td></tr>
<tr><td>QNaN (±0 × ±∞)</td><td colspan="6">dQNaN, NV***</td><td>QNaN3, NV***</td></tr>
<tr><td>QSNaN1</td><td colspan="7">QSNaN1, NV***</td></tr>
<tr><td>QSNaN2</td><td colspan="7">QSNaN2, NV***</td></tr>
</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.

R may be any number; its generation may cause OF, UF, and/or NX.

The multiply operation in fused floating-point negative multiply-subtract (FNMSUB) instructions cannot cause inexact, underflow, or overflow exceptions.

Note that rounding occurs after the negation. Thus, FNMSUB is not equivalent to FMSUB followed by FNEG when the rounding mode is towards ±∞.

See the earlier sections on Nonstandard Mode and unfinished_FPop for additional details.

8.5.8 Floating-Point Divide (FDIV)

TABLE 8-9 Floating-Point Divide operation (F[rs1] ÷ F[rs2])

<table>
<tr><th colspan="2"></th><th colspan="8">F[rs2]</th></tr>
<tr><th colspan="2"></th><th>–∞</th><th>– N2</th><th>–0</th><th>+ 0</th><th>+ N2</th><th>+∞</th><th>QNaN2</th><th>SNaN2</th></tr>
<tr><th rowspan="9">F[rs1]</th><th>–∞</th><td>dQNaN, NV</td><td colspan="2">+∞</td><td colspan="2">–∞</td><td>dQNaN, NV</td><td rowspan="8">QNaN2</td><td rowspan="9">QSNaN2, NV</td></tr>
<tr><th>–N1</th><td></td><td>+R</td><td>+∞, DZ</td><td>–∞, DZ</td><td>–R</td><td></td></tr>
<tr><th>–0</th><td colspan="2">+0</td><td colspan="2" rowspan="2">dQNaN, NV</td><td colspan="2">–0</td></tr>
<tr><th>+ 0</th><td colspan="2">–0</td><td colspan="2">+0</td></tr>
<tr><th>+ N1</th><td></td><td>–R</td><td>–∞, DZ</td><td>+∞, DZ</td><td>+R</td><td></td></tr>
<tr><th>+∞</th><td>dQNaN, NV</td><td colspan="2">–∞</td><td colspan="2">+∞</td><td>dQNaN, NV</td></tr>
<tr><th>QNaN1</th><td colspan="6">QNaN1</td></tr>
<tr><th>SNaN1</th><td colspan="7">QSNaN1, NV</td></tr>
</table>

R may be any number; its generation may cause OF, UF, and/or NX.

8.5.9 Floating-Point Square Root (FSQRT)

TABLE 8-10 Floating-Point Square Root operation ($\sqrt{F[rs2]}$)

<table>
<tr><th colspan="8">F[rs2]</th></tr>
<tr><th>–∞</th><th>–N2</th><th>– 0</th><th>+0</th><th>+ N2</th><th>+∞</th><th>QNaN2</th><th>SNaN2</th></tr>
<tr><td colspan="2">dQNaN, NV</td><td>–0</td><td>+0</td><td>+R</td><td>+∞</td><td>QNaN2</td><td>QSNaN2, NV</td></tr>
</table>

R may be any number; its generation may cause NX.

Square root cannot cause DZ, OF, or UF.

8.5.10 Floating-Point Compare (FCMP, FCMPE)

TABLE 8-11 Floating-Point Compare (FCMP, FCMPE) operation (F[rs1] ? F[rs2])

		F[rs2]							
		−∞	−N2	−0	+0	+N2	+∞	QNaN2	SNaN2
F[rs1]	−∞	0	1	1	1	1	1	3, NV*	3, NV
	−N1	2	0, 1, 2	1	1	1	1	3, NV*	3, NV
	−0	2	2	0	0	1	1	3, NV*	3, NV
	+0	2	2	0	0	1	1	3, NV*	3, NV
	+N1	2	2	2	2	0,1,2	1	3, NV*	3, NV
	+∞	2	2	2	2	2	0	3, NV*	3, NV
	QNaN1	3, NV*	3, NV*	3, NV*	3, NV*	3, NV*	3, NV*	3, NV*	3, NV
	SNaN1	3, NV	3, NV	3, NV	3, NV	3, NV	3, NV	3, NV	3, NV

* NV for FCMPE, but not for FCMP.

TABLE 8-12 FSR.fcc Encoding for Result of FCMP, FCMPE

fcc result	meaning
0	=
1	<
2	>
3	unordered

NaN is considered to be unequal to anything else, even the identical NaN bit pattern.

FCMP/FCMPE cannot cause DZ, OF, UF, NX.

8.5.11 Floating-Point to Floating-Point Conversions (F<s | d | q>TO<s | d | q>)

TABLE 8-13 Floating-Point to Float-Point Conversions (convert(F[rs2]))

F[rs2]									
−SNaN2	−QNaN2	−∞	−N2	−0	+0	+N2	+∞	+QNaN2	+SNaN2
−QSNaN2, NV	−QNaN2	−∞	−R	−0	+0	+R	+∞	+QNaN2	+QSNaN2, NV

For FsTOd:

- the least-significant fraction bits of a normal number are filled with zero to fit in double-precision format
- the least-significant bits of a NaN result operand are filled with zero to fit in double-precision format

For FsTOq and FdTOq:

- the least-significant fraction bits of a normal number are filled with zero to fit in quad-precision format

- the least-significant bits of a NaN result operand are filled with zero to fit in quad-precision format

For FqTOs and FdTOs:

- the fraction is rounded according to the current rounding mode
- the lower-order bits of a NaN source are discarded to fit in single-precision format; this discarding is not considered a rounding operation, and will not cause an NX exception

For FqTOd:

- the fraction is rounded according to the current rounding mode
- the least-significant bits of a NaN source are discarded to fit in double-precision format; this discarding is not considered a rounding operation, and will not cause an NX exception

TABLE 8-14 Floating-Point to Float-Point Conversion Exception Conditions

NV	• SNaN operand
OF	• FdTOs, FqTOs: the input is larger than can be expressed in single precision • 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

8.5.12 Floating-Point to Integer Conversions (F<s | d | q>TO<i | x>)

TABLE 8-15 Floating-Point to Integer Conversions (convert(F[rs2]))

<table>
<tr><th></th><th colspan="10">F[rs2]</th></tr>
<tr><th></th><th>–SNaN2</th><th>–QNaN2</th><th>–∞</th><th>–N2</th><th>–0</th><th>+0</th><th>+N2</th><th>+∞</th><th>+QNaN2</th><th>+SNaN2</th></tr>
<tr><td>FdTOx
FsTOx
FqTOx</td><td colspan="2">-2^{63},
NV</td><td>-2^{63},
NV</td><td rowspan="2">–R</td><td colspan="2" rowspan="2">0</td><td rowspan="2">+R</td><td>$2^{63}-1$,
NV</td><td colspan="2">$2^{63}-1$,
NV</td></tr>
<tr><td>FdTOi
FsTOi
FqTOi</td><td colspan="2">-2^{31},
NV</td><td>-2^{31},
NV</td><td>$2^{31}-1$,
NV</td><td colspan="2">$2^{31}-1$,
NV</td></tr>
</table>

R may be any integer, and may cause NV, NX.

Float-to-Integer conversions are always treated as round-toward-zero (truncated).

These operations are invalid (due to integer overflow) under the conditions described in *Integer Overflow Definition* on page 293.

TABLE 8-16 Floating-point to Integer Conversion Exception Conditions

NV	• SNaN operand • QNaN operand • ±∞ operand • integer overflow
NX	• non-integer source (truncation occurred)

8.5.13 Integer to Floating-Point Conversions (F<i | x>TO<s | d | q>)

TABLE 8-17 Integer to Floating-Point Conversions (convert(F[rs2]))

F[rs2]		
−int	0	+int
−R	+0	+R

R may be any number; its generation may cause NX.

TABLE 8-18 Floating-Point Conversion Exception Conditions

NX	• FxTOd, FxTOs, FiTOs (possible loss of precision) • not applicable to FiTOd, FxTOq, or FiTOq (FSR.cexc will always be cleared)

CHAPTER **9**

Memory

The UltraSPARC Architecture *memory models* define the semantics of memory operations. The instruction set semantics require that loads and stores behave *as if* they are performed in the order in which they appear in the dynamic control flow of the program. The *actual* order in which they are processed by the memory may be different. The purpose of the memory models is to specify what constraints, if any, are placed on the order of memory operations.

The memory models apply both to uniprocessor and to shared memory multiprocessors. Formal memory models are necessary for precise definitions of the interactions between multiple virtual processors and input/output devices in a shared memory configuration. Programming shared memory multiprocessors requires a detailed understanding of the operative memory model and the ability to specify memory operations at a low level in order to build programs that can safely and reliably coordinate their activities. For additional information on the use of the models in programming real systems, see *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

This chapter contains a great deal of theoretical information so that the discussion of the UltraSPARC Architecture TSO memory model has sufficient background.

This chapter describes memory models in these sections:

9.1 Memory Location Identification

A memory location is identified by an 8-bit address space identifier (ASI) and a 64-bit memory address. The 8-bit ASI can be obtained from an ASI register or included in a memory access instruction. The ASI used for an access can distinguish among different 64-bit address spaces, such as Primary memory space, Secondary memory space, and internal control registers. It can also apply attributes to the access, such as whether the access should be performed in big- or little-endian byte order, or whether the address should be taken as a virtual or real.

9.2 Memory Accesses and Cacheability

Memory is logically divided into real memory (cached) and I/O memory (noncached with and without side effects) spaces.

Real memory stores information without side effects. A load operation returns the value most recently stored. Operations are side-effect-free in the sense that a load, store, or atomic load-store to a location in real memory has no program-observable effect, except upon that location (or, in the case of a load or load-store, on the destination register).

I/O locations may not behave like memory and may have side effects. Load, store, and atomic load-store operations performed on I/O locations may have observable side effects, and loads may not return the value most recently stored. The value semantics of operations on I/O locations are *not* defined by the memory models, but the constraints on the order in which operations are performed is the same as it would be if the I/O locations were real memory. The storage properties, contents, semantics, ASI assignments, and addresses of I/O registers are implementation dependent.

9.2.1 Coherence Domains

Two types of memory operations are supported in the UltraSPARC Architecture: cacheable and noncacheable accesses. The manner in which addresses are differentiated is implementation dependent. In some implementations, it is indicated in the page translation entry (TTE.cp).

Although SPARC V9 does not specify memory ordering between cacheable and noncacheable accesses, the UltraSPARC Architecture maintains TSO ordering between memory references regardless of their cacheability.

9.2.1.1 Cacheable Accesses

Accesses within the coherence domain are called cacheable accesses. They have these properties:

- Data reside in real memory locations.
- Accesses observe supported cache coherency protocol(s).
- The cache line size is 2^n bytes (where $n \geq 4$), and can be different for each cache.

9.2.1.2 Noncacheable Accesses

Noncacheable accesses are outside of the coherence domain. They have the following properties:

- Data might not reside in real memory locations. Accesses may result in programmer-visible side effects. An example is memory-mapped I/O control registers.
- Accesses do not observe supported cache coherency protocol(s).
- The smallest unit in each transaction is a single byte.

The UltraSPARC Architecture MMU optionally includes an attribute bit in each page translation, TTE.e, which when set signifies that this page has side effects.

Noncacheable accesses without side effects (TTE.e = 0) are processor-consistent and obey TSO memory ordering. In particular, processor consistency ensures that a noncacheable load that references the same location as a previous noncacheable store will load the data from the previous store.

Noncacheable accesses with side effects (TTE.e = 1) are processor consistent and are strongly ordered. These accesses are described in more detail in the following section.

9.2.1.3 Noncacheable Accesses with Side-Effect

Loads, stores, and load-stores to I/O locations might not behave with memory semantics. Loads and stores could have side effects; for example, a read access could clear a register or pop an entry off a FIFO. A write access could set a register address port so that the next access to that address will read or write a particular internal register. Such devices are considered order sensitive. Also, such devices may only allow accesses of a fixed size, so store merging of adjacent stores or stores within a 16-byte region would cause an error (see *Store Coalescing* on page 320).

Noncacheable accesses (other than block loads and block stores) to pages with side effects (TTE.e = 1) exhibit the following behavior:

- Noncacheable accesses are strongly ordered with respect to each other. Bus protocol should guarantee that IO transactions to the same device are delivered in the order that they are received.
- Noncacheable loads with the TTE.e bit = 1 will not be issued to the system until all previous instructions have completed, and the store queue is empty.
- Noncacheable store coalescing is disabled for accesses with TTE.e = 1.
- A MEMBAR may be needed between side-effect and non-side-effect accesses. See TABLE 9-3 on page 317.

Whether block loads and block stores adhere to the above behavior or ignore TTE.e and always behave as if TTE.e = 0 is implementation-dependent (impl. dep. #410-S10, #411-S10).

On UltraSPARC Architecture virtual processors, noncacheable and side-effect accesses do not observe supported cache coherency protocols (impl. dep. #120).

Non-faulting loads (using `ASI_PRIMARY_NO_FAULT[_LITTLE]` or `ASI_SECONDARY_NO_FAULT[_LITTLE])` with the TTE.e bit = 1 cause a *DAE_side_effect_page* trap.

Prefetches to noncacheable addresses result in nops.

The processor does speculative instruction memory accesses and follows branches that it predicts are taken. Instruction addresses mapped by the MMU can be accessed even though they are not actually executed by the program. Normally, locations with side effects or that generate timeouts or bus errors are not mapped as instruction addresses by the MMU, so these speculative accesses will not cause problems.

IMPL. DEP. #118-V9: The manner in which I/O locations are identified is implementation dependent.

IMPL. DEP. #120-V9: The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent.

V9 Compatibility Note | Operations to I/O locations are *not* guaranteed to be sequentially consistent among themselves, as they are in SPARC V8.

Systems supporting SPARC V8 applications that use memory-mapped I/O locations must ensure that SPARC V8 sequential consistency of I/O locations can be maintained when those locations are referenced by a SPARC V8 application. The MMU either must enforce such consistency or cooperate with system software or the virtual processor to provide it.

IMPL. DEP. #121-V9: An implementation may choose to identify certain addresses and use an implementation-dependent memory model for references to them.

9.3 Memory Addressing and Alternate Address Spaces

An address in SPARC V9 is a tuple consisting of an 8-bit address space identifier (ASI) and a 64-bit byte-address offset within the specified address space. Memory is byte-addressed, with halfword accesses aligned on 2-byte boundaries, word accesses (which include instruction fetches) aligned on 4-byte boundaries, extended-word and doubleword accesses aligned on 8-byte boundaries, and quadword quantities aligned on 16-byte boundaries. With the possible exception of the cases described in *Memory Alignment Restrictions* on page 73, an improperly aligned address in a load, store, or load-store instruction always causes a trap to occur. The largest datum that is guaranteed to be atomically read or written is an aligned doubleword[1]. Also, memory references to different bytes, halfwords, and words in a given doubleword are treated for ordering purposes as references to the same location. Thus, the unit of ordering for memory is a doubleword.

Notes The doubleword is the coherency unit for update, but programmers should not assume that doubleword floating-point values are updated as a unit unless they are doubleword-aligned and always updated with double-precision loads and stores. Some programs use pairs of single-precision operations to load and store double-precision floating-point values when the compiler cannot determine that they are doubleword aligned.

Also, although quad-precision operations are defined in the SPARC V9 architecture, the granularity of loads and stores for quad-precision floating-point values may be word or doubleword.

9.3.1 Memory Addressing Types

The UltraSPARC Architecture supports the following types of memory addressing:

Virtual Addresses (VA). Virtual addresses are addresses produced by a virtual processor that maps all systemwide, program-visible memory. Virtual addresses can be presented in nonprivileged mode and privileged mode

Real addresses (RA). A real address is provided to privileged software to describe the underlying physical memory allocated to it. Translation storage buffers (TSBs) maintained by privileged software are used to translate privileged or nonprivileged mode virtual addresses into real addresses. MMU bypass addresses in privileged mode are also real addresses.

Nonprivileged software only uses virtual addresses. Privileged software uses virtual and real addresses.

9.3.2 Memory Address Spaces

The UltraSPARC Architecture supports accessing memory using virtual or real addresses. Multiple virtual address spaces within the same real address space are distinguished by a *context identifier* (context ID).

[1] Two exceptions to this are the special `ASI_TWIN_DW_NUCLEUS[_L]` and `ASI_TWINX_REAL[_L]` which provide hardware support for an atomic quad load to be used for `TTE` loads from TSBs.

Privileged software can create multiple virtual address spaces, using the primary and secondary context registers to associate a context ID with every virtual address. Privileged software manages the allocation of context IDs.

The full representation of a real address is as follows:

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
- To provide additional access control and attribute information, for example, to specify the endianness of the reference
- To specify the address of an internal control register in the virtual processor, cache, or memory management hardware

Memory management hardware can associate an independent 2^{64}-byte memory address space with each ASI. In practice, the three independent memory address spaces (contexts) created by the MMU are Primary, Secondary, and Nucleus.

Programming Note | Independent address spaces, accessible through ASIs, make it possible for system software to easily access the address space of faulting software when processing exceptions or to implement access to a client program's memory space by a server program.

Alternate-space load, store, load-store and prefetch instructions specify an *explicit* ASI to use for their data access. The behavior of the access depends on the current privilege mode.

Non-alternate space load, store, load-store, and prefetch instructions use an *implicit* ASI value that is determined by current virtual processor state (the current privilege mode, trap level (TL), and the value of the PSTATE.cle). Instruction fetches use an implicit ASI that depends only on the current mode and trap level.

The architecturally specified ASIs are listed in Chapter 10, *Address Space Identifiers (ASIs)*. The operation of each ASI in nonprivileged and privileged modes is indicated in TABLE 10-1 on page 323.

Attempts by nonprivileged software (PSTATE.priv = 0) to access restricted ASIs (ASI bit 7 = 0) cause a *privileged_action* exception. Attempts by privileged software (PSTATE.priv = 1) to access ASIs 30_{16}–$7F_{16}$ cause a *privileged_action* exception.

When TL = 0, normal accesses by the virtual processor to memory when fetching instructions and performing loads and stores implicitly specify `ASI_PRIMARY` or `ASI_PRIMARY_LITTLE`, depending on the setting of PSTATE.cle.

When TL = 1 or 2 (> 0 but ≤ *MAXPTL*), the implicit ASI in privileged mode is:

- for instruction fetches, `ASI_NUCLEUS`
- for loads and stores, `ASI_NUCLEUS` if PSTATE.cle = 0 or `ASI_NUCLEUS_LITTLE` if PSTATE.cle = 1 (impl. dep. #124-V9).

SPARC V9 supports the `PRIMARY[_LITTLE]`, `SECONDARY[_LITTLE]`, and `NUCLEUS[_LITTLE]` address spaces.

Accesses to other address spaces use the load/store alternate instructions. For these accesses, the ASI is either contained in the instruction (for the register+register addressing mode) or taken from the ASI register (for register+immediate addressing).

ASIs are either nonrestricted or restricted-to-privileged:

- A nonrestricted ASI (ASI range 80_{16} – FF_{16}) is one that may be used independently of the privilege level (PSTATE.priv) at which the virtual processor is running.
- A restricted-to-privileged ASI (ASI range 00_{16} – $2F_{16}$) requires that the virtual processor be in privileged mode for a legal access to occur.

The relationship between virtual processor state and ASI restriction is shown in TABLE 9-1.

TABLE 9-1 Allowed Accesses to ASIs

ASI Value	Type	Result of ASI Access in NP Mode	Result of ASI Access in P Mode
00_{16} –- $2F_{16}$	Restricted-to-privileged	*privileged_action* exception	Valid Access
80_{16} – FF_{16}	Nonrestricted	Valid Access	Valid Access

Some restricted ASIs are provided as mandated by SPARC V9: `ASI_AS_IF_USER_PRIMARY[_LITTLE]` and `ASI_AS_IF_USER_SECONDARY[_LITTLE]`. The intent of these ASIs is to give privileged software efficient, yet secure access to the memory space of nonprivileged software.

The normal address space is *primary address space*, which is accessed by the unrestricted `ASI_PRIMARY[_LITTLE]` ASIs. The *secondary address space*, which is accessed by the unrestricted `ASI_SECONDARY[_LITTLE]` ASIs, is provided to allow server software to access client software's address space.

`ASI_PRIMARY_NOFAULT[_LITTLE]` and `ASI_SECONDARY_NOFAULT[_LITTLE]` support *nonfaulting loads***.** These ASIs may be used to color (that is, distinguish into classes) loads in the instruction stream so that, in combination with a judicious mapping of low memory and a specialized trap handler, an optimizing compiler can move loads outside of conditional control structures.

9.4 SPARC V9 Memory Model

The SPARC V9 processor architecture specified the organization and structure of a central processing unit but did not specify a memory system architecture. This section summarizes the MMU support required by an UltraSPARC Architecture processor.

The memory models specify the possible order relationships between memory-reference instructions issued by a virtual processor and the order and visibility of those instructions as seen by other virtual processors. The memory model is intimately intertwined with the program execution model for instructions.

9.4.1 SPARC V9 Program Execution Model

The SPARC V9 strand model of a virtual processor consists of three units: an Issue Unit, a Reorder Unit, and an Execute Unit, as shown in FIGURE 9-1.

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.

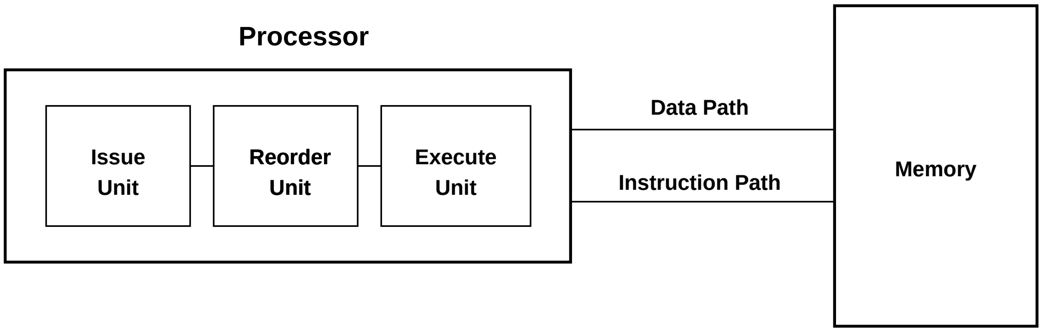


FIGURE 9-1 Processor Model: Uniprocessor System

Issued instructions are collected and potentially reordered in the Reorder Unit, and then dispatched to the Execute Unit. Instruction reordering allows an implementation to perform some operations in parallel and to better allocate resources. The reordering of instructions is constrained to ensure that the results of program execution are the same as they would be if the instructions were performed in program order. This property is called *processor self-consistency*.

Processor self-consistency requires that the result of execution, in the absence of any shared memory interaction with another virtual processor, be identical to the result that would be observed if the instructions were performed in program order. In the model in FIGURE 9-1, instructions are issued in program order and placed in the reorder buffer. The virtual processor is allowed to reorder instructions, provided it does not violate any of the data-flow constraints for registers or for memory.

The data-flow order constraints for register reference instructions are these:

1. An instruction that reads from or writes to a register cannot be performed until all earlier instructions that write to that register have been performed (read-after-write hazard; write-after-write hazard).

2. An instruction cannot be performed that writes to a register until all earlier instructions that read that register have been performed (write-after-read hazard).

V9 Compatibility Note | An implementation can avoid blocking instruction execution in case 2 and the write-after-write hazard in case 1 by using a renaming mechanism that provides the old value of the register to earlier instructions and the new value to later uses.

The data-flow order constraints for memory-reference instructions are those for register reference instructions, plus the following additional constraints:

1. A memory-reference instruction that uses (loads or stores) the value at a location cannot be performed until all earlier memory-reference instructions that set (store to) that location have been performed (read-after-write hazard, write-after-write hazard).

2. A memory-reference instruction that writes (stores to) a location cannot be performed until all previous instructions that read (load from) that location have been performed (write-after-read hazard).

Memory-barrier instruction (MEMBAR) and the TSO memory model also constrain the issue of memory-reference instructions. See *Memory Ordering and Synchronization* on page 316 and *The UltraSPARC Architecture Memory Model — TSO* on page 313 for a detailed description.

The constraints on instruction execution assert a partial ordering on the instructions in the reorder buffer. Every one of the several possible orderings is a legal execution ordering for the program. See Appendix D, *Formal Specification of the Memory Models*, for more information.

9.4.2 Virtual Processor/Memory Interface Model

Each UltraSPARC Architecture virtual processor in a multiprocessor system is modeled as shown in FIGURE 9-2; that is, having two independent paths to memory: one for instructions and one for data.

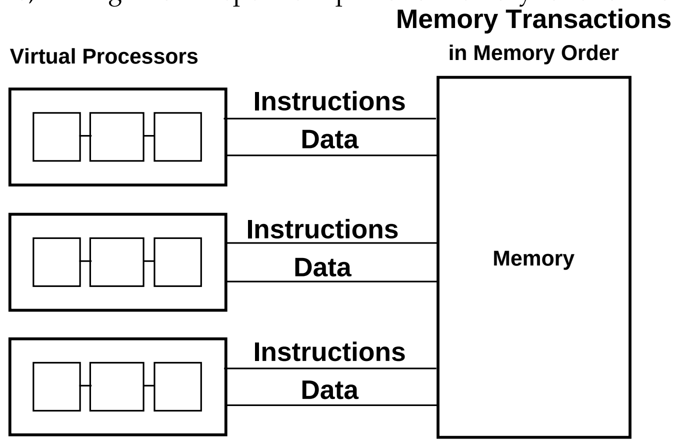


FIGURE 9-2 Data Memory Paths: Multiprocessor System

Data caches are maintained by hardware so their contents always appear to be consistent (coherent). Instruction caches are *not* required to be kept consistent with data caches and therefore require explicit program (software) action to ensure consistency when a program modifies an executing instruction stream. See *Synchronizing Instruction and Data Memory* on page 318 for details. Memory is shared in terms of address space, but it may be nonhomogeneous and distributed in an implementation.Caches are ignored in the model, since their functions are transparent to the memory model[1].

In real systems, addresses may have attributes that the virtual processor must respect. The virtual processor executes loads, stores, and atomic load-stores in whatever order it chooses, as constrained by program order and the memory model.

Instructions are performed in an order constrained by local dependencies. Using this dependency ordering, an execution unit submits one or more pending memory transactions to the memory. The memory performs transactions in *memory order*. The memory unit may perform transactions submitted to it out of order; hence, the execution unit must not concurrently submit two or more transactions that are required to be ordered, unless the memory unit can still guarantee in-order semantics.

The memory accepts transactions, performs them, and then acknowledges their completion. Multiple memory operations may be in progress at any time and may be initiated in a nondeterministic fashion in any order, provided that all transactions to a location preserve the per-virtual processor partial orderings. Memory transactions may complete in any order. Once initiated, all memory operations are performed atomically: loads from one location all see the same value, and the result of stores is visible to all potential requestors at the same instant.

The order of memory operations observed at a single location is a *total order* that preserves the partial orderings of each virtual processor's transactions to this address. There may be many legal total orders for a given program's execution.

1. The model described here is only a model; implementations of UltraSPARC Architecture systems are unconstrained as long as their observable behaviors match those of the model.

9.5 The UltraSPARC Architecture Memory Model — TSO

The UltraSPARC Architecture is a *model* that specifies the behavior observable by software on UltraSPARC Architecture systems. Therefore, access to memory can be implemented in any manner, as long as the behavior observed by software conforms to that of the models described here.

The SPARC V9 architecture defines three different memory models: *Total Store Order (TSO)*, *Partial Store Order (PSO)*, and *Relaxed Memory Order (RMO)*.

All SPARC V9 processors must provide Total Store Order (or a more strongly ordered model, for example, Sequential Consistency) to ensure compatibility for SPARC V8 application software.

All UltraSPARC Architecture virtual processors implement TSO ordering. The PSO and RMO models from SPARC V9 are not described in this UltraSPARC Architecture specification. UltraSPARC Architecture 2007 processors do not implement the PSO memory model directly, but all software written to run under PSO will execute correctly on an UltraSPARC Architecture 2007 processor (using the TSO model).

Whether memory models represented by PSTATE.mm = 10_2 or 11_2 are supported in an UltraSPARC Architecture processor is implementation dependent (impl. dep. #113-V9-Ms10). If the 10_2 model is supported, then when PSTATE.mm = 10_2 the implementation must correctly execute software that adheres to the RMO model described in *The SPARC Architecture Manual-Version 9*. If the 11_2 model is supported, its definition is implementation dependent and will be described in implementation-specific documentation.

Programs written for Relaxed Memory Order will work in both Partial Store Order and Total Store Order. Programs written for Partial Store Order will work in Total Store Order. Programs written for a weak model, such as RMO, may execute more quickly when run on hardware directly supporting that model, since the model exposes more scheduling opportunities, but use of that model may also require extra instructions to ensure synchronization. Multiprocessor programs written for a stronger model will behave unpredictably if run in a weaker model.

Machines that implement *sequential consistency* (also called "strong ordering" or "strong consistency") automatically support programs written for TSO. Sequential consistency is not a SPARC V9 memory model. In sequential consistency, the loads, stores, and atomic load-stores of all virtual processors are performed by memory in a serial order that conforms to the order in which these instructions are issued by individual virtual processors. A machine that implements sequential consistency may deliver lower performance than an equivalent machine that implements TSO order. Although particular SPARC V9 implementations may support sequential consistency, portable software must not rely on the sequential consistency memory model.

9.5.1 Memory Model Selection

The active memory model is specified by the 2-bit value in PSTATE.mm,. The value 00_2 represents the TSO memory model; increasing values of PSTATE.mm indicate increasingly weaker (less strongly ordered) memory models.

Writing a new value into PSTATE.mm causes subsequent memory reference instructions to be performed with the order constraints of the specified memory model.

IMPL. DEP. #119-Ms10: The effect of an attempt to write an unsupported memory model designation into PSTATE.mm is implementation dependent; however, it should never result in a value of PSTATE.mm value greater than the one that was written. In the case of an UltraSPARC Architecture implementation that only supports the TSO memory model, PSTATE.mm always reads as zero and attempts to write to it are ignored.

9.5.2 Programmer-Visible Properties of the UltraSPARC Architecture TSO Model

Total Store Order must be provided for compatibility with existing SPARC V8 programs. Programs that execute correctly in either RMO or PSO will execute correctly in the TSO model.

The rules for TSO, in addition to those required for self-consistency (see page 311), are:

- Loads are blocking and ordered with respect to earlier loads
- Stores are ordered with respect to stores.
- Atomic load-stores are ordered with respect to loads and stores.
- Stores cannot bypass earlier loads.

Programming Note | Loads *can* bypass earlier stores to other addresses, which maintains processor self-consistency.

Atomic load-stores are treated as both a load and a store and can only be applied to cacheable address spaces.

Thus, TSO ensures the following behavior:

- Each load instruction behaves as if it were followed by a MEMBAR `#LoadLoad` and #LoadStore.
- Each store instruction behaves as if it were followed by a MEMBAR `#StoreStore`.
- Each atomic load-store behaves as if it were followed by a MEMBAR `#LoadLoad`, `#LoadStore`, and `#StoreStore`.

In addition to the above TSO rules, the following rules apply to UltraSPARC Architecture memory models:

- A MEMBAR `#StoreLoad` must be used to prevent a load from bypassing a prior store, if Strong Sequential Order (as defined in *The UltraSPARC Architecture Memory Model — TSO* on page 313) is desired.
- Accesses that have side effects are all strongly ordered with respect to each other.
- A MEMBAR `#Lookaside` is not needed between a store and a subsequent load to the same noncacheable address.
- Load (LDXA) and store (STXA) instructions that reference certain internal ASIs perform both an intra-virtual processor synchronization (i.e. an implicit MEMBAR `#Sync` operation before the load or store is executed) and an inter-virtual processor synchronization (that is, all active virtual processors are brought to a point where synchronization is possible, the load or store is executed, and all virtual processors then resume instruction fetch and execution). The model-specific PRM should indicate which ASIs require intra-virtual processor synchronization, inter-virtual processor synchronization, or both.

9.5.3 TSO Ordering Rules

TABLE 9-2 summarizes the cases where a MEMBAR must be inserted between two memory operations on an UltraSPARC Architecture virtual processor running in TSO mode, to ensure that the operations appear to complete in a particular order. Memory operation *ordering* is not to be confused with processor consistency or deterministic operation; MEMBARs are required for deterministic operation of certain ASI register updates.

Programming Note To ensure software portability across systems, the MEMBAR rules in this section should be followed (which may be stronger than the rules in SPARC V9).

TABLE 9-2 is to be read as follows: Reading from row to column, the first memory operation in program order in a row is followed by the memory operation found in the column. Symbols used as table entries:

- # — No intervening operation is required.
- M — an intervening MEMBAR `#StoreLoad` or MEMBAR `#Sync` or MEMBAR `#MemIssue` is required
- S — an intervening MEMBAR `#Sync` or MEMBAR `#MemIssue` is required
- nc — Noncacheable
- e — Side effect
- ne — No side effect

TABLE 9-2 Summary of UltraSPARC Architecture Ordering Rules (TSO Memory Model)

	***To* Memory Operation C (column):**										
***From* Memory Operation R (row):**	**load**	**store**	**atomic**	**bload**	**bstore**	**load_nc_e**	**store_nc_e**	**load_nc_ne**	**store_nc_ne**	**bload_nc**	**bstore_nc**
load	#	#	#	S	S	#	#	#	#	S	S
store	M[2]	#	#	M	S	M	#	M	#	M	S
atomic	#	#	#	M	S	#	#	#	#	M	S
bload	S	S	S	S	S	S	S	S	S	S	S
bstore	M	S	M	M	S	M	S	M	S	M	S
load_nc_e	#	#	#	S	S	#[1]	#[1]	#[1]	#[1]	S	S
store_nc_e	S	#	#	S	S	#[1]	#[1]	M[2]	#[1]	M	S
load_nc_ne	#	#	#	S	S	#[1]	#[1]	#[1]	#[1]	S	S
store_nc_ne	S	#	#	S	S	M[2]	#[1]	M[2]	#[1]	M	S
bload_nc	S	S	S	S	S	S	S	S	S	S	S
bstore_nc	S	S	S	S	S	M	S	M	S	M	S

1. This table assumes that both noncacheable operations access the same device.

2. When the store and subsequent load access the *same* location, no intervening MEMBAR is required.

Note that transitivity applies; if operation X is always ordered before operation Y ("#" in TABLE 9-2) and operation Y is always ordered before operation Z (again, "#" in the table), then the sequence of operations X ... Y ... Z may safely be executed with no intervening MEMBAR, even if the table shows that a MEMBAR is normally needed between X and Z. For example, a MEMBAR is normally needed between a store and a load ("M" in TABLE 9-2); however, the sequence "store ... atomic ... load" may be executed safely with no intervening MEMBAR because stores are always ordered before atomics and atomics are always ordered before loads.

9.5.4 Hardware Primitives for Mutual Exclusion

In addition to providing memory-ordering primitives that allow programmers to construct mutual-exclusion mechanisms in software, the UltraSPARC Architecture provides three hardware primitives for mutual exclusion:

- Compare and Swap (CASA and CASXA)
- Load Store Unsigned Byte (LDSTUB and LDSTUBA)
- Swap (SWAP and SWAPA)

Each of these instructions has the semantics of both a load and a store in all three memory models. They are all *atomic*, in the sense that no other store to the same location can be performed between the load and store elements of the instruction. All of the hardware mutual-exclusion operations conform to the TSO memory model and may require barrier instructions to ensure proper data visibility.

Atomic load-store instructions can be used only in the cacheable domains (not in noncacheable I/O addresses). An attempt to use an atomic load-store instruction to access a noncacheable page results in a *DAE_nc_page* exception.

The atomic load-store alternate instructions can use a limited set of the ASIs. See the specific instruction descriptions for a list of the valid ASIs. An attempt to execute an atomic load-store alternate instruction with an invalid ASI results in a *DAE_invalid_asi* exception.

9.5.4.1 Compare-and-Swap (CASA, CASXA)

Compare-and-swap is an atomic operation that compares a value in a virtual processor register to a value in memory and, if and only if they are equal, swaps the value in memory with the value in a second virtual processor register. Both 32-bit (CASA) and 64-bit (CASXA) operations are provided. The compare-and-swap operation is atomic in the sense that once it begins, no other virtual processor can access the memory location specified until the compare has completed and the swap (if any) has also completed and is potentially visible to all other virtual processors in the system.

Compare-and-swap is substantially more powerful than the other hardware synchronization primitives. It has an infinite consensus number; that is, it can resolve, in a wait-free fashion, an infinite number of contending processes. Because of this property, compare-and-swap can be used to construct wait-free algorithms that do not require the use of locks. For examples, see *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

9.5.4.2 Swap (SWAP)

SWAP atomically exchanges the lower 32 bits in a virtual processor register with a word in memory. SWAP has a consensus number of two; that is, it cannot resolve more than two contending processes in a wait-free fashion.

9.5.4.3 Load Store Unsigned Byte (LDSTUB)

LDSTUB loads a byte value from memory to a register and writes the value FF_{16} into the addressed byte atomically. LDSTUB is the classic test-and-set instruction. Like SWAP, it has a consensus number of two and so cannot resolve more than two contending processes in a wait-free fashion.

9.5.5 Memory Ordering and Synchronization

The UltraSPARC Architecture provides some level of programmer control over memory ordering and synchronization through the MEMBAR and FLUSH instructions.

MEMBAR serves two distinct functions in SPARC V9. One variant of the MEMBAR, the ordering MEMBAR, provides a way for the programmer to control the order of loads and stores issued by a virtual processor. The other variant of MEMBAR, the sequencing MEMBAR, enables the programmer to explicitly control order and completion for memory operations. Sequencing MEMBARs are needed only when a program requires that the effect of an operation becomes globally visible rather than simply being scheduled.[1] Because both forms are bit-encoded into the instruction, a single MEMBAR can function both as an ordering MEMBAR and as a sequencing MEMBAR.

The SPARC V9 instruction set architecture does not guarantee consistency between instruction and data spaces. A problem arises when instruction space is dynamically modified by a program writing to memory locations containing instructions (Self-Modifying Code). Examples are Lisp, debuggers, and dynamic linking. The FLUSH instruction synchronizes instruction and data memory after instruction space has been modified.

9.5.5.1 Ordering MEMBAR Instructions

Ordering MEMBAR instructions induce an ordering in the instruction stream of a single virtual processor. Sets of loads and stores that appear before the MEMBAR in program order are ordered with respect to sets of loads and stores that follow the MEMBAR in program order. Atomic operations (LDSTUB(A), SWAP(A), CASA, and CASXA) are ordered by MEMBAR as if they were both a load and a store, since they share the semantics of both. An STBAR instruction, with semantics that are a subset of MEMBAR, is provided for SPARC V8 compatibility. MEMBAR and STBAR operate on all pending memory operations in the reorder buffer, independently of their address or ASI, ordering them with respect to all future memory operations. This ordering applies only to memory-reference instructions issued by the virtual processor issuing the MEMBAR. Memory-reference instructions issued by other virtual processors are unaffected.

The ordering relationships are bit-encoded as shown in TABLE 9-3. For example, MEMBAR 01_{16}, written as "`membar #LoadLoad`" in assembly language, requires that all load operations appearing before the MEMBAR in program order complete before any of the load operations following the MEMBAR in program order complete. Store operations are unconstrained in this case. MEMBAR 08_{16} (`#StoreStore`) is equivalent to the STBAR instruction; it requires that the values stored by store instructions appearing in program order prior to the STBAR instruction be visible to other virtual processors before issuing any store operations that appear in program order following the STBAR.

In TABLE 9-3 these ordering relationships are specified by the "<*m*" symbol, which signifies memory order. See Appendix D, *Formal Specification of the Memory Models*, for a formal description of the <*m* relationship.

TABLE 9-3 Ordering Relationships Selected by Mask

Ordering Relation, Earlier <*m* Later	Assembly Language Constant Mnemonic	Effective Behavior in TSO model	Mask Value	nmask Bit #
Load <*m* Load	`#LoadLoad`	nop	01_{16}	0
Store <*m* Load	`#StoreLoad`	`#StoreLoad`	02_{16}	1
Load <*m* Store	`#LoadStore`	nop	04_{16}	2
Store <*m* Store	`#StoreStore`	nop	08_{16}	3

Implementation Note An UltraSPARC Architecture 2007 implementation that only implements the TSO memory model may implement MEMBAR `#LoadLoad`, MEMBAR `#LoadStore`, and MEMBAR `#StoreStore` as nops and MEMBAR `#Storeload` as a MEMBAR `#Sync.`

[1] Sequencing MEMBARs are needed for some input/output operations, forcing stores into specialized stable storage, context switching, and occasional other system functions. Using a sequencing MEMBAR when one is not needed may cause a degradation of performance. See *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*, for examples of the use of sequencing MEMBARs.

9.5.5.2 Sequencing MEMBAR Instructions

A sequencing MEMBAR exerts explicit control over the completion of operations. The three sequencing MEMBAR options each have a different degree of control and a different application.

- **Lookaside Barrier —** Ensures that loads following this MEMBAR are from memory and not from a lookaside into a write buffer. Lookaside Barrier requires that pending stores issued prior to the MEMBAR be completed before any load from that address following the MEMBAR may be issued. A Lookaside Barrier MEMBAR may be needed to provide lock fairness and to support some plausible I/O location semantics. See the example in “Control and Status Registers” in *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*.
- **Memory Issue Barrier —** Ensures that all memory operations appearing in program order before the sequencing MEMBAR complete before any new memory operation may be initiated. See the example in “I/O Registers with Side Effects” in *Programming with the Memory Models*, contained in the separate volume *UltraSPARC Architecture Application Notes*.
- **Synchronization Barrier —** Ensures that all instructions (memory reference and others) preceding the MEMBAR complete and that the effects of any fault or error have become visible before any instruction following the MEMBAR in program order is initiated. A Synchronization Barrier MEMBAR fully synchronizes the virtual processor that issues it.

TABLE 9-4 shows the encoding of these functions in the MEMBAR instruction.

TABLE 9-4 Sequencing Barrier Selected by Mask

Sequencing Function	Assembler Tag	Mask Value	cmask Bit #
Lookaside Barrier	`#Lookaside`	10_{16}	0
Memory Issue Barrier	`#MemIssue`	20_{16}	1
Synchronization Barrier	`#Sync`	40_{16}	2

Implementation Note | In UltraSPARC Architecture 2007 implementations, MEMBAR `#Lookaside` and MEMBAR `#MemIssue` are typically implemented as a MEMBAR `#Sync.`

For more details, see the MEMBAR instruction on page 201 of Chapter 7, *Instructions*.

9.5.5.3 Synchronizing Instruction and Data Memory

The SPARC V9 memory models do not require that instruction and data memory images be consistent at all times. The instruction and data memory images may become inconsistent if a program writes into the instruction stream. As a result, whenever instructions are modified by a program in a context where the data (that is, the instructions) in the memory and the data cache hierarchy may be inconsistent with instructions in the instruction cache hierarchy, some special programmatic (software) action must be taken.

The FLUSH instruction will ensure consistency between the in-flight instruction stream and the data references in the virtual processor executing FLUSH. The programmer must ensure that the modification sequence is robust under multiple updates and concurrent execution. Since, in general, loads and stores may be performed out of order, appropriate MEMBAR and FLUSH instructions must be interspersed as needed to control the order in which the instruction data are modified.

The FLUSH instruction ensures that subsequent instruction fetches from the doubleword target of the FLUSH by the virtual processor executing the FLUSH appear to execute after any loads, stores, and atomic load-stores issued by the virtual processor to that address prior to the FLUSH. FLUSH acts as a barrier for instruction fetches in the virtual processor on which it executes and has the properties of a store with respect to MEMBAR operations.

IMPL. DEP. #122-V9: The latency between the execution of FLUSH on one virtual processor and the point at which the modified instructions have replaced outdated instructions in a multiprocessor is implementation dependent.

> **Programming Note** Because FLUSH is designed to act on a doubleword and because, on some implementations, FLUSH may trap to system software, it is recommended that system software provide a user-callable service routine for flushing arbitrarily sized regions of memory. On some implementations, this routine would issue a series of FLUSH instructions; on others, it might issue a single trap to system software that would then flush the entire region.

On an UltraSPARC Architecture virtual processor:

- A FLUSH instruction causes a synchronization with the virtual processor, which flushes the instruction pipeline in the virtual processor on which the FLUSH instruction is executed.
- Coherency between instruction and data memories may or may not be maintained by hardware. If it is, an UltraSPARC Architecture implementation may ignore the address in the operands of a FLUSH instruction.

> **Programming Note** UltraSPARC Architecture virtual processors are not required to maintain coherency between instruction and data caches in hardware. Therefore, portable software must do the following:
>
> (1) must always assume that store instructions (except Block Store with Commit) do not coherently update instruction cache(s);
>
> (2) must, in every FLUSH instruction, supply the address of the instruction or instructions that were modified.

For more details, see the FLUSH instruction on page 133 of Chapter 7, *Instructions*.

9.6 Nonfaulting Load

A nonfaulting load behaves like a normal load, with the following exceptions:

- A nonfaulting load from a location with side effects (TTE.e = 1) causes a *DAE_side_effect_page* exception.
- A nonfaulting load from a page marked for nonfault access only (TTE.nfo = 1) is allowed; other types of accesses to such a page cause a *DAE_nfo_page* exception.
- These loads are issued with `ASI_PRIMARY_NO_FAULT[_LITTLE]` or `ASI_SECONDARY_NO_FAULT[_LITTLE]`. A store with a `NO_FAULT` ASI causes a *DAE_invalid_asi* exception.

Typically, optimizers use nonfaulting loads to move loads across conditional control structures that guard their use. This technique potentially increases the distance between a load of data and the first use of that data, in order to hide latency. The technique allows more flexibility in instruction scheduling and improves performance in certain algorithms by removing address checking from the critical code path.

For example, when following a linked list, nonfaulting loads allow the null pointer to be accessed safely in a speculative, read-ahead fashion; the page at virtual address 0_{16} can safely be accessed with no penalty. The TTE.nfo bit marks pages that are mapped for safe access by nonfaulting loads but that can still cause a trap by other, normal accesses.

Thus, programmers can trap on “wild” pointer references—many programmers count on an exception being generated when accessing address 0_{16} to debug software—while benefiting from the acceleration of nonfaulting access in debugged library routines.

9.7 Store Coalescing

Cacheable stores may be coalesced with adjacent cacheable stores within an 8 byte boundary offset in the store buffer to improve store bandwidth. Similarly non-side-effect-noncacheable stores may be coalesced with adjacent non-side-effect noncacheable stores within an 8-byte boundary offset in the store buffer.

In order to maintain strong ordering for I/O accesses, stores with side-effect attribute (`e` bit set) will not be combined with any other stores.

Stores that are separated by an intervening MEMBAR `#Sync` will not be coalesced.

CHAPTER **10**

Address Space Identifiers (ASIs)

This appendix describes address space identifiers (ASIs) in the following sections:

10.1 Address Space Identifiers and Address Spaces

An UltraSPARC Architecture processor provides an address space identifier (ASI) with every address sent to memory. The ASI does the following:

- Distinguishes between different address spaces
- Provides an attribute that is unique to an address space
- Maps internal control and diagnostics registers within a virtual processor

The memory management unit uses a 64-bit virtual address and an 8-bit ASI to generate a memory, I/O, or internal register address.

10.2 ASI Values

The range of address space identifiers (ASIs) is 00_{16}-FF_{16}. That range is divided into restricted and unrestricted portions. ASIs in the range 80_{16}–FF_{16} are unrestricted; they may be accessed by software running in any privilege mode.

ASIs in the range 00_{16}–$7F_{16}$ are restricted; they may only be accessed by software running in a mode with sufficient privilege for the particular ASI. ASIs in the range 00_{16}–$2F_{16}$ may only be accessed by software running in privileged or hyperprivileged mode and ASIs in the range 30_{16}–$7F_{16}$ may only be accessed by software running in hyperprivileged mode.

SPARC V9 Compatibility Note In SPARC V9, the range of ASIs was evenly divided into restricted (00_{16}-$7F_{16}$) and unrestricted (80_{16}-FF_{16}) halves.

An attempt by nonprivileged software to access a restricted (privileged or hyperprivileged) ASI (00_{16}–$7F_{16}$) causes a *privileged_action* trap.

An attempt by privileged software to access a hyperprivileged ASI (30_{16}–$7F_{16}$) also causes a *privileged_action* trap.

An ASI can be categorized based on how it affects the MMU's treatment of the accompanying address, into one of three categories:

- A *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 UltraSPARC Architecture processor has an 8-bit Address Space Identifier (ASI) appended to the virtual address (VA). The VA plus the ASI fully specify the address.

For instruction fetches and for data loads, stores, and load-stores that do not use the load or store alternate instructions, the ASI is an implicit ASI generated by the virtual processor.

If a load alternate, store alternate, or load-store alternate instruction is used, the value of the ASI (an "explicit ASI") can be specified in the ASI register or as an immediate value in the instruction.

In practice, ASIs are not only used to differentiate address spaces but are also used for other functions like referencing registers in the MMU unit.

10.3.1 Supported ASIs

TABLE 10-1 lists architecturally-defined ASIs; some are in all UltraSPARC Architecture implementations and some are only present in some implementations.

An ASI marked with a closed bullet (●) is required to be implemented on all UltraSPARC Architecture 2007 processors.

An ASI marked with an open bullet (❍) is defined by the UltraSPARC Architecture 2007 but is not necessarily implemented in all UltraSPARC Architecture 2007 processors; its implemention is optional. Across all implementations on which it is implemented, it appears to software to behave identically.

Some ASIs may only be used with certain load or store instructions; see table footnotes for details.

The word "decoded" in the Virtual Address column of TABLE 10-1 indicates that the the supplied virtual address is decoded by the virtual processor.

The "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 implemenations. ASIs marked "implementation dependent" may be used for implementation-specific purposes.

Attempting to access an address space described as “Implementation dependent” in TABLE 10-1 produces implementation-dependent results.

TABLE 10-1 UltraSPARC Architecture ASIs *(1 of 5)*

ASI Value	req'd (●) opt'l (❍)	ASI Name (and Abbreviation)	Access Type(s)	Virtual Address (VA)	V/ non-T/ R	Shared /per strand	Description
00_{16}– 03_{16}	❍	—	—[2,12]	—	—	—	Implementation dependent[1]
04_{16}	●	ASI_NUCLEUS (ASI_N)	RW[2,4]	(decoded)	V	—	Implicit address space, nucleus context, TL > 0
05_{16}– $0B_{16}$	❍	—	—[2,12]	—	—	—	Implementation dependent[1]
$0C_{16}$	●	ASI_NUCLEUS_LITTLE (ASI_NL)	RW[2,4]	(decoded)	V	—	Implicit address space, nucleus context, TL > 0, little-endian
$0D_{16}$– $0F_{16}$	❍	—	—[2,12]	—	—	—	Implementation dependent[1]
10_{16}	●	ASI_AS_IF_USER_PRIMARY (ASI_AIUP)	RW[2,4,18]	(decoded)	V	—	Primary address space, as if user (nonprivileged)
11_{16}	●	ASI_AS_IF_USER_SECONDARY (ASI_AIUS)	RW[2,4,18]	(decoded)	V	—	Secondary address space, as if user (nonprivileged)
12_{16}– 13_{16}	❍	—	—[2,12]	—	—	—	Implementation dependent[1]
14_{16}	❍	ASI_REAL	RW[2,4]	(decoded)	R	—	Real address
15_{16}	❍	ASI_REAL_IO[D]	RW[2,5]	(decoded)	R	—	Real address, noncacheable, with side effect (deprecated)
16_{16}	❍	ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)	RW[2,8,14,18]	(decoded)	V	—	Primary address space, block load/store, as if user (nonprivileged)
17_{16}	❍	ASI_BLOCK_AS_IF_USER_SECONDAR Y (ASI_BLK_AIUS)	RW[2,8,14,18]	(decoded)	V	—	Secondary address space, block load/store, as if user (nonprivileged)
18_{16}	●	ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)	RW[2,4,18]	(decoded)	V	—	Primary address space, as if user (nonprivileged), little-endian
19_{16}	●	ASI_AS_IF_USER_SECONDARY_ LITTLE (ASI_AIUSL)	RW[2,4,18]	(decoded)	V	—	Secondary address space, as if user (nonprivileged), little-endian
$1A_{16}$– $1B_{16}$	❍	—	—[2,12]	—	—	—	Implementation dependent[1]
$1C_{16}$	❍	ASI_REAL_LITTLE (ASI_REAL_L)	RW [2,4]	(decoded)	R	—	Real address, little-endian
$1D_{16}$	❍	ASI_REAL_IO_LITTLE[D] (ASI_REAL_IO_L[D])	RW [2,5]	(decoded)	R	—	Real address, noncacheable, with side effect, little-endian (deprecated)
$1E_{16}$	❍	ASI_BLOCK_AS_IF_USER_PRIMARY_ LITTLE (ASI_BLK_AIUPL)	RW[2,8,14,18]	(decoded)	V	—	Primary address space, block load/store, as if user (nonprivileged), little-endian
$1F_{16}$	❍	ASI_BLOCK_AS_IF_USER_ SECONDARY_LITTLE (ASI_BLK_AIUS_L)	RW[2,8,14,18]	(decoded)	V	—	Secondary address space, block load/store, as if user (nonprivileged), little-endian

TABLE 10-1 UltraSPARC Architecture ASIs *(2 of 5)*

ASI Value	req'd (●) opt'l (❍)	ASI Name (and Abbreviation)	Access Type(s)	Virtual Address (VA)	V/ non-T/ R	Shared /per strand	Description
20_{16}	❍	ASI_SCRATCHPAD	RW[2,6]	(decoded; see below)	non-T	per strand	Privileged Scratchpad registers; implementation dependent[1]
	❍		"	0_{16}	"	"	Scratchpad Register 0[1]
	❍		"	8_{16}	"	"	Scratchpad Register 1[1]
	❍		"	10_{16}	"	"	Scratchpad Register 2[1]
	❍		"	18_{16}	"	"	Scratchpad Register 3[1]
	❍			20_{16}	"	"	Scratchpad Register 4[1]
	❍		"	28_{16}	"	"	Scratchpad Register 5[1]
	❍		"	30_{16}	"	"	Scratchpad Register 6[1]
	❍		"	38_{16}	"	"	Scratchpad Register 7[1]
21_{16}	❍	ASI_MMU_CONTEXTID	RW[2,6]	(decoded; see below)	non-T	per strand	MMU context registers
	❍		"	8_{16}	"	"	I/D MMU Primary Context ID register 0
	❍		"	10_{16}	"	"	I/D MMU Secondary Context ID register_0
	❍		"	108_{16}	"	"	I/D Primary Context ID register 1
	❍		"	110_{16}	"	"	I/D MMU Secondary Context ID register 1
22_{16}	❍	ASI_TWINX_AS_IF_USER_PRIMARY (ASI_TWINX_AIUP)	R[2,7,11]	(decoded)	V	—	Primary address space, 128-bit atomic load twin extended word, as if user (nonprivileged)
23_{16}	❍	ASI_TWINX_AS_IF_USER_SECONDARY (ASI_TWINX_AIUS)	R[2,7,11]	(decoded)	V	—	Secondary address space, 128-bit atomic load twin extended word, as if user (nonprivileged)
24_{16}	❍	—	—	—	—	—	Implementation dependent[1]
25_{16}	❍	ASI_QUEUE	(see below)	(decoded; see below)	non-T	per strand	
	❍		RW[2,6]	$3C0_{16}$	"	"	CPU Mondo Queue Head Pointer
	❍		RW[2,6,17]	$3C8_{16}$	"	"	CPU Mondo Queue Tail Pointer
	❍		RW[2,6]	$3D0_{16}$	"	"	Device Mondo Queue Head Pointer
	❍		RW[2,6,17]	$3D8_{16}$	"	"	Device Mondo Queue Tail Pointer
	❍		RW[2,6]	$3E0_{16}$	"	"	Resumable Error Queue Head Pointer
	❍		RW[2,6,17]	$3E8_{16}$	"	"	Resumable Error Queue Tail Pointer
	❍		RW[2,6]	$3F0_{16}$	"	"	Nonresumable Error Queue Head Pointer
	❍		RW[2,6,17]	$3F8_{16}$	"	"	Nonresumable Error Queue Tail Pointer

TABLE 10-1 UltraSPARC Architecture ASIs *(3 of 5)*

ASI Value	req'd (●) opt'l (❍)	ASI Name (and Abbreviation)	Access Type(s)	Virtual Address (VA)	V/ non-T/ R	Shared /per strand	Description
26_{16}	❍	`ASI_TWINX_REAL` (`ASI_TWINX_R`) `ASI_QUAD_LDD_REAL`[D†]	R[2,7,11]	(decoded)	R	—	128-bit atomic twin extended-word load from real address
27_{16}	❍	`ASI_TWINX_NUCLEUS` (`ASI_TWINX_N`)	R[2,7,11]	(decoded)	V	—	Nucleus context, 128-bit atomic load twin extended-word
28_{16}–29_{16}	❍	—	—[2,12]	—	—	—	Implementation dependent[1]
$2A_{16}$	❍	`ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE` (`ASI_TWINXAIUPL`)	R[2,7,11]	(decoded)	V	—	Primary address space, 128-bit atomic load twin extended-word, as if user (nonprivileged), little-endian
$2B_{16}$	❍	`ASI_TWINX_AS_IF_USER_SECONDARY_LITTLE` (`ASI_TWINX_AIUS_L`)	R[2,7,11]	(decoded)	V	—	Secondary address space, 128-bit atomic load twin extended-word, as if user (nonprivileged), little-endian
$2C_{16}$	❍	—	—[2]	—	—	—	Implementation dependent[1]
$2D_{16}$	❍	—	—[2,12]	—	—	—	Implementation dependent[1]
$2E_{16}$	❍	`ASI_TWINX_REAL_LITTLE` (`ASI_TWINX_REAL_L`) `ASI_QUAD_LDD_REAL_LITTLE`[D†]	R[2,7,11]	(decoded)	R	—	128-bit atomic twin-extended-word load from real address, little-endian
$2F_{16}$	❍	`ASI_TWINX_NUCLEUS_LITTLE` (`ASI_TWINX_NL`)	R[2,7,11]	(decoded)	V	—	Nucleus context, 128-bit atomic load twin extended-word, little-endian
30_{16}–$7F_{16}$	●	—	—[3]	—	—	—	Reserved for use in hyperprivilege mode
45_{16}	❍	—	—[3,13]	—	—	—	Implementation dependent[1]
46_{16}–48_{16}	❍	—	—[3,13]	—	—	—	Implementation dependent[1]
49_{16}	❍	—	—[3,13]	—	—	—	Implementation dependent[1]
$4A_{16}$–$4B_{16}$	❍	—	—[3,13]	—	—	—	Implementation dependent[1]
$4C_{16}$	❍	Error Status and Enable Registers					Implementation dependent[1]
75_{16}–$7F_{16}$	●	—	—[3,13]	—	—	—	*Reserved*
80_{16}	●	`ASI_PRIMARY` (`ASI_P`)	RW[4]	(decoded)	V	—	Implicit primary address space
81_{16}	●	`ASI_SECONDARY` (`ASI_S`)	RW[4]	(decoded)	V	—	Secondary address space
82_{16}	●	`ASI_PRIMARY_NO_FAULT` (`ASI_PNF`)	R[9,11]	(decoded)	V	—	Primary address space, no fault
83_{16}	●	`ASI_SECONDARY_NO_FAULT` (`ASI_SNF`)	R[9,11]	(decoded)	V	—	Secondary address space, no fault
84_{16}–87_{16}	●	—	—[16]	—	—	—	*Reserved*
88_{16}	●	`ASI_PRIMARY_LITTLE` (`ASI_PL`)	RW[4]	(decoded)	V	—	Implicit primary address space, little-endian
89_{16}	●	`ASI_SECONDARY_LITTLE` (`ASI_SL`)	RW[4]	(decoded)	V	—	Secondary address space, little-endian
$8A_{16}$	●	`ASI_PRIMARY_NO_FAULT_LITTLE` (`ASI_PNFL`)	R[9,11]	(decoded)	V	—	Primary address space, no fault, little-endian

TABLE 10-1 UltraSPARC Architecture ASIs *(4 of 5)*

ASI Value	req'd (●) opt'l (❍)	ASI Name (and Abbreviation)	Access Type(s)	Virtual Address (VA)	V/ non-T/ R	Shared /per strand	Description
$8B_{16}$	●	ASI_SECONDARY_NO_FAULT_LITTLE (ASI_SNFL)	R[9,11]	(decoded)	V	—	Seondary address space, no fault, little-endian
$8C_{16}$– BF_{16}	●	—	—[16]	—	—	—	*Reserved*
$C0_{16}$	❍	ASI_PST8_PRIMARY (ASI_PST8_P)	W[8,10,14]	(decoded)	V	—	Primary address space, 8×8-bit partial store
$C1_{16}$	❍	ASI_PST8_SECONDARY (ASI_PST8_S)	W[8,10,14]	(decoded)	V	—	Secondary address space, 8x8-bit partial store
$C2_{16}$	❍	ASI_PST16_PRIMARY (ASI_PST16_P)	W[8,10,14]	(decoded)	V	—	Primary address space, 4×16-bit partial store
$C3_{16}$	❍	ASI_PST16_SECONDARY (ASI_PST16_S)	W[8,10,14]	(decoded)	V	—	Secondary address space, 4×16-bit partial store
$C4_{16}$	❍	ASI_PST32_PRIMARY (ASI_PST32_P)	W[8,10,14]	(decoded)	V	—	Primary address space, 2x32-bit partial store
$C5_{16}$	❍	ASI_PST32_SECONDARY (ASI_PST32_S)	W[8,10,14]	(decoded)	V	—	Secondary address space, 2×32-bit partial store
$C6_{16}$– $C7_{16}$	●	—	—[15]	—	—	—	Implementation dependent[1]
$C8_{16}$	❍	ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL)	W[8,10,14]	(decoded)	V	—	Primary address space, 8x8-bit partial store, little-endian
$C9_{16}$	❍	ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL)	W[8,10,14]	(decoded)	V	—	Secondary address space, 8×8-bit partial store, little-endian
CA_{16}	❍	ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL)	W[8,10,14]	(decoded)	V	—	Primary address space, 4x16-bit partial store, little-endian
CB_{16}	❍	ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL)	W[8,10,14]	(decoded)	V	—	Secondary address space, 4×16-bit partial store, little-endian
CC_{16}	❍	ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL)	W[8,10,14]	(decoded)	V	—	Primary address space, 2×32-bit partial store, little-endian
CD_{16}	❍	ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL)	W[8,10,14]	(decoded)	V	—	Second address space, 2×32-bit partial store, little-endian
CE_{16}– CF_{16}	●	—	—[15]	—	—	—	Implementation dependent[1]
$D0_{16}$	❍	ASI_FL8_PRIMARY (ASI_FL8_P)	RW[8,14]	(decoded)	V	—	Primary address space, one 8-bit floating-point load/store
$D1_{16}$	❍	ASI_FL8_SECONDARY (ASI_FL8_S)	RW[8,14]	(decoded)	V	—	Second address space, one 8-bit floating-point load/store
$D2_{16}$	❍	ASI_FL16_PRIMARY (ASI_FL16_P)	RW[8,14]	(decoded)	V	—	Primary address space, one 16-bit floating-point load/store
$D3_{16}$	❍	ASI_FL16_SECONDARY (ASI_FL16_S)	RW[8,14]	(decoded)	V	—	Second address space, one 16-bit floating-point load/store
$D4_{16}$– $D7_{16}$	●	—	—[15]	—	—	—	Implementation dependent[1]
$D8_{16}$	❍	ASI_FL8_PRIMARY_LITTLE (ASI_FL8_PL)	RW[8,14]	(decoded)	V	—	Primary address space, one 8-bit floating point load/store, little-endian

TABLE 10-1 UltraSPARC Architecture ASIs *(5 of 5)*

ASI Value	req'd(●) opt'l (❍)	ASI Name (and Abbreviation)	Access Type(s)	Virtual Address (VA)	V/ non-T/ R	Shared /per strand	Description
$D9_{16}$	❍	ASI_FL8_SECONDARY_LITTLE (ASI_FL8_SL)	RW[8,14]	(decoded)	V	—	Second address space, one 8-bit floating point load/store, little-endian
DA_{16}	❍	ASI_FL16_PRIMARY_LITTLE (ASI_FL16_PL)	RW[8,14]	(decoded)	V	—	Primary address space, one 16-bit floating-point load/store, little-endian
DB_{16}	❍	ASI_FL16_SECONDARY_LITTLE (ASI_FL16_SL)	RW[8,14]	(decoded)	V	—	Second address space, one 16-bit floating point load/store, little-endian
DC_{16} –DF_{16}	●	—	—[15]	—	—	—	Implementation dependent[1]
$E0_{16}$	❍	ASI_BLOCK_COMMIT_PRIMARY (ASI_BLK_COMMIT_P)	W[8,11,14]	(decoded)	V	—	Primary address space, 8x8-byte block store commit operation
$E1_{16}$	❍	ASI_BLOCK_COMMIT_SECONDARY (ASI_BLK_COMMIT_S)	W[8,11,14]	(decoded)	V	—	Secondary address space, 8x8-byte block store commit operation
$E2_{16}$	❍	ASI_TWINX_PRIMARY (ASI_TWINX_P)	R[19]	(decoded)	V	—	Primary address space, 128-bit atomic load twin extended word
$E3_{16}$	❍	ASI_TWINX_SECONDARY (ASI_TWINX_S)	R[19]	(decoded)	V	—	Secondary address space, 128-bit atomic load twin extended-word
$E4_{16}$– $E9_{16}$	●	—	—[15]	—	—	—	Implementation dependent[1]
EA_{16}	❍	ASI_TWINX_PRIMARY_LITTLE (ASI_TWINX_PL)	R[19]	(decoded)	V	—	Primary address space, 128-bit atomic load twin extended word, little endian
EB_{16}	❍	ASI_TWINX_SECONDARY_LITTLE (ASI_TWINX_SL)	R[19]	(decoded)	V	—	Secondary address space, 128-bit atomic load twin extended word, little endian
EC_{16}– EF_{16}	❍	—	—[15]	—	—	—	Implementation dependent[1]
$F0_{16}$	❍	ASI_BLOCK_PRIMARY (ASI_BLK_P)	RW[8,14]	(decoded)	V	—	Primary address space, 8x8-byte block load/store
$F1_{16}$	❍	ASI_BLOCK_SECONDARY (ASI_BLK_S)	RW[8,14]	(decoded)	V	—	Secondary address space, 8x8- byte block load/store
$F2_{16}$– $F5_{16}$	●	—	—[15]	—	—	—	Implementation dependent[1]
$F6_{16}$– $F7_{16}$	●	—	—	—	—	—	Implementation dependent[1]
$F8_{16}$	❍	ASI_BLOCK_PRIMARY_LITTLE (ASI_BLK_PL)	RW[8,14]	(decoded)	V	—	Primary address space, 8x8-byte block load/store, little endian
$F9_{16}$	❍	ASI_BLOCK_SECONDARY_LITTLE (ASI_BLK_SL)	RW[8,14]	(decoded)	V	—	Secondary address space, 8x8- byte block load/store, little endian
FA_{16}– FD_{16}	●	—	—[15]	—	—	—	Implementation dependent[1]
FE_{16}– FF_{16}	●	—	—[15]	—	—	—	Implementation dependent[1]

† This ASI name has been changed, for consistency; although use of this name is deprecated and software should use the new name, the old name is listed here for compatibility.

1 Implementation dependent ASI (impl. dep. #29); available for use by implementors. Software that references this ASI may not be portable.

2 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in nonprivileged mode causes a *privileged_action* exception.

3 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in nonprivileged mode or privileged mode causes a *privileged_action* exception.

4 May be used with all load alternate, store alternate, atomic alternate and prefetch alternate instructions (CASA, CASXA, LDSTUBA, LDTWA, LDDFA, LDFA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDXA, PREFETCHA, STBA, STTWA, STDFA, STFA, STHA, STWA, STXA, SWAPA).

5 May be used with all of the following load alternate and store alternate instructions: LDTWA, LDDFA, LDFA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDXA, STBA, STTWA, STDFA, STFA, STHA, STWA, STXA. Use with an atomic alternate or prefetch alternate instruction (CASA, CASXA, LDSTUBA, SWAPA or PREFETCHA) causes a *DAE_invalid_asi* exception.

6 May only be used in a LDXA or STXA instruction for RW ASIs, LDXA for read-only ASIs and STXA for write-only ASIs. Use of LDXA for write-only ASIs, STXA for read-only ASIs, or any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a *DAE_invalid_asi* exception.

7 May only be used in an LDTXA instruction. Use of this ASI in any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a *DAE_invalid_asi* exception.

8 May only be used in a LDDFA or STDFA instruction for RW ASIs, LDDFA for read-only ASIs and STDFA for write-only ASIs. Use of LDDFA for write-only ASIs, STDFA for read-only ASIs, or any other load alternate, store alternate, atomic alternate or prefetch alternate instruction causes a *DAE_invalid_asi* exception.

9 May be used with all of the following load and prefetch alternate instructions: LDTWA, LDDFA, LDFA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, LDUWA, LDXA, PREFETCHA. Use with an atomic alternate or store alternate instruction causes a *DAE_invalid_asi* exception.

10 Write(store)-only ASI; an attempted load alternate, atomic alternate, or prefetch alternate instruction to this ASI causes a *DAE_invalid_asi* exception.

11 Read(load)-only ASI; an attempted store alternate or atomic alternate instruction to this ASI causes a *DAE_invalid_asi* exception.

12 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in privileged mode causes a *DAE_invalid_asi* exception.

14 An attempted access to this ASI may cause an exception (see *Special Memory Access ASIs* on page 329 for details).

15 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to this ASI in any mode causes a *DAE_invalid_asi* exception if this ASI is not implemented by the model dependent implementation.

16 An attempted load alternate, store alternate, atomic alternate or prefetch alternate instruction to a reserved ASI in any mode causes a *DAE_invalid_asi* exception.

17 The Queue Tail Registers (ASI 25_{16}) are read-only. An attempted write to the Queue Tail Registers causes a *DAE_invalid_asi* exception

19 May only be used in an LDTXA (load twin-extended-word) instruction (which shares an opcode with LDTWA). Use of this ASI in any other load instruction causes a *DAE_invalid_asi* exception.

10.4 Special Memory Access ASIs

This section describes special memory access ASIs that are not described in other sections.

10.4.1 ASIs 10_{16}, 11_{16}, 16_{16}, 17_{16} and 18_{16} (ASI_*AS_IF_USER_*)

These ASI are intended to be used in accesses from privileged mode, but are processed as if they were issued from nonprivileged mode. Therefore, they are subject to privilege-related exceptions. They are distinguished from each other by the context from which the access is made, as described in TABLE 10-2.

When one of these ASIs is specified in a load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- In any other privilege mode:
  - If U/DMMU TTE.p = 1, a *DAE_privilege_violation* exception occurs
  - Otherwise, the access occurs and its endianness is determined by the U/DMMU TTE.ie bit. If U/DMMU TTE.ie = 0, the access is big-endian; otherwise, it is little-endian.

TABLE 10-2 Privileged ASI_*AS_IF_USER_* ASIs

ASI	Names	Addressing (Context)	Endianness of Access
10_{16}	ASI_AS_IF_USER_PRIMARY (ASI_AIUP)	Virtual (Primary)	Big-endian when U/DMMU TTE.ie = 0; little-endian when U/DMMU TTE.ie = 1
11_{16}	ASI_AS_IF_USER_SECONDARY (ASI_AIUS)	Virtual (Secondary)	
16_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)	Virtual (Primary)	
17_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS)	Virtual (Secondary)	

10.4.2 ASIs 18_{16}, 19_{16}, $1E_{16}$, and $1F_{16}$ (ASI_*AS_IF_USER_*_LITTLE)

These ASIs are little-endian versions of ASIs 10_{16}, 11_{16}, 16_{16}, and 17_{16} (ASI_AS_IF_USER_*), described in section 10.4.1. Each operates identically to the corresponding non-little-endian ASI, except that if an access occurs its endianness is the opposite of that for the corresponding non-little-endian ASI.

These ASI are intended to be used in accesses from privileged mode, but are processed as if they were issued from nonprivileged mode. Therefore, they are subject to privilege-related exceptions. They are distinguished from each other by the context from which the access is made, as described in TABLE 10-3.

When one of these ASIs is specified in a load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- In any other privilege mode:

- If U/DMMU TTE.p = 1, a *DAE_privilege_violation* exception occurs
- Otherwise, the access occurs and its endianness is determined by the U/DMMU TTE.ie bit. If U/DMMU TTE.ie = 0, the access is little-endian; otherwise, it is big-endian.

TABLE 10-3 Privileged `ASI_*AS_IF_USER_*_LITTLE` ASIs

ASI	Names	Addressing (Context)	Endianness of Access
18_{16}	ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)	Virtual (Primary)	Little-endian when U/DMMU TTE.ie = 0; big-endian when U/DMMU TTE.ie = 1
19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)	Virtual (Secondary)	
$1E_{16}$	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUP)	Virtual (Primary)	
$1F_{16}$	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL)	Virtual (Secondary)	

10.4.3 ASI 14_{16} (ASI_REAL)

When ASI_REAL is specified in any load alternate, store alternate or prefetch alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- In any other privilege mode:
  - VA is passed through to RA
  - During the address translation, context values are disregarded.
  - The endianness of the access is dertermined by the U/DMMU TTE.ie bit; if U/DMMU TTE.ie = 0, the access is big-endian, otherwise it is little-endian.

Even if data address translation is disabled, an access with this ASI is still a cacheable access.

10.4.4 ASI 15_{16} (ASI_REAL_IO)

Accesses with ASI_REAL_IO bypass the external cache and behave as if the side effect bit (TTE.e bit) is set. When this ASI is specified in any load alternate or store alternate instruction, the virtual processor behaves as follows:

- In nonprivileged mode, a *privileged_action* exception occurs
- If used with a CASA, CASXA, LDSTUBA, SWAPA, or PREFETCHA instruction, a *DAE_invalid_asi* exception occurs
- Used with any other load alternate or store alternate instuction, in privileged mode:
  - VA is passed through to RA
  - During the address translation, context values are disregarded.
  - The endianness of the access is dertermined by the U/DMMU TTE.ie bit; if U/DMMU TTE.ie = 0, the access is big-endian, otherwise it is little-endian.

10.4.5 ASI $1C_{16}$ (ASI_REAL_LITTLE)

ASI_REAL_LITTLE is a little-endian version of ASI 14_{16} (ASI_REAL). It operates identically to ASI_REAL, except if an access occurs, its endianness the opposite of that for ASI_REAL.

10.4.6 ASI $1D_{16}$ (ASI_REAL_IO_LITTLE)

ASI_REAL_IO_LITTLE is a little-endian version of ASI 15_{16} (ASI_REAL_IO). It operates identically to ASI_REAL_IO, except if an access occurs, its endianness the opposite of that for ASI_REAL_IO.

10.4.7 ASIs 22_{16}, 23_{16}, 27_{16}, $2A_{16}$, $2B_{16}$, $2F_{16}$ (Privileged Load Integer Twin Extended Word)

ASIs 22_{16}, 23_{16}, 27_{16}, $2A_{16}$, $2B_{16}$ and $2F_{16}$ exist for use with the (nonportable) LDTXA instruction as atomic Load Integer Twin Extended Word operations (see *Load Integer Twin Extended Word from Alternate Space* on page 197). These ASIs are distinguished by the context from which the access is made and the endianness of the access, as described in TABLE 10-4.

TABLE 10-4 Privileged Load Integer Twin Extended Word ASIs

ASI	Names	Addressing (Context)	Endianness of Access
22_{16}	ASI_TWINX_AS_IF_USER_PRIMARY (ASI_TWINX_AIUP)	Virtual (Primary)	Big-endian when U/DMMU TTE.ie = 0; little-endian when U/DMMU TTE.ie = 1
23_{16}	ASI_TWINX_AS_IF_USER_SECONDARY (ASI_TWINX_AIUS)	Virtual (Secondary)	
27_{16}	ASI_TWINX_NUCLEUS (ASI_TWINX_N)	Virtual (Nucleus)	
$2A_{16}$	ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE (ASI_TWINX_AIUP_L)	Virtual (Primary)	Little-endian when U/DMMU TTE.ie = 0; big-endian when U/DMMU TTE.ie = 1
$2B_{16}$	ASI_TWINX_AS_IF_USER_SECONDARY_LITTLE (ASI_TWINX_AIUS_L)	Virtual (Secondary)	
$2F_{16}$	ASI_TWINX_NUCLEUS_LITTLE (ASI_TWINX_NL)	Virtual (Nucleus)	

When these ASIs are used with LDTXA, a *mem_address_not_aligned* exception is generated if the operand address is not 16-byte aligned.

If these ASIs are used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

Compatibility Note These ASIs replaced ASIs 24_{16} and $2C_{16}$ used in earlier UltraSPARC implementations; see the detailed Compatibility Note on page 335 for details.

10.4.8 ASIs 26_{16} and $2E_{16}$ (Privileged Load Integer Twin Extended Word, Real Addressing)

ASIs 26_{16} and $2E_{16}$ exist for use with the LDTXA instruction as atomic Load Integer Twin Extended Word operations using Real addressing (see *Load Integer Twin Extended Word from Alternate Space* on page 197). These two ASIs are distinguished by the endianness of the access, as described in TABLE 10-5.

TABLE 10-5 Load Integer Twin Extended Word (Real) ASIs

ASI	Name	Addressing (Context)	Endianness of Access
26_{16}	ASI_TWINX_REAL (ASI_TWINX_R)	Real (—)	Big-endian when U/DMMU TTE.ie = 0; little-endian when U/DMMU TTE.ie = 1
$2E_{16}$	ASI_TWINX_REAL_LITTLE (ASI_TWINX_REAL_L)	Real (—)	Little-endian when U/DMMU TTE.ie = 0; big-endian when U/DMMU TTE.ie = 1

When these ASIs are used with LDTXA, a *mem_address_not_aligned* exception is generated if the operand address is not 16-byte aligned.

If these ASIs are used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

Compatibility Note These ASIs replaced ASIs 34_{16} and $3C_{16}$ used in earlier UltraSPARC implementations; see the Compatibility Note on page 335 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 197). These ASIs are distinguished by the address space accessed (Primary or Secondary) and the endianness of the access, as described in TABLE 10-6.

TABLE 10-6 Load Integer Twin Extended Word ASIs

ASI	Names	Addressing (Context)	Endianness of Access
$E2_{16}$	ASI_TWINX_PRIMARY (ASI_TWINX_P)	Virtual (Primary)	Big-endian when U/DMMU TTE.ie = 0, little-endian when U/DMMU TTE.ie = 1
$E3_{16}$	ASI_TWINX_SECONDARY (ASI_TWINX_S)	Virtual (Secondary)	
EA_{16}	ASI_TWINX_PRIMARY_LITTLE (ASI_TWINX_PL)	Virtual (Primary)	Little-endian when U/DMMU TTE.ie = 0, big-endian when U/DMMU TTE.ie = 1
EB_{16}	ASI_TWINX_SECONDARY_LITTLE (ASI_TWINX_SL)	Virtual (Secondary)	

When these ASIs are used with LDTXA, a *mem_address_not_aligned* exception is generated if the operand address is not 16-byte aligned.

If these ASIs are used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

10.4.10 Block Load and Store ASIs

ASIs 16_{16}, 17_{16}, $1E_{16}$, $1F_{16}$, $E0_{16}$, $E1_{16}$, $F0_{16}$, $F1_{16}$, $F8_{16}$, and $F9_{16}$ exist for use with LDDFA and STDFA instructions as Block Load (LDBLOCKF) and Block Store (STBLOCKF) operations (see *Block Load* on page 178 and *Block Store* on page 250).

When these ASIs are used with the LDDFA (STDFA) opcode for Block Load (Store), a *mem_address_not_aligned* exception is generated if the operand address is not 64-byte aligned.

ASIs $E0_{16}$ and $E1_{16}$ are only defined for use in Block Store with Commit operations (see page 250). Neither ASI $E0_{16}$ nor $E1_{16}$ should be used with the LDDFA opcode; however, if either *is* used, the resulting behavior is specified in the LDDFA instruction description on page 184.

If a Block Load or Block Store ASI is used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

10.4.11 Partial Store ASIs

ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–CD_{16} exist for use with the STDFA instruction as Partial Store (STPARTIALF) operations (see *Store Partial Floating-Point* on page 260).

When these ASIs are used with STDFA for Partial Store, a *mem_address_not_aligned* exception is generated if the operand address is not 8-byte aligned and an *illegal_instruction* exception is generated if i = 1 in the instruction and the ASI register contains one of the Partial Store ASIs.

If one of these ASIs is used with a Store Alternate instruction other than STDFA, a Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is generated and *mem_address_not_aligned*, *LDDF_mem_address_not_aligned*, and *illegal_instruction* (for i = 1) are not generated.

ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–CD_{16} are only defined for use in Partial Store operations (see page 260). None of them should be used with LDDFA; however, if any of those ASIs *is* used with LDDFA, the resulting behavior is specified in the LDDFA instruction description on page 185.

10.4.12 Short Floating-Point Load and Store ASIs

ASIs $D0_{16}$–$D3_{16}$ and $D8_{16}$–DB_{16} exist for use with the LDDFA and STDFA instructions as Short Floating-point Load and Store operations (see *Load Floating-Point Register on page 181* and *Store Floating-Point* on page 253).

When ASI $D2_{16}$, $D3_{16}$, DA_{16}, or DB_{16} is used with LDDFA (STDFA) for a 16-bit Short Floating-point Load (Store), a *mem_address_not_aligned* exception is generated if the operand address is not halfword-aligned.

If any of these ASIs are used with any other Load Alternate, Store Alternate, Atomic Load-Store Alternate, or PREFETCHA instruction, a *DAE_invalid_asi* exception is always generated and *mem_address_not_aligned* is not generated.

10.5 ASI-Accessible Registers

In this section the Data Watchpoint registers, and scratchpad registers are described.

A list of UltraSPARC Architecture 2007 ASIs is shown in TABLE 10-1 on page 323.

10.5.1 Privileged Scratchpad Registers (ASI_SCRATCHPAD) D1

An UltraSPARC Architecture virtual processor includes eight Scratchpad registers (64 bits each, read/write accessible) (impl.dep. #302-U4-Cs10). The use of the Scratchpad registers is completely defined by software.

For conventional uses of Scratchpad registers, see "Scratchpad Register Usage" in *Software Considerations*, contained in the separate volume *UltraSPARC Architecture Application Notes*.

The Scratchpad registers are intended to be used by performance-critical trap handler code.

The addresses of the privileged scratchpad registers are defined in TABLE 10-7.

TABLE 10-7 Scratchpad Registers

Assembly Language ASI Name	ASI #	Virtual Address	Privileged Scratchpad Register #
ASI_SCRATCHPAD	20_{16}	00_{16}	0
		08_{16}	1
		10_{16}	2
		18_{16}	3
		20_{16}	4
		28_{16}	5
		30_{16}	6
		38_{16}	7

IMPL. DEP. #404-S10: The degree to which Scratchpad registers 4–7 are accessible to privileged software is implementation dependent. Each may be
(1) fully accessible,
(2) accessible, with access much slower than to scratchpad registers 0–3, or
(3) inaccessible (cause a *DAE_invalid_asi* exception).

V9 Compatibility Note | Privileged scratchpad registers are an UltraSPARC Architecture extension to SPARC V9.

10.5.2 ASI Changes in the UltraSPARC Architecture

The following Compatibility Note summarize the UltraSPARC ASI changes in UltraSPARC Architecture.

Compatibility Note | The names of several ASIs used in earlier UltraSPARC implementations have changed in UltraSPARC Architecture. Their functions have not changed; just their names have changed.

ASI#	Previous UltraSPARC	UltraSPARC Architecture
14_{16}	ASI_PHYS_USE_EC	ASI_REAL
15_{16}	ASI_PHYS_BYPASS_EC_WITH_EBIT	ASI_REAL_IO
$1C_{16}$	ASI_PHYS_USE_EC_LITTLE (ASI_PHYS_USE_EC_L)	ASI_REAL_LITTLE
$1D_{16}$	ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE (ASI_PHY_BYPASS_EC_WITH_EBIT_L)	ASI_REAL_IO_LITTLE

Compatibility Note The names *and* ASI assignments (but not functions) changed between earlier UltraSPARC implementations and UltraSPARC Architecture, for the following ASIs:

Previous UltraSPARC		UltraSPARC Architecture	
ASI#	**Name**	**ASI#**	**Name**
24_{16}	ASI_NUCLEUS_QUAD_LDD[D]	27_{16}	ASI_TWINX_NUCLEUS (ASI_TWINX_N)
$2C_{16}$	ASI_NUCLEUS_QUAD_LDD_ LITTLE[D] (ASI_NUCLEUS_QUAD_LDD_L[D])	$2F_{16}$	ASI_TWINX_NUCLEUS_ LITTLE (ASI_TWINX_NL)

DDD

CHAPTER **11**

Performance Instrumentation

This chapter describes the architecture for performance monitoring hardware on UltraSPARC Architecture processors. The architecture is based on the design of performance instrumentation counters in previous UltraSPARC Architecture processors, with an extension for the selective sampling of instructions.

11.1 High-Level Requirements

11.1.1 Usage Scenarios

The performance monitoring hardware on UltraSPARC Architecture processors addresses the needs of various kinds of users. There are four scenarios envisioned:

- *System-wide performance monitoring.* In this scenario, someone skilled in system performance analysis (e.g, a Systems Engineer) is using analysis tools to evaluate the performance of the entire system. An example of such a tool is cpustat. The objective is to obtain performance data relating to the configuration and behavior of the system, e.g., the utilization of the memory system.
- *Self-monitoring of performance by the operating system.* In this scenario the OS is gathering performance data in order to tune the operation of the system. Some examples might be:
  - (a) determining whether the processors in the system should be running in single- or multi-stranded mode.
  - (b) determining the affinity of a process to a processor by examining that process's memory behavior.
- *Performance analysis of an application by a developer*. In this scenario a developer is trying to optimize the performance of a specific application, by altering the source code of the application or the compilation options. The developer needs to know the performance characteristics of the components of the application at a coarse grain, and where these are problematic, to be able to determine fine-grained performance information. Using this information, the developer will alter the source or compilation parameters, re-run the application, and observe the new performance characteristics. This process is repeated until performance is acceptable, or no further improvements can be found.

  An example might be that a loop nest is measured to be not performing well. Upon closer inspection, the developer determines that the loop has poor cache behavior, and upon more detailed inspection finds a specific operation which repeatedly misses the cache. Reorganizing the code and/or data may improve the cache behavior.
- *Monitoring of an application's performance, e.g., by a Java Virtual Machine*. In this scenario the application is not executing directly on the hardware, but its execution is being mediated by a piece of system software, which for the purposes of this document is called a Virtual Machine. This may

be a Java VM, or a binary translation system running software compiled for another architecture, or for an earlier version of the UltraSPARC Architecture. One goal of the VM is to optimize the behavior of the application by monitoring its performance and dynamically reorganizing the execution of the application (e.g., by selective recompilation of the application).

This scenario differs from the previous one principally in the time allowed to gather performance data. Because the data are being gathered during the execution of the program, the measurements must not adversely affect the performance of the application by more than, say, a few percent, and must yield insight into the performance of the application in a relatively short time (otherwise, optimization opportunities are deferred for too long). This implies an observation mechanism which is of very low overhead, so that many observations can be made in a short time.

In contrast, a developer optimizing an application has the luxury of running or re-running the application for a considerable period of time (minutes or even hours) to gather data. However, the developer will also expect a level of precision and detail in the data which would overwhelm a virtual machine, so the accuracy of the data required by a virtual machine need not be as high as that supplied to the developer.

Scenarios 1 and 2 are adequately dealt with by a suitable set of performance counters capable of counting a variety of performance-related events. Counters are ideal for these situations because they provide low-overhead statistics without any intrusion into the behavior of the system or disruption to the code being monitored. However, counters may not adequately address the latter two scenarios, in which detailed and timely information is required at the level of individual instructions. Therefore, UltraSPARC Architecture processors may also implement an instruction sampling mechanism.

11.1.2 Metrics

There are two classes of data reported by a performance instrumentation mechanism:

- *Architectural performance metrics.* These are metrics related to the observable execution of code at the architectural level (UltraSPARC Architecture). Examples include:
  - The number of instructions executed
  - The number of floating point instructions executed
  - The number of conditional branch instructions executed
- *Implementation performance metrics.* These describe the behavior of the microprocessor in terms of its implementation, and would not necessarily apply to another implementation of the architecture.

In optimizing the performance of an application or system, attention will first be paid to the first class of metrics, and so these are more important. Only in performance-critical cases would the second class receive attention, since using these metrics requires a fairly extensive understanding of the specific implementation of the UltraSPARC Architecture.

11.1.3 Accuracy Requirements

Accuracy requirements for performance instrumentation vary depending on the scenario. The requirements are complicated by the possibly speculative nature of UltraSPARC Architecture processor implementations. For example, an implementation may include in its cache miss statistics the misses induced by speculative executions which were subsequently flushed, or provide two separate statistics, one for the misses induced by flushed instructions and one for misses induced by retired instructions. Although the latter would be desirable, the additional implementation complexity of associating events with specific instructions is significant, and so all events may be counted without distinction. The instruction sampling mechanism may distinguish between instructions that retired and those that were flushed, in which case sampling can be used to obtain statistical estimates of the frequencies of operations induced by mis-speculation.

For critical performance measurements, architectural event counts must be accurate to a high degree (1 part in 10^5). Which counters are considered performance-critical (and therefore accurate to 1 part in 10^5) are specified in implementation-specific documentation.

Implementation event counts must be accurate to 1 part in 10^3, not including the speculative effects mentioned above. An upper bound on counter skew must be stated in implementation-specific documentation.

Programming Note | Increasing the time between counter reads will mitigate the inaccurcies that could be introduced by counter skew (due to speculative effects).

11.2 Performance Counters and Controls

The performance instrumentation hardware provides performance instrumentation counters (PICs). The number and size of performance counters is implementation dependent, but each performance counter register contains at least one 32-bit counter. It is implementation dependent whether the performance counter registers are accessed as ASRs or are accessed through ASIs.

There are one or more performance counter control registers (PCRs) associated with the counter registers. It is implementation dependent whether the PCRs are accessed as ASRs or are accessed through ASIs.

Each counter in a counter register can count one kind of event at a time. The number of the kinds of events that can be counted is implementation dependent. For each performance counter register, the corresponding control register is used to select the event type being counted. A counter is incremented whenever an event of the matching type occurs. A counter may be incremented by an event caused by an instruction which is subsequently flushed (for example, due to mis-speculation). Counting of events may be controlled based on privilege mode or on the strand in which they occur. Masking may be provided to allow counting of subgroups of events (for example, various occurrences of different opcode groups).

A field that indicates when a counter has overflowed must be present in either each performance instrumentation counter or in a separate performance counter control register.

Performance counters are usually provided on a per-strand basis.

11.2.1 Counter Overflow

Overflow of a counter must cause a disrupting trap to be generated, when enabled by a Trap Overflow Enable bit (in an implementation-specific location). There must be a separate toe bit for each performance counter, so that overflow traps can be enabled on a per-counter basis. Overflow of a counter is recorded in the overflow-indication field of either a performance instrumentation counter or a separate performance counter control register.

Programming Note | Counter overflow traps can also be used for sampling, by setting the initial counter value so that an interrupt occurs *n* counts later.

Counter overflow traps are provided so that large counts can be maintained in software, beyond the range directly supported in hardware. The counters continue to count after an overflow, and software can utilize the overflow traps to maintain additional high-order bits.

CHAPTER **12**

Traps

A *trap* is a vectored transfer of control to software running in a privilege mode (see page 342) with (typically) greater privileges. A trap in nonprivileged mode can be delivered to privileged mode or hyperprivileged mode. A trap that occurs while executing in privileged mode can be delivered to privileged mode or hyperprivileged mode.

The actual transfer of control occurs through a trap table that contains the first eight instructions (32 instructions for *clean_window*, window spill, and window fill, traps) of each trap handler. The virtual base address of the trap table for traps to be delivered in privileged mode is specified in the Trap Base Address (TBA) register. The displacement within the table is determined by the trap type and the current trap level (TL). One-half of each table is reserved for hardware traps; the other half is reserved for software traps generated by Tcc instructions.

A trap behaves like an unexpected procedure call. It causes the hardware to do the following:

1. Save certain virtual processor state (such as program counters, CWP, ASI, CCR, PSTATE, and the trap type) on a hardware register stack.
2. Enter privileged execution mode with a predefined PSTATE.
3. Begin executing trap handler code in the trap vector.

When the trap handler has finished, it uses either a DONE or RETRY instruction to return.

A trap may be caused by a Tcc instruction, an instruction-induced exception, a reset, an asynchronous error, or an interrupt request not directly related to a particular instruction. The virtual processor must appear to behave as though, before executing each instruction, it determines if there are any pending exceptions or interrupt requests. If there are pending exceptions or interrupt requests, the virtual processor selects the highest-priority exception or interrupt request and causes a trap.

Thus, an *exception* is a condition that makes it impossible for the virtual processor to continue executing the current instruction stream without software intervention. A *trap* is the action taken by the virtual processor when it changes the instruction flow in response to the presence of an exception, interrupt, reset, or Tcc instruction.

V9 Compatibility Note | Exceptions referred to as "catastrophic error exceptions" in the SPARC V9 specification do not exist in the UltraSPARC Architecture; they are handled using normal error-reporting exceptions. (impl. dep. #31-V8-Cs10)

An *interrupt* is a request for service presented to a virtual processor by an external device.

Traps are described in these sections:

■ **Register Window Traps** on page 362.

12.1 Virtual Processor Privilege Modes

An UltraSPARC Architecture virtual processor is always operating in a discrete privilege mode. The privilege modes are listed below in order of increasing privilege:

- Nonprivileged mode (also known as "user mode")
- Privileged mode, in which supervisor (operating system) software primarily operates
- Hyperprivileged mode (not described in this document)

The virtual processor's operating mode is determined by the state of two mode bits, as shown in TABLE 12-1.

TABLE 12-1 Virtual Processor Privilege Modes

PSTATE.priv	Virtual Processor Privilege Mode
0	Nonprivileged
1	Privileged

A trap is delivered to the virtual processor in either privileged mode or hyperprivileged mode; in which mode the trap is delivered depends on:

- Its trap type
- The trap level (TL) at the time the trap is taken
- The privilege mode at the time the trap is taken

Traps detected in nonprivileged and privileged mode can be delivered to the virtual processor in privileged mode or hyperprivileged mode.

TABLE 12-4 on page 351 indicates in which mode each trap is processed, based on the privilege mode at which it was detected.

A trap delivered to privileged mode uses the privileged-mode trap vector, based upon the TBA register. See *Trap-Table Entry Address to Privileged Mode* on page 348 for details.

The maximum trap level at which privileged software may execute is *MAXPTL* (which, on an UltraSPARC Architecture 2007 virtual processor, is 2)..

Notes | Execution in nonprivileged mode with TL > 0 is an invalid condition that privileged software should never allow to occur.

FIGURE 12-1 shows how a virtual processor transitions between privilege modes, excluding transitions that can occur due to direct software writes to PSTATE.priv. In this figure, PT indicates a "trap destined for privileged mode" and HT indicates a "trap destined for hyperprivileged mode".

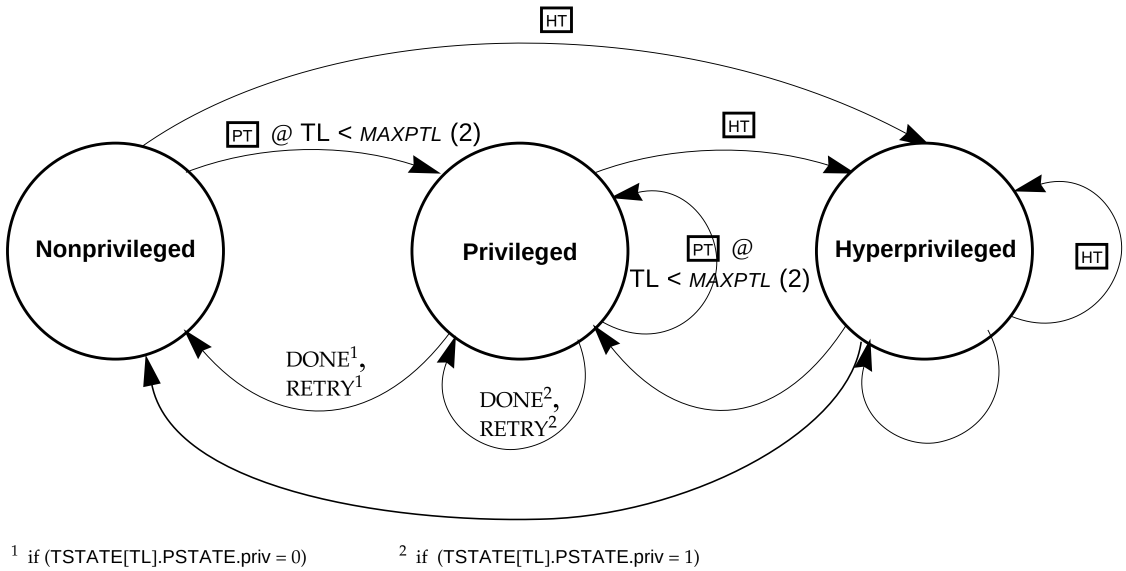


FIGURE 12-1 Virtual Processor Privilege Mode Transition Diagram

12.2 Virtual Processor States and Traps

The value of TL affects the generated trap vector address. TL also determines where (that is, into which element of the TSTATE array) the states are saved.

12.2.0.1 Usage of Trap Levels

If *MAXPTL* = 2 in an UltraSPARC Architecture implementation, the trap levels might be used as shown in TABLE 12-2.

TABLE 12-2 Typical Usage for Trap Levels

TL	Corresponding Execution Mode	Usage
0	Nonprivileged	Normal execution
1	Privileged	System calls; interrupt handlers; instruction emulation
2	Privileged	Window spill/fill handler

12.3 Trap Categories

An exception, error, or interrupt request can cause any of the following trap types:

- Precise trap
- Deferred trap
- Disrupting trap

- Reset trap

12.3.1 Precise Traps

A *precise trap* is induced by a particular instruction and occurs before any program-visible state has been changed by the trap-inducing instructions. When a precise trap occurs, several conditions must be true:

- The PC saved in TPC[TL] points to the instruction that induced the trap and the NPC saved in TNPC[TL] points to the instruction that was to be executed next.
- All instructions issued before the one that induced the trap have completed execution.
- Any instructions issued after the one that induced the trap remain unexecuted.

Among the actions that trap handler software might take when processing a precise trap are:

- Return to the instruction that caused the trap and reexecute it by executing a RETRY instruction (PC ← old PC, NPC ← old NPC).
- Emulate the instruction that caused the trap and return to the succeeding instruction by executing a DONE instruction (PC ← old NPC, NPC ← old NPC + 4).
- Terminate the program or process associated with the trap.

12.3.2 Deferred Traps

A *deferred trap* is also induced by a particular instruction, but unlike a precise trap, a deferred trap may occur after program-visible state has been changed. Such state may have been changed by the execution of either the trap-inducing instruction itself or by one or more other instructions.

There are two classes of deferred traps:

- *Termination deferred traps* — The instruction (usually a store) that caused the trap has passed the retirement point of execution (the TPC has been updated to point to an instruction beyond the one that caused the trap). The trap condition is an error that prevents the instruction from completing and its results becoming globally visible. A termination deferred trap has high trap priority, second only to the priority of resets.

  Programming Note | Not enough state is saved for execution of the instruction stream to resume with the instruction that caused the trap. Therefore, the trap handler must terminate the process containing the instruction that caused the trap.

- *Restartable deferred traps* — The program-visible state has been changed by the trap-inducing instruction or by one or more other instructions after the trap-inducing instruction.

  SPARC V9 Compatibility Note | A *restartable* deferred trap is the "deferred trap" defined in the SPARC V9 specification.

The fundamental characteristic of a *restartable* deferred trap is that the state of the virtual processor on which the trap occurred may not be consistent with any precise point in the instruction sequence being executed on that virtual processor. When a restartable deferred trap occurs, TPC[TL] and TNPC[TL] contain a PC value and an NPC value, respectively, corresponding to a point in the instruction sequence being executed on the virtual processor. This PC may correspond to the trap-inducing instruction or it may correspond to an instruction following the trap-inducing instruction. With a restartable deferred trap, program-visible updates may be missing from instructions prior to the instruction to which TPC[TL] refers. The missing updates are limited to instructions in the range from (and including) the actual trap-inducing instruction up to (but not including) the instruction to which TPC[TL] refers. By definition, the instruction to which TPC[TL] refers has not yet executed, therefore it cannot have any updates, missing or otherwise.

With a restartable deferred trap there must exist sufficient information to report the error that caused the deferred trap. If system software can recover from the error that caused the deferred trap, then there must be sufficient information to generate a consistent state within the processor so that execution can resume. Included in that information must be an indication of the mode (nonprivileged, privileged, or hyperprivileged) in which the trap-inducing instruction was issued.

How the information necessary for repairing the state to make it consistent state is maintained and how the state is repaired to a consistent state are implementation dependent. It is also implementation dependent whether execution resumes at the point of the trap-inducing instruction or at an arbitrary point between the trap-inducing instruction and the instruction pointed to by the TPC[TL], inclusively.

Associated with a particular restartable deferred trap implementation, the following must exist:

- An instruction that causes a potentially outstanding restartable deferred trap exception to be taken as a trap
- Instructions with sufficient privilege to access the state information needed by software to emulate the restartable deferred trap-inducing instruction and to resume execution of the trapped instruction stream.

Programming Note | Resuming execution may require the emulation of instructions that had not completed execution at the time of the restartable deferred trap, that is, those instructions in the deferred-trap queue.

Software should resume execution with the instruction starting at the instruction to which TPC[TL] refers. Hardware should provide enough information for software to recreate virtual processor state and update it to the point just before execution of the instruction to which TPC[TL] refers. After software has updated virtual processor state up to that point, it can then resume execution by issuing a RETRY instruction.

IMPL. DEP. #32-V8-Ms10: Whether any restartable deferred traps (and, possibly, associated deferred-trap queues) are present is implementation dependent.

Among the actions software can take after a restartable deferred trap are these:

- Emulate the instruction that caused the exception, emulate or cause to execute any other execution-deferred instructions that were in an associated restartable deferred trap state queue, and use RETRY to return control to the instruction at which the deferred trap was invoked.
- Terminate the program or process associated with the restartable deferred trap.

A deferred trap (of either of the two classes) is always delivered to the virtual processor in hyperprivileged mode.

12.3.3 Disrupting Traps

12.3.3.1 Disrupting versus Precise and Deferred Traps

A *disrupting trap* is caused by a condition (for example, an interrupt) rather than directly by a particular instruction. This distinguishes it from *precise* and *deferred* traps.

When a disrupting trap has been serviced, trap handler software normally arranges for program execution to resume where it left off. This distinguishes disrupting traps from *reset* traps, since a reset trap vectors to a unique reset address and execution of the program that was running when the reset occurred is generally not expected to resume.

When a disrupting trap occurs, the following conditions are true:

1. The PC saved in TPC[TL] points to an instruction in the disrupted program stream and the NPC value saved in TNPC[TL] points to the instruction that was to be executed after that one.

2. All instructions issued before the instruction indicated by TPC[TL] have retired.

3. The instruction to which TPC[TL] refers and any instruction(s) that were issued after it remain unexecuted.

A disrupting trap may be due to an interrupt request directly related to a previously-executed instruction; for example, when a previous instruction sets a bit in the SOFTINT register.

12.3.3.2 Causes of Disrupting Traps

A disrupting trap may occur due to either an interrupt request or an error not directly related to instruction processing. The source of an interrupt request may be either internal or external. An interrupt request can be induced by the assertion of a signal not directly related to any particular virtual processor or memory state, for example, the assertion of an "I/O done" signal.

A condition that causes a disrupting trap persists until the condition is cleared.

12.3.3.3 Conditioning of Disrupting Traps

How disrupting traps are conditioned is affected by:

- The privilege mode in effect when the trap is outstanding, just before the trap is actually taken (regardless of the privilege mode that was in effect when the exception was detected).
- The privilege mode for which delivery of the trap is destined

Outstanding in Nonprivileged or Privileged mode, destined for delivery in Privileged mode. An outstanding disrupting trap condition in either nonprivileged mode or privileged mode and destined for delivery to privileged mode is held pending while the Interrupt Enable (ie) field of PSTATE is zero (PSTATE.ie = 0). *interrupt_level_n* interrupts are further conditioned by the Processor Interrupt Level (PIL) register. An interrupt is held pending while either PSTATE.ie = 0 or the condition's interrupt level is less than or equal to the level specified in PIL. When delivery of this disrupting trap is enabled by PSTATE.ie = 1, it is delivered to the virtual processor in privileged mode if TL < *MAXPTL* (2, in UltraSPARC Architecture 2007 implementations).

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

Type of Disrupting Trap Condition	Current Virtual Processor Privilege Mode	Disposition of Disrupting Traps, based on privilege mode in which the trap is destined to be delivered	
		Privileged	Hyperprivileged
Interrupt_level_n	Nonprivileged or Privileged	Held pending while PSTATE.ie = 0 or interrupt level ≤ PIL	—
All other disrupting traps	Nonprivileged or Privileged	Held pending while PSTATE.ie = 0	Delivered immediately

12.3.3.4 Trap Handler Actions for Disrupting Traps

Among the actions that trap-handler software might take to process a disrupting trap are:

- Use RETRY to return to the instruction at which the trap was invoked (PC ← old PC, NPC ← old NPC).
- Terminate the program or process associated with the trap.

12.3.4 Uses of the Trap Categories

The SPARC V9 *trap model* stipulates the following:

1. Reset traps occur asynchronously to program execution.
2. When recovery from an exception can affect the interpretation of subsequent instructions, such exceptions shall be precise. See TABLE 12-4, TABLE 12-5, and *Exception and Interrupt Descriptions* on page 358 for identification of which traps are precise.
3. In an UltraSPARC Architecture implementation, all exceptions that occur as the result of program execution are precise (impl. dep. #33-V8-Cs10).
4. An error detected after the initial access of a multiple-access load instruction (for example, LDTX or LDBLOCKF) should be precise. Thus, a trap due to the second memory access can occur. However, the processor state should not have been modified by the first access.
5. Exceptions caused by external events unrelated to the instruction stream, such as interrupts, are disrupting.

A deferred trap may occur one or more instructions after the trap-inducing instruction is dispatched.

12.4 Trap Control

Several registers control how any given exception is processed, for example:

- The interrupt enable (ie) field in PSTATE and the Processor Interrupt Level (PIL) register control interrupt processing. See *Disrupting Traps* on page 345 for details.
- The enable floating-point unit (fef) field in FPRS, the floating-point unit enable (pef) field in PSTATE, and the trap enable mask (tem) in the FSR control floating-point traps.
- The TL register, which contains the current level of trap nesting, affects whether the trap is processed in privileged mode or hyperprivileged mode.
- PSTATE.tle determines whether implicit data accesses in the trap handler routine will be performed using big-endian or little-endian byte order.

Between the execution of instructions, the virtual processor prioritizes the outstanding exceptions, errors, and interrupt requests. At any given time, only the highest-priority exception, error, or interrupt request is taken as a trap. When there are multiple interrupts outstanding, the interrupt with the highest interrupt level is selected. When there are multiple outstanding exceptions, errors, and/or interrupt requests, a trap occurs based on the exception, error, or interrupt with the highest priority (numerically lowest priority number in TABLE 12-5). See *Trap Priorities* on page 356.

12.4.1 PIL Control

When an interrupt request occurs, the virtual processor compares its interrupt request level against the value in the Processor Interrupt Level (PIL) register. If the interrupt request level is greater than PIL and no higher-priority exception is outstanding, then the virtual processor takes a trap using the appropriate *interrupt_level_n* trap vector.

12.4.2 FSR.tem Control

The occurrence of floating-point traps of type IEEE_754_exception can be controlled with the user-accessible trap enable mask (tem) field of the FSR. If a particular bit of FSR.tem is 1, the associated IEEE_754_exception can cause an *fp_exception_ieee_754* trap.

If a particular bit of FSR.tem is 0, the associated IEEE_754_exception does not cause an *fp_exception_ieee_754* trap. Instead, the occurrence of the exception is recorded in the FSR's accrued exception field (aexc).

If an IEEE_754_exception results in an *fp_exception_ieee_754* trap, then the destination F register, FSR.fcc*n*, and FSR.aexc fields remain unchanged. However, if an IEEE_754_exception does not result in a trap, then the F register, FSR.fcc*n*, and FSR.aexc fields are updated to their new values.

12.5 Trap-Table Entry Addresses

Traps are delivered to the virtual processor in either privileged mode or hyperprivileged mode, depending on the trap type, the value of TL at the time the trap is taken, and the privilege mode at the time the exception was detected. See TABLE 12-4 on page 351 and TABLE 12-5 on page 354 for details.

Unique trap table base addresses are provided for traps being delivered in privileged mode and in hyperprivileged mode.

12.5.1 Trap-Table Entry Address to Privileged Mode

Privileged software initializes bits 63:15 of the Trap Base Address (TBA) register (its most significant 49 bits) with bits 63:15 of the desired 64-bit privileged trap-table base address.

At the time a trap to privileged mode is taken:

- Bits 63:15 of the trap vector address are taken from TBA{63:15}.
- Bit 14 of the trap vector address (the "TL>0" field) is set based on the value of TL just before the trap is taken; that is, if TL = 0 then bit 14 is set to 0 and if TL > 0 then bit 14 is set to 1.
- Bits 13:5 of the trap vector address contain a copy of the contents of the TT register (TT[TL]).
- Bits 4:0 of the trap vector address are always 0; hence, each trap table entry is at least 2^5 or 32 bytes long. Each entry in the trap table may contain the first eight instructions of the corresponding trap handler.

FIGURE 12-2 illustrates the trap vector address for a trap delivered to privileged mode. In FIGURE 12-2, the "TL>0" bit is 0 if TL = 0 when the trap was taken, and 1 if TL > 0 when the trap was taken. This implies, as detailed in the following section, that there are two trap tables for traps to privileged mode: one for traps from TL = 0 and one for traps from TL > 0.

from TBA{63:15} (TBA.tba_high49)	TL>0	TT[TL]	0 0000
63 … 15	14	13 … 5	4 … 0

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.

Value of TL (before trap)	Software Trap Type	Hardware Trap Type (TT[TL])	Trap Table Offset (from TBA)	Contents of Trap Table
TL = 0	—	000_{16}–$07F_{16}$	0_{16}– $FE0_{16}$	Hardware traps
	—	080_{16}–$0FF_{16}$	1000_{16}–$1FE0_{16}$	*Spill / fill* traps
	0_{16}– $7F_{16}$	100_{16}–$17F_{16}$	2000_{16}–$2FE0_{16}$	Software traps to Privileged level
	—	180_{16}–$1FF_{16}$	3000_{16}–$3FE0_{16}$	*unassigned*
TL = 1 (TL = *MAXPTL*–1)	—	000_{16}–$07F_{16}$	4000_{16}–$4FE0_{16}$	Hardware traps
	—	080_{16}–$0FF_{16}$	5000_{16}–$5FE0_{16}$	*Spill / fill* traps
	0_{16}– $7F_{16}$	100_{16}–$17F_{16}$	6000_{16}–$6FE0_{16}$	Software traps to Privileged level
	—	180_{16}–$1FF_{16}$	7000_{16}–$7FE0_{16}$	*unassigned*

FIGURE 12-3 Privileged-mode Trap Table Layout

The trap table for TL = 0 comprises 512 thirty-two-byte entries; the trap table for TL > 0 comprises 512 more thirty-two-byte entries. Therefore, the total size of a full privileged trap table is $2 \times 512 \times 32$ bytes (32 Kbytes). However, if privileged software does not use software traps (Tcc instructions) at TL > 0, the table can be made 24 Kbytes long.

12.5.3 Trap Type (TT)

When a normal trap occurs, a value that uniquely identifies the type of the trap is written into the current 9-bit TT register (TT[TL]) by hardware. Control is then transferred into the trap table to an address formed by the trap's destination privilege mode:

- The TBA register, (TL > 0), and TT[TL] (see *Trap-Table Entry Address to Privileged Mode* on page 348)

TT values 000_{16}–$0FF_{16}$ are reserved for hardware traps. TT values 100_{16}–$17F_{16}$ are reserved for software traps (caused by execution of a Tcc instruction) to privileged-mode trap handlers.

IMPL. DEP. #35-V8-Cs20: TT values 060_{16} to $07F_{16}$ were reserved for *implementation_dependent_exception_n* exceptions in the SPARC V9 specification, but are now all defined as standard UltraSPARC Architecture exceptions. See TABLE 12-4 for details.

The assignment of TT values to traps is shown in TABLE 12-4; TABLE 12-5 provides the same list, but sorted in order of trap priority. The key to both tables follows:

Symbol	Meaning
●	This trap type is associated with a feature that is architecturally required in an implementation of UltraSPARC Architecture 2007. Hardware must detect this exception or interrupt, trap on it (if not masked), and set the specified trap type value in the TT register.
❍	This trap type is associated with a feature that is architecturally defined in UltraSPARC Architecture 2007, but its implementation is optional.
P	Trap is taken via the Privileged trap table, in Privileged mode (PSTATE.priv = 1)
H	Trap is taken in Hyperprivileged mode
-x-	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.
—	This trap is reserved for future use.
(ie)	When the outstanding disrupting trap condition occurs in this privilege mode, it may be conditioned (masked out) by PSTATE.ie = 0 (but remains pending).
(nm)	Never Masked — when the condition occurs in this running mode, it is never masked out and the trap is always taken.
(pend)	Held Pending — the condition can *occur* in this running mode, but can't be *serviced* in this mode. Therefore, it is held pending until the mode changes to one in which the exception *can* be serviced.

TABLE 12-4 Exception and Interrupt Requests, by TT Value *(1 of 3)*

UA-2007 ●=Req'd. ❍=Opt'l	Exception or Interrupt Request	TT (Trap Type)	Trap Category	Priority (0 = High-est)	Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode	
					NP	Priv
—	*Reserved*	000_{16}	—	—	—	—
●	*(used at higher privilege levels)*	001_{16}–005_{16}	—	—	—	—
—	*Reserved*	005_{16}	—	—	—	—
—	*implementation-dependent*	006_{16}	—	—	—	—
●	*IAE_privilege_violation*	008_{16}	precise	3.1	H	-x-
●	*(used at higher privilege levels)*	009_{16}	—	—	—	—
●	*(used at higher privilege levels)*	$00A_{16}$	—	—	—	—
●	*IAE_unauth_access*	$00B_{16}$	precise	3.2	H	H
●	*IAE_nfo_page*	$00C_{16}$	precise	3.3	H	H
—	*Reserved*	$00F_{16}$	—	—	—	—
●	*illegal_instruction*	010_{16}	precise	6.2	H	H
●	*privileged_opcode*	011_{16}	precise	7	P (nm)	-x-
❍	*unimplemented_LDTW*	012_{16}	precise	6.3	H	H
❍	*unimplemented_STTW*	013_{16}	precise	6.3	H	H
●	*DAE_invalid_asi*	014_{16}	precise	12.01	H	H
●	*DAE_privilege_violation*	015_{16}	precise	12.04	H	H
●	*DAE_nc_page*	016_{16}	precise	12.05	H	H
●	*DAE_nfo_page*	017_{16}	precise	12.06	H	H
—	*Reserved*	018_{16}–$01F_{16}$	—	—	—	—
●	*fp_disabled*	020_{16}	precise	8	P (nm)	P (nm)
❍	*fp_exception_ieee_754*	021_{16}	precise	11.1	P (nm)	P (nm)
❍	*fp_exception_other*	022_{16}	precise	11.1	P (nm)	P (nm)
●	*tag_overflow*[D]	023_{16}	precise	14	P (nm)	P (nm)
●	*clean_window*	024_{16}‡–027_{16}	precise	10.1	P (nm)	P (nm)
●	*division_by_zero*	028_{16}	precise	15	P (nm)	P (nm)
—	*Reserved*	$02C_{16}$	—	—	—	—
●	*DAE_side_effect_page*	030_{16}	precise	12.06	H	H
—	*Reserved*	032_{16}	—	—	—	—

TABLE 12-4 Exception and Interrupt Requests, by TT Value *(2 of 3)*

UA-2007 ●=Req'd. ❍=Opt'l	Exception or Interrupt Request	TT (Trap Type)	Trap Category	Priority (0 = High-est)	Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode: NP	Priv
●	*mem_address_not_aligned*	034_{16}	precise	10.2	H	H
●	*LDDF_mem_address_not_aligned*	035_{16}	precise	10.1	H	H
●	*STDF_mem_address_not_aligned*	036_{16}	precise	10.1	H	H
●	*privileged_action*	037_{16}	precise	11.1	H	H
❍	*LDQF_mem_address_not_aligned*	038_{16}	precise	10.1	H	H
❍	*STQF_mem_address_not_aligned*	039_{16}	precise	10.1	H	H
—	*Reserved*	$03A_{16}$	—	—	—	—
—	*Reserved*	$03B_{16}$	—	—	—	—
—	*Reserved*	$03C_{16}$–$03D_{16}$	—	—	—	—
❍	*(used at higher privilege levels)*	0401_{16}	—	—	—	—
●	*interrupt_level_n* (n = 1–15)	041_{16} – $04F_{16}$	disrupting	32-n (31 to 17)	P (ie)	P (ie)
—	*Reserved*	050_{16}–$05D_{16}$	—	—	—	—
●	*(used at higher privilege levels)*	$05F_{16}$–061_{16}	—	—	—	—
❍	*(used at higher privilege levels)*	060_{16}	—	—	—	—
❍	*VA_watchpoint*	062_{16}	precise	11.2	P (nm)	P (nm)
●	*(used at higher privilege levels)*	063_{16}–$06C_{16}$	—	—	—	—
❍	*implementation_dependent_exception_n* (impl. dep. #35-V8-Cs20)	070_{16}	—	∇	—	—
●	*(used at higher privilege levels)*	071_{16}–072_{16}	—	—	—	—
❍	*implementation_dependent_exception_n* (impl. dep. #35-V8-Cs20)	073_{16}	—	∇	—	—
●	*control_transfer_instruction*	074_{16}	precise	11.1	P	P
❍	*instruction_VA_watchpoint*	075_{16}	precise	2.05	P (nm)	P (nm)
❑	*implementation_dependent_exception_n* (impl. dep. #35-V8-Cs20)	077_{16} – 078_{16}	—	∇	—	—
❑	*implementation_dependent_exception_n* (impl. dep. #35-V8-Cs20)	079_{16}–$07B_{16}$	—	∇	—	—
●	*cpu_mondo*	$07C_{16}$	disrupting	16.08	P (ie)	P (ie)
●	*dev_mondo*	$07D_{16}$	disrupting	16.11	P (ie)	P (ie)

TABLE 12-4 Exception and Interrupt Requests, by TT Value *(3 of 3)*

UA-2007 ●=Req'd. ❍=Opt'l	Exception or Interrupt Request	TT (Trap Type)	Trap Category	Priority (0 = High-est)	Mode in which Trap is Delivered (and Conditioning Applied), based on Current Privilege Mode	
					NP	Priv
●	*resumable_error*	$07E_{16}$	disrupting	33.3	P (ie)	P (ie)
—	*nonresumable_error*	$07F_{16}$	—	—	—	—
●	*spill_n_normal* (n = 0–7)	080_{16}‡– $09F_{16}$	precise	9	P (nm)	P (nm)
●	*spill_n_other* (n = 0–7)	$0A0_{16}$‡– $0BF_{16}$	precise	9	P (nm)	P (nm)
●	*fill_n_normal* (n = 0–7)	$0C0_{16}$‡– $0DF_{16}$	precise	9	P (nm)	P (nm)
●	*fill_n_other* (n = 0–7)	$0E0_{16}$‡– $0FF_{16}$	precise	9	P (nm)	P (nm)
●	*trap_instruction*	100_{16}– $17F_{16}$	precise	16.02	P (nm)	P (nm)
●	*htrap_instruction*	180_{16}– $1FF_{16}$	precise	16.02	-x-	

* Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36-V8 on page 356), including relative priorities within a given priority level.

‡ The trap vector entry (32 bytes) for this trap type plus the next three trap types (total of 128 bytes) are permanently reserved for this exception.

D This exception is deprecated, because the only instructions that can generate it have been deprecated.

TABLE 12-5 Exception and Interrupt Requests, by Priority *(1 of 2)*

UA-2007 ●=Req'd. ❍=Opt'l ❑.=Impl-Specific	Exception or Interrupt Request	TT (Trap Type)	Trap Category	Priority (0 = Highest)	Mode in which Trap is Delivered and (and Conditioning Applied), based on Current Privilege Mode NP	Priv
❍	*instruction_VA_watchpoint*	075_{16}	precise	2.05	P (nm)	P (nm)
●	*IAE_privilege_violation*	008_{16}	precise	3.1	H	-x-
●	*IAE_unauth_access*	$00B_{16}$	precise	3.2	H	H
●	*IAE_nfo_page*	$00C_{16}$	precise	3.3	H	H
●	*illegal_instruction*	010_{16}	precise	6.2	H	H
❍	*unimplemented_LDTW*	012_{16}	precise	6.3	H	H
❍	*unimplemented_STTW*	013_{16}	precise		H	H
●	*privileged_opcode*	011_{16}	precise	7	P (nm)	-x-
●	*fp_disabled*	020_{16}	precise	8	P (nm)	P (nm)
●	*spill_n_normal* (n = 0–7)	080_{16}‡– $09F_{16}$	precise	9	P (nm)	P (nm)
●	*spill_n_other* (n = 0–7)	$0A0_{16}$‡– $0BF_{16}$	precise		P (nm)	P (nm)
●	*fill_n_normal* (n = 0–7)	$0C0_{16}$‡– $0DF_{16}$	precise		P (nm)	P (nm)
●	*fill_n_other* (n = 0–7)	$0E0_{16}$‡– $0FF_{16}$	precise		P (nm)	P (nm)
●	*clean_window*	024_{16}‡– 027_{16}	precise	10.1	P (nm)	P (nm)
●	*LDDF_mem_address_not_aligned*	035_{16}	precise		H	H
●	*STDF_mem_address_not_aligned*	036_{16}	precise		H	H
❍	*LDQF_mem_address_not_aligned*	038_{16}	precise		H	H
❍	*STQF_mem_address_not_aligned*	039_{16}	precise		H	H
●	*mem_address_not_aligned*	034_{16}	precise	10.2	H	H
❍	*fp_exception_other*	022_{16}	precise	11.1	P (nm)	P (nm)
❍	*fp_exception_ieee_754*	021_{16}	precise		P (nm)	P (nm)
●	*privileged_action*	037_{16}	precise		H	H
●	*control_transfer_instruction*	074_{16}	precise		P	H

TABLE 12-5 Exception and Interrupt Requests, by Priority *(2 of 2)*

UA-2007 ●=Req'd. ❍=Opt'l ❑.=Impl-Specific	Exception or Interrupt Request	TT (Trap Type)	Trap Category	Priority (0 = High-est)	Mode in which Trap is Delivered and (and Conditioning Applied), based on Current Privilege Mode NP	Priv
❍	*VA_watchpoint*	062_{16}	precise	11.2	P (nm)	P (nm)
●	*DAE_invalid_asi*	014_{16}	precise	12.01	H	H
●	*DAE_privilege_violation*	015_{16}	precise	12.04	H	H
●	*DAE_nc_page*	016_{16}	precise	12.05	H	H
●	*DAE_nfo_page*	017_{16}	precise	12.06	H	H
●	*DAE_side_effect_page*	030_{16}	precise	12.06	H	H
●	*tag_overflow*[D]	023_{16}	precise	14	P (nm)	P (nm)
●	*division_by_zero*	028_{16}	precise	15	P (nm)	P (nm)
●	*trap_instruction*	100_{16}–$17F_{16}$	precise	16.02	P (nm)	P (nm)
●	*htrap_instruction*	180_{16}–$1FF_{16}$	precise	16.02	-x-	
●	*cpu_mondo*	$07C_{16}$	disrupting	16.08	P (ie)	P (ie)
●	*dev_mondo*	$07D_{16}$	disrupting	16.11	P (ie)	P (ie)
●	*interrupt_level_n* (n = 1–15)	041_{16}–$04F_{16}$	disrupting	32-n (31 to 17)	P (ie)	P (ie)
●	*resumable_error*	$07E_{16}$	disrupting	33.3	P (ie)	P (ie)
—	*nonresumable_error*	$07F_{16}$	—	—	—	—

* Although these trap priorities are recommended, all trap priorities are implementation dependent (impl. dep. #36-V8 on page 356), including relative priorities within a given priority level.

‡ The trap vector entry (32 bytes) for this trap type plus the next three trap types (total of 128 bytes) are permanently reserved for this exception.

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.

Bit	Field	Description
8:6	spill_or_fill	010_2 for spill traps; 011_2 for fill traps
5	other	(OTHERWIN ≠ 0)
4:2	wtype	If (other) then WSTATE.other; else WSTATE.normal

Trap Type	spill_or_fill	other	wtype	0	0
	8 – 6	5	4 – 2	1	0

FIGURE 12-4 Trap Type Encoding for Spill/Fill Traps

12.5.4 Trap Priorities

TABLE 12-4 on page 351 and TABLE 12-5 on page 354 show the assignment of traps to TT values and the relative priority of traps and interrupt requests. A trap priority is an ordinal number, with 0 indicating the highest priority and greater priority numbers indicating decreasing priority; that is, if $x < y$, a pending exception or interrupt request with priority x is taken instead of a pending exception or interrupt request with priority y. Traps within the same priority class (0 to 33) are listed in priority order in TABLE 12-5 (impl. dep. #36-V8).

IMPL. DEP. #36-V8: The relative priorities of traps defined in the UltraSPARC Architecture are fixed. However, the absolute priorities of those traps are implementation dependent (because a future version of the architecture may define new traps). The priorities (both absolute and relative) of any new traps are implementation dependent.

However, the TT values for the exceptions and interrupt requests shown in TABLE 12-4 and TABLE 12-5 must remain the same for every implementation.

The trap priorities given above always need to be considered within the context of how the virtual processor actually issues and executes instructions.

12.6 Trap Processing

The virtual processor's action during trap processing depends on various virtual processor states, including the trap type, the current level of trap nesting (given in the TL register), and PSTATE. When a trap occurs, the GL register is normally incremented by one (described later in this section), which replaces the set of eight global registers with the next consecutive set.

During normal operation, the virtual processor is in `execute_state`. It processes traps in `execute_state` and continues.

TABLE 12-6 describes the virtual processor mode and trap-level transitions involved in handling traps.

TABLE 12-6 Trap Received While in `execute_state`

Original State	New State, After Receiving Trap or Interrupt
`execute_state` TL < *MAXPTL* – 1	`execute_state` TL ← TL + 1

12.6.1 Normal Trap Processing

A trap is delivered in either privileged mode or hyperprivileged mode, depending on the type of trap, the trap level (TL), and the privilege mode in effect when the exception was detected.

During normal trap processing, the following state changes occur (conceptually, in this order):

- The trap level is updated. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack.

TL ← 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 offPSTATE.priv ← 1 // the virtual processor enters privileged mode
PSTATE.cle ← PSTATE.tle //set endian mode for traps
endif
PSTATE.ie ← 0 // interrupts are disabled
PSTATE.tle is unchanged
PSTATE.tct ← 0 // trap on CTI disabled

- For a register-window trap (*clean_window*, window spill, or window fill) only, CWP is set to point to the register window that must be accessed by the trap-handler software, that is:

if TT[TL] = 024_{16} // a *clean_window* trap
then CWP ← CWP + 1
endif

if (080_{16} ≤ TT[TL] ≤ $0BF_{16}$) // window spill trap
then CWP ← CWP + CANSAVE + 2
endif

if ($0C0_{16}$ ≤ TT[TL] ≤ $0FF_{16}$) // window fill trap
then CWP ← CWP – 1
endif

For non-register-window traps, CWP is not changed.

- Control is transferred into the trap table:

// Note that at this point, TL has already been incremented (above)
if ((trap is to privileged mode) **and** (TL ≤ *MAXPTL*))
then
//the trap is handled in privileged mode
//Note: The expression "(TL > 1)" below evaluates to the
//value 0_2 if TL was 0 just before the trap (in which
//case, TL = 1 now, since it was incremented above,
//during trap entry). "(TL > 1)" evaluates to 1_2 if
//TL was > 0 before the trap.
PC ← TBA{63:15} :: (TL > 1) :: TT[TL] :: 0 0000_2
NPC ← TBA{63:15} :: (TL > 1) :: TT[TL] :: 0 0100_2
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 UltraSPARC Architecture differs from that shown in *The SPARC Architecture Manual-Version 9*. Each trap is marked as precise, deferred, disrupting, or reset. Example exception conditions are included for each exception type. Chapter 7, *Instructions*, enumerates which traps can be generated by each instruction.

The following traps are generally expected to be supported in all UltraSPARC Architecture 2007 implementations. A given trap is not required to be supported in an implementation in which the conditions that cause the trap can never occur.

- ***clean_window*** [TT = 024_{16}–027_{16}] (Precise) — A SAVE instruction discovered that the window about to be used contains data from another address space; the window must be cleaned before it can be used.

  IMPL. DEP. #102-V9: An implementation may choose either to implement automatic cleaning of register windows in hardware or to generate a *clean_window* trap, when needed, so that window(s) can be cleaned by software. If an implementation chooses the latter option, then support for this trap type is mandatory.

- ***control_transfer_instruction*** [TT = 074_{16}] (Precise) — This exception is generated if PSTATE.tct = 1 and the processor determines that a successful control transfer will occur as a result of execution of that instruction. If such a transfer will occur, the processor generates a *control_transfer_instruction* precise trap (trap type = 74_{16}) instead of completing the control transfer. The pc stored in TPC[TL] is the address of the CTI, and the TNPC[TL] is set to the value of NPC before the CTI is executed. (impl. dep. #450-S20). PSTATE.tct is 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 = $07C_{16}$] (Disrupting) — This interrupt is generated when another virtual processor has enqueued a message for this virtual processor. It is used to deliver a trap in privileged mode, to inform privileged software that an interrupt report has been appended to the virtual processor's CPU mondo queue. A direct message between virtual processors is sent via a CPU mondo interrupt. When the CPU mondo queue 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.

SPARC V9 Compatibility Note The *data_access_exception* exception from SPARC V9 and UltraSPARC Architecture 2005 has been replaced by more specific exceptions, such as *DAE_invalid_asi*, *DAE_nc_page*, *DAE_nfo_page*, *DAE_privilege_violation*, and *DAE_side_effect_page*.

- ***DAE_invalid_asi*** [TT = 014_{16}] (Precise) — An attempt was made to execute an invalid combination of instruction and ASI. See the instruction descriptions in Chapter 7 for a detailed list of valid ASIs for each instruction that can access alternate address spaces. The following invalid combinations of instruction, ASI, and virtual address cause a *DAE_invalid_asi* exception:
  - A load, store, load-store, or PREFETCHA instruction with either an invalid ASI or an invalid virtual address for a valid ASI.
  - A disallowed combination of instruction and ASI (see *Block Load and Store ASIs* on page 333 and *Partial Store ASIs* on page 333). This includes the following:
    - an attempt to use a (deprecated) atomic quad load ASI (24_{16}, $2C_{16}$, 34_{16}, or $3C_{16}$) with any load alternate opcode other than LDTXA's (which is shared by LDDA)
    - an attempt to use a nontranslating ASI value with any load or store alternate instruction other than LDXA, LDDFA, STXA, or STDFA
    - an attempt to read from a write-only ASI-accessible register, or load from a store-only ASI (for example, a block commit store ASI, $E0_{16}$ or $E1_{16}$)
    - an attempt to write to a read-only ASI-accessible register
- ***DAE_nc_page*** [TT = 016_{16}] (Precise) —An access to a noncacheable page (TTE.cp = 0) was attempted by an atomic load-store instruction (CASA, CASXA, SWAP, SWAPA, LDSTUB, or LDSTUBA), an LDTXA instruction, a LDBLOCKF instruction, or a STPARTIALF instruction.
- ***DAE_nfo_page*** [TT = 017_{16}] (Precise) — An attempt was made to access a non-faulting-only page (TTE.nfo = 1) by any type of load, store, load-store, or FLUSH instruction with an ASI other than a nonfaulting ASI (`PRIMARY_NO_FAULT[_LITTLE]` or `SECONDARY_NO_FAULT[_LITTLE]`).
- ***DAE_privilege_violation*** [TT = 015_{16}] (Precise) — A privilege violation occurred, due to an attempt to access a privileged page (TTE.p = 1) by any type of load, store, or load-store instruction when executing in nonprivileged mode (PSTATE.priv = 0). This includes the special case of an access by privileged software using one of the `ASI_AS_IF_USER_PRIMARY[_LITTLE]` or `ASI_AS_IF_USER_SECONDARY[_LITTLE]` ASIs.
- ***DAE_side_effect_page*** [TT = 030_{16}] (Precise) — An attempt was made to access a page which may cause side effects (TTE.e = 1) 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.

- ***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 362.
- ***IAE_nfo_page*** [TT = $00C_{16}$] (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a memory page which was marked for access only by nonfaulting loads (TTE.nfo = 1).
- ***IAE_privilege_violation*** [TT = 008_{16}] (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a privileged memory page (TTE.p = 1) while the virtual processor was executing in nonprivileged mode.
- ***IAE_unauth_access*** [TT = $00B_{16}$] (Precise) — An instruction-access exception occurred as a result of an attempt to fetch an instruction from a memory page which was missing “execute” permission (TTE.ep = 0).
- ***illegal_instruction*** [TT = 010_{16}] (Precise) — An attempt was made to execute an ILLTRAP instruction, an instruction with an unimplemented opcode, an instruction with invalid field usage, or an instruction that would result in illegal processor state.

  Examples of cases in which *illegal_instruction* is generated include the following:
  - An instruction encoding does not match any of the opcode map definitions (see Appendix A, *Opcode Maps*).
  - An instruction is not implemented in hardware.
  - A reserved instruction field in Tcc instruction is nonzero.

    If a reserved instruction field in an instruction other than Tcc is nonzero, an *illegal_instruction* exception should be, but is not required to be, generated. (See *Reserved Opcodes and Instruction Fields* on page 86.)
  - An illegal value is present in an instruction i field.
  - An illegal value is present in a field that is explicitly defined for an instruction, such as cc2, cc1, cc0, fcn, impl, rcond, or opf_cc.
  - Illegal register alignment (such as odd rd value in a doubleword load instruction).

- Illegal rd value for LDXFSR, STXFSR, or the deprecated instructions LDFSR or STFSR.
- ILLTRAP instruction.
- DONE or RETRY when TL = 0.

All causes of an *illegal_instruction* exception are described in individual instruction descriptions in Chapter 7, *Instructions*.

> **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 = 075_{16}] (Precise) — The virtual processor has detected that the Program Counter (PC) matches the VA Watchpoint register, when instruction VA watchpoints are enabled and the PC is being translated from a virtual address to a 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 = 041_{16}–$04F_{16}$] (Disrupting) — SOFTINT{n} was set to 1 or an external interrupt request of level n was presented to the virtual processor and n > PIL.

> **Implementation Note** | interrupt_level_14 can be caused by (1) setting SOFTINT{14} to 1, (2) occurrence of a "TICK match", or (3) occurrence of a "STICK match" (see *SOFTINT^P Register (ASRs 20, 21, 22)* on page 54).

- ***LDDF_mem_address_not_aligned*** [TT = 035_{16}] (Precise) — An attempt was made to execute an LDDF or LDDFA instruction and the effective address was not doubleword aligned. (impl. dep. #109)
- ***mem_address_not_aligned*** [TT = 034_{16}] (Precise) — A load/store instruction generated a memory address that was not properly aligned according to the instruction, or a JMPL or RETURN instruction generated a non-word-aligned address. (See also *Special Memory Access ASIs* on page 329.)
- ***nonresumable_error*** [TT = $07F_{16}$] (Disrupting) — There is a valid entry in the nonresumable error queue. This interrupt is not generated by hardware, but is used by hyperprivileged software to inform privileged software that an error report has been appended to the nonresumable error queue.
- ***privileged_action*** [TT = 037_{16}] (Precise) — An action defined to be privileged has been attempted while in nonprivileged mode (PSTATE.priv = 0), or an action defined to be hyperprivileged has been attempted while in nonprivileged or privileged mode. Examples:
  - A data access by nonprivileged software using a restricted (privileged or hyperprivileged) ASI, that is, an ASI in the range 00_{16} to $7F_{16}$ (inclusively)
  - A data access by nonprivileged or privileged software using a hyperprivileged ASI, that is, an ASI in the range 30_{16} to $7F_{16}$ (inclusively)
  - Execution by nonprivileged software of an instruction with a privileged operand value
  - An attempt to read the TICK register by nonprivileged software when nonprivileged access to TICK is disabled (TICK.npt = 1).
  - An attempt to execute a nonprivileged instruction with an operand value requiring more privilege than available in the current privilege mode.
- ***privileged_opcode*** [TT = 011_{16}] (Precise) — An attempt was made to execute a privileged instruction while in nonprivileged mode (PSTATE.priv = 0).
- ***resumable_error*** [TT = $07E_{16}$] (Disrupting) — There is a valid entry in the resumable error queue. This interrupt is used to inform privileged software that an error report has been appended to the resumable error queue, and the current instruction stream is in a consistent state so that execution can be resumed after the error is handled.
- ***spill_n_normal*** [TT = 080_{16}–$09F_{16}$] (Precise)
- ***spill_n_other*** [TT = $0A0_{16}$–$0BF_{16}$] (Precise)

A SAVE or FLUSHW instruction has determined that the contents of a register window must be saved to memory.

- ***STDF_mem_address_not_aligned*** [TT = 036_{16}] (Precise) — An attempt was made to execute an STDF or STDFA instruction and the effective address was not doubleword aligned. (impl. dep. #110)
- ***tag_overflow*** [TT = 023_{16}] (Precise) (deprecated C2) — A TADDccTV or TSUBccTV instruction was executed, and either 32-bit arithmetic overflow occurred or at least one of the tag bits of the operands was nonzero.
- ***trap_instruction*** [TT = 100_{16}–$17F_{16}$] (Precise) — A Tcc instruction was executed and the trap condition evaluated to TRUE, and the software trap number operand of the instruction is 127 or less.
- ***unimplemented_LDTW*** [TT = 012_{16}] (Precise) — An attempt was made to execute an LDTW instruction that is not implemented in hardware on this implementation (impl. dep. #107-V9).
- ***unimplemented_STTW*** [TT = 013_{16}] (Precise) — An attempt was made to execute an STTW instruction that is not implemented in hardware on this implementation (impl. dep. #108-V9).
- ***VA_watchpoint*** [TT = 062_{16}] (Precise) — The virtual processor has detected an attempt to access (load from or store to) a virtual address specified by the VA Watchpoint register, while VA watchpoints are enabled and the address is being translated from a virtual address to a hardware address. If the load or store address is not being translated from a virtual address (for example, the address is being treated as a real address), then a *VA_watchpoint* exception will not be generated even if a match is detected between the VA Watchpoint register and a load or store address.

12.7.1 SPARC V9 Traps Not Used in UltraSPARC Architecture 2007

The following traps were optional in the SPARC V9 specification and are not used in UltraSPARC Architecture 2007:

- ***implementation_dependent_exception_n*** [TT = 077_{16} - $07B_{16}$] This range of implementation-dependent exceptions has been replaced by a set of architecturally-defined exceptions. (impl.dep. #35-V8-Cs20)
- ***LDQF_mem_address_not_aligned*** [TT = 038_{16}] (Precise) — An attempt was made to execute an LDQF instruction and the effective address was word aligned but not quadword aligned. Use of this exception is implementation dependent (impl. dep. #111-V9-Cs10). A separate trap entry for this exception supports fast software emulation of the LDQF instruction when the effective address is word aligned but not quadword aligned. See *Load Floating-Point Register* on page 181. (impl. dep. #111)
- ***STQF_mem_address_not_aligned*** [TT = 039_{16}] (Precise) — An attempt was made to execute an STQF instruction and the effective address was word aligned but not quadword aligned. Use of this exception is implementation dependent (impl. dep. #112-V9-Cs10). A separate trap entry for the exception supports fast software emulation of the STQF instruction when the effective address is word aligned but not quadword aligned. See *Store Floating-Point* on page 253. (impl. dep. #112)

12.8 Register Window Traps

Window traps are used to manage overflow and underflow conditions in the register windows, support clean windows, and implement the FLUSHW instruction.

12.8.1 Window Spill and Fill Traps

A window overflow occurs when a SAVE instruction is executed and the next register window is occupied (CANSAVE = 0). An overflow causes a spill trap that allows privileged software to save the occupied register window in memory, thereby making it available for use.

A window underflow occurs when a RESTORE instruction is executed and the previous register window is not valid (CANRESTORE = 0). An underflow causes a fill trap that allows privileged software to load the registers from memory.

12.8.2 *clean_window* Trap

The virtual processor provides the *clean_window* trap so that system software can create a secure environment in which it is guaranteed that data cannot inadvertently leak through register windows from one software program to another.

A clean register window is one in which all of the registers, including uninitialized registers, contain either 0 or data assigned by software executing in the address space to which the window belongs. A clean window cannot contain register values from another process, that is, from software operating in a different address space.

Supervisor software specifies the number of windows that are clean with respect to the current address space in the CLEANWIN register. This number includes register windows that can be restored (the value in the CANRESTORE register) and the register windows following CWP that can be used without cleaning. Therefore, the number of clean windows available to be used by the SAVE instruction is

CLEANWIN – CANRESTORE

The SAVE instruction causes a *clean_window* exception if this value is 0. This behavior allows supervisor software to clean a register window before it is accessed by a user.

12.8.3 Vectoring of Fill/Spill Traps

To make handling of fill and spill traps efficient, the SPARC V9 architecture provides multiple trap vectors for the fill and spill traps. These trap vectors are determined as follows:

- Supervisor software can mark a set of contiguous register windows as belonging to an address space different from the current one. The count of these register windows is kept in the OTHERWIN register. A separate set of trap vectors (*fill_n_other* and *spill_n_other*) is provided for spill and fill traps for these register windows (as opposed to register windows that belong to the current address space).
- Supervisor software can specify the trap vectors for fill and spill traps by presetting the fields in the WSTATE register. This register contains two subfields, each three bits wide. The WSTATE.normal field determines one of eight spill (fill) vectors to be used when the register window to be spilled (filled) belongs to the current address space (OTHERWIN = 0). If the OTHERWIN register is nonzero, the WSTATE.other field selects one of eight *fill_n_other* (*spill_n_other*) trap vectors.

See *Trap-Table Entry Addresses* on page 348, for more details on how the trap address is determined.

12.8.4 CWP on Window Traps

On a window trap, the CWP is set to point to the window that must be accessed by the trap handler, as follows.

Note | All arithmetic on CWP is done **modulo** *N_REG_WINDOWS*.

- If the spill trap occurs because of a SAVE instruction (when CANSAVE = 0), there is an overlap window between the CWP and the next register window to be spilled:

  CWP ← (CWP + 2) **mod** *N_REG_WINDOWS*

  If the spill trap occurs because of a FLUSHW instruction, there can be unused windows (CANSAVE) in addition to the overlap window between the CWP and the window to be spilled:

  CWP ← (CWP + CANSAVE + 2) **mod** *N_REG_WINDOWS*

  Implementation Note | All spill traps can set CWP by using the calculation:
  CWP ← (CWP + CANSAVE + 2) **mod** *N_REG_WINDOWS*
  since CANSAVE is 0 whenever a trap occurs because of a SAVE instruction.

- On a fill trap, the window preceding CWP must be filled:

  CWP ← (CWP – 1) **mod** *N_REG_WINDOWS*

- On a *clean_window* trap, the window following CWP must be cleaned. Then

  CWP ← (CWP + 1) **mod** *N_REG_WINDOWS*

12.8.5 Window Trap Handlers

The trap handlers for fill, spill, and *clean_window* traps must handle the trap appropriately and return, by using the RETRY instruction, to reexecute the trapped instruction. The state of the register windows must be updated by the trap handler, and the relationships among CLEANWIN, CANSAVE, CANRESTORE, and OTHERWIN must remain consistent. Follow these recommendations:

- A spill trap handler should execute the SAVED instruction for each window that it spills.
- A fill trap handler should execute the RESTORED instruction for each window that it fills.
- A *clean_window* trap handler should increment CLEANWIN for each window that it cleans:

  CLEANWIN ← (CLEANWIN + 1)

CHAPTER 13

Interrupt Handling

Virtual processors and I/O devices can interrupt a selected virtual processor by assembling and sending an interrupt packet. The contents of the interrupt packet are defined by software convention. Thus, hardware interrupts and cross-calls can have the same hardware mechanism for interrupt delivery and share a common software interface for processing.

The interrupt mechanism is a two-step process:

- sending of an interrupt request (through an implemenation-specific hardware mechanism) to an interrupt queue of the target virtual processor
- receipt of the interrupt request on the target virtual processor and scheduling software handling of the interrupt request

Privileged software running on a virtual processor can schedule interrupts to *itself* (typically, to process queued interrupts at a later time) by setting bits in the privileged SOFTINT register (see *Software Interrupt Register (softint)* on page 366).

Programming Note An interrupt request packet is sent by an interrupt source and is received by the specified target in an interrupt queue. Upon receipt of an interrupt request packet, a special trap is invoked on the target virtual processor. The trap handler software invoked in the target virtual processor then schedules itself to later handle the interrupt request by posting an interrupt in the SOFTINT register at the desired interrupt level.

In the following sections, the following aspects of interrupt handling are described:

13.1 Interrupt Packets

Each interrupt is accompanied by data, referred to as an "interrupt packet". An interrupt packet is 64 bytes long, consisting of eight 64-bit doublewords. The contents of these data are defined by software convention.

13.2 Software Interrupt Register (SOFTINT)

To schedule interrupt vectors for processing at a later time, privileged software running on a virtual processor can send itself signals (interrupts) by setting bits in the privileged SOFTINT register.

See *softintP Register (ASRs 20, 21, 22)* on page 54 for a detailed description of the SOFTINT register.

Programming Note | The SOFTINT register (ASR 16_{16}) is used for communication from nucleus (privileged, TL > 0) software to privileged software running with TL = 0. Interrupt packets and other service requests can be scheduled in queues or mailboxes in memory by the nucleus, which then sets SOFTINT{*n*} to cause an interrupt at level *n*.

Programming Note | The SOFTINT mechanism is independent of the "mondo" interrupt mechanism mentioned in *Interrupt Queues* on page 366. The two mechanisms do not interact.

13.2.1 Setting the Software Interrupt Register

SOFTINT{*n*} is set to 1 by executing a WRSOFTINT_SET^{P} instruction (WRasr using ASR 20) with a '1' in bit *n* of the value written (bit *n* corresponds to interrupt level *n*). The value written to the SOFTINT_SET register is effectively **or**ed into the SOFTINT register. This approach allows the interrupt handler to set one or more bits in the SOFTINT register with a single instruction.

See *softint_setP 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 **and**ed with the SOFTINT register. This approach allows the interrupt handler to clear one or more bits in the SOFTINT register with a single instruction.

Programming Note | To avoid a race condition between operating system kernel software clearing an interrupt bit and nucleus software setting it, software should (again) examine the queue for any valid entries after clearing the interrupt bit.

See *softint_clrP Pseudo-Register (ASR 21)* on page 56 for a detailed description of the SOFTINT_CLR pseudo-register.

13.3 Interrupt Queues

Interrupts are indicated to privileged mode via circular interrupt queues, each with an associated trap vector. There are 4 interrupt queues, one for each of the following types of interrupts:

- Device mondos[1]

- CPU mondos
- Resumable errors
- Nonresumable errors

New interrupt entries are appended to the tail of a queue and privileged software reads them from the head of the queue.

Programming Note Software conventions for cooperative management of interrupt queues and the format of queue entries are specified in the separate *Hypervisor API Specification* document.

13.3.1 Interrupt Queue Registers

The active contents of each queue are delineated by a 64-bit head register and a 64-bit tail register.

The interrupt queue registers are accessed through ASI `ASI_QUEUE` (25_{16}). The ASI and address assignments for the interrupt queue registers are provided in TABLE 13-1.

TABLE 13-1 Interrupt Queue Register ASI Assignments

Register	ASI	Virtual Address	Privileged mode Access
CPU Mondo Queue Head	25_{16} (`ASI_QUEUE`)	$3C0_{16}$	RW
CPU Mondo Queue Tail	25_{16} (`ASI_QUEUE`)	$3C8_{16}$	R or RW†
Device Mondo Queue Head	25_{16} (`ASI_QUEUE`)	$3D0_{16}$	RW
Device Mondo Queue Tail	25_{16} (`ASI_QUEUE`)	$3D8_{16}$	R or RW†
Resumable Error Queue Head	25_{16} (`ASI_QUEUE`)	$3E0_{16}$	RW
Resumable Error Queue Tail	25_{16} (`ASI_QUEUE`)	$3E8_{16}$	R or RW†
Nonresumable Error Queue Head	25_{16} (`ASI_QUEUE`)	$3F0_{16}$	RW
Nonresumable Error Queue Tail	25_{16} (`ASI_QUEUE`)	$3F8_{16}$	R or RW†

† 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

Programming Note

By current convention, the format of a Queue Head or Tail register is as follows:

head/tail offset	000000
63 … 6	5 … 0

Under this convention:

- updating a Queue Head register involves incrementing it by 64 (size of a queue entry, in bytes)
- Queue Head and Tail registers are updated using modular arithmetic (modulo the size of the circular queue, in bytes)
- bits 5:0 always read as zeros, and attempts to write to them are ignored
- the maximum queue offset for an interrupt queue is implementation dependent
- behavior when a queue register is written with a value larger than the maximum queue offset (queue length minus the length of the last entry) is undefined

This is merely a convention and is subject to change.

13.4 Interrupt Traps

The following interrupt traps are defined in the UltraSPARC Architecture 2007: *cpu_mondo*, *dev_mondo*, *resumable_error*, and *nonresumable_error*. See Chapter 12, *Traps*, for details.

UltraSPARC Architecture 2007 also supports the *interrupt_level_n* traps defined in the SPARC V9 specification.pt trans

How interrupts are delivered is implementation-specific; see the relevant implementation-specific Supplement to this specification for details.

CHAPTER **14**

Memory Management

An UltraSPARC Architecture Memory Management Unit (MMU) conforms to the requirements set forth in the *SPARC V9 Architecture Manual*. In particular, it supports a 64-bit virtual address space, simplified protection encoding, and multiple page sizes.

IMPL. DEP. # 451-S20: The width of the virtual address supported is implementation dependent. If fewer than 64 bits are supported, the unsupported bits must have the same value as the most significant supported bit. For example, if the model supports 48 virtual address bits, then bits 63:48 must have the same value as bit 47.

This appendix describes the Memory Management Unit, as observed by privileged software, in these sections:

14.1 Virtual Address Translation

The MMUs may support up to eight page sizes: 8 KBytes, 64 KBytes, 512 KBytes, 4 MBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes. 8-KByte, 64-KByte and 4- MByte page sizes must be supported; the other page sizes are optional.

IMPL. DEP. #310-U4: Which, if any, of the following optional page sizes are supported by the MMU in an UltraSPARC Architecture 2007 implementation is implementation dependent: 512 KBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes.

An UltraSPARC Architecture MMU supports a 64-bit virtual address (VA) space.

IMPL. DEP. #452-S20: The number of real address (RA) bits supported is implementation dependent. A minimum of 40 bits and maximum of 56 bits can be provided for real addresses (RA). See implementation-specific documentation for details.

In each translation, the virtual page number is replaced by a physical page number, which is concatenated with the page offset to form the full hardware address, as illustrated in FIGURE 14-1 and FIGURE 14-2.

IMPL. DEP. #453-S20: It is implementation dependent whether there is a unified MMU (UMMU) or a separate IMMU (for instruction accesses) and DMMU (for data accesses). The UltraSPARC Architecture supports both configurations.

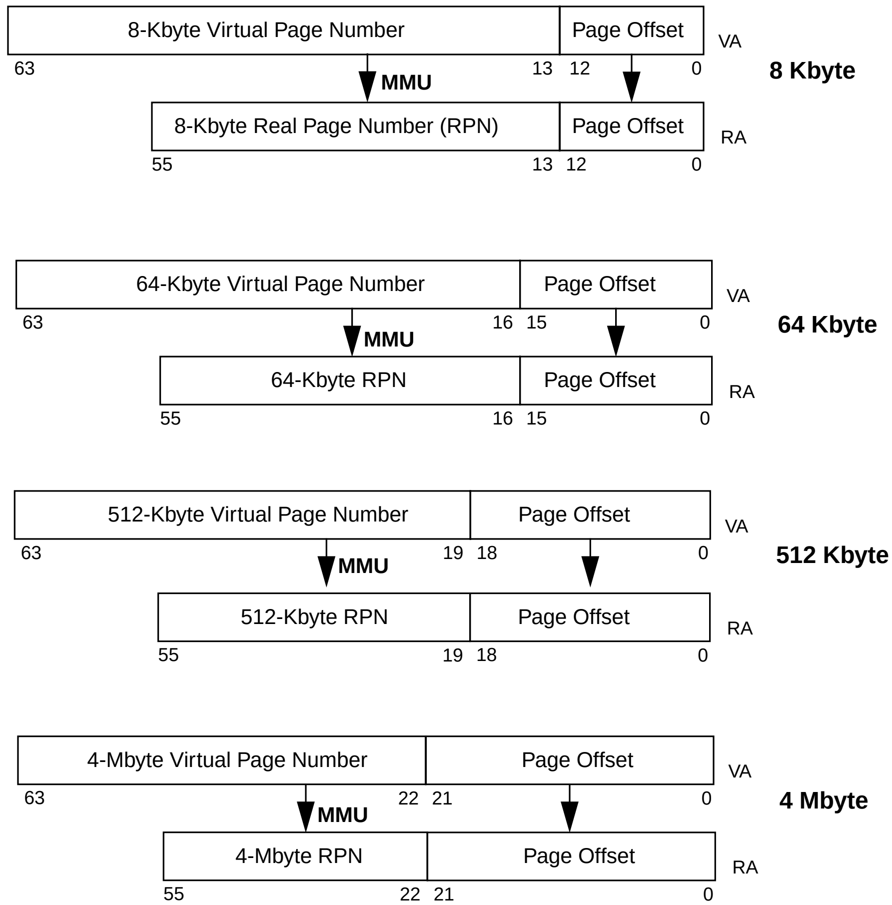


FIGURE 14-1 Virtual-to--Real Address Translation for 8-Kbyte, 64-Kbyte, 512-Kbyte, and 4-Mbyte Page Sizes

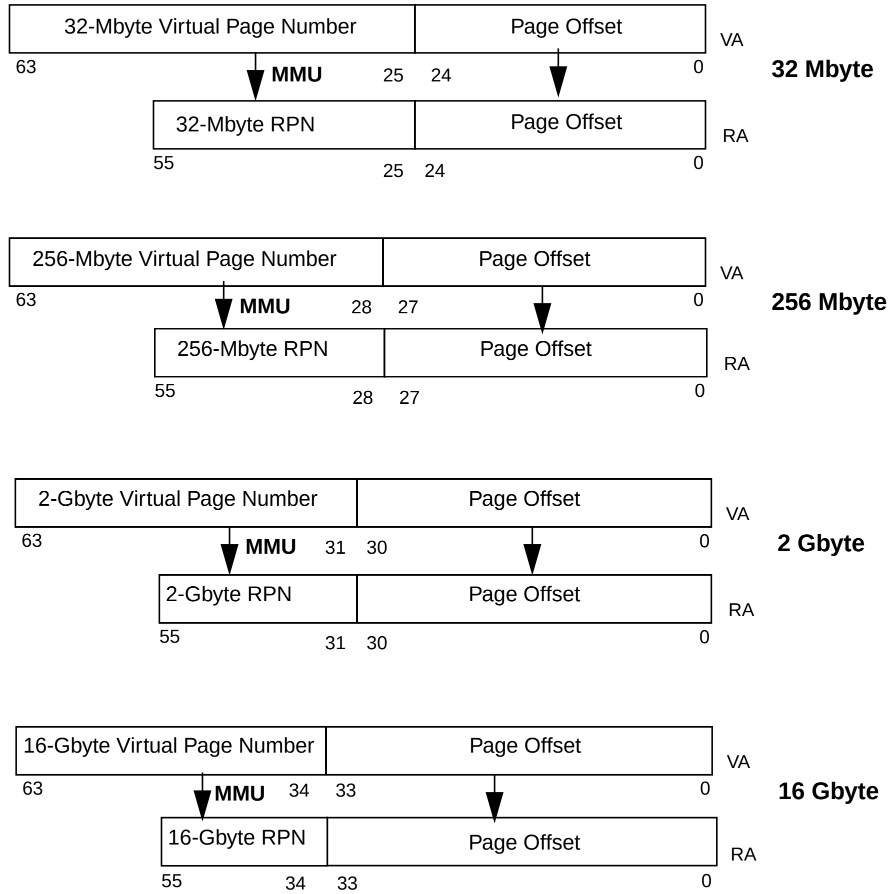


FIGURE 14-2 Virtual-Real Address Translation for 32-Mbyte, 256-Mbyte, 2-Gbyte, and 16-Gbyte Page Sizes

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.

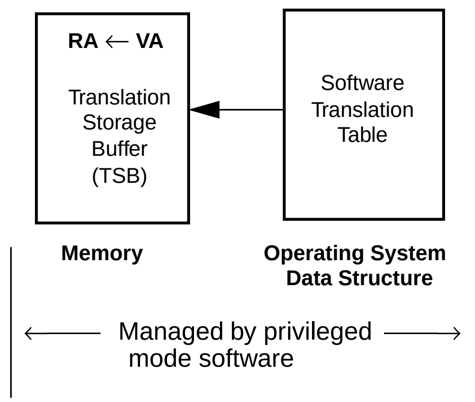


FIGURE 14-3 Conceptual View of the MMU

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

Access Type	Privilege Mode	Under What Conditions each Context is Used		
		Primary Context	**Secondary Context**	**Nucleus Context**
Instruction Access	Nonprivileged or Privileged	(when TL = 0)	†	(when TL > 0)
Data access using *implicit* ASI	Nonprivileged or Privileged	(when TL = 0)	†	(when TL > 0)
Data access using *explicit* ASI	Nonprivileged	`ASI_PRIMARY*`	`ASI_SECONDARY*`	†
	Privileged	`ASI_PRIMARY*` `ASI_AS_IF_USER_PRIMARY*`	`ASI_SECONDARY*` `ASI_AS_IF_USER_SECONDARY*`	`ASI_NUCLEUS*`

†. no context is listed becuase this case cannot occur

Note The UltraSPARC Architecture provides the capability of private and shared contexts. Multiple primary and secondary context IDs, which allow different processes to share TTEs, are defined. See *Context ID Registers* on page 380 for details.

Programming Note Privileged software (operating sytems) intended to be portable across all UltraSPARC Architecture implementations should always ensure that, for memory accesses made in privileged mode, private and shared context IDs are set to the same value. The exception to this is privileged-mode accesses using the `ASI_AS_IF_USER*` ASIs, which remain portable even if the private and shared context IDs differ.

IMPL. DEP. #___: The UltraSPARC Architecture defines a 16-bit context ID. The size of the context ID field is implementation dependent. At least 13 bits must be implemented. If fewer than 16 bits are supported, the unused high order bits are ignored on writes to the context ID, and read as zeros.

14.3 TSB Translation Table Entry (TTE)

The Translation Storage Buffer (TSB) Translation Table Entry (TTE) is the equivalent of a page table entry as defined in the *Sun4v Architecture Specification*; it holds information for a single page mapping. The TTE is divided into two 64-bit words representing the *tag* and *data* of the translation. Just as in a hardware cache, the tag is used to determine whether there is a hit in the TSB; if there is a hit, the data are used by either the hardware tablewalker or privileged software.

The TTE configuration is illustrated in FIGURE 14-4 and described in TABLE 14-2.

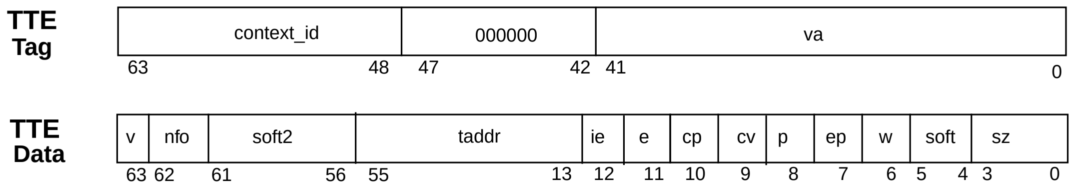


FIGURE 14-4 Translation Storage Buffer (TSB) Translation Table Entry (TTE)

TABLE 14-2 TSB TTE Bit Description *(1 of 3)*

Bit	Field	Description
Tag– **63:48**	context_id	The 16-bit context ID associated with the TTE.
Tag– **47:42**	—	These bits must be zero for a tag match.
Tag– **41:0**	va	Bits 63:22 of the Virtual Address (the virtual page number). Bits 21:13 of the VA are not maintained because these bits index the minimally sized, direct-mapped TSBs.
Data – **63**	v	Valid. If v = 1, then the remaining fields of the TTE are meaningful, and the TTE can be used; otherwise, the TTE cannot be used to translate a virtual address. **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.

TABLE 14-2 TSB TTE Bit Description *(2 of 3)*

<table>
<tr><th>Bit</th><th>Field</th><th>Description</th></tr>
<tr><td>Data – 12</td><td>ie</td><td>Invert Endianness. If ie = 1 for a page, accesses to the page are processed with inverse endianness from that specified by the instruction (big for little, little for big). See page 377 for details.

Programming Notes (1) The primary purpose of this bit is to aid in the mapping of I/O devices (through noncacheable memory addresses) whose registers contain and expect data in little-endian format. Setting TTE.ie = 1 allows those registers to be accessed correctly by big-endian programs using ordinary loads and stores, such as those typically issued by compilers; otherwise little-endian loads and stores would have be issued by hand-written assembler code.

(2) This bit can also be used when mapping cacheable memory. However, cacheable accesses to pages marked with TTE.ie = 1 may be slower than accesses to the page with TTE.ie = 0. For example, an access to a cacheable page with TTE.ie = 1 may perform as if there was a miss in the first-level data cache.

Implementation Note Some implementations may require cacheable accesses to pages tagged with TTE.ie = 1 to bypass the data cache, adding latency to those accesses.

IMPL. DEP. #__: 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.</td></tr>
<tr><td>Data – 11</td><td>e</td><td>Side effect. If the side-effect bit is set to 1, loads with <code>ASI_PRIMARY_NO_FAULT</code>, <code>ASI_SECONDARY_NO_FAULT</code>, and their <code>*_LITTLE</code> variations will trap for addresses within the page, noncacheable memory accesses other than block loads and stores are strongly ordered against other e-bit accesses, and noncacheable stores are not merged. This bit should be set to 1 for pages that map I/O devices having side effects. Note, also, that the e bit causes the prefetch instruction to be treated as a nop, but does not prevent normal (hardware) instruction prefetching.
Note 1: The e bit does not force a noncacheable access. It is expected, but not required, that the cp and cv bits will be set to 0 when the e bit is set to 1. If both the cp and cv bits are set to 1 along with the e bit, the result is undefined.
Note 2: The e bit and the nfo bit are mutually exclusive; both bits should never be set to 1 in any TTE.</td></tr>
<tr><td>Data – 10
Data – 9</td><td>cp,
cv</td><td>The cacheable-in-physically-indexed-cache bit and cacheable-in-virtually-indexed-cache bit determine the cacheability of the page. Given an implementation with a physically indexed instruction cache, a virtually indexed data cache, and a physically indexed unified second-level cache, the following table illustrates how the cp and cv bits could be used:
<table>
<tr><th rowspan="2">Cacheable (cp:cv)</th><th colspan="2">Meaning of TTE when placed in:</th></tr>
<tr><th>I-TLB (Instruction Cache PA-indexed)</th><th>D-TLB (Data Cache VA-indexed)</th></tr>
<tr><td>00, 01</td><td>Noncacheable</td><td>Noncacheable</td></tr>
<tr><td>10</td><td>Cacheable L2-cache, I-cache</td><td>Cacheable L2-cache</td></tr>
<tr><td>11</td><td>Cacheable L2-cache, I-cache</td><td>Cacheable L2-cache, D-cache</td></tr>
</table>
The MMU does not operate on the cacheable bits but merely passes them through to the cache subsystem. The cv bit in the IMMU is read as zero and ignored when written.
IMPL. DEP. #226-U3: Whether the cv bit is supported in hardware is implementation dependent in the UltraSPARC Architecture. The cv bit in hardware should be provided if the implementation has virtually indexed caches, and the implementation should support hardware unaliasing for the caches.</td></tr>
</table>

TABLE 14-2 TSB TTE Bit Description *(3 of 3)*

Bit	Field	Description
Data – **8**	p	Privileged. If p = 1, only privileged software can access the page mapped by the TTE. If p = 1 and an access to the page is attempted by nonprivileged mode (PSTATE.priv = 0), then the MMU signals an*IAE_privilege_violation* exception or*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 *IAE_unauth_access* exception. **IMPL. DEP. #___:** An UltraSPARC Architecture ITLB implementation may elect to not implement the ep bit, and instead present the *IAE_unauth_access* exception if there is an attempt to load an ITLB entry with ep = 0 during a hardware tablewalk. In this case, the MMU miss trap handler software must also detect the ep = 0 case when the IMMU miss is handled by software.
Data – **6**	w	**IMPL. DEP.** #Writable. If w = 1, the page mapped by this TTE has write permission granted. Otherwise, write permission is not granted
Data – **5:4**	soft	Software-defined field, provided for use by the operating system. The soft field can be written with any value in the TSB. Hardware is not required to maintain this field in any TLB (or uTLB), so when it is read from the TLB (or uTLB), it may read as zero.
Data – 3:**0**	sz	The page size of this entry, encoded as shown below. sz — Page Size 0000 — 8 Kbyte 0001 — 64 Kbyte 0010 — 512 Kbyte 0011 — 4 Mbyte 0100 — 32 Mbyte 0101 — 256 Mbyte 0110 — 2 Gbyte 0111 — 16 Gbyte 1000-1111 — *Reserved*

14.4 Translation Storage Buffer (TSB)

The Translation Storage Buffer (TSB) is an array of Translation Table Entries managed entirely by privileged software. It serves as a cache of the software translation table, used to quickly reload the TLB in the event of a TLB miss.

14.4.1 TSB Indexing Support

Hardware TSB indexing support via TSB pointers should be provided for the TTEs.

14.4.2 TSB Cacheability and Consistency

The TSB exists as a data structure in memory and therefore can be cached. Indeed, the speed of the TLB miss handler relies on the TSB accesses hitting the level-2 cache at a substantial rate. This policy may result in some conflicts with normal instruction and data accesses, but the dynamic sharing of the level-2 cache resource will provide a better overall solution than that provided by a fixed partitioning.

Programming Note When software updates the TSB, it is responsible for ensuring that the store(s) used to perform the update are made visible in the memory system (for access by subsequent loads, stores, and load-stores) by use of an appropriate MEMBAR instruction.

Making a TSB update visible to fetches of instructions subsequent to the store(s) that updated the TSB may require execution of instructions such as FLUSH, DONE, or RETRY, in addition to the MEMBAR.

14.4.3 TSB Organization

The TSB is arranged as a direct-mapped cache of TTEs.

In each case, n least significant bits of the respective virtual page number are used as the offset from the TSB base address, with n equal to log 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.

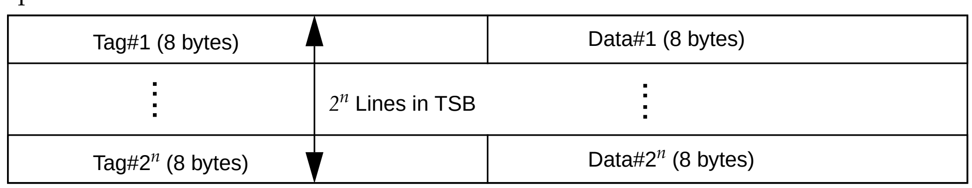


FIGURE 14-5 TSB Organization

IMPL. DEP. #227-U3: The maximum number of entries in a TSB is implementation-dependent in the UltraSPARC Architecture (to a maximum of 16 million).

14.5 ASI Value, Context ID, and Endianness Selection for Translation

The selection of the context ID for a translation is the result of a two-step process:

1. The ASI is determined (conceptually by the Integer Unit) from the instruction, ASI register, trap level, privilege level (PSTATE.priv) and the virtual processor endian mode (PSTATE.cle).

2. The context ID is determined directly from the ASI. The context ID value is read by the context ID selected by the ASI.

The ASI value and endianness (little or big) are determined, according to TABLE 14-3 through TABLE 14-4.

When using the Primary Context ID, the values stored in the Primary Context IDs are used by the Data (or Unified) MMU. The Secondary Context ID is never used for instruction accesses.

The endianness of a data access is specified by three conditions:

- The ASI specified in the opcode or ASI register
- The PSTATE current little-endian bit (cle)
- The TTE "invert endianness" bit (ie). The TTEbit 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 14-3 ASI Mapping for Instruction Access

Mode	TL	PSTATE.cle	Endianness	ASI Used	Resulting Address Type
Nonprivileged	0	—	Big	ASI_PRIMARY	VA
Privileged	0	—	Big	ASI_PRIMARY	VA
	1–2	—	Big	ASI_NUCLEUS	VA

TABLE 14-4 ASI Mapping for Data Accesses *(1 of 2)*

Access Type	Privi-lege Mode	TL	PSTATE.cle	TTE .ie	Endian-ness	ASI Used	Resulting Address Type
Load, Store, Atomic Load-Store, or Prefetch with implicit ASI	NP	0[1]	0	0	Big	ASI_PRIMARY	VA
				1	Little		
		0[1]	1	0	Little	ASI_PRIMARY_LITTLE	VA
				1	Big		
	P	0	0	0	Big	ASI_PRIMARY	VA
				1	Little		
		0	1	0	Little	ASI_PRIMARY_LITTLE	VA
				1	Big		
		1-2[1]	0	0	Big	ASI_NUCLEUS	VA
				1	Little		
		1-2[1]	1	0	Little	ASI_NUCLEUS_LITTLE	VA
				1	Big		
Load, Store, Atomic Load-Store, or Prefetch alternate with ASI name *not* ending in _LITTLE	NP	0[1]	any	0	Big[2]	Explicitly specified in instruction	VA
				1	Little[1]		
	P	0-2[1]	any	0	Big[1]	Explicitly specified in instruction	VA
				1	Little[1]		
		0-2[1]	any	0	Big	ASI_*REAL* ASI	RA
				1	Little		
		0-2[1]	any	any	Big	Nontranslating ASIs	—

TABLE 14-4 ASI Mapping for Data Accesses *(2 of 2)*

Access Type	Privilege Mode	TL	PSTATE.cle	TTE.ie	Endianness	ASI Used	Resulting Address Type
Load, Store, Atomic Load-Store, or Prefetch alternate with ASI name ending in `_LITTLE`	NP	0[1]	any	0	Little	Explicitly specified in instruction	VA
				1	Big		
	P	0-2[1]	any	0	Little	Explicitly specified in instruction)	VA
				1	Big		
		0-2[1]	any	0	Little	`ASI_*REAL*` ASI	RA
				1	Big		

1. *MAXPTL* = 2 for UltraSPARC Architecture 2007 processors. Privilege mode operation is valid only for TL = 0, 1 or 2. Nonprivileged mode operation is valid only for TL = 0. See section 5.6.7 for details.

2. Accesses to nontranslating ASIs are always made in big endian mode, regardless of the setting of TTE.ie. See *ASI Values* on page 321 for information about nontranslating ASIs.

The Context ID used by the data and instruction MMUs is determined according to TABLE 14-5. The Context ID selection is not affected by the endianness of the access. For a comprehensive list of ASI values in the ASI map, see Chapter 10, *Address Space Identifiers (ASIs)*.

TABLE 14-5 IMMU, DMMU and UMMU Context ID Usage

ASI Value	Context ID Register
`ASI_*NUCLEUS*` (any ASI name containing the string "`NUCLEUS`")	Nucleus (0000_{16}, hard-wired)
`ASI_*PRIMARY*` (any ASI name containing the string "`PRIMARY`")	All Primary Context IDs
`ASI_*SECONDARY*` (any ASI name containing the string "`SECONDARY`")	All Secondary Context IDs
All other ASI values	(Not applicable; no translation)

14.6 SPARC V9 "MMU Attributes"

The UltraSPARC Architecture MMU complies completely with the SPARC V9 "MMU Attributes" as described in Appendix F.3.2.

With regard to Read, Write and Execute Permissions, SPARC V9 says "An MMU may allow zero or more of read, write and execute permissions, on a per-mapping basis. Read permission is necessary for data read accesses and atomic accesses. Write permission is necessary for data write accesses and atomic accesses. Execute permission is necessary for instruction accesses. At a minimum, an MMU must allow for 'all permissions', 'no permissions', and 'no write permission'; optionally, it can provide 'execute only' and 'write only', or any combination of 'read/write/execute' permissions."

TABLE 14-6 shows how various protection modes can be achieved, if necessary, through the presence or absence of a translation in the instruction or data MMU. Note that this behavior requires specialized TLB-miss handler code to guarantee these conditions.

TABLE 14-6 MMU SPARC V9 Appendix F.3.2 Protection Mode Compliance

Condition					Resultant Protection Mode
TTE in DMMU	TTE in IMMU	TTE in UMMU	ep Bit	Writable Attribute Bit	
Yes	No	Yes	0	0	Read-only[1]
No	Yes	N/A	1	N/A	Execute-only[1]
Yes	No	Yes	0	1	Read/Write[1]
Yes	Yes	Yes	1	0	Read-only/Execute
Yes	Yes	Yes	1	1	Read/Write/Execute
No	No	No	N/A	N/A	No Access

1. These protection modes are optional, according to SPARC V9.

14.6.1 Accessing MMU Registers

All internal MMU registers can be accessed directly by the virtual processor through defined ASIs, using LDXA and STXA instructions. UltraSPARC Architecture-compatible processors do not require a MEMBAR #Sync, FLUSH, DONE, or RETRY instruction after a store to an MMU register for proper operation.

TABLE 14-7 lists the MMU registers and provides references to sections with more details.

TABLE 14-7 MMU Internal Registers and ASI Operations

IMMU ASI	D/UMMU ASI	VA{63:0}	Access	Register or Operation Name
21_{16}		8_{16}	RW	Primary Context ID 0 register
—	21_{16}	10_{16}	RW	Secondary Context ID 0 register
21_{16}		108_{16}	RW	Primary Context ID 1 register
—	21_{16}	110_{16}	RW	Secondary Context ID 1 register

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

Register	ASI	Virtual Address
Primary Context ID 0	21_{16}	008_{16}
Primary Context ID 1	21_{16}	108_{16}
Secondary Context ID 0	21_{16}	010_{16}
Secondary Context ID 1	21_{16}	110_{16}

Programming Note

> For platforms that implement more than one primary context ID and one secondary context ID, privileged code must ensure that no more than one page translation is allowed to match at any time. An illustration of erroneous behavior is as follows:
>
> 1. An operating system constructs a mapping for virtual address *A* valid for context ID *P*;
> 2. it then constructs a mapping for address *A* for context ID *Q*.
>
> By setting Primary Context ID 0 to *P* and Primary Context ID 1 to *Q*, both mappings would be active simultaneously, with conflicting translations for address *A*. Care must be taken not to construct such scenarios.

UltraSPARC Architecture processors must prevent errors or data corruption due to multiple valid translations for a given virtual address using different contexts. TLBs may need to detect this scenario as a multiple tag hit error and cause an exception for such an access.

The Primary Context ID register is illustrated in FIGURE 14-6, where pcontext is the context ID for the primary address space.

Primary Context ID

—	pcontextid
63 16	15 0

FIGURE 14-6 IMMU, DMMU, and UMMU Primary Context ID

The Secondary Context ID register is illustrated in FIGURE 14-7, where scontextid is the context ID for the secondary address space.

Secondary Context ID

—	scontextid
63 16	15 0

FIGURE 14-7 D/UMMU Secondary Context ID

The Nucleus Context ID register is hardwired to zero, as illustrated in FIGURE 14-6.

Nucleus Context ID

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
63   0

FIGURE 14-8 IMMU, DMMU, and UMMU Nucleus Context ID

IMPL. DEP. #415-S10: The size of context ID fields in MMU context registers is implementation-dependent and may range from 13 to 16 bits.

APPENDIX **A**

Opcode Maps

This appendix contains the UltraSPARC Architecture 2007 instruction opcode maps.

In this appendix and in Chapter 7, *Instructions*, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE 7-1 on page 87. For preferred substitute instructions for deprecated opcodes, see the individual opcodes in Chapter 7 that are labeled "Deprecated".

In the tables in this appendix, *reserved* (—) and shaded entries (as defined below) indicate opcodes that are not implemented in UltraSPARC Architecture 2007 strands.

Shading	**Meaning**
(shaded)	An attempt to execute opcode will cause an *illegal_instruction* exception.

An attempt to execute a reserved opcode behaves as defined in *Reserved Opcodes and Instruction Fields* on page 86.

TABLE A-1 op{1:0}

op {1:0}			
0	**1**	**2**	**3**
Branches and SETHI (*See* TABLE A-2)	CALL	Arithmetic & Miscellaneous (*See* TABLE A-3)	Loads/Stores (*See* TABLE A-4)

TABLE A-2 op2{2:0} (op = 0)

op2 {2:0}							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ILLTRAP	BPcc (*See* TABLE A-7)	Bicc^D(*See* TABLE A-7)	BPr (bit 28 = 0) (*See* TABLE A-8) — (bit 28 = 1)[1]	SETHI, NOP[2]	FBPfcc (*See* TABLE A-7)	FBfcc^D (*See* TABLE A-7)	—

1. See the footnote regarding bit 28 on page 109.
2. rd = 0, imm22 = 0

TABLE A-3 op3{5:0} (op = 10_2) *(1 of 2)*

		op3{5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**0**	ADD	ADDcc	TADDcc	WRY[D] (rd = 0) — (rd = 1) WRCCR (rd = 2 WRASI (rd = 3) — (rd = 4, 5) — (rd = 15, rs1 = 0, i = 1) — (rd = 15) **and** (rs1 ≠ 0 **or** i ≠ 1)) — (rd = 7 – 14) WRFPRS (rd = 6) WRasr[PASR] (7 ≤ rd ≤ 14) WRPCR[P] (rd = 16) WRPIC (rd = 17) — (rd = 18) WRGSR (rd = 19) WRSOFTINT_SET[P] (rd = 20) WRSOFTINT_CLR[P] (rd = 21) WRSOFTINT[P] (rd = 22) WRTICK_CMPR[P] (rd = 23) WRSTICK_CMPR[P] (rd = 25) — (rd = 26) — (rd = 27) — (rd = 28 - 31)
	1	AND	ANDcc	TSUBcc	SAVED[P] (fcn = 0) RESTORED[P] (fcn = 1) ALLCLEAN[P] (fcn = 2) OTHERW[P] (fcn = 3) NORMALW[P] (fcn = 4) INVALW[P] (fcn = 5) — (fcn ≥ 6)
	2	OR	ORcc	TADDccTV[D]	—
	2	OR	ORcc	TADDccTV[D]	WRPR[P] (rd = 0-14 **or** 16) — (rd = 15 **or** 17–31)
	3	XOR	XORcc	TSUBccTV[D]	—
	4	SUB	SUBcc	MULScc[D]	FPop1 (*See* TABLE A-5)
	5	ANDN	ANDNcc	SLL (x = 0), SLLX (x = 1)	FPop2 (*See* TABLE A-6)
	6	ORN	ORNcc	SRL (x = 0), SRLX (x = 1)	(VIS) (*See* TABLE A-12)
	7	XNOR	XNORcc	SRA (x = 0), SRAX (x = 1)	

TABLE A-3 op3{5:0} (op = 10_2) *(2 of 2)*

		op3{5:4}			
		0	**1**	**2**	**3**
	8	ADDC	ADDCcc	RDY[D] (rs1 = 0, i = 0) — (rs1 = 1, i = 0) RDCCR (rs1= 2, i = 0) RDASI (rs1 = 3, i = 0) RDTICK[Pnpt] (rs1 = 4, i = 0) RDPC (rs1 = 5, i = 0) RDFPRS (rs1 = 6, i = 0) RDasr[PASR] (7 ≤ rd ≤ 14, i = 0) MEMBAR (rs1 = 15, rd = 0, i = 1, instruction bit 12 = 0) — (rs1 = 15, rd = 0, i = 1, instruction bit 12 = 1) — (i = 1, (rs1 ≠ 15 or rd ≠ 0)) — (rs1 = 15, rd = 0, i = 0) — (rs1 = 15 **and** rd > 0 **and** i = 0) RDPCR[P] (rs1 = 16 **and** i = 0) RDPIC (rs1 = 17 **and** i = 0) — (rs1 = 18 **and** i = 0) RDGSR (rs1 = 19 **and** i = 0) — (rs1 = 20 or 21) **and** (i = 0)) RDSOFTINT[P] (rs1 = 22 **and** i = 0) RDTICK_CMPR[P] (rs1 = 23 **and** i = 0) RDSTICK (rs1 = 24 **and** i = 0) RDSTICK_CMPR[P] (rs1 = 25 **and** i = 0) — ((rs1 = 26) **and** (i = 0)) — ((rs1 = 27 – 31) **and** (i = 0))	JMPL
op3 {3:0}	**A**	UMUL[D]	UMULcc[D]	RDPR[P] (rs1 = 1–14 or 16) — (rs1 = 15 or 17 – 31)	Tcc ((i = 0 **and** inst{10:5} = 0) **or** ((i = 1) **and** (inst{10:8} = 0))) (*See* TABLE A-7) — (bit 29 = 1) — ((i = 0 **and** (inst{10:5} ≠ 0)) **or** (i = 1 **and** (inst{10:8} ≠ 0))
	B	SMUL[D]	SMULcc[D]	FLUSHW	FLUSH
	C	SUBC	SUBCcc	MOVcc	SAVE
	D	UDIVX	—	SDIVX	RESTORE
	E	UDIV[D]	UDIVcc[D]	POPC (rs1 = 0) — (rs1 > 0)	DONE[P] (fcn = 0) RETRY[P] (fcn = 1) — (fcn = 16..31)
	F	SDIV[D]	SDIVcc[D]	MOVr (*See* TABLE A-8)	—

op3 {3:0}

TABLE A-4 op3{5:0} (op = 11_2)

		op3{5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**0**	LDUW	LDUWA[PASI]	LDF	LDFA[PASI]
	1	LDUB	LDUBA[PASI]	(rd = 0) LDFSR[D] (rd = 1) LDXFSR — (rd > 1)	*Reserved*
	2	LDUH	LDUHA[PASI]	LDQF	LDQFA[PASI]
	3	LDTW[D] — (rd odd)	LDTWA[D, PASI] LDTXA — (rd odd)	LDDF	LDDFA[PASI] LDBLOCKF LDSHORTF
	4	STW	STWA[PASI]	STF	STFA[PASI]
	5	STB	STBA[PASI]	STFSR[D], STXFSR — (rd > 1)	*Reserved*
	6	STH	STHA[PASI]	STQF	STQFA[PASI]
	7	STTW[D] — (rd odd)	STTWA[PASI] — (rd odd)	STDF	STDFA[PASI] STLBLOCKF STPARTIALF STSHORTF
	8	LDSW	LDSWA[PASI]	*Reserved*	*Reserved*
	9	LDSB	LDSBA[PASI]	*Reserved*	*Reserved*
	A	LDSH	LDSHA[PASI]	*Reserved*	*Reserved*
	B	LDX	LDXA[PASI]	*Reserved*	*Reserved*
	C	*Reserved*	*Reserved*	*Reserved*	CASA[PASI]
	D	LDSTUB	LDSTUBA[PASI]	PREFETCH — (fcn = 5 – 15)	PREFETCHA[PASI] — (fcn = 5 – 15)
	E	STX	STXA[PASI]	*Reserved*	CASXA[PASI]
	F	SWAP[D]	SWAPA[D, PASI]	*Reserved*	*Reserved*

TABLE A-5 opf{8:0} (op = 10_2, op3 = 34_{16} = FPop1)

	opf{3:0}							
opf{8:4}	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
00_{16}	—	FMOVs	FMOVd	FMOVq	—	FNEGs	FNEGd	FNEGq
01_{16}	—	—	—	—	—	—	—	—
02_{16}	—	—	—	—	—	—	—	—
03_{16}	—	—	—	—	—	—	—	—
04_{16}	—	FADDs	FADDd	FADDq	—	FSUBs	FSUBd	FSUBq
05_{16}	—	—	—	—	—	—	—	—
06_{16}	—	—	—	—	—	—	—	—
07_{16}	—	—	—	—	—	—	—	—
08_{16}	—	FsTOx	FdTOx	FqTOx	FxTOs	—	—	—
09_{16}	—	—	—	—	—	—	—	—
$0A_{16}$	—	—	—	—	—	—	—	—
$0B_{16}$	—	—	—	—	—	—	—	—
$0C_{16}$	—	—	—	—	FiTOs	—	FdTOs	FqTOs
$0D_{16}$	—	FsTOi	FdTOi	FqTOi	—	—	—	—
$0E_{16}$–$1F_{16}$	—	—	—	—	—	—	—	—
	8	**9**	**A**	**B**	**C**	**D**	**E**	**F**
00_{16}	—	FABSs	FABSd	FABSq	—	—	—	—
01_{16}	—	—	—	—	—	—	—	—
02_{16}	—	FSQRTs	FSQRTd	FSQRTq	—	—	—	—
03_{16}	—	—	—	—	—	—	—	—
04_{16}	—	FMULs	FMULd	FMULq	—	FDIVs	FDIVd	FDIVq
05_{16}	—	—	—	—	—	—	—	—
06_{16}	—	FsMULd	—	—	—	—	FdMULq	—
07_{16}	—	—	—	—	—	—	—	—
08_{16}	FxTOd	—	—	—	FxTOq	—	—	—
09_{16}	—	—	—	—	—	—	—	—
$0A_{16}$	—	—	—	—	—	—	—	—
$0B_{16}$	—	—	—	—	—	—	—	—
$0C_{16}$	FiTOd	FsTOd	—	FqTOd	FiTOq	FsTOq	FdTOq	—
$0D_{16}$	—	—	—	—	—	—	—	—
$0E_{16}$–$1F_{16}$	—	—	—	—	—	—	—	—

TABLE A-6 opf{8:0} (op = 10_2, op3 = 35_{16} = FPop2)

	opf{3:0}								
opf{8:4}	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8–F**
00_{16}	—	FMOVs (fcc0)	FMOVd (fcc0)	FMOVq (fcc0)	—	† ‡	† ‡	† ‡	—
01_{16}	—	—	—	—	—	—	—	—	—
02_{16}	—	—	—	—	—	FMOVRsZ ‡	FMOVRdZ ‡	FMOVRqZ ‡	—
03_{16}	—	—	—	—	—	—	—	—	—
04_{16}	—	FMOVs (fcc1)	FMOVd (fcc1)	FMOVq (fcc1)	—	FMOVRsLEZ ‡	FMOVRdLEZ ‡	FMOVRqLEZ ‡	—
05_{16}	—	FCMPs	FCMPd	FCMPq	—	FCMPEs ‡	FCMPEd ‡	FCMPEq ‡	—
06_{16}	—	—	—	—	—	FMOVRsLZ ‡	FMOVRdLZ ‡	FMOVRqLZ ‡	—
07_{16}	—	—	—	—	—	—	—	—	—
08_{16}	—	FMOVs (fcc2)	FMOVd (fcc2)	FMOVq (fcc2)	—	†	†	†	—
09_{16}	—	—	—	—	—	—	—	—	—
$0A_{16}$	—	—	—	—	—	FMOVRsNZ ‡	FMOVRdNZ ‡	FMOVRqNZ ‡	—
$0B_{16}$	—	—	—	—	—	—	—	—	—
$0C_{16}$	—	FMOVs (fcc3)	FMOVd (fcc3)	FMOVq (fcc3)	—	FMOVRsGZ ‡	FMOVRdGZ ‡	FMOVRqGZ ‡	—
$0D_{16}$	—	—	—	—	—	—	—	—	—
$0E_{16}$	—	—	—	—	—	FMOVRsGEZ ‡	FMOVRdGEZ ‡	FMOVRqGEZ ‡	—
$0F_{16}$	—	—	—	—	—	—	—	—	—
10_{16}	—	FMOVs (icc)	FMOVd (icc)	FMOVq (icc)	—	—	—	—	—
11_{16}–17_{16}	—	—	—	—	—	—	—	—	—
18_{16}	—	FMOVs (xcc)	FMOVd (xcc)	FMOVq (xcc)	—	—	—	—	—
19_{16}–$1F_{16}$	—	—	—	—	—	—	—	—	—

† Reserved variation of FMOVR ‡ bit 13 of instruction = 0

TABLE A-7 cond{3:0}

		BPcc op = 0 op2 = 1 bit 28 = 0		Bicc op = 0 op2 = 2	FBPfcc op = 0 op2 = 5	FBfcc[D] op = 0 op2 = 6	Tcc op = 2 op3 = $3A_{16}$
cond {3:0}	0	BPN		BN[D]	FBPN	FBN[D]	TN
	1	BPE		BE[D]	FBPNE	FBNE[D]	TE
	2	BPLE		BLE[D]	FBPLG	FBLG[D]	TLE
	3	BPL		BL[D]	FBPUL	FBUL[D]	TL
	4	BPLEU		BLEU[D]	FBPL	FBL[D]	TLEU
	5	BPCS		BCS[D]	FBPUG	FBUG[D]	TCS
	6	BPNEG		BNEG[D]	FBPG	FBG[D]	TNEG
	7	BPVS		BVS[D]	FBPU	FBU[D]	TVS
	8	BPA		BA[D]	FBPA	FBA[D]	TA
	9	BPNE		BNE[D]	FBPE	FBE[D]	TNE
	A	BPG		BG[D]	FBPUE	FBUE[D]	TG
	B	BPGE		BGE[D]	FBPGE	FBGE[D]	TGE
	C	BPGU		BGU[D]	FBPUGE	FBUGE[D]	TGU
	D	BPCC		BCC[D]	FBPLE	FBLE[D]	TCC
	E	BPPOS		BPOS[D]	FBPULE	FBULE[D]	TPOS
	F	BPVC		BVC[D]	FBPO	FBO[D]	TVC

TABLE A-8 Encoding of rcond{2:0} Instruction Field

		BPr op = 0 op2 = 3	MOVr op = 2 op3 = $2F_{16}$	FMOVr op = 2 op3 = 35_{16}
rcond {2:0}	0	—	—	—
	1	BRZ	MOVRZ	FMOVR<s|d|q>Z
	2	BRLEZ	MOVRLEZ	FMOVR<s|d|q>LEZ
	3	BRLZ	MOVRLZ	FMOVR<s|d|q>LZ
	4	—	—	—
	5	BRNZ	MOVRNZ	FMOVR<s|d|q>NZ
	6	BRGZ	MOVRGZ	FMOVR<s|d|q>GZ
	7	BRGEZ	MOVRGEZ	FMOVR<s|d|q>GEZ

TABLE A-9 cc / opf_cc Fields (MOVcc and FMOVcc)

opf_cc			Condition Code Selected
cc2	cc1	cc0	
0	0	0	fcc0
0	0	1	fcc1
0	1	0	fcc2
0	1	1	fcc3
1	0	0	icc

TABLE A-9 cc / opf_cc Fields (MOVcc and FMOVcc)

1	0	1	—
1	1	0	xcc
1	1	1	—

TABLE A-10 cc Fields (FBPfcc, FCMP, and FCMPE)

cc1	cc0	Condition Code Selected
0	0	fcc0
0	1	fcc1
1	0	fcc2
1	1	fcc3

TABLE A-11 cc Fields (BPcc and Tcc)

cc1	cc0	Condition Code Selected
0	0	icc
0	1	—
1	0	xcc
1	1	—

TABLE A-12 opf{8:0} for VIS opcodes (op = 10_2, op3 = 36_{16})

		opf {8:4}									
		00	01	02	03	04	05	06	07	08	09-0F
	0	EDGE8cc	ARRAY8	FCMPLE16	—	—	FPADD16	FZERO	FAND	—	
	0	EDGE8cc	ARRAY8	FCMPLE16	—		FPADD16	FZERO	FAND	—	
	1	EDGE8N	—	—	FMUL 8x16	—	FPADD16S	FZEROS	FANDS	SIAM	
	2	EDGE8Lcc	ARRAY16	FCMPNE16	—	—	FPADD32	FNOR	FXNOR	—	
	3	EDGE8LN	—	—	FMUL 8x16AU	—	FPADD32S	FNORS	FXNORS	—	
	4	EDGE16cc	ARRAY32	FCMPLE32	—		FPSUB16	FANDNOT2	FSRC1	—	
	5	EDGE16N	—	—	FMUL 8x16AL	—	FPSUB16S	FANDNOT2S	FSRC1S	—	
	6	EDGE16Lcc	—	FCMPNE32	FMUL 8SUx16	—	FPSUB32	FNOT2	FORNOT2	—	
	7	EDGE16LN	—	—	FMUL 8ULx16	—	FPSUB32S	FNOT2S	FORNOT2S	—	
opf {3:0}	8	EDGE32cc	ALIGN ADDRESS	FCMPGT16	FMULD 8SUx16	FALIGN DATA	—	FANDNOT1	FSRC2	—	*Reserved*
	9	EDGE32N	BMASK	—	FMULD 8ULx16	—	—	FANDNOT1S	FSRC2S	—	
	A	EDGE32Lcc	ALIGNADDRESS _LITTLE	FCMPEQ16	FPACK32	—	—	FNOT1	FORNOT1	—	
	B	EDGE32LN	—	—	FPACK16	FPMERGE	—	FNOT1S	FORNOT1S	—	
	C	—	—	FCMPGT32	—	BSHUFFLE	—	FXOR	FOR	—	
	D	—	—	—	FPACKFIX	FEXPAND	—	FXORS	FORS	—	
	E	—	—	FCMPEQ32	PDIST	—	—	FNAND	FONE	—	
	F	—	—	—	—	—	—	FNANDS	FONES	—	

TABLE A-14 opf{8:0} for VIS opcodes (op = 10_2, op3 = 36_{16}) *(3 of 3)*

opf {3:0}	opf {8:4}: 10	11	12	13	14	15	16	17	18–1F
0	—	—	—	—	—	—	—	—	
1	—	—	—	—	—	—		—	
2	—	—	—	—	—	—		—	
3	—	—	—	—	—	—		—	
4	—	—	—	—	—	—		—	
5	—	—	—	—	—	—		—	
6	—	—	—	—	—	—		—	
7	—	—	—	—	—	—		—	
8	—	—	—	—	—	—		—	*Reserved*
9	—	—	—	—	—	—			
A	—	—	—	—	—	—			
B	—	—	—	—	—	—			
C	—	—	—	—	—	—			
D	—	—	—	—	—	—			
E	—	—	—	—	—	—			
F	—	—	—	—	—	—			

TABLE A-13 op5{3:0} (op = 10_2, op3 = 37_{16} = FMAf

		op5{1:0}			
		0	1	2	3
op5{3:2}	0	—	FMADDs	FMADDd	—
	1	—	FMSUBs	FMSUBd	—
	2	—	FNMSUBs	FNMSUBd	—
	3	—	FNMADDs	FNMADDd	—

Note: This chapter is undergoing final review; please check back later for a copy of UltraSPARC Architecture 2007 containing the final version of this chapter.

Implementation Dependencies

This appendix summarizes implementation dependencies in the SPARC V9 standard. In SPARC V9, the notation "**IMPL. DEP. #*nn*:**" identifies the definition of an implementation dependency; the notation "(impl. dep. #*nn*)" identifies a reference to an implementation dependency. These dependencies are described by their number *nn* in TABLE B-1 on page 397.

The appendix contains these sections:

B.1 Definition of an Implementation Dependency

The SPARC V9 architecture is a *model* that specifies unambiguously the behavior observed by *software* on SPARC V9 systems. Therefore, it does not necessarily describe the operation of the *hardware* of any actual implementation.

An implementation is *not* required to execute every instruction in hardware. An attempt to execute a SPARC V9 instruction that is not implemented in hardware generates a trap. Whether an instruction is implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.

The two levels of SPARC V9 compliance are described in *UltraSPARC Architecture 2007 Compliance with SPARC V9 Architecture* on page 16.

Some elements of the architecture are defined to be implementation dependent. These elements include certain registers and operations that may vary from implementation to implementation; they are explicitly identified as such in this appendix.

Implementation elements (such as instructions or registers) that appear in an implementation but are not defined in this document (or its updates) are not considered to be SPARC V9 elements of that implementation.

B.2 Hardware Characteristics

Hardware characteristics that do not affect the behavior observed by software on SPARC V9 systems are not considered architectural implementation dependencies. A hardware characteristic may be relevant to the user system design (for example, the speed of execution of an instruction) or may be transparent to the user (for example, the method used for achieving cache consistency). The SPARC International document, *Implementation Characteristics of Current SPARC V9-based Products, Revision 9.x*, provides a useful list of these hardware characteristics, along with the list of implementation-dependent design features of SPARC V9-compliant implementations.

In general, hardware characteristics deal with

- Instruction execution speed
- Whether instructions are implemented in hardware
- The nature and degree of concurrency of the various hardware units constituting a SPARC V9 implementation

B.3 Implementation Dependency Categories

Many of the implementation dependencies can be grouped into four categories, abbreviated by their first letters throughout this appendix:

- **Value (v)**
  The semantics of an architectural feature are well defined, except that a value associated with the feature may differ across implementations. A typical example is the number of implemented register windows (impl. dep. #2-V8).
- **Assigned Value (a)**
  The semantics of an architectural feature are well defined, except that a value associated with the feature may differ across implementations and the actual value is assigned by SPARC International. Typical examples are the impl field of the Version register (VER) (impl. dep. #13-V8) and the FSR.ver field (impl. dep. #19-V8).
- **Functional Choice (f)**
  The SPARC V9 architecture allows implementors to choose among several possible semantics related to an architectural function. A typical example is the treatment of a catastrophic error exception, which may cause either a deferred or a disrupting trap (impl. dep. #31-V8-Cs10).
- **Total Unit (t)**
  The existence of the architectural unit or function is recognized, but details are left to each implementation. Examples include the handling of I/O registers (impl. dep. #7-V8) and some alternate address spaces (impl. dep. #29-V8).

B.4 List of Implementation Dependencies

TABLE B-1 provides a complete list of the SPARC V9 implementation dependencies. The Page column lists the page for the context in which the dependency is defined; bold face indicates the main page on which the implementation dependency is described.

TABLE B-1 SPARC V9 Implementation Dependencies *(1 of 7)*

Nbr	Category	Description	Page
1-V8	f	**Software emulation of instructions** Whether an instruction complies with UltraSPARC Architecture 2007 by being implemented directly by hardware, simulated by software, or emulated by firmware is implementation dependent.	**16**
2-V8	v	**Number of IU registers** An UltraSPARC Architecture implementation may contain from 72 to 640 general-purpose 64-bit R registers. This corresponds to a grouping of the registers into *MAXPGL* + 1 sets of global R registers plus a circular stack of *N_REG_WINDOWS* sets of 16 registers each, known as register windows. The number of register windows present (*N_REG_WINDOWS*) is implementation dependent, within the range of 3 to 32 (inclusive).	**17**, 34
3-V8	f	**Incorrect IEEE Std 754-1985 results** 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.	**86**
4, 5		*Reserved.*	
6-V8	f	**I/O registers privileged status** Whether I/O registers can be accessed by nonprivileged code is implementation dependent.	**19**
7-V8	t	**I/O register definitions** The contents and addresses of I/O registers are implementation dependent.	**19**
8-V8-Cs20	t	**RDasr/WRasr target registers** Ancillary state registers (ASRs) in the range 0–27 that are not defined in UltraSPARC Architecture 2007 are reserved for future architectural use. ASRs in the range 28–31 are available to be used for implementation-dependent purposes.	**20**, 48, 225, 285
9-V8-Cs20	f	**RDasr/WRasr privileged status** The privilege level required to execute each of the implementation-dependent read/write ancillary state register instructions (for ASRs 28–31) is implementation dependent.	**20**, 48, 225, 285
10-V8–12-V8		*Reserved.*	
13-V8	a	(this implementation dependency applies to execution modes with greater privileges)	
14-V8–15-V8		*Reserved.*	
16-V8-Cu3		*Reserved.*	
17-V8		*Reserved.*	

TABLE B-1 SPARC V9 Implementation Dependencies *(2 of 7)*

Nbr	Category	Description	Page
18-V8-Ms10	f	**Nonstandard IEEE 754-1985 results** When FSR.ns = 1, the FPU produces implementation-dependent results that may not correspond to IEEE Standard 754-1985.	
		a: When FSR.ns = 1 and a floating-point *source operand* is subnormal, an implementation may treat the subnormal operand as if it were a floating-point zero value of the same sign. The cases in which this replacement is performed are implementation dependent. However, if it occurs, (1) it should *not* apply to FABS, FMOV, or FNEG instructions and (2) FADD, FSUB, and FCMP should give identical treatment to subnormal source operands. Treating a subnormal source operand as zero may generate an IEEE 754 floating-point “inexact”, “division by zero”, or “invalid” condition (see *Current Exception (cexc)* on page 46). Whether the generated condition(s) trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem.	**294**
		b: When a floating-point operation generates a subnormal *result* value, an UltraSPARC Architecture implementation may either write the result as a subnormal value or replace the subnormal result by a floating-point zero value of the same sign and generate IEEE 754 floating-point “inexact” and “underflow” conditions. Whether these generated conditions trigger an *fp_exception_ieee_754* exception or not depends on the setting of FSR.tem.	**294**
		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.	**294**
19-V8	a	**FPU version, FSR.ver** Bits 19:17 of the FSR, FSR.ver, identify one or more implementations of the FPU architecture.	**43**
20-V8–21-V8		*Reserved.*	
22-V8	f	**FPU tem, cexc, and aexc** An UltraSPARC Architecture implementation implements the tem, cexc, and aexc fields in hardware, conformant to IEEE Std 754-1985.	48
23-V8		*Reserved.*	
24-V8		*Reserved.*	
25-V8	f	**RDPR of FQ with nonexistent FQ** An UltraSPARC Architecture implementation does not contain a floating-point queue (FQ). Therefore, FSR.ftt = 4 (sequence_error) does not occur, and an attempt to read the FQ with the RDPR instruction causes an *illegal_instruction* exception.	45, 229
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 UltraSPARC Architecture. Others remain implementation dependent in the UltraSPARC Architecture. See *ASI Assignments* on page 322 and *Block Load and Store ASIs* on page 333 for details.	**78**
30-V8-Cu3	f	**ASI address decoding** In SPARC V9, an implementation could choose to decode only a subset of the 8-bit ASI specifier. In UltraSPARC Architecture implementations, all 8 bits of each ASI specifier must be decoded. Refer to Chapter 10, *Address Space Identifiers (ASIs)*, of this specification for details.	78

TABLE B-1 SPARC V9 Implementation Dependencies *(3 of 7)*

Nbr	Category	Description	Page
31-V8-Cs10	f	This implementation dependency is no longer used in the UltraSPARC Architecture, since “catastrophic” errors are now handled using normal error-reporting mechanisms.	—
32-V8-Ms10	t	**Restartable deferred traps** Whether any restartable deferred traps (and associated deferred-trap queues) are present is implementation dependent.	345
33-V8-Cs10	f	**Trap precision** In an UltraSPARC Architecture implementation, all exceptions that occur as the result of program execution are precise.	347
34-V8	f	**Interrupt clearing** **a**: The method by which an interrupt is removed is now defined in the UltraSPARC Architecture (see *Clearing the Software Interrupt Register* on page 366). **b**: How quickly a virtual processor responds to an interrupt request, like all timing-related issues, is implementation dependent.	**366**
35-V8-Cs20	t	**Implementation-dependent traps** Trap type (TT) values 060_{16}–$07F_{16}$ were reserved for *implementation_dependent_exception_n* exceptions in SPARC V9 but are now all defined as standard UltraSPARC Architecture exceptions.	**349**
36-V8	f	**Trap priorities** The relative priorities of traps defined in the UltraSPARC Architecture are fixed. However, the absolute priorities of those traps are implementation dependent (because a future version of the architecture may define new traps). The priorities (both absolute and relative) of any new traps are implementation dependent.	**356**
41-V8		*Reserved.*	
42-V8-Cs10	t, f, v	**FLUSH instruction** FLUSH is implemented in hardware in all UltraSPARC Architecture 2007 implementations, so never causes a trap as an unimplemented instruction.	
43-V8		*Reserved.*	
44-V8-Cs10	f	**Data access FPU trap** **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.	**182**, **199** **185**
45-V8–46-V8		*Reserved.*	
47-V8-Cs20	t	**RDasr** 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	**226**
48-V8-Cs20	t	**WRasr** 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	**286**

TABLE B-1 SPARC V9 Implementation Dependencies *(4 of 7)*

Nbr	Category	Description	Page
49-V8–54-V8		*Reserved.*	
55-V8-Cs10	f	**Tininess detection** In SPARC V9, it is implementation-dependent whether "tininess" (an IEEE 754 term) is detected before or after rounding. In all UltraSPARC Architecture implementations, tininess is detected before rounding.	48
56–100		*Reserved.*	
101-V9-CS10	v	**Maximum trap level (*MAXPTL*)** The architectural parameter *MAXPTL* is a constant for each implementation; its legal values are from 2 to 6 (supporting from 2 to 6 levels of saved trap state). In a typical implementation *MAXPTL* = *MAXPGL* (see impl. dep. #401-S10). Architecturally, *MAXPTL* must be ≥ 2.	**68**, 70
102-V9	f	**Clean windows trap** An implementation may choose either to implement automatic "cleaning" of register windows in hardware or to generate a *clean_window* trap, when needed, for window(s) to be cleaned by software.	**358**
103-V9-Ms10	f	**Prefetch instructions** The following aspects of the PREFETCH and PREFETCHA instructions are implementation dependent:	
		a: the attributes of the block of memory prefetched: its size (minimum = 64 bytes) and its alignment (minimum = 64-byte alignment)	**220**
		b: whether each defined prefetch variant is implemented (1) as a NOP, (2) with its full semantics, or (3) with common-case prefetching semantics	**220**, 222
		c: whether and how variants 16, 18, 19 and 24–31 are implemented; if not implemented, a variant must execute as a NOP	**224**C
		The following aspects of the PREFETCH and PREFETCHA instructions used to be (but are no longer) implementation dependent:	
		d: while in nonprivileged mode (PSTATE.priv = 0), an attempt to reference an ASI in the range $0_{16}..7F_{16}$ by a PREFETCHA instruction executes as a NOP; specifically, it does not cause a *privileged_action* exception.	—
		e: PREFETCH and PREFETCHA have no observable effect in privileged code	—
		g: while in privileged mode (PSTATE.priv = 1), an attempt to reference an ASI in the range $30_{16}..7F_{16}$ by a PREFETCHA instruction executes as a NOP (specifically, it does not cause a *privileged_action* exception)	—
105-V9	f	**TICK register**	**52**
		a: If an accurate count cannot always be returned when TICK is read, any inaccuracy should be small, bounded, and documented.	
		b: An implementation may implement fewer than 63 bits in TICK.counter; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any upper bits not implemented must read as 0.	
106-V9cS10	f	**IMPDEP2A instructions** The IMPDEP2A instructions were defined to be completely implementation dependent in SPARC V9. The opcodes that have not been used in this space are now just documented as reserved opcodes.	
107-V9	f	**Unimplemented LDTW(A) trap**	
		a: It is implementation dependent whether LDTW is implemented in hardware. If not, an attempt to execute an LDTW instruction will cause an *unimplemented_LDTW* exception.	**192** **194**
		b: It is implementation dependent whether LDTWA is implemented in hardware. If not, an attempt to execute an LDTWA instruction will cause an *unimplemented_LDTW* exception.	

TABLE B-1 SPARC V9 Implementation Dependencies *(5 of 7)*

Nbr	Category	Description	Page
108-V9	f	**Unimplemented STTW(A) trap**	
		a: It is implementation dependent whether STTW is implemented in hardware. If not, an attempt to execute an STTW instruction will cause an *unimplemented_STTW* exception.	**265** **267**
		b: It is implementation dependent whether STDA is implemented in hardware. If not, an attempt to execute an STTWA instruction will cause an *unimplemented_STTW* exception.	
109-V9-Cs10	f	***LDDF(A)_mem_address_not_aligned***	
		a: LDDF requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid (i = 1 or instruction bits 12:5 = 0) LDDF instruction may cause an *LDDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the LDDF instruction and return. (In an UltraSPARC Architecture processor, the *LDDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDDF instruction)	73, 73, 181, 361
		b: LDDFA requires only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid (i = 1 or instruction bits 12:5 = 0) LDDFA instruction may cause an *LDDF_mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the LDDFA instruction and return. (In an UltraSPARC Architecture processor, the *LDDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDDFA instruction)	183
110-V9-Cs10	f	***STDF(A)_mem_address_not_aligned***	
		a: STDF requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid (i = 1 or instruction bits 12:5 = 0) STDF instruction may cause an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STDF instruction and return. (In an UltraSPARC Architecture processor, the *STDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STDF instruction)	73, **253**, 362
		b: STDFA requires only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, an attempt to execute a valid (i = 1 or instruction bits 12:5 = 0) STDFA instruction may cause an *STDF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STDFA instruction and return. (In an UltraSPARC Architecture processor, the *STDF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STDFA instruction)	255

TABLE B-1 SPARC V9 Implementation Dependencies *(6 of 7)*

Nbr	Category	Description	Page
111-V9-Cs10	f	***LDQF(A)_mem_address_not_aligned***	
		a: LDQF requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an LDQF instruction may cause an *LDQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the LDQF instruction and return. (In an UltraSPARC Architecture processor, the *LDQF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDQF instruction) (this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the LDQF instruction in hardware)	74, 73, 181, 362
		b: LDQFA requires only word alignment. However, if the effective address is word-aligned but not quadword-aligned, an attempt to execute an LDQFA instruction may cause an *LDQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the LDQF instruction and return. (In an UltraSPARC Architecture processor, the *LDQF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the LDQFA instruction) (this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the LDQFA instruction in hardware)	183
112-V9-Cs10	f	***STQF(A)_mem_address_not_aligned***	
		a: STQF requires only word alignment in memory. However, if the effective address is word aligned but not quadword aligned, an attempt to execute an STQF instruction may cause an *STQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STQF instruction and return. (In an UltraSPARC Architecture processor, the *STQF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STQF instruction) (this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the STQF instruction in hardware)	74, 253, 362
		b: STQFA requires only word alignment in memory. However, if the effective address is word aligned but not quadword aligned, an attempt to execute an STQFA instruction may cause an *STQF_mem_address_not_aligned* exception. In this case, the trap handler software must emulate the STQFA instruction and return. (In an UltraSPARC Architecture processor, the *STQF_mem_address_not_aligned* exception occurs in this case and trap handler software emulates the STQFA instruction) (this exception does not occur in hardware on UltraSPARC Architecture 2007 implementations, because they do not implement the STQFA instruction in hardware)	255
113-V9-Ms10	f	**Implemented memory models** Whether memory models represented by PSTATE.mm = 10_2 or 11_2 are supported in an UltraSPARC Architecture processor is implementation dependent. If the 10_2 model is supported, then when PSTATE.mm = 10_2 the implementation must correctly execute software that adheres to the RMO model described in *The SPARC Architecture Manual-Version 9*. If the 11_2 model is supported, its definition is implementation dependent.	**66**, 313
118-V9	f	**Identifying I/O locations** The manner in which I/O locations are identified is implementation dependent.	**307**
119-Ms10	f	**Unimplemented values for PSTATE.mm** The effect of an attempt to write an unsupported memory model designation into PSTATE.mm is implementation dependent; however, it should never result in a value of PSTATE.mm value greater than the one that was written. In the case of an UltraSPARC Architecture implementation that only supports the TSO memory model, PSTATE.mm always reads as zero and attempts to write to it are ignored.	**66**, **314**

TABLE B-1 SPARC V9 Implementation Dependencies *(7 of 7)*

Nbr	Category	Description	Page
120-V9	f	**Coherence and atomicity of memory operations** The coherence and atomicity of memory operations between virtual processors and I/O DMA memory accesses are implementation dependent.	**307**
121-V9	f	**Implementation-dependent memory model** An implementation may choose to identify certain addresses and use an implementation-dependent memory model for references to them.	**307**
122-V9	f	**FLUSH latency** The latency between the execution of FLUSH on one virtual processor and the point at which the modified instructions have replaced outdated instructions in a multiprocessor is implementation dependent.	133, **319**
123-V9	f	**Input/output (I/O) semantics** The semantic effect of accessing I/O registers is implementation dependent.	**19**
124-V9	v	**Implicit ASI when TL > 0** In SPARC V9, when TL > 0, the implicit ASI for instruction fetches, loads, and stores is implementation dependent. In all UltraSPARC Architecture implementations, when TL > 0, the implicit ASI for instruction fetches is ASI_NUCLEUS; loads and stores will use ASI_NUCLEUS if PSTATE.cle = 0 or ASI_NUCLEUS_LITTLE if PSTATE.cle = 1.	309
125-V9-Cs10	f	**Address masking** (1) When PSTATE.am = 1, only the less-significant 32 bits of the PC register are stored in the specified destination register(s) in CALL, JMPL, and RDPC instructions, while the more-significant 32 bits of the destination registers(s) are set to 0. ((2) When PSTATE.am = 1, during a trap, only the less-significant 32 bits of the PC and NPC are stored (respectively) to TPC[TL] and TNPC[TL]; the more-significant 32 bits of TPC[TL] and TNPC[TL] are set to 0.	**67**, **67**, 111, 174, 226, 357
126-V9-Ms10		**Register Windows State registers width** Privileged registers CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN contain values in the range 0 to *N_REG_WINDOWS* – 1. An attempt to write a value greater than *N_REG_WINDOWS* – 1 to any of these registers causes an implementation-dependent value between 0 and *N_REG_WINDOWS* – 1 (inclusive) to be written to the register. Furthermore, an attempt to write a value greater than *N_REG_WINDOWS* – 2 violates the register window state definition in *Register Window Management Instructions* on page 83. Although the width of each of these five registers is architecturally 5 bits, the width is implementation dependent and shall be between $\lceil \log_2(N\_REG\_WINDOWS) \rceil$ and 5 bits, inclusive. If fewer than 5 bits are implemented, the unimplemented upper bits shall read as 0 and writes to them shall have no effect. All five registers should have the same width. For UltraSPARC Architecture 2007 processors, = 8. Therefore, each register window state register is implemented with 3 bits, the maximum value for CWP and CLEANWIN is 7, and the maximum value for CANSAVE, CANRESTORE, and OTHERWIN is 6. When these registers are written by the WRPR instruction, bits 63:3 of the data written are ignored.	**58**
127–199		*Reserved.*	—

TABLE B-2 provides a list of implementation dependencies that, in addition to those in TABLE B-1, apply to UltraSPARC Architecture processors. Bold face indicates the main page on which the implementation dependency is described. See Appendix C in the Extensions Documents for further information.

TABLE B-2 UltraSPARC Architecture Implementation Dependencies *(1 of 5)*

Nbr	Description	Page
200–201	*Reserved.*	—
203-U3-Cs10	**Dispatch Control register (DCR) bits 13:6 and 1** *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	
204-U3-CS10	**DCR bits 5:3 and 0** *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	
205-U3-Cs10	**Instruction Trap Register** *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	
206-U3-Cs10	**SHUTDOWN instruction**	
208-U3	**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.	—
209-U3	**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.	—
211-U3	**Error logging registers' information** The information that the error logging registers preserves beyond the reset induced by an ERROR signal is implementation dependent.	—
212-U3-Cs10	***Trap with fatal error*** *This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
213-U3	**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.	—
226-U3	**TTE support for cv bit** Whether the cv bit is supported in TTE is implementation dependent in the UltraSPARC Architecture. When the cv bit in TTE is not provided and the implementation has virtually indexed caches, the implementation should support hardware unaliasing for the caches.	**375**
227-U3	**TSB number of entries** The maximum number of entries in a TSB is implementation dependent in the UltraSPARC Architecture (to a maximum of 16 million).	**377**
228-U3-Cs10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
229-U3-Cs10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.***TSB Base address generation** Whether the implementation generates the TSB Base address by **exclusive-OR**ing the TSB Base register and a TSB register or by taking the tsb_base field directly from a TSB register is implementation dependent in UltraSPARC Architecture. This implementation dependency existed for UltraSPARC III/IV, only to maintain compatibility with the TLB miss handling software of UltraSPARC I/II.	—
230	*Reserved.*	—
230-U3-Cs20	*This implementation dependency no longer applies, in UltraSPARC Architecture 2007*	—
232-U3-Cs10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
233-U3-Cs10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
235-U3-Cs10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—

TABLE B-2 UltraSPARC Architecture Implementation Dependencies *(2 of 5)*

Nbr	Description	Page
236-U3-Cs10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.***t**	—
239-U3-Cs10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	
240-U3-Cs10	*Reserved.*	—
243-U3	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
244-U3-Cs10	**Data Watchpoint** Reliability Data Watchpoint traps are completely implementation-dependent in UltraSPARC Architecture processors.	—
245-U3-Cs10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
248-U3	**Conditions for *fp_exception_other* with unfinished_FPop** The conditions under which an *fp_exception_other* exception with floating-point trap type of unfinished_FPop can occur are implementation dependent. An implementation may cause *fp_exception_other* with unfinished_FPop under a different (but specified) set of conditions.	**45**
249-U3-Cs10	**Data Watchpoint for Partial Store Instruction** For an STPARTIAL instruction, the following aspects of data watchpoints are implementation dependent: (a) whether data watchpoint logic examines the byte store mask in R[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.	**262**
250-U3-Cs10	*Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2007.*	—
251	*Reserved.*	
252-U3-Cs10	*Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
253-U3-Cs10	*Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
255-U3-Cs10	**LDDFA with ASI $E0_{16}$ or $E1_{16}$ and misaligned destination register number** If an LDDFA opcode is used with an ASI of $E0_{16}$ or $E1_{16}$ (Block Store Commit ASI, an illegal combination with LDDFA) and a destination register number rd is specified which is not a multiple of 8 ("misaligned" rd), an UltraSPARC Architecture virtual processor generates an *illegal_instruction* exception.	185
256-U3	**LDDFA with ASI $E0_{16}$ or $E1_{16}$ and misaligned memory address** If an LDDFA opcode is used with an ASI of $E0_{16}$ or $E1_{16}$ (Block Store Commit ASI, an illegal combination with LDDFA) and a memory address is specified with less than 64-byte alignment, the virtual processor generates an exception. It is implementation dependent whether the exception generated is *DAE_invalid_asi*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned*.	**185**
257-U3	**LDDFA with ASI $C0_{16}$–$C5_{16}$ or $C8_{16}$–CD_{16} and misaligned memory address** If an LDDFA opcode is used with an ASI of $C0_{16}$–$C5_{16}$ or $C8_{16}$–CD_{16} (Partial Store ASIs, which are an illegal combination with LDDFA) and a memory address is specified with less than 8-byte alignment, the virtual processor generates n exception. It is implementation dependent whether the exception generated is *DAE_invalid_asi*, *mem_address_not_aligned*, or *LDDF_mem_address_not_aligned*.	**185**
259–299	*Reserved.*	—

TABLE B-2 UltraSPARC Architecture Implementation Dependencies *(3 of 5)*

Nbr	Description	Page
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.	195
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.	268
302-U4-Cs10	**Scratchpad registers** An UltraSPARC Architecture processor includes eight privileged Scratchpad registers (64 bits each, read/write accessible).	334
303-U4-CS10	*This implementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
305-U4-Cs10	*Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
306-U4-Cs10	**Trap type generated upon attempted access to noncacheable page with LDTXA** When an LDTXA instruction attempts access from an address that is not mapped to cacheable memory space, a *DAE_nc_page* exception is generated.	198
307-U4-Cs10	*Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
308-U3-Cs10	*Thisimplementation dependency no longer applies, as of UltraSPARC Architecture 2005.*	—
309-U4-Cs10	*Reserved.*	—
310-U4	**Large page sizes** Which, if any, of the following optional page sizes are supported by the MMU in an UltraSPARC Architecture implementation is implementation dependent: 512 KBytes, 32 MBytes, 256 MBytes, 2 GBytes, and 16 GBytes.	**369**
311–319	*Reserved.*	
327–399	*Reserved*	
400-S10	**Global Level register (GL) implementation** Although GL is defined as a 4-bit register, an implementation may implement any subset of those bits sufficient to encode the values from 0 to *MAXPGL* for that implementation. If any bits of GL are not implemented, they read as zero and writes to them are ignored.	**70**
401-S10	**Maximum Global Level (*MAXPGL*)** The architectural parameter *MAXPGL* is a constant for each implementation; its legal values are from 2 to 15 (supporting from 3 to 16 sets of global registers). In a typical implementation *MAXPGL* = *MAXPTL* (see impl. dep. #101-V9-CS10). Architecturally, *MAXPTL* must be ≥ 2.	68, **70**
403-S10	**Setting of "dirty" bits in FPRS** A "dirty" bit (du or dl) in the FPRS register must be set to '1' if any of its corresponding F registers is actually modified. If an instruction that normally writes to an F register is executed and causes an *fp_disabled* exception, FPRS.du and FPRS.dl are unchanged. Beyond that, the specific conditions under which a dirty bit is set are implementation dependent.	53, 53
404-S10	**Scratchpad registers 4 through 7** The degree to which Scratchpad registers 4–7 are accessible to privileged software is implementation dependent. Each may be (1) fully accessible, (2) accessible, with access much slower than to scratchpad register 0–3, or (3) inaccessible (cause a *DAE_invalid_asi* exception).	**334**

TABLE B-2 UltraSPARC Architecture Implementation Dependencies *(4 of 5)*

Nbr	Description	Page
405-S10	**Virtual address range** An UltraSPARC Architecture implementation may support a full 64-bit virtual address space or a more limited range of virtual addresses. In an implementation that does not support a full 64-bit virtual address space, the supported range of virtual addresses is restricted to two equal-sized ranges at the extreme upper and lower ends of 64-bit addresses; that is, for *n*-bit virtual addresses, the valid address ranges are 0 to $2^{n-1} - 1$ and $2^{64} - 2^{n-1}$ to $2^{64} - 1$. (see also impl. dep. #451-S20)	**18**
409-S10	**FLUSH instruction and memory consistency** 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.	**134**
410-S10	**Block Load behavior** The following aspects of the behavior of block load (LDBLOCKF) instructions are implementation dependent: • What memory ordering model is used by LDBLOCKF (LDBLOCKF is not required to follow TSO memory ordering) • Whether LDBLOCKF follows memory ordering with respect to stores (including block stores), including whether the virtual processor detects read-after-write and write-after-read hazards to overlapping addresses • Whether LDBLOCKF appears to execute out of order, or follow LoadLoad ordering (with respect to older loads, younger loads, and other LDBLOCKFs) • Whether LDBLOCKF follows register-dependency interlocks, as do ordinary load instructions •	**179**
	• Whether the MMU ignores the side-effect bit (TTE.e) for LDBLOCKF accesses (in which case, LDBLOCKFs behave as if TTE.e = 0)	307
	• Whether *VA_watchpoint* exceptions are recognized on accesses to all 64 bytes of a LDBLOCKF (the recommended behavior), or only on accesses to the first eight bytes	179, 180
411-S10	**Block Store behavior** The following aspects of the behavior of block store (STBLOCKF) instructions are implementation dependent: • The memory ordering model that STBLOCKF follows (other than as constrained by the rules outlined on page 251). • Whether *VA_watchpoint* exceptions are recognized on accesses to all 64 bytes of a STBLOCKF (the recommended behavior), or only on accesses to the first eight bytes. • Whether STBLOCKFs to non-cacheable pages execute in strict program order or not. If not, a STBLOCKF to a non-cacheable page causes a *DAE_nc_page* exception. • Whether STBLOCKF follows register dependency interlocks (as ordinary stores do). • Whether a non-Commit STBLOCKF forces the data to be written to memory and invalidates copies in all caches present (as the Commit variants of STBLOCKF do).	**251**, 252
	• Whether the MMU ignores the side-effect bit (TTE.e) for STBLOCKF accesses (in which case, STBLOCKFs behave as if TTE.e = 0)	307
	• Any other restrictions on the behavior of STBLOCKF, as described in implementation-specific documentation.	
412-S10	**MEMBAR behavior** An UltraSPARC Architecture implementation may define the operation of each MEMBAR variant in any manner that provides the required semantics.	**202**

TABLE B-2 UltraSPARC Architecture Implementation Dependencies *(5 of 5)*

Nbr	Description	Page
413-S10	**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.	**198**
414	*Reserved.*	—
415-S10	**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.	**381**
417-S10	**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 (zeroes) the more-significant 32 bits of the values it places into PC and NPC.	68, **114233**
442-S10	**STICK register** **a**: If an accurate count cannot always be returned when STICK is read, any inaccuracy should be small, bounded, and documented. **b**: An implementation may implement fewer than 63 bits in STICK.counter; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any high-order bits not implemented must read as 0.	**57**
444–449	*Reserved for UltraSPARC Architecture 2005*	
450-S20	**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.	**65**,358
451-S20	**Width of Virtual Addresses supported** The width of the virtual address supported is implementation dependent. If fewer than 64 bits are supported, the unsupported bits must have the same value as the most significant supported bit. For example, if the model supports 48 virtual address bits, then bits 63:48 must have the same value as bit 47. (see also impl. dep. #405-S10)	**369**,
452-S20	**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.	**369**
453-S20	**Unified vs. Split Instruction and Data MMUs** It is implementation dependent whether there is a unified MMU (UMMU) or a separate IMMU (for instruction accesses) and DMMU (for data accesses). The UltraSPARC Architecture supports both configurations.	**369**
453-S20	**Unified vs. Split Instruction and Data MMUs** It is implementation dependent whether there is a unified MMU (UMMU) or a separate IMMU (for instruction accesses) and DMMU (for data accesses). The UltraSPARC Architecture supports both configurations.	**369**
454-499	*Reserved for UltraSPARC Architecture 2007*	
500 and up	*Reserved for future use*	

APPENDIX **C**

Assembly Language Syntax

This appendix supports Chapter 7, *Instructions*. Each instruction description in Chapter 7 includes a table that describes the suggested assembly language format for that instruction. This appendix describes the notation used in those assembly language syntax descriptions and lists some synthetic instructions provided by UltraSPARC Architecture assemblers for the convenience of assembly language programmers.

The appendix contains these sections:

- **Notation Used** on page 409.
- **Syntax Design** on page 414.
- **Synthetic Instructions** on page 414.

C.1 Notation Used

The notations defined here are also used in the assembly language syntax descriptions in Chapter 7, *Instructions*.

Items in `typewriter font` are literals to be written exactly as they appear. Items in *italic font* are metasymbols that are to be replaced by numeric or symbolic values in actual SPARC V9 assembly language code. For example, "*imm_asi*" would be replaced by a number in the range 0 to 255 (the value of the *imm_asi* bits in the binary instruction) or by a symbol bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, reg_{rs2} is a *reg* (register name) whose binary value will be placed in the `rs2` field of the resulting instruction.

C.1.1 Register Names

reg. A reg is an intveger 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`, `%g`*n*, `%o`*n*, `%l`*n*, and `%i`*n* forms are preferred over `%r`*n*.

reg$_{rs1}$ (rs1 field)
reg$_{rs2}$ (rs2 field)
reg$_{rd}$ (rd field)

freg. An *freg* is a floating-point register name. It may have the following values:

`%f0`, `%f1`, `%f2`, ... `%f31`

`%f32`, `%f34`, ... `%f60`, `%f62` (even-numbered only, from `%f32` to `%f62`)

`%d0`, `%d2`, `%d4`, ... `%d60`, `%d62` (`%d`*n*, where *n* **mod** 2 = 0, only)

`%q0`, `%q4`, `%q8`, ... `%q56`, `%q60` (`%q`*n*, where *n* **mod** 4 = 0, only)

See *Floating-Point Registers* on page 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 `fcc`*n* specifies a set of floating-point condition codes. It can have any of the following values:

`%fcc0`
`%fcc1`
`%fcc2`
`%fcc3`

C.1.2 Special Symbol Names

Certain special symbols appear in the syntax table in typewriter font. They must be written exactly as they are shown, including the leading percent sign (%).

The symbol names and the registers or operators to which they refer are as follows:

`%asi`	Address Space Identifier (ASI) register
`%canrestore`	Restorable Windows register
`%cansave`	Savable Windows register
`%ccr`	Condition Codes register
`%cleanwin`	Clean Windows register
`%cwp`	Current Window Pointer (CWP) register

%fprs	Floating-Point Registers State (FPRS) register
%fsr	Floating-Point State register
%gsr	General Status Register (GSR)
%otherwin	Other Windows (OTHERWIN) register
%pc	Program Counter (PC) register
%pil	Processor Interrupt Level register
%pstate	Processor State register
%softint	Soft Interrupt register
%softint_clr	Soft Interrupt register (clear selected bits)
%softint_set	Soft Interrupt register (set selected bits)
%stick †	System Timer (STICK) register
%stick_cmpr †	System Timer Compare (STICK_CMPR) register
%tba	Trap Base Address (TBA) register
%tick	Cycle count (TICK) register
%tick_cmpr	Timer Compare (TICK_CMPR) register
%tl	Trap Level (TL) register
%tnpc	Trap Next Program Counter (TNPC) register
%tpc	Trap Program Counter (TPC) register
%tstate	Trap State (TSTATE) register
%tt	Trap Type (TT) register
%wstate	Window State register
%y	Y register

† The original assembly language names for `%stick` and `%stick_cmpr` were, respectively, `%sys_tick` and `%sys_tick_cmpr`, which are now deprecated. Over time, assemblers will support the new `%stick` and `%stick_cmpr` names for these registers (which are consistent with `%tick` and %tick_cmpr). In the meantime, some existing assemblers may only recognize the original names.

The following special symbol names are prefix unary operators that perform the functions described, on an argument that is a constant, symbol, or expression that evaluates to a constant offset from a symbol:

%hh	Extracts bits 63:42 (high 22 bits of upper word) of its operand
%hm	Extracts bits 41:32 (low-order 10 bits of upper word) of its operand
%hi or %lm	Extracts bits 31:10 (high-order 22 bits of low-order word) of its operand
%lo	Extracts bits 9:0 (low-order 10 bits) of its operand

For example, the value of "`%lo(`*symbol*`)`" is the least-significant 10 bits of *symbol*.

Certain predefined value names appear in the syntax table in `typewriter font`. They must be written exactly as they are shown, including the leading sharp sign (#). The value names and the constant values to which they are bound are listed in TABLE C-1.

TABLE C-1 Value Names and Values *(1 of 2)*

Value Name in Assembly Language	Value	Comments
for PREFETCH instruction "fcn" field		
#n_reads	0	
#one_read	1	
#n_writes	2	
#one_write	3	
#page	4	
#unified	17 (11_{16})	
#n_reads_strong	20 (14_{16})	

TABLE C-1 Value Names and Values *(2 of 2)*

Value Name in Assembly Language	Value	Comments
`#one_read_strong`	21 (15_{16})	
`#n_writes_strong`	22 (16_{16})	
`#one_write_strong`	23 (17_{16})	
for MEMBAR instruction "mmask" field		
`#LoadLoad`	01_{16}	
`#StoreLoad`	02_{16}	
`#LoadStore`	04_{16}	
for MEMBAR instruction "cmask" field		
`#StoreStore`	08_{16}	
`#Lookaside`	10_{16}	
`#MemIssue`	20_{16}	
`#Sync`	40_{16}	

C.1.3 Values

Some instructions use operand values as follows:

const4	A constant that can be represented in 4 bits
const22	A constant that can be represented in 22 bits
imm_asi	An alternate address space identifier (0–255)
siam_mode	A 3-bit mode value for the SIAM instruction
simm7	A signed immediate constant that can be represented in 7 bits
simm8	A signed immediate constant that can be represented in 8 bits
simm10	A signed immediate constant that can be represented in 10 bits
simm11	A signed immediate constant that can be represented in 11 bits
simm13	A signed immediate constant that can be represented in 13 bits
value	Any 64-bit value
shcnt32	A shift count from 0–31
shcnt64	A shift count from 0–63

C.1.4 Labels

A label is a sequence of characters that comprises alphabetic letters (a–z, A–Z [with upper and lower case distinct]), underscores (_), dollar signs ($), periods (.), and decimal digits (0-9). A label may contain decimal digits, but it may not begin with one. A local label contains digits only.

C.1.5 Other Operand Syntax

Some instructions allow several operand syntaxes, as follows:

reg_plus_imm Can be any of the following:

reg_{rs1} (equivalent to reg_{rs1} + `%g0`)
reg_{rs1} + *simm13*

reg_{rs1} – *simm13*
simm13 (equivalent to `%g0` + *simm13*)
simm13 + reg_{rs1} (equivalent to reg_{rs1} + *simm13*)

address Can be any of the following:

reg_{rs1} (equivalent to reg_{rs1} + `%g0`)
reg_{rs1} + *simm13*
reg_{rs1} – *simm13*
simm13 (equivalent to `%g0` + *simm13*)
simm13 + reg_{rs1} (equivalent to reg_{rs1} + *simm13*)
reg_{rs1} + reg_{rs2}

membar_mask Is the following:

const7 A constant that can be represented in 7 bits. Typically, this is an expression involving the logical OR of some combination of `#Lookaside`, `#MemIssue`, `#Sync`, `#StoreStore`, `#LoadStore`, `#StoreLoad`, and `#LoadLoad` (see TABLE 7-7 and TABLE 7-8 on page 202 for a complete list of mnemonics).

prefetch_fcn (prefetch function) Can be any of the following:

0–31

Predefined constants (the values of which fall in the 0-31 range) useful as *prefetch_fcn* values can be found in TABLE C-1 on page 411.

regaddr (register-only address) Can be any of the following:

reg_{rs1} (equivalent to reg_{rs1} + `%g0`)
reg_{rs1} + reg_{rs2}

reg_or_imm (register or immediate value) Can be either of:

reg_{rs2}
simm13

reg_or_imm5 (register or immediate value) Can be either of:

reg_{rs2}
simm5

reg_or_imm10 (register or immediate value) Can be either of:

reg_{rs2}
simm10

reg_or_imm11 (register or immediate value) Can be either of:

reg_{rs2}
simm11

reg_or_shcnt (register or shift count value) Can be any of:

reg_{rs2}

shcnt32
shcnt64

software_trap_number Can be any of the following:

reg_{rs1}	(equivalent to reg_{rs1} + `%g0`)
reg_{rs1} + reg_{rs2}	
reg_{rs1} + *simm8*	
reg_{rs1} – *simm8*	
simm8	(equivalent to `%g0 +` *simm8*)
simm8 + reg_{rs1}	(equivalent to reg_{rs1} + *simm8*)

The resulting operand value (software trap number) must be in the range 0–255, inclusive.

C.1.6 Comments

Two types of comments are accepted by the SPARC V9 assembler: C-style "`/*...*/`" comments, which may span multiple lines, and "`!...`" comments, which extend from the "`!`" to the end of the line.

C.2 Syntax Design

The SPARC V9 assembly language syntax is designed so that the following statements are true:

- The destination operand (if any) is consistently specified as the last (rightmost) operand in an assembly language instruction.
- A reference to the *contents* of a memory location (for example, in a load, store, or load-store instruction) is always indicated by square brackets ([]); a reference to the *address* of a memory location (such as in a JMPL, CALL, or SETHI) is specified directly, without square brackets.

The follow additional syntax constraints have been adopted for UltraSPARC Architecture:

- Instruction mnemonics should be limited to a maximum of 15 characters.

C.3 Synthetic Instructions

TABLE C-2 describes the mapping of a set of synthetic (or "pseudo") instructions to actual instructions. These synthetic instructions are provided by the SPARC V9 assembler for the convenience of assembly language programmers.

Note: Synthetic instructions should not be confused with "pseudo ops," which typically provide information to the assembler but do not generate instructions. Synthetic instructions always generate instructions; they provide more mnemonic syntax for standard SPARC V9 instructions.

TABLE C-2 Mapping Synthetic to SPARC V9 Instructions *(1 of 3)*

Synthetic Instruction		SPARC V9 Instruction(s)		Comment
`cmp`	reg_{rs1}, *reg_or_imm*	`subcc`	reg_{rs1}, *reg_or_imm*, `%g0`	Compare.
`jmp`	*address*	`jmpl`	*address*, `%g0`	
`call`	*address*	`jmpl`	*address*, `%o7`	

TABLE C-2 Mapping Synthetic to SPARC V9 Instructions *(2 of 3)*

Synthetic Instruction	SPARC V9 Instruction(s)	Comment
iprefetch *label*	bn,a,pt %xcc,*label*	Originally envisioned as an encoding for an "instruction prefetch" operation, but functions as a NOP on all UltraSPARC Architecture implementations. (See PREFETCH function 17 on page 219 for an alternative method of prefetching instructions.)
tst reg_{rs1}	orcc %g0, reg_{rs1}, %g0	Test.
ret	jmpl %i7+8, %g0	Return from subroutine.
retl	jmpl %o7+8, %g0	Return from leaf subroutine.
restore	restore %g0, %g0, %g0	Trivial RESTORE.
save	save %g0, %g0, %g0	Trivial SAVE. **(Warning:** trivial SAVE should only be used in kernel code!)
setuw *value*, reg_{rd}	sethi %hi(*value*), reg_{rd}	(When ((*value*&3FF$_{16}$) == 0).)
	— or —	
	or %g0, *value*, reg_{rd}	(When $0 \le value \le 4095$).
	— or —	
	sethi %hi(*value*), reg_{rd};	(Otherwise)
	or reg_{rd}, %lo(*value*), reg_{rd}	Warning: do not use setuw in the delay slot of a DCTI.
set *value*, reg_{rd}		synonym for setuw.
setsw *value*, reg_{rd}	sethi %hi(*value*), reg_{rd}	(When (value> = 0) and ((value & 3FF$_{16}$) == 0).)
	— or —	
	or %g0, *value*, reg_{rd}	(When $4096 \le value \le 4095$).
	— or —	
	sethi %hi(*value*), reg_{rd}	(Otherwise, if (*value* < 0) and ((*value* & 3FF$_{16}$) = = 0))
	sra reg_{rd}, %g0, reg_{rd}	
	— or —	
	sethi %hi(*value*), reg_{rd};	(Otherwise, if *value* 0)
	or reg_{rd}, %lo(*value*), reg_{rd}	
	— or —	
	sethi %hi(*value*), reg_{rd};	(Otherwise, if *value* < 0)
	or reg_{rd}, %lo(*value*), reg_{rd}	
	sra reg_{rd}, %g0, reg_{rd}	Warning: do not use setsw in the delay slot of a CTI.
setx *value*, *reg*, reg_{rd}	sethi %hh(*value*), *reg*	Create 64-bit constant.
	or *reg*, %hm(*value*), *reg*	("*reg*" is used as a temporary register.)
	sllx *reg*,32,*reg*	
	sethi %hi(*value*), reg_{rd}	Note: setx optimizations are possible but not enumerated here. The worst case is shown. Warning: do not use setx in the delay slot of a CTI.
	or reg_{rd}, *reg*, reg_{rd}	
	or reg_{rd}, %lo(*value*), reg_{rd}	
signx reg_{rs1}, reg_{rd}	sra reg_{rs1}, %g0, reg_{rd}	Sign-extend 32-bit value to 64 bits.
signx reg_{rd}	sra reg_{rd}, %g0, reg_{rd}	

TABLE C-2 Mapping Synthetic to SPARC V9 Instructions *(3 of 3)*

Synthetic Instruction		SPARC V9 Instruction(s)		Comment
not	reg_{rs1}, reg_{rd}	xnor	reg_{rs1}, %g0, reg_{rd}	One's complement.
not	reg_{rd}	xnor	reg_{rd}, %g0, reg_{rd}	One's complement.
neg	reg_{rs2}, reg_{rd}	sub	%g0, reg_{rs2}, reg_{rd}	Two's complement.
neg	reg_{rd}	sub	%g0, reg_{rd}, reg_{rd}	Two's complement.
cas	[reg_{rs1}], reg_{rs2}, reg_{rd}	casa	[reg_{rs1}]#ASI_P, reg_{rs2}, reg_{rd}	Compare and swap.
casl	[reg_{rs1}], reg_{rs2}, reg_{rd}	casa	[reg_{rs1}]#ASI_P_L, reg_{rs2}, reg_{rd}	Compare and swap, little-endian.
casx	[reg_{rs1}], reg_{rs2}, reg_{rd}	casxa	[reg_{rs1}]#ASI_P, reg_{rs2}, reg_{rd}	Compare and swap extended.
casxl	[reg_{rs1}], reg_{rs2}, reg_{rd}	casxa	[reg_{rs1}]#ASI_P_L, reg_{rs2}, reg_{rd}	Compare and swap extended, little-endian.
inc	reg_{rd}	add	reg_{rd}, 1, reg_{rd}	Increment by 1.
inc	*const13*, reg_{rd}	add	reg_{rd}, *const13*, reg_{rd}	Increment by *const13*.
inccc	reg_{rd}	addcc	reg_{rd}, 1, reg_{rd}	Increment by 1; set icc & xcc.
inccc	*const13*, reg_{rd}	addcc	reg_{rd}, *const13*, reg_{rd}	Incr by *const13*; set icc & xcc.
dec	reg_{rd}	sub	reg_{rd}, 1, reg_{rd}	Decrement by 1.
dec	*const13*, reg_{rd}	sub	reg_{rd}, *const13*, reg_{rd}	Decrement by *const13*.
deccc	reg_{rd}	subcc	reg_{rd}, 1, reg_{rd}	Decrement by 1; set icc & xcc.
deccc	*const13*, reg_{rd}	subcc	reg_{rd}, *const13*, reg_{rd}	Decr by *const13*; set icc & xcc.
btst	*reg_or_imm*, reg_{rs1}	andcc	reg_{rs1}, *reg_or_imm*, %g0	Bit test.
bset	*reg_or_imm*, reg_{rd}	or	reg_{rd}, *reg_or_imm*, reg_{rd}	Bit set.
bclr	*reg_or_imm*, reg_{rd}	andn	reg_{rd}, *reg_or_imm*, reg_{rd}	Bit clear.
btog	*reg_or_imm*, reg_{rd}	xor	reg_{rd}, *reg_or_imm*, reg_{rd}	Bit toggle.
clr	reg_{rd}	or	%g0, %g0, reg_{rd}	Clear (zero) register.
clrb	[*address*]	stb	%g0, [*address*]	Clear byte.
clrh	[*address*]	sth	%g0, [*address*]	Clear half-word.
clr	[*address*]	stw	%g0, [*address*]	Clear word.
clrx	[*address*]	stx	%g0, [*address*]	Clear extended word.
clruw	reg_{rs1}, reg_{rd}	srl	reg_{rs1}, %g0, reg_{rd}	Copy and clear upper word.
clruw	reg_{rd}	srl	reg_{rd}, %g0, reg_{rd}	Clear upper word.
mov	*reg_or_imm*, reg_{rd}	or	%g0, *reg_or_imm*, reg_{rd}	
mov	%y, reg_{rd}	rd	%y, reg_{rd}	
mov	%asr*n*, reg_{rd}	rd	%asr*n*, reg_{rd}	
mov	*reg_or_imm*, %y	wr	%g0, *reg_or_imm*, %y	
mov	*reg_or_imm*, %asr*n*	wr	%g0, *reg_or_imm*, %asr*n*	

Index

B

C

E

F

J

L

M

N

Q

R

S

V

W

X

Y

Z