UltraSPARC® IIIi Processor

User's Manual

Version 1.0

June 2003

Table of Contents

Section II: Architecture and Functions

Section III: Execution Environment

Section IV: Memory and Cache

Section V: Supervisor Programming

Section VI: Performance Programming

Section VII: Special Topics

Section VIII: Appendix

Section IX: Index

List of Figures

List of Tables

Preface

Welcome to the UltraSPARC® IIIi Processor User's Manual. This book contains information about the architecture and programming of the UltraSPARC IIIi processor, one of Sun Microsystems' family of SPARC® V9-compliant processors.

Target Audience

This user's manual is mainly targeted for programmers who write software for the UltraSPARC IIIi processor. This user's manual contains a depository of information that is useful to operating system programmers, application software programmers, logic designers and third party vendors who are trying to understand the architecture and operation of the UltraSPARC IIIi processor. This manual is both a guide and a reference manual for low-level programming of the processor.

A Brief History of SPARC

SPARC stands for **S**calable **P**rocessor **ARC**hitecture, which was first announced in 1987. Unlike more traditional processor architectures, SPARC is an open standard freely available through license from SPARC International, Inc. Any company that obtains a license can manufacture and sell a SPARC-compliant processor.

By the early 1990s, SPARC processors were available from over a dozen different vendors, and over 8,000 SPARC-compliant applications had been certified.

In 1994, SPARC International, Inc. published *The SPARC Architecture Manual, Version 9*, which defined a powerful 64-bit enhancement to the SPARC architecture. SPARC V9 provided support for the following:

- 64-bit virtual addresses and 64-bit integer data

- Fault tolerance
- Fast trap handling and context switching
- Big- and little-endian byte orders

UltraSPARC is the first family of SPARC V9-compliant processors available from Sun Microsystems, Inc.

Prerequisites

This user's manual is a companion to *The SPARC Architecture Manual, Version 9*. The reader of this user's manual should be familiar with the contents of *The SPARC Architecture Manual, Version 9*, which is available from many technical bookstores or directly from its copyright holder:

SPARC International, Inc.
2242 Camden Ave, Suite #105
San Jose, CA 95124
(408) 558-8111
http://www.sparc.org

The SPARC Architecture Manual, Version 9 provides a complete description of the SPARC V9 architecture. Since SPARC V9 is an open architecture, many of the implementation decisions have been left to the manufacturers of SPARC-compliant processors. These "implementation dependencies" are introduced in *The SPARC Architecture Manual, Version 9*.

User's Manual Overview

This manual is focused on the treatment of the UltraSPARC IIIi processor. However, it sometimes refers to the UltraSPARC III family of processors to indicate generality of a certain feature. The term "UltraSPARC III family of processors" refers to processors that are similar to the UltraSPARC IIIi processor.

This manual is divided into multiple sections. These sections are described next.

Processor Introduction

The processor introduction section describes the high level features of the UltraSPARC IIIi processor. This section also discusses how the UltraSPARC IIIi processor is used in a system.

Architecture and Functions

This section discusses the details of the UltraSPARC IIIi architecture and the functions of various processor units. An entire chapter is devoted to a discussion on the instruction execution pipeline.

Execution Environment

This section describes the details necessary to understand the execution environment. Various topics such as memory models, data formats, registers, and instruction types are discussed.

Memory and Cache

This section describes the details of memories and caches. Topics such as memory models, memory sub-system, and caches are discussed.

Supervisor Programming

Supervisor software controls the processor and the instruction execution environment for itself and application programs. Chapters are devoted to interrupt handling and error handling.

Performance Programming

This section explores the opportunities to exploit the high-performance architecture of the processor, that is, performance instrumentation.

Instruction Definitions Appendix

This section describes, in detail, each instruction for the UltraSPARC IIIi processor.

SPARC V9 Architecture

The SPARC Architecture Manual, Version 9 was used to implement the processor to insure SPARC compatibility for user and application programs. The SPARC V9 manual provides important theoretical information for operating system programmers who write memory management software, compiler writers who write machine-specific optimizers, and anyone who writes code to run on all SPARC V9-compatible machines. Book copies of the *The SPARC Architecture Manual, Version 9* are readily available at bookstores or from SPARC International, Inc.

Software that is intended to be portable across all SPARC V9 processors should adhere to *The SPARC Architecture Manual, Version 9*.

In this book, the word *architecture* refers to the machine details that are visible to an assembly language programmer or to the compiler code generator. It does not, necessarily, include details of the implementation that are not visible or easily observable by software. Where such details are provided, the intent is to enable faster and better programs.

Textual Usage

Fonts

Fonts are used as follows:

- *Italic sans serif* font is used for exception and trap names. “The *privileged_action* exception...” is an example of how this font is used, it is also used for assembly language terms, emphasis, book titles, and the first instance of a word that is defined.
- `Courier` font is used for register fields (named bits), instruction fields, and read-only register fields. “The `rs1` field contains...” is an example of how this font is used. It is also used for literals, instruction names, register names, and software examples.
- UPPERCASE items are acronyms, instruction names, or writable register fields. Some common acronyms are listed in Acronyms and Definitions. **Note:** Names of some instructions contain both uppercase and lowercase letters.
- Underbar characters join words in register, register field, exception, and trap names. **Note:** Such words can be split across lines at the underbar without an intervening hyphen. “This is true whenever the integer_condition_code field...” is an example of how the underbar characters are used.

Notational Conventions

The following notational conventions are used:

- Square brackets, [ ], indicate a numbered register in a register file. For example, r[0] translates to register 0.
- Angle brackets, < >, indicate a bit number or colon-separated range of bit numbers within a field. “Bits FSR<29:28> and FSR<12> are...” is an example of how the angle brackets are used.
- Curly braces, {}, indicate textual substitution. For example, the string “PRIMARY{_LITTLE}” expands to “ASI_PRIMARY” and “ASI_PRIMARY_LITTLE.”
- If the bar, |, is used with the curly braces, it represents multiple substitutions. For example, the string “ASI_DMMU_TSB_{8KB|64KB|DIRECT}_PTR_REG” expands to “ASI_DMMU_TSB_8KB_PTR_REG”, “ASI_DMMU_TSB_64KB_PTR_REG”, and “ASI_DMMU_TSB_DIRECT_PTR_REG.”
- The $\square$ symbol designates concatenation of bit vectors. A comma (,) on the left side of an assignment separates quantities that are concatenated for the purpose of assignment. For example, if X, Y, and Z are 1-bit vectors and the 2-bit vector T equals 11_2, then

  $$(X, Y, Z) \leftarrow 0 \square T$$

  results in X = 0, Y = 1, and Z = 1.
- “A mod B” means “A modulus B,” where the calculated value is the remainder when A is divided by B.

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}). In some cases, numbers may be preceded by “0x” to indicate hexadecimal (base-16) notation (for example, 0xFFFF.0000). Long binary and hexadecimal numbers within the text have spaces or periods inserted every four characters to improve readability.

The notation 7h’1F indicates a hexadecimal number of $1F_{16}$ with 7 binary bits of width.

Informational Notes

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

Programming Note – Programming notes contain incidental information about programming the UltraSPARC IIIi processor unless otherwise restricted to a particular processor in the family.

Implementation Note – Implementation notes contain information that contains implementation specific information of the UltraSPARC IIIi processor compared to other UltraSPARC processors.

Compatibility Note – Compatibility notes contain information relevant to the previous SPARC V8 architecture.

UltraSPARC Note – UltraSPARC notes highlight the differences between the UltraSPARC I and UltraSPARC II processors and the UltraSPARC III family of processors. This note shows architectural and functional differences that may be generalized or applicable to one particular processor in one of the families. Check the appropriate User's Manual or section in this User's Manual to determine individual processor functionality as needed.

Note – This highlights a useful note regarding important and informative processor architecture or functional operation. This may be used for purposes not covered in one of the other notes.

Acronyms and Definitions

This chapter defines concepts and terminology common to all implementations of SPARC V9.

address space identifier — *See* **ASI**

AFAR — Asynchronous Fault Address Register

AFSR — Asynchronous Fault Status Register

aliased — Two virtual addresses that refer to the same physical address

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

ASI — Address Space Identifier. An 8-bit value that identifies an address space. For each instruction or data access, the integer unit appends an ASI to the address. *See also* **implicit ASI**.

ASR — Ancillary State Register

Ax — Either the A0 or A1 pipeline

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 — Block Load

BST — Block Store

byte — Eight consecutive bits of data

clean window — A register window in which all of the registers contain zero, 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 A memory transaction is completed when an idealized memory has executed the transaction with respect to all processors. A load is considered completed when no subsequent memory transaction can affect the value returned by the load. A store is considered completed when no subsequent load can return the value that was overwritten by the store.

consistency *See* **coherence**

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

copyback The process of copying back a dirty cache line in response to a cache hit while snooping.

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

cross-call An interprocessor call in a multiprocessor system

CSR Control Status Register

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

D-cache Level-1 data memory cache

DCTI Delayed Control Transfer Instruction

DCU Data Cache Unit. Includes controller and Tag and Data RAM arrays

demap To invalidate a mapping in the MMU

deprecated The term applied to an architectural feature (such as an instruction or register) for which a SPARC V9 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 SPARC V9 software and may not be supported in future versions of the architecture.

DFT Designed for Test

DIMM Dual In-line Memory Module. Provides a single or double bank of SDRAM devices 72 bits or 144 bits of data width.

dispatch To send a previously fetched instruction to one or more functional units for execution. Typically, the instruction is dispatched from a reservation station or other buffer of instructions waiting to be executed. *See also* **issued**.

doublet Two bytes (16 bits) of data

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

DQM Data input/output Mask. Q stands for either input or output.

ECU External or embedded Cache Unit controller

EMU External Memory Unit. A combination of the ECU and the Memory Control Unit (MCU).

exception A condition that makes it impossible for the processor to continue executing the current instruction stream without software intervention. *See also* **trap**.

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

`f` register A floating-point register. SPARC V9 includes single-, double-, and quad-precision `f` registers.

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

FFA or FGA or FP1 Floating-Point/Graphics ALU pipeline

FGM or FP0 Floating-Point/Graphics Multiply pipeline

FGU Floating Point and Graphics Unit (FP0 and FP1)

floating-point exception An exception that occurs during the execution of a Floating-point operate (FPop) instruction while the corresponding bit in `FSR.TEM` is set to one. The exceptions are *unfinished_FPop*, *unimplemented_FPop*, *sequence_error*, *hardware_error*, *invalid_fp_register*, or *IEEE_754_exception*.

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

floating-point operate (FPop) instructions Instructions that perform floating-point calculations, as defined by the `FPop1` and `FPop2` opcodes. FPop instructions do not include `FBfcc` instructions or loads and stores between memory and the floating-point unit.

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.

FPRS Floating Point Register State

FPU Floating-Point Unit

FRF Floating-Point Register File

FSR Floating-Point Status Register

halfword An aligned doublet. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

HBM Hierarchical Bus Mode

hexlet Sixteen bytes (128 bits) of data

HPE Hardware Prefetch Enable

I-cache Level-2 Instruction memory cache

IEU Instruction Execution Unit

IIU Instruction Issue Unit

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

implementation dependent An aspect of the architecture that can legitimately vary among implementations. In many cases, the permitted range of variation is specified in the SPARC V9 standard. When a range is specified, compliant implementations must not deviate from that range.

implicit ASI The ASI that is supplied by the hardware on all instruction accesses and on data accesses that do not contain an explicit ASI or a reference to the contents of the ASI register.

informative appendix An appendix containing information that is useful but not required to create an implementation that conforms to the SPARC V9 specification. *See also* **normative appendix**.

initiated *Synonym*: **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 *See* **ISA**

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

interrupt request A request for service presented to the processor by an external device

ISA 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. It does not define clock cycle times, cycles per instruction, datapaths, etc.

issued (1) A memory transaction (load, store, or atomic load-store) is “issued” when a processor has sent the transaction to the memory subsystem and the completion of the request is out of the processor’s control. *Synonym*: **initiated**.
(2) An instruction (or sequence of instructions) is said to be *issued* when released from the processor's in-order instruction fetch unit. Typically, instructions are issued to a

reservation station or other buffer of instructions waiting to be executed. (Other conventions for this term exist, but this document attempts to use "issue" consistently as defined here). *See also* **dispatched**.

IU Integer Unit

L2-cache External or embedded unified, instruction/data, Level-2 memory cache

leaf procedure A procedure that is a leaf in the program's call graph, that is, one that does not call (by using `CALL` or `JMPL`) any other procedures.

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. *Load* includes loads into integer or floating-point registers, block loads, Load Quadword Atomic, 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.

MCU Memory Control Unit. Controls the SDRAM signals

Memory Management Unit *See* **MMU**

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

module A master or slave device that attaches to the shared-memory bus

MOESI A cache-coherence protocol. Each of the letters stands for one of the states that a cache line can be in, as follows: **M**, modified, dirty data with no outstanding shared copy; **O**, owned, dirty data with outstanding shared copy(s); **E**, exclusive, clean data with no outstanding shared copy; **S**, shared, clean data with outstanding shared copy(s); **I**, invalid, invalid data.

must *Synonym*: **shall**

NaN Not a Number

NCPQ Noncoherent Pending Queue

next program counter *See* **nPC**

NFO Nonfault access only

non-faulting load A load operation that, in the absence of faults or in the presence of a recoverable fault, completes correctly, and in the presence of a nonrecoverable fault returns (with the assistance of system software) a known data value (nominally zero). *See also* **speculative load**.

non-privileged An adjective that describes:
(1) the state of the processor when PSTATE.PRIV = 0, that is, non-privileged mode;
(2) processor state information that is accessible to software while the processor is in either privileged mode or non-privileged mode; for example, non-privileged registers, non-privileged ASRs, or, in general, non-privileged state;
(3) an instruction that can be executed when the processor is in either privileged mode or non-privileged mode.

non-privileged mode The mode in which a processor is operating when PSTATE.PRIV = 0. *See also* **privileged**.

normative appendix An appendix containing specifications that must be met by an implementation conforming to the SPARC V9 specification. *See also* **informative appendix**.

nPC Next program counter. A register that contains the address of the next executed instruction if a trap does not occur.

NPT Non-Privileged Trap

NWINDOWS The number of register windows present in a particular implementation

OBP OpenBoot™ PROM

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.

opcode A bit pattern that identifies a particular instruction

optional A feature not required for SPARC V9 compliance

ORQ Outgoing Request Queue

PA Physical Address. An address that maps real physical memory or I/O device space. *See also* **virtual address**.

Page Table Entry *See* **PTE**

PC Program Counter. A register that contains the address of the instruction currently being executed by the IU.

PCR Performance Control Register

physical address *See* **PA**

PIC Performance Instrumentation Counter

PIO Programmed I/O

PIPT Physically Indexed, Physically Tagged

PIVT Physically Indexed, Virtually Tagged

POR Power-on Reset. The most aggressive reset.

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

Non-prefetchable 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 processor when PSTATE.PRIV = 1, that is, *privileged mode*;
(2) processor state that is only accessible to software while the processor is in *privileged mode*; for example, privileged registers, privileged ASRs, or, in general, privileged state;
(3) an instruction that can be executed only when the processor is in *privileged mode*.

privileged mode The mode in which a processor is operating when PSTATE.PRIV = 1. *See also* **non-privileged.**

processor The combination of the integer unit and the floating-point unit

program counter *See* **PC**.

PSO Partial Store Order

PTA Pending Tag Array

PTE Page Table Entry. Describes the virtual-to-physical translation and page attributes for a specific page. A PTE generally means an entry in the page table or in the TLB; however, it is sometimes used as an entry in the translation storage buffer (TSB). In general, a PTE contains fewer fields than a TTE. *See also* **TLB** and **TSB**.

QNaN Quiet Not a Number

quadlet Four bytes (32 bits) of data

quadword Aligned hexlet. **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.

RD Rounding Direction

RDPR Read Privileged Register

RED_state **R**eset, **E**rror, and **D**ebug state. The processor state when PSTATE.RED = 1. A restricted execution environment used to process resets and traps that occur when TL = MAXTL – 1.

reserved Describes 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.

Reserved instruction fields shall read as zero, unless the implementation supports extended instructions within the field. The behavior of SPARC V9 processors when they encounter nonzero values in reserved instruction fields is undefined.

Reserved bit combinations within instruction fields are defined in Appendix A, Instruction Definitions. In all cases, SPARC V9 processors shall decode and trap on these reserved combinations.

Reserved register fields should always be written by software with values of those fields previously read from that register or with zeroes; they should read as zero in hardware. Software intended to run on future versions of SPARC V9 should not assume that these fields will read as zero or any other particular value. Throughout this specification, figures and tables illustrating registers and instruction encodings indicate reserved fields and combinations with an em dash (—).

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

restricted Describes an ASI that may be accessed only while the processor is operating in privileged mode.

RMO Relaxed Memory Order

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

RTO Read to Own

RTOR Read to Own Remote. A reissued RTO transaction.

RTS Read to Share

RTSM Read to Share Mtag. An RTS to modify MTag transaction.

SAM SPARC Architecture Manual, Version 9

scrub Writes data from the W-cache to the L2-cache

SDRAM Synchronous Dynamic Random Access Memory. May be prefaced with DDR, double data rate SDRAM.

SFAR Synchronous Fault Address Register

SFSR Synchronous Fault Status Register

shall A keyword indicating a mandatory requirement. Designers shall implement all such mandatory requirements to ensure interoperability with other SPARC V9-compliant products. *Synonym:* **must**.

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

SIAM Set Interval Arithmetic Mode instruction

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**.

SIG Single-Instruction Group. Sometimes shortened to “single-group.”

SIR Software-Initiated Reset

SNaN Signalling Not a Number

snooping The process of maintaining coherency between caches in a shared-memory bus architecture. All cache controllers monitor (snoop) the bus to determine whether they have a copy of the shared cache block.

SPE Software Prefetch Enable

speculative load A load operation that is issued by the 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* **non-faulting load**, which is an explicit load that always completes, even in the presence of recoverable faults.

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. *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*.

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

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

TBA Trap Base Address

TLB Translation Lookaside Buffer. A cache within an MMU that contains recent partial translations. TLBs speed up closely following translations by often eliminating the need to reread PTE from memory.

TLB hit The desired translation is present in the on-chip TLB

TLB miss The desired translation is not present in the on-chip TLB

TPC Trap-saved PC

Translation Lookaside Buffer *See* **TLB**

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

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

TSO Total Store Order

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

UE User process error

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

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

Notwithstanding any of the above, undefined aspects of the architecture shall not cause security holes (such as allowing user software to access privileged state), put the processor into supervisor mode or an unrecoverable state.

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

unpredictable *Synonym:* **undefined**

unrestricted Describes an ASI that can be used regardless of the processor mode; that is, regardless of the value of PSTATE.PRIV.

user application program *Synonym:* **application program**

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

victimize [Error handling]

VIPT Virtually Indexed, Physically Tagged

virtual address *See* **VA**

VIS Visual Instruction Set. Performs partitioned integer arithmetic and other small integer operations.

VIVT Virtually Indexed, Virtually Tagged (cache)

WAW Write After Write

WDR WatchDog trap-level Reset

word An aligned quadlet. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

WRF Working Register File

writeback The process of writing a dirty cache line back to memory before it is refilled.

WRPR Write Privileged Register

XIR Externally Initiated Reset

SECTION I

Processor Introduction

CHAPTER 1

Introducing the UltraSPARC IIIi Processor

1.1 Overview

The UltraSPARC IIIi processor is derived from Sun Microsystems high-end UltraSPARC III processor, providing many of the same performance, reliability, and security features, but in a highly integrated format that brings the power of the UltraSPARC architecture to cost-efficient high-end desktop systems and inexpensive 1-4 way servers. It implements both the full 64-bit, SPARC V9 architecture and version 2.0 of Sun Microsystems' VIS™ instruction set. The VIS instruction set provides a wide range of "Single Instruction, Multiple Data" (SIMD) acceleration functions for working with 8-, 16-, and 32-bit data values, doing pixel manipulation, 2D image processing, 3D graphics, data compression, and other specialized performance-critical operations.

Major functional blocks included in the UltraSPARC IIIi processor are:

- Integer execution unit
- Floating-point execution unit
- 32 KB primary (Level 1 or L1) instruction cache
- 64 KB primary (L1) data cache
- 1 MB L2 unified cache (used for both instructions and data)
- 2 KB prefetch cache for floating-point data
- 2 KB write cache
- Synchronous DRAM (SDRAM) memory controller
- JBUS controller

In common with all other members of the UltraSPARC III family of processors, the UltraSPARC IIIi processor is a 4-way superscalar processor, meaning it attempts to fetch 4 instructions at a time from the L1 instruction cache, and (given the appropriate instruction mix) is capable of sustaining an execution rate of 4 instructions per clock cycle. Each instruction is processed through a 14-stage pipeline that starts with address generation and

ends with the final retirement of any valid execution result. A 16-entry instruction queue decouples instruction fetch from instruction issue, working to buffer any discrepancies between these two rates. Thus, if more instructions are fetched than can be issued repeatedly, an empty instruction queue gradually will fill. Or, if the next instruction fetch misses in the L1 cache, a filled instruction queue can hide this break in the flow of instructions through the pipeline, by continuing to supply the execution units with instructions for the several clock cycles needed to retrieve the missing block of instructions from the on-chip L2 cache.

To enhance throughput, while instructions enter and exit the instruction queue in strict program order, they can complete executing out-of-order. For example, if a short latency instruction (like an integer add) follows a long latency instruction (like an integer divide) in the pipeline, the fast operation does not need to wait on the slow one to finish. Instructions fetched together will enter the queue in parallel, but, within the constraints imposed by program order, they may exit the queue in company with instructions fetched either earlier or later (depending on the specific instruction mix and availability of the necessary functional units).

The UltraSPARC IIIi processor is supported by Sun's popular Solaris™ operating system, providing access to the more than ten thousand applications that have been developed for the SPARC/Solaris platform over the years. Comprehensive sets of programs are available for many fields, including engineering, manufacturing, telecommunications, financial services, health, retail, ecommerce, and a variety of other industry segments. Additional operating systems available for use with UltraSPARC processors include Linux and leading real-time operating systems. A robust set of tools for developing software also can be readily acquired, either from Sun Microsystems or independent software vendors.

1.2 Features

The UltraSPARC IIIi processor is richly featured, providing all of the following capabilities:

- Binary compatibility with the entire base of SPARC application code.
- Full 64-bit virtual address space.
- 64-bit internal operation, including 64-bit datapaths, 64-bit ALUs, and 64-bit address arithmetic.
- 43-bit physical address space, supporting up to 8 Terabytes of memory.
- Low latency and high bandwidth for memory operations, due in part to a memory hierarchy that incorporates separate on-chip L1 instruction and data caches, a 1 MB on-chip unified L2 cache, a prefetch cache, a write cache, and an on-chip SDRAM controller.
- 1 to 4-way glueless multiprocessing.
- Introductory frequency above 1 GHz, scaling up over time, propelled by a 14-stage non-stalling pipeline.
- 4-way superscalar instruction dispatch to nine separate execution units.

- High-performance JBUS system interface.
- Sophisticated power management.
- Extensive RAS protection, starting with error detection and correction (EDC) on the primary and secondary caches.

Compared to the previous generation UltraSPARC IIi processor, the UltraSPARC IIIi processor offers several useful new features, including version 2.0 of the VIS instruction set, support for interval arithmetic, better prefetch capabilities, an extended interrupt scheme, and 4 times as much physical address space. It combines these advantages with far greater levels of performance as well as greatly improved data reliability.

The UltraSPARC IIIi processor brings all the advantages of full 64-bit computing to both desktop systems and entry-level servers, together with up to 4-way glueless MP operation, in a very cost-competitive form.

1.3 Summary

Detailed information about specific functional blocks and features of the UltraSPARC IIIi processor can be found in the following chapters of this manual. This section attempts to summarize the more significant elements of the UltraSPARC IIIi processor, for the benefit of readers seeking to quickly acquire a relatively comprehensive understanding of it.

Register Windows
In addition to the usual assortment of registers used for control purposes, status information, condition codes, etc., the UltraSPARC architecture includes 160 64-bit integer registers, and another set of 32 64-bit registers for use by the FPU and VIS instructions. The 160 integer registers are organized into 8 overlapping register “windows” of 32-registers each. In each register window, 8 registers are shared with the previous window, and are used to hold input parameters from a calling routine; 8 registers are shared with the next window, and are used to hold output parameters for use by a called routine; 8 registers are unshared, and are used to hold local parameters; while 8 registers are global, and are used to hold values shared by all routines. The 8 output registers for one window are the 8 input registers for the next window. There are four sets of 8 global registers, designated for different uses, as appropriate: normal, MMU, interrupt, and alternate. (8 x 8 in/out registers + 8 x 8 local registers + 4 x 8 global registers = 160 integer registers.) Register windows are a distinctive feature of the SPARC architecture, designed to provide a very fast means to handle context switches, interrupts, and traps.

32 KB Primary Instruction Cache Memory (4-way set associative)
Holds 8K fixed-width 4-byte SPARC instructions for immediate access by the pipeline. Instructions in this cache are protected against single bit errors by parity checking. If an error is detected, the cache line with the erring byte is marked as invalid; as a consequence, the next access to that line forces it to be refilled with valid instructions from the L2 cache.

64 KB Primary Data Cache Memory (4-way set associative)
Holds data items for rapid loads to and stores from the register file. (In common with other RISC architectures, all SPARC instructions operate register-to-register, accessing their operands from the register file and return their results to it.) Uses the same parity checking/ line invalidation scheme for EDC as the instruction cache. Cache is write-through, so data in the primary cache is always "clean."

2 KB Prefetch Cache Memory (4-way set associative)
A special cache used to hold floating-point data that can be fetched well ahead of use. This cache increases the effective size of the primary data cache when executing floating-point programs, and provides specific hardware support for speculative loads, including both software and hardware data prefetch operations.

2 KB Write Cache Memory (4-way set associative)
A special cache used to coalesce data being stored back to memory. By reducing the number of separate store operations needed, effectively increases the memory bandwidth of the processor.

Non-cacheable Store Compression
The UltraSPARC IIIi processor uses a 16-byte buffer to merge adjacent non-cacheable stores into a single external data transaction, greatly increasing store bandwidth to the graphics frame buffer. In addition, a flow control signal is available through the Graphics Status Register that allows software to interrogate a FIFO status signal on the graphics card, without requiring completion of a non-cacheable read to the device. This prevents stalling due to waiting for prior non-cacheable stores to be pushed to the device, and eliminates bubbles in the store throughput due to the pipeline depth between the processor and the graphics device.

1 MB Unified Secondary Cache (4-way set associative)
This large, on-chip L2 cache buffers the impact of L1 cache misses by providing fast, local access to a much larger pool of instructions and data than will fit into the several L1 caches. The effect is to substantially reduce the overall latency of memory operations. The tags for the L2 cache are protected by parity checking, while data in the cache is protected by full ECC, providing single-bit error correction and double-bit error detection. The L2 cache uses a write-back policy to reduce store traffic to main memory. Any uncorrectable double-bit errors are marked on write-back, so they will not propagate to other processors in an MP configuration.

JBUS Interface
A Sun-proprietary system interface new to the UltraSPARC IIIi processor, developed to provide a combination of the high performance expected of Sun systems with the low cost demanded by the desktop and entry-level server marketplaces. A companion JI0 chip is available from Sun Microsystems. In addition to supporting the shared address/data JBus itself, the companion chip also provides support for up to 2 industry-standard PCI buses, as well as for Sun's proprietary UPA64S graphics bus (in place of the secondary PCI bus).

SDRAM Controller

Provides direct connectivity of the processor to main memory through a 2-channel DDR SDRAM interface. Full ECC protection is provided on all stored memory data, and transactions on the memory/address bus are protected by parity checking. In the interests of simplicity, any system or DRAM-related, non-correctable errors are handled as deferred traps.

Low Power Operating Modes

The UltraSPARC IIIi processor features low-power modes. When signalled to conserve power, the on-chip Clock Control Unit instantaneously switches the processor's clock rate to lower power modes.

CHAPTER 2

UltraSPARC IIIi Processor in a System

The UltraSPARC IIIi processor can reside either on the system motherboard itself or in a separate module attached to the motherboard. The UltraSPARC IIIi processor is intended to operate with a special support bridge chip that provides I/O functions (called “JIO”). The UltraSPARC IIIi processor and its companion I/O chip can be used to scale systems from a minimum 1-way desktop or blade configuration up to a 4-way stand-alone server.

2.1 System Configurations

The UltraSPARC IIIi processor is designed to operate efficiently in 1-way, 2-way, or 4-way systems.

2.1.1 Four-Processor System

FIGURE 2-1 illustrates a typical configuration for a high-performance, 4-way, entry-level server. This system incorporates 4 UltraSPARC IIIi processors and two companion JIO chips (configured as master-slave) to provide maximum I/O bandwidth. In the system shown, JBUS uses a “Bell Repeater”, a bit-sliced pipeline register chip to reduce loading on JBUS. A lower cost 4-way system with half the bandwidth can be build using a single master JIO chip.

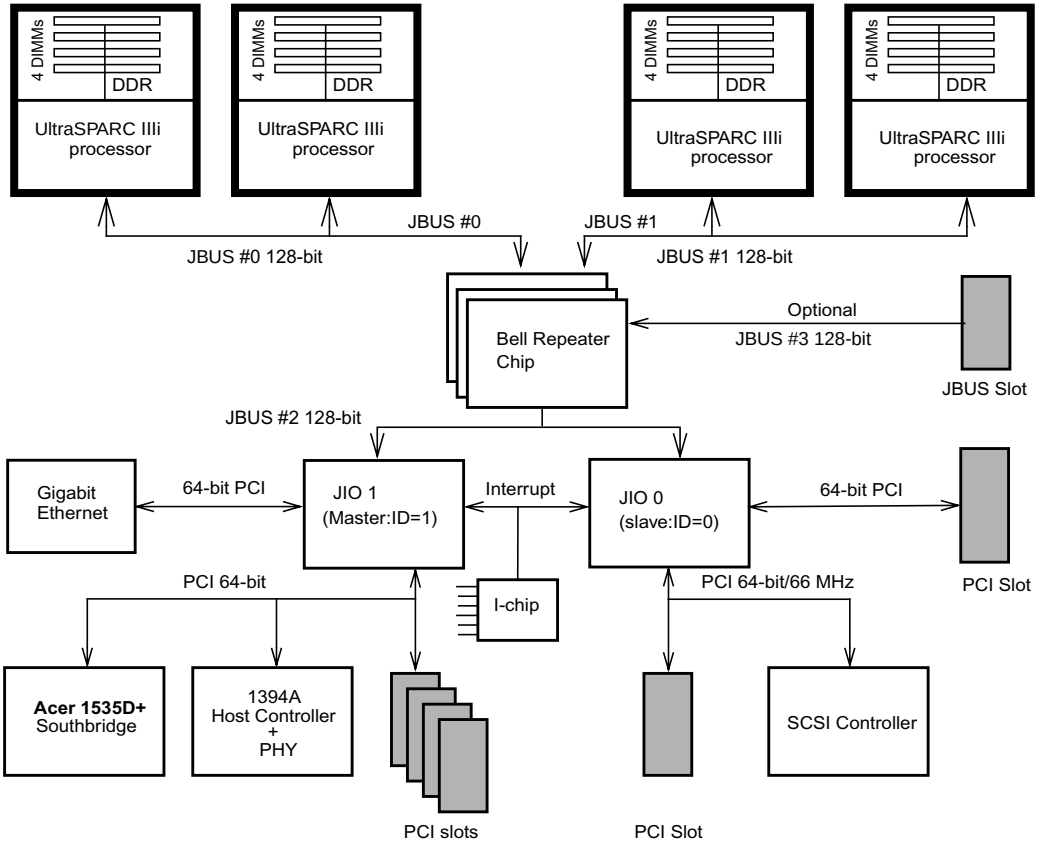


FIGURE 2-1 Four-Processor System with the UltraSPARC IIIi Processor

Note that, in the configuration shown, four possible JBUS segments, JBUS #0, JBUS #1, JBUS #2 and an optional JBUS #3, propagate through the Bell Repeater. The Bell Repeater is only needed when the JBUS is required to run at maximum frequency with more than three loads, to reduce loading on the JBUS. The Bell Repeater forwards the signals from each of the four segments of the JBUS on to the other three segments. Propagating JBUS signals through the Bell Repeater introduces a one cycle delay, i.e., any signals the Bell Repeater receives in one cycle. it forwards in the next. The Bell Repeater operates entirely automatically, i.e., it requires no control signals.

2.1.2 Two-Processor System

FIGURE 2-2 illustrates a typical configuration for inexpensive 2-way desktops or servers based on the UltraSPARC IIIi processor. This system incorporates 2 UltraSPARC IIIi processors with two companion JIO chips. Since this configuration, like the 4-way system, may involve placing 4 loads on the JBUS, it also requires addition of a Bell Repeater to achieve maximum JBUS performance. In the 4-load configuration shown, however, no Bell Repeater is needed, since the JBUS in this example has been designed to run lower than maximum frequency.

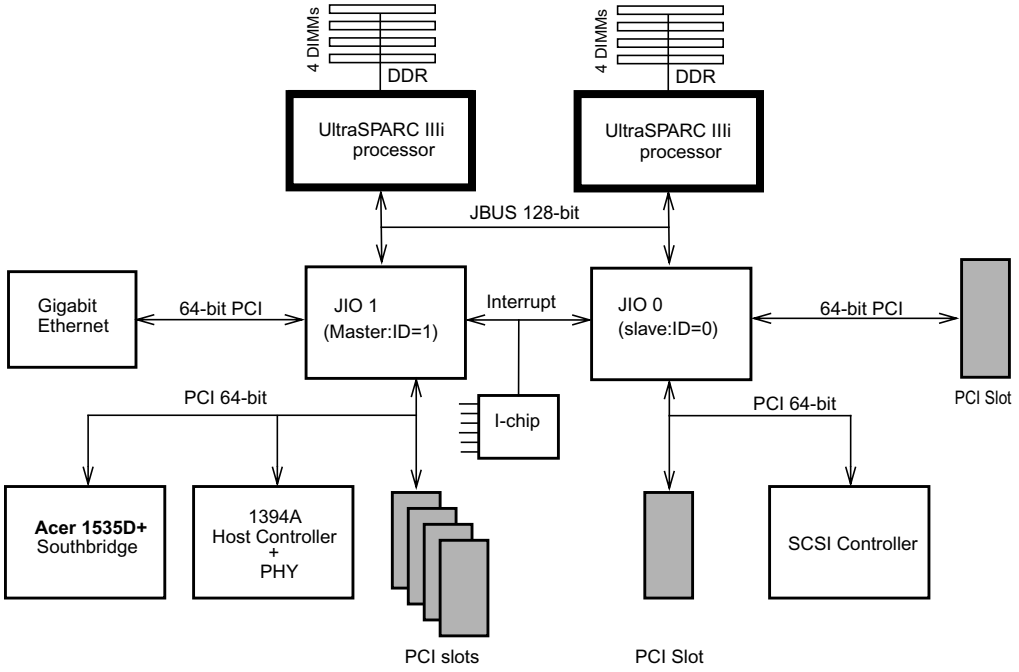


FIGURE 2-2 Two-Processor System with the UltraSPARC IIIi Processor

2.1.3 One-Processor System

FIGURE 2-3 illustrates a typical configuration for a minimum-cost, 1-way system based on the UltraSPARC IIIi processor. This system involves no Bell Repeater and only 1 JIO chip. To reduce cost still further, note that the UltraSPARC IIIi processor can be configured to use a minimum memory of only two DIMMs on the DDR interface. In this sort of cost optimized single processor configuration, PCI slots are only provided where PCI devices can be added to a system.

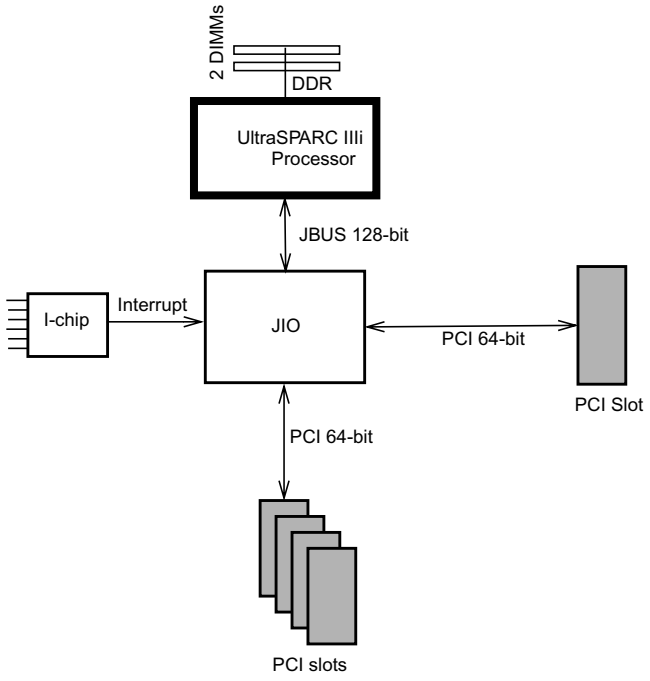


FIGURE 2-3 One-Processor System with the UltraSPARC IIIi Processor

2.2 JBUS Interface

The UltraSPARC III processor has a companion JIO chip that features a 183-pin interface to connect to the JBUS. The JBUS is a 16-byte (128-bit), split transaction, shared address/data bus.

2.3 Memory System

The memory system consists of the Memory Control Unit (MCU) in the processor, and two channels of DDR Synchronous DRAM memory. Each channel supports either one or two registered DIMMs, allowing systems to be configured with less memory (for lower cost) or more memory (for higher performance). Each channel has an address/ control bus as well as an 8-byte data bus (plus 1 byte for ECC check bits). Clock buffering with a PLL is provided on the DIMMs.

Since both memory channels are controlled identically by the memory controller, DIMMs always must be loaded in pairs. Each DIMM pair consists of two 72-bit DDR SDRAM DIMMs. Since each DIMM could be dual sided (single/double), there are a maximum of four data loads per memory channel.

The UltraSPARC IIIi processor modules have a total of four DIMM slots. In order, these are termed 1A, 1B, 2A, 2B. DIMMs 1A and 2A correspond to memory channel 1. DIMMs 1B and 2B correspond to memory channel 2. DIMM pair #1 contains DIMMs 1A and 1B. DIMM pair #2 contains DIMMs 2A and 2B. FIGURE 2-4 summarizes the high level architecture of the UltraSPARC IIIi memory system, including placement of the four DIMMs.

Each cache line is split across the DIMMs in memory channel 1 and memory channel 2. In FIGURE 2-4, DIMM 1A belongs to memory channel 1 and DIMM 1B belongs to memory channel 2. Similarly, DIMM 2A belongs to memory channel 1 and DIMM 2B belongs to memory channel 2.

In exactly the same way, each External Bank of memory is split across the two memory channels. As shown in FIGURE 2-4, External Banks 0 and 1 are split across DIMM 1A and DIMM 1B, and External Banks 2 and 3 are split across DIMM 2A and DIMM 2B.

Each External Bank contains four Internal Banks. The memory controller pipelines requests to memory, making use of all 16 of the internal memory banks available (4 External Banks times 4 Internal Banks each), when all DIMM slots are fully loaded.

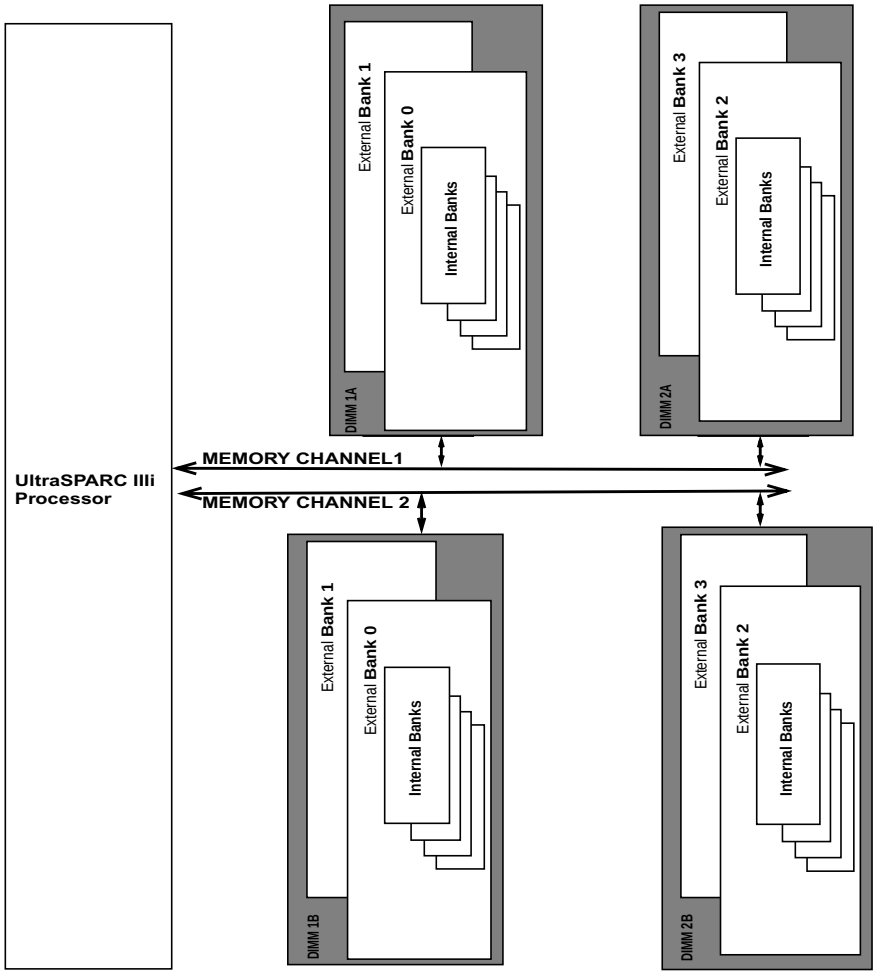


FIGURE 2-4 DDR Memory System Architecture

2.4 Power Management

The UltraSPARC IIIi processor features two low power modes: a 1/2 speed mode and a 1/32 speed mode for clock operation.

SECTION II

Architecture and Functions

CHAPTER 3

UltraSPARC IIIi Processor Architecture Basics

The UltraSPARC IIIi processor is a high-performance, highly-integrated, 4-way superscalar processor. In addition to wide parallel instruction dispatch to exploit instruction-level parallelism in code, the processor is designed to offer high clock speeds. To reduce instruction execution latencies, the processor incorporates on-chip level-1 instruction and data caches, a 1 MB unified level-2 cache, a memory controller, and large, flexible memory management units (MMUs). The processor was designed specifically to work in inexpensive desktop systems and entry-level servers, in configurations ranging from 1-4 processors.

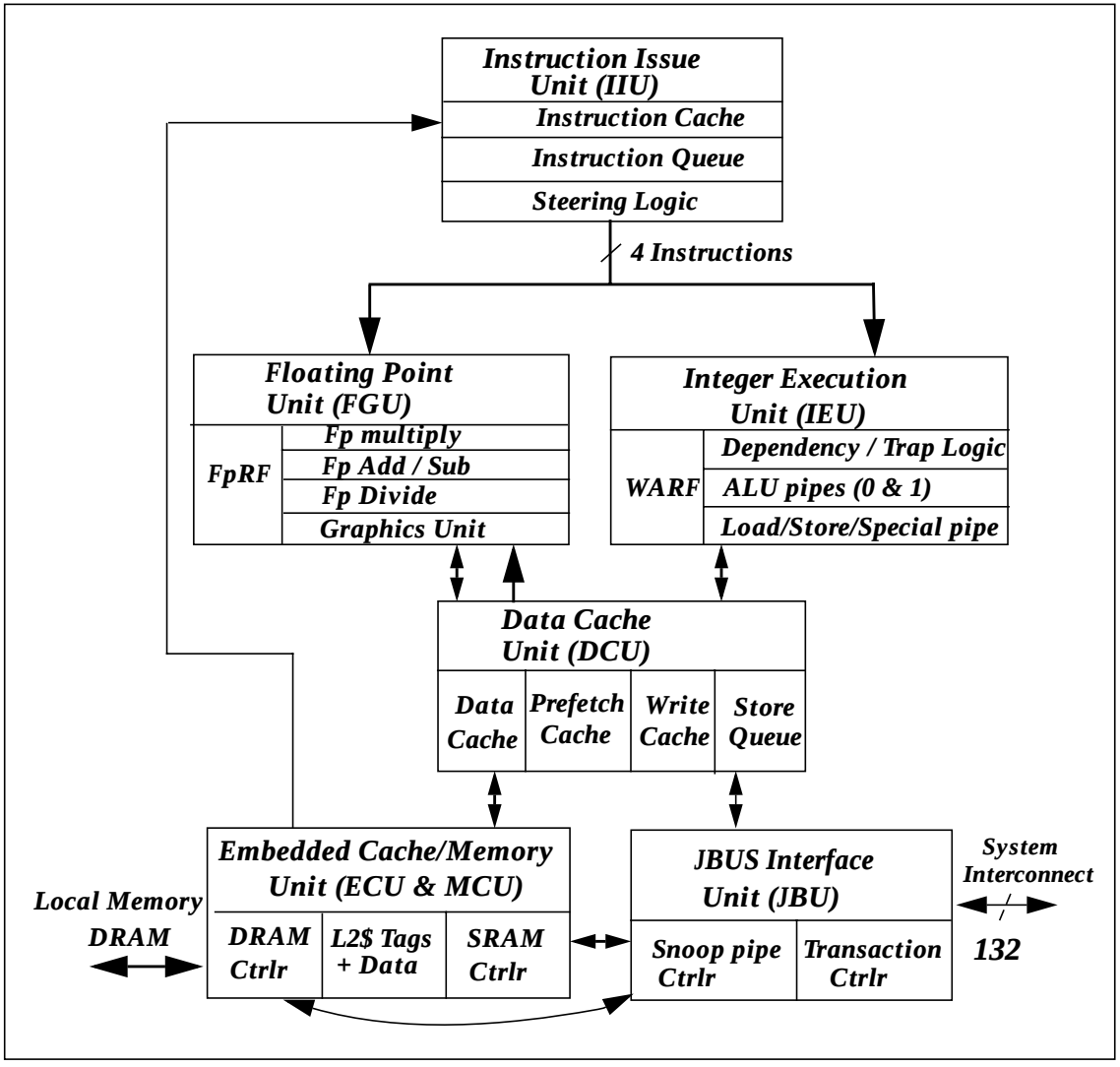
The UltraSPARC IIIi processor also offers a number of performance enhancements over previous UltraSPARC processors. The processor incorporates multiple data prefetching mechanisms to enable long latency load operations to be overlapped with earlier operations. The processor offers an enhanced data memory management unit (D-MMU) with 3 separate TLBs providing a total of 1040 entries, and flexible support for page sizes ranging from 8 KB up to 4 MB, enabling the processor to effectively map both small and large memory systems.

3.1 Component Overview

The processor includes a high-performance, instruction fetch engine, called the *instruction issue unit*, that is decoupled from the rest of the pipeline by a 16-entry instruction buffer. Four instructions at a time are fetched from the level-1 instruction cache and queued for issue in the instruction buffer. Up to 4 instructions in a clock cycle can be steered from this queue into 6 execution buffers. Up to 6 instructions in a clock cycle can be dispatched from the 6 execution buffers into the 6 parallel execution units in UltraSPARC IIIi processor: 2 integer ALUs, 1 branch unit, 1 load/store unit (also handles certain special operations, like integer multiplication and division), 1 floating-point add/subtract unit, and 1 floating-point multiply/divide unit. The two floating-point units also handle the specialized SIMD VIS instructions for accelerating graphics, media, and network functions.

In addition to a 32 KB primary instruction cache, a 64 KB primary data cache, an instruction fetch engine, a 16-entry instruction buffer, and the 6 parallel execution units, the processor also integrates on-chip a 1 MB L2-cache, a 2 KB prefetch cache, a 2 KB write cache, an I/O interface (to the JBUS), and a memory controller. FIGURE 3-1 shows a simplified block diagram of the UltraSPARC IIIi processor.

FIGURE 3-1 UltraSPARC IIIi Processor Architecture



3.1.1 Instruction Fetch and Buffering

The instruction issue unit in the UltraSPARC IIIi processor is responsible for fetching, queuing, and steering instructions as appropriate to one of the six parallel execution units included in the UltraSPARC IIIi processor design. Up to four instructions are fetched and decoded at a time. Assuming the fetch request hits in the level-1 instruction cache (and certain other conditions are met, e.g., the instruction queue is not full), instruction fetching is possible in every clock cycle. If a fetch request misses in the level-1 instruction cache, a fill request is sent to the lower memory hierarchy for the 32-byte line containing the missing instruction block.

The instruction cache uses a 32-byte line, containing 8 fixed-width 4-byte SPARC instructions. The unified L2 cache uses a 64-byte line. If the instruction request hits in the first half of an L2 cache line, the second half of that line is also fetched, and placed in a special 32-byte Instruction Prefetch Buffer (IPB), accessed in parallel with the instruction cache. This precaution avoids a potential L1 cache miss, in those cases where instruction fetching does move on sequentially to use the next group of 8 instructions.

The UltraSPARC IIIi processor instruction cache contains 1K lines, with a total capacity of 8,192 instructions. Cache lines are virtually indexed but physically tagged. The cache is 4-way set-associative. It requires 2 cycles of latency to fetch an item, but access is pipelined, so sequential requests have single cycle throughput, after the two cycle delay for the first item is satisfied. Other cache features besides the usual data and tag arrays include a microtag, predecode bits, a Load Prediction Bit (LPB), and a snoop tag array. The microtag uses 8 bits of virtual address to enable fast way-selection of a potentially matching cache line, without waiting for the physical address translation to complete. The predecode bits include information about which pipeline each instruction will be issued to, and other information to optimize execution. The LPB is used to dynamically learn those load instructions that frequently see a read-after-write (RAW) hazard with preceding stores. The snoop tag is a copy of the tags dedicated for snoops caused by either stores from the same, or different, processors. The instruction cache in the UltraSPARC IIIi processor is kept completely coherent so the cache never needs to be flushed.

The instruction fetch engine is also dependent upon control transfer instructions such as branches and jumps. The UltraSPARC IIIi processor uses a 16K-entry branch predictor to predict the fetch direction of conditional branches. For branches that are either known to be taken or predicted taken, the branch target must be determined. For PC relative branches, the target of the branch is computed. This adds a one-cycle penalty to the branch taken case, but avoids any penalties from target misprediction. For predicting the target of return instructions an 8-entry Return Address Stack (RAS) is used. For other indirect branches (branches whose targets are determined by a register value), the software can provide a branch target prediction with a jump target preparation instruction.

The 16-entry instruction buffer decouples the front-end instruction fetch from the back-end instruction execution, allowing these two parts of the pipeline to operate at different rates. If more instructions are fetched than can be issued, an empty instruction buffer gradually fills up. If instruction fetch is interrupted by a taken branch penalty or an instruction cache miss, a full instruction buffer gradually drains, hiding some or all of the ensuing latency.

3.1.2 Execution Pipelines

The UltraSPARC IIIi processor has six parallel execution units. Buffered instructions can be issued to all six units in a single cycle, and sustained issue to any 4 of these units is possible. The six executions are:

- 2 integer Arithmetic and Logic Units (ALU)
- 1 Branch pipeline
- 1 Load/store pipeline (also handles special instructions)
- 1 Floating-point multiply pipeline (also handles SIMD instructions)
- 1 Floating-point addition pipeline (also handles SIMD instructions)

The ALUs perform integer addition and subtraction, logic operations, and shifts. These units have single-cycle latency and throughput. The branch pipeline handles all branch instructions and can resolve one branch each cycle. Load/store operations are discussed in the next section. The load/store pipeline also handles Integer multiplication and division. Integer multiplication has a latency of 6 to 9 cycles depending on the size of the operands. Division is also iterative and requires 40 to 70 cycles.

The floating-point units each have 4-cycles of latency, but are fully pipelined (one instruction per cycle per pipeline). These pipelines handle double and single precision floating-point operations and a set of SIMD instructions that operate on 8 or 16-bit fields. Floating-point division and square root operations use the floating-point multiplication pipeline and are iterative computations. Floating-point division requires 17 cycle for single precision, 20 cycles for double precision computations. Floating-point square root requires 23 cycles for single precision, 29 cycles for double precision computations.

3.1.3 Load/Store Unit

A load or store instruction can be issued each cycle to the load/store pipeline. The load/store unit consists of the load/store pipeline, a store queue, a data cache and a write cache.

Integer loads of unsigned words and double words have a 2-cycle latency. All other loads have a 3-cycle latency. There is an 8-entry store queue to buffer stores. Stores reside in the store queue from the time they are issued until they complete an update to the write cache. The store queue can effectively isolate the processor from the latency of completing stores. If the store queue fills up, the processor will block on a subsequent store.

The store queue allows successive separate stores to the same cache line to collect. For non-catchable stores (for example, stores to a graphics frame buffer), this function can greatly reduce the amount of store traffic generated, effectively raising the bandwidth to external devices.

The UltraSPARC IIIi processor supports store forwarding, the ability to pass data still in the store queue directly to a quickly following load that attempts to access the same target location in memory (a Read After Write or RAW hazard). Since 3 cycles of latency is required for a load to communicate with the store queue, the LPB bit in the instruction cache is used to force 2-cycle loads to issue as 3-cycle loads. If a 2-cycle load is not correctly predicted to have a RAW hazard, the load must be re-issued.

The data cache holds 64 KB. Cache lines are virtually indexed but physically tagged. The cache is 4-way set-associative. It requires 2 cycles of latency to fetch an item, but access is pipelined, so sequential requests have single-cycle throughput. Like the instruction cache, the data cache uses 8-bit microtags to do way-selection based on virtual addresses. The update policy is write-through, no write-allocate. The line size is 32 bytes with no subblocking. The data cache only needs to be flushed if an alias is created using virtual address bit 13. VA[13] is the only virtual bit used to index the data cache.

The write cache is a write-back cache used to reduce the amount of store bandwidth required to the L2-cache. It exploits both temporal and spatial locality in the store stream. The small (2 KB) structure achieves a store bandwidth equivalent to a 64 KB write-back data cache while maintaining TSO compatibility. The write cache is kept fully coherent with both the processor pipeline and the system memory state. The write cache is 4-way set-associative and has 64-byte lines. The write cache maintains dirty bits on a per byte basis.

3.1.3.1 Data Prefetching Support

The UltraSPARC IIIi processor makes use of advanced data prefetching mechanisms in both software and hardware. Software prefetching allows compilers (of Java JITs) to explicitly expose the memory-level parallelism in programs and to schedule memory operations. There are a number of variations of software prefetches. Software prefetches can specify if the data should be brought into the processor for reading or for both reading and writing. Software can also specify if the data should be installed into the L2-cache, for data that will be reused frequently, or only brought into the prefetch cache.

Hardware prefetching is an automatic facility that looks for common data sequences, and attempts to fetch ahead based on detected patterns.

Prefetch mechanisms are used to both hide load-miss activity and overlap load misses to increase memory-level parallelism. Robust prefetch mechanisms that avoid as many load misses as possible are especially important for the UltraSPARC IIIi processor since load misses block program execution, i.e., on load misses, the processor waits for the load to complete before executing any other instructions.

Specifically to benefit data-intensive floating-point programs, the UltraSPARC IIIi processor features a special prefetch cache. The prefetch cache is a small (2 KB) cache that is accessed in parallel with the data cache for floating-point loads. In effect, it expands the size of the data cache when executing floating-point programs, and can noticeably reduce load misses with a correspondingly favorable impact on performance. Floating-point load misses,

hardware prefetches and software prefetches bring data into the prefetch cache. The prefetch cache is 4-way set-associative and has 64-byte lines which are broken into two 32-byte subblocks with separate valid bits. The prefetch cache is write invalidate.

3.1.4 Memory Management Units

There are separate Memory Management Units (MMUs) for instruction and data address translation. MMUs have two primary functions: memory protection, preventing processes from accessing each other's memory spaces, and address translation -- the conversion of virtual addresses in the processor's logical 64-bit address space into real addresses in the system's physical memory. The first time a virtual address is encountered, the processor traps to software to walk a set of page tables in memory to locate the corresponding physical address. Since the process of translating a virtual address into a physical address is slow, the MMUs contain a set of Translation Lookaside Buffers (TLBs). These are specialized caches used to store recently mapped pairs of virtual-physical addresses together with associated page protection and usage information. Since TLB lookup is fast (unlike the initial translation process itself), memory operations can proceed without interruption as long as their virtual address "hits" in a TLB.

The instruction MMU contains two TLBs accessed in parallel. The first TLB is a 16-entry fully-associative TLB. This small TLB is perfectly flexible, in the sense that it can hold pages of various sizes (8K, 64K, 512 KB, or 4 MB), and pages can be either locked or unlocked. The second TLB is a 128-entry, 2-way set-associative TLB. This large TLB is used exclusively to hold unlocked pages of the "default" 8 KB size.

The data MMU of the UltraSPARC IIIi processor is enhanced to provide more translation entries and to provide more support for using large pages for translation. It contains three TLBs accessed in parallel. The first TLB is a 16-entry, fully-associative TLB, identical in nature to the small TLB in the instruction MMU. The other two TLBs are both 512-entry, 2-way set-associative caches. Like the large TLB in the instruction MMU, these large data TLBs only store entries for unlocked pages. Unlike the large TLB in the instruction MMU, the large TLBs in the data MMU can be set to any of the four page sizes, although only pages of the same size can accessed/filled at a time (but multiple pages of that size can be handled at once). The two TLBs can be set to either both store pages of the same size, or each store pages of different sizes.

Having the two large TLBs is very important for general use of large pages for translation, in systems that need to map large physical memories. One of the TLBs can be set for large pages (such as 4 MB pages) while the other can be set to the default page size (usually 8 KB pages). With this configuration the processor provides robust support for large pages.

3.1.5 Embedded Cache Unit (Level-2 Unified Cache)

The UltraSPARC IIIi processor supports an on-chip 1 MB, 4-way set-associative Level 2 cache. A separate, 4-way set-associative cache is used to store tags for the L2 cache. Tags are protected by parity checking, date is fully protected with error correcting code (ECC) that allows all single-bit errors to be corrected and double-bit errors to be detected and marked to prevent use.

3.1.6 JBUS Interface Unit

The UltraSPARC IIIi processor communicates with the JIO chip through JBUS. All transactions with the JBUS are routed through the JBUS interface unit. The outgoing control logic arbitrates for issuing transactions and for driving data. The incoming control logic enqueues all transactions issued on the bus and accumulates snoop results from internal caches before driving data on the system bus. The error control logic handles error logging and trap generation.

3.1.7 Memory Controller Unit

The Memory Control Unit (MCU) handles all data transfers between the system and the main memory of the UltraSPARC IIIi processor. The MCU accepts read and write transactions from the ECU and JBU. The local memory supports up to 16 GB of DDR 266 MHz SDRAM. Data transfers between memory and the JBU are handled by the MCU. The local memory consists of two DDR channels each of which are composed of two 72-bit DIMMs. Nine bits of ECC are stored with each 16-bytes of data. The ECC is checked by the MCU when data is read from memory. The MCU also handles the memory refresh and Low Power operation of memory.

A major goal of the MCU is to aggressively reduce memory latencies. Methods to reduce latency include the following:

- Allowing reads to bypass writes while preserving the system bus order
- Reads from the ECU are started speculatively before reaching the system bus
- Holding internal SDRAM banks open to reduce the latency due to row access strobe (RAS)

3.2 Processor Operating Modes

The UltraSPARC IIIi processor operates in various modes.

3.2.1 Privileged Mode

This mode is a **“supervisor”** mode. In this mode, the software is allowed to access both privileged and non-privileged registers and address space identifiers (ASIs). There are certain features of the processor that can be accessed only in privileged mode. Privileged mode execution typically is used by the kernel and operating system.

3.2.2 Non-Privileged Mode

This mode is a “**non-supervisor**” operating mode, in which programs are allowed to access only non-privileged registers and ASIs. If non-privileged software tries to access privileged registers or ASIs, exceptions are generated and handled by the operating system. Non-privileged mode execution is typically used by the application programmers.

3.2.3 Reset and RED_State

The UltraSPARC IIIi processor can be reset using various mechanisms. This section deals with the reset and RED_state for the UltraSPARC IIIi processor.

3.2.3.1 RED_state Characteristics

A processor enters RED_state in one of the following two ways:

- First, by trapping when already at the maximum trap level.
- Second, by setting `PSTATE.RED`.

When the processor enters RED_state, it will clear the DCU Control Register, including enable bits for I-cache, D-cache, I-MMU, D-MMU, and virtual and physical watchpoints.

Note – Exiting RED_state by writing zero to PSTATE.RED in the delay slot of a JMPL is not recommended. A non-cacheable instruction prefetch can be made to the JMPL target, which may be in a cacheable memory area. This condition could result in a bus error on some systems and cause an *instruction_access_error* trap. You can mask the trap by setting the NCEEN bit in the ESTATE_ERR_EN register to zero, but this approach will mask all noncorrectable error checking. Exiting RED_state with DONE or RETRY avoids the problem.

3.2.3.2 Resets

Reset priorities from highest to lowest are power-on resets (POR, hard or soft), externally initiated reset (XIR), watchdog reset (WDR), and software-initiated reset (SIR).

Power-on Reset (Hard Reset)

A Power-on Reset (POR) occurs when the J_POR_L pin is activated and stays asserted until the processor is within its specified operating range. When the J_POR_L pin is active, all other resets and traps are ignored. POR has a trap type of 1 at physical address offset 0x20. Any pending external transactions are canceled.

After POR, software must initialize values of certain registers and state that is unknown after POR. The following bits must be initialized before the caches are enabled:

- In the I-cache, valid bits must be cleared and microtag bits must be set so that each way within a set has a unique microtag value.
- In the D-cache, valid bits must be cleared and microtag bits must be set so that each way within a set has a unique microtag value.
- All L2-cache tags and data

The I-MMU and D-MMU TLBs must also be initialized. The P-cache valid bits must be initialized before any floating-point loads are executed.

Caution – Executing a DONE or RETRY instruction when TSTATE is uninitialized after a POR can damage the chip. The POR boot code should initialize TSTATE<3:0>, using wrpr writes, before any DONE or RETRY instructions are executed.

However, these operations can only be executed in privileged mode. Therefore, user code is not at the risk of damaging the chip.

System Reset (Soft Reset)

A system reset occurs when the J_RST_L pin is activated. When the J_RST_L pin is active, all other resets and traps are ignored. System reset has a trap type of 1 at physical address offset 0x20. Any pending external transactions are canceled.

Note – Memory refresh continues uninterrupted during a system reset. The system interface, L2-cache configuration, and memory controller configuration are preserved across a system reset.

Externally Initiated Reset (XIR)

An XIR is sent to the processor through the XIR transaction on the JBUS. It causes a SPARC-V9 XIR, which has a trap type 3_{16} at physical address offset 0x60. XIR has higher priority than all other resets except Power-on Reset and System Reset.

XIR affects only one processor, rather than the entire system. Memory state, cache state, and most Control Status Register state are unchanged. System coherency is *not* guaranteed to be maintained through an XIR reset. The saved `PC` and `nPC` will only be approximate because the trap is not precise with respect to pipeline state.

Watchdog Reset (WDR) and error_state

The processor enters `error_state` when a trap occurs at `TL` = `MAXTL`.

The processor automatically exits `error_state` using WDR. The processor signals itself internally to take a WDR and sets `TT` = 2. The WDR traps to the address at RSTVaddr + $0x40_{16}$. WDR sets the processor in a state where it is prepared for diagnosis of failures.

WDR affects only one processor, rather than the entire system. `CWP` updates due to window traps that cause watchdog traps are the same as the no watchdog trap case.

Software-Initiated Reset (SIR)

An SIR is initiated by an `SIR` instruction within any processor. This per-processor reset has a trap type 4 at physical address offset 0x80. SIR affects only one processor, rather than the entire system.

RED_state Trap Vector

When the UltraSPARC IIIi processor processes a reset or trap that enters `RED_state`, it takes a trap at an offset relative to the `RED_state` trap vector base address (`RSTVaddr`); the base address is at virtual address `FFFF FFFF F000 0000`$_{16}$, which passes through to physical address `7FF F000 0000`$_{16}$.

3.2.4 Error Handling

The UltraSPARC IIIi processor provides extensive support for detecting and correcting errors. Note that some errors may still be uncorrectable.

3.2.4.1 Error Classes in Severity

The classes of error in order of severity are as follows:

1. **Hardware-corrected errors.** Hardware tries to correct the error automatically. A trap is generated to log the error conditions when the error is corrected to enable the actions for preventive maintenance.

2. **Software-correctable errors.** Hardware does not correct the error automatically. Instead, it invokes a trap requesting the recovery software to correct the error. Corrective actions are expected from the recovery software. If recovery is successful, the system should continue the operation.

3. **Uncorrectable errors.** By its nature the error is uncorrectable, and hardware invokes a trap to signal the occurrence of the error to appropriate recovery software. Depending on the condition under which the error occurs, the system may be able to recover from the error and continue operation. If not, it may be able to isolate the error to a particular process and terminate it. Otherwise, the software should shut down the system gracefully.

4. **Fatal errors.** By its nature, the error indicates either loss of system consistency or a system interconnect protocol error. It is dangerous to continue operation in this situation because of the impending threat of a failure to maintain data integrity. Therefore, upon the detection of the error, the processor generates an error signaling sequence to its interconnect, expecting to be halted/reset by the system. System actions induced by the error signaling sequence are dependent on system implementation.

3.2.4.2 Corrective Actions

Errors are handled by invocation of one of the following actions:

- **Reset-inducing error sequence.** Any fatal error causes the error signaling sequence to induce a system reset. Some errors asynchronous to instruction execution may generate this error signaling sequence.
- **Precise traps.** Most errors detected in the course of an instruction execution generate a precise trap. If the error is hardware correctable, software just logs it. If the error is software correctable, software corrects it before continuing execution. If the error is uncorrectable, software takes appropriate action.
- **Deferred traps.** Some uncorrectable errors requiring immediate attention generate a deferred trap to request software intervention. The recovery software examines the recorded error information to determine the extent of the damage caused by the error. Depending on the observed effect, the system may need to be brought down, or it may continue to run when the effect is isolated within the user program. In any event, the error does not require immediate reset of the system.
- **Disrupting traps.** An error asynchronous to instruction execution generates a disrupting trap to request logging and clearing. The error may already be corrected by hardware and may only require logging. If the error is software correctable, software corrects it before continuing execution. If the error is uncorrectable, software takes appropriate action.

3.2.4.3 Errors Synchronous and Asynchronous to Instruction Execution

Some errors can be detected asynchronously to instruction execution. Other errors are detected in the course of an instruction execution, that is, synchronous to instruction execution. Separate error recording mechanisms are used for synchronous and asynchronous errors.

An error asynchronous to instruction execution is signaled by either a disruption or deferred trap to the processor, or through an error signaling sequence to system hardware which induces a system reset depending on the severity of the error. The errors signalled through a disrupting trap do not directly correspond to an instruction. Traps may or may not be recoverable. Errors signalled are meant to indicate either a loss of system consistency or a protocol error on system interconnect.

An error detected in the course of an instruction execution is signalled through an error trap to the instruction, with additional information recorded in hardware. The trap is either precise or deferred. The program (process) affected by the error should be given a corrected response, or if the error is uncorrectable, the process should be terminated appropriately. Precise traps are used wherever possible.

3.2.5 Debug and Diagnostics Mode

The UltraSPARC IIIi processor provides interfaces for diagnostic access to most internal state of the processor. This is important for diagnosing, and when possible recovering from failures. There are several different diagnostic interfaces. All the diagnostic interfaces are accessible only from software running in privileged mode or from an external system controller in a server. All internal diagnostic and configuration registers are 8-bytes wide, and must be accessed as 8-byte units with 8-byte aligned addresses.

There are a number of diagnostic registers that are mapped to internal ASI registers. These registers are accessed by load and store alternate ASI instructions that specify certain configurations of ASI numbers and virtual addresses. Diagnostic registers are provided for recording various fault conditions as well as important information and state associated with the fault to help diagnosis and possibly recover.

For diagnostic and error recovery in the large memories on chip, such as caches, each element of these memory arrays can be individually read and written. Accesses are performed with load and store alternate ASIs that use specific ASIs that point to the memory array. These accesses can only be done by privileged software.

Special ASI numbers are used for diagnostic accesses to structures where the virtual address is used to specify the portion of the structure to be read. Most structures can be directly read and many structures can also be directly written or quickly cleared.

The UltraSPARC IIIi processor also provides a serial JTAG interface that can be used by a system controller for diagnostics. A system controller can perform a shadow scan where various configuration and diagnostic information is scanned out of the processor without interfering with the operation of the processor. The system controller can also use the JTAG interface to scan in information to configure or control various aspects of the processor.

The JTAG interface also can be used to perform a full scan dump. When a full scan dump is performed, most of the flops in the processor are scanned out through a scan chain. A full scan dump is a destructive action and the processor must be reset after completion of the dump. The full scan provides an important tool for diagnosis of serious failures.

For controlling diagnostics mode, there is a range of configuration registers, which can enable and disable many features of the processor. The configuration registers are only accessible in privileged mode. Some of the configuration registers are implemented as ASRs. These registers are accessible from the RDASR/WRASR interface. Most of the configuration registers are mapped as internal ASI registers. These registers are accessed by load and store alternate ASI instructions that specify certain configurations of ASI numbers and virtual addresses.

CHAPTER 4

Instruction Execution

This chapter focuses on the needs of compiler writers and others who are interested in scheduling instructions to optimize program performance. The chapter discusses the following topics:

- Section 4.1, “Introduction”
- Section 4.2, “Processor Pipeline”
- Section 4.3, “Pipeline Recirculation”
- Section 4.4, “Grouping Rules”
- Section 4.5, “Conditional Moves”
- Section 4.6, “Instruction Latencies and Dispatching Properties”

4.1 Introduction

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

4.1.1 NOP, Neutralized, and Helper Instructions

The distinction between NOP and neutralized instructions is subtle.

4.1.1.1 NOP Instruction

The architected `NOP` instruction is coded as a `SETHI` instruction with the destination register `%g0`. This instruction is groupable in the A0 or A1 pipeline.

4.1.1.2 Neutralized Instruction

Some instructions have no visible effects on the software. They have been de-implemented or assigned to not have an effect if the processor is in a certain mode. These instructions are often referred to as `NOP` instructions, but they are not the same as the `NOP` instruction in that they execute in the pipeline that is assigned to them. These are versions of instructions that have no effect because they only access the `%g0` register and do not have any side effects. Hence, these instructions are functionally neutral.

4.1.1.3 Helper Instructions

Helper instructions are generated by the hardware to help in the execution or re-execution of an instruction. The hardware partitions a single instruction into multiple instructions that flow through the pipeline, consecutively. They have no software visibility and are part of the hardware function of the pipeline.

4.2 Processor Pipeline

The processor pipeline consists of fourteen stages plus an extra stage that is occasionally used by the hardware. The pipeline stages are referred to by the following mnemonic single-letter names and are shown in TABLE 4-1.

TABLE 4-1 Processor Pipeline Stages

Pipeline Stage	Definition
A	**A**ddress generation
P	**P**reliminary Fetch
F	**F**etch instructions from I-cache
B	**B**ranch target computation
I	**I**nstruction group formation
J	**J**: grouping
R	**R**egister access (dispatch/dependency checking stage)

TABLE 4-1 Processor Pipeline Stages *(Continued)*

Pipeline Stage	Definition
E	**E**xecute
C	**C**ache
M	**M**iss detect
W	**W**rite
X	e**X**tend
T	**T**rap
D	**D**one

Rather than executing the instructions in a single pipeline, several separate pipelines are each dedicated to execution of a particular class of instructions. The execution pipelines start after the R-stage of the pipeline. Some instructions take a cycle or two to execute, others take a few cycles within the pipeline. As long as the execution fits within the fixed pipeline depth, execution can in general be fully pipelined. Some instructions have extended execution times that sometimes vary in duration depending on the state of the processor.

The following sections provide a stage-by-stage description of the pipeline. Chapter 3 “UltraSPARC IIIi Processor Architecture Basics” describes the functions of the various execution units. This chapter explains how the pipeline operates the execution units to process the instructions.

FIGURE 4-1 on page 34 illustrates each pipeline stage in detail and the relationship between high level, large architectural structures.

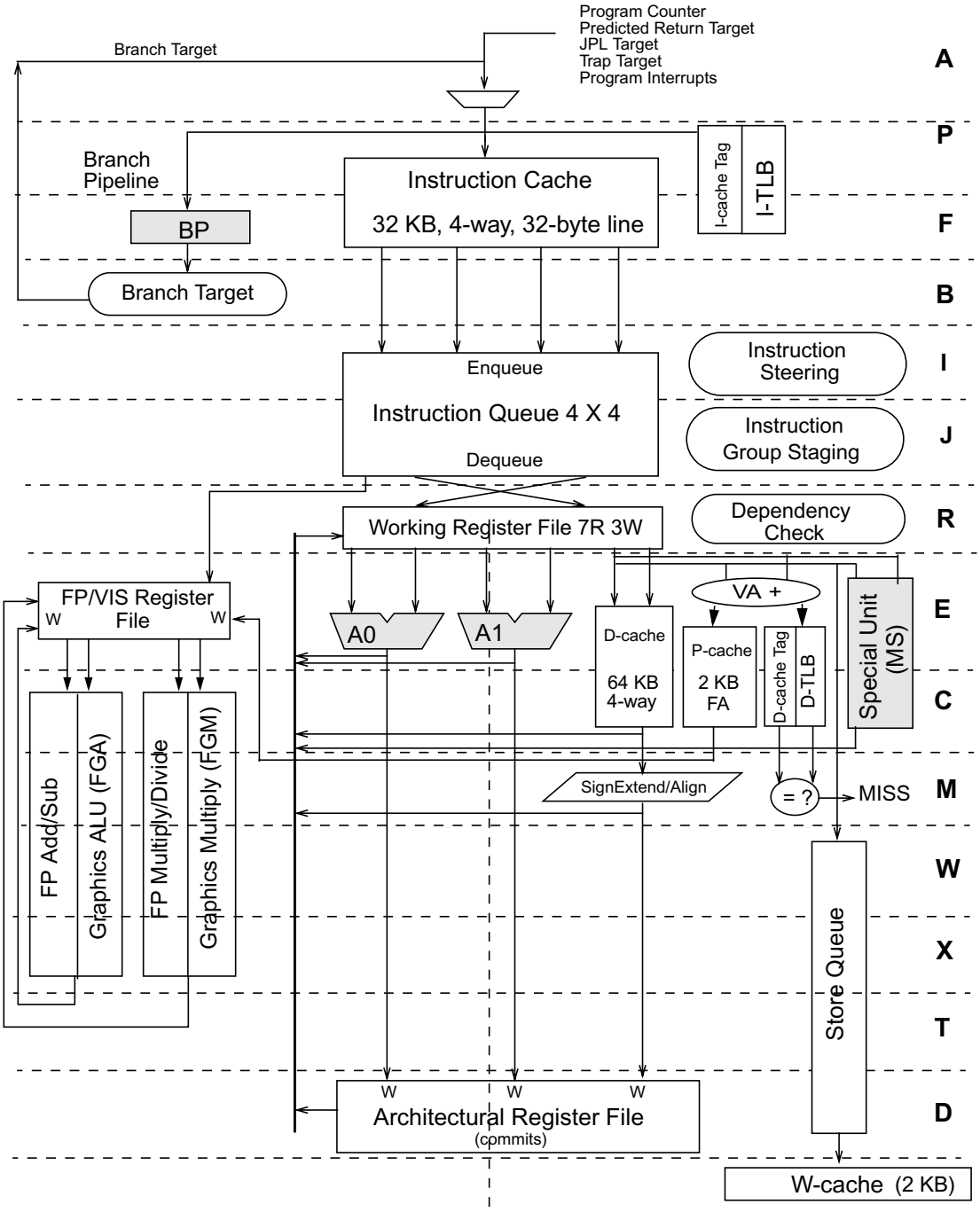


FIGURE 4-1 Instruction Pipeline Diagram

4.2.1 Instruction Dependencies

Instruction dependencies exist in the grouping, dispatching, and execution of instructions.

4.2.1.1 Grouping Dependencies

Up to four instructions can be grouped together for simultaneous dispatch. The number of instructions that can be grouped together depends on the consecutive instructions that are present in the instruction fetch stream, the availability of execution resources (execution units), and the state of the system. Instructions are grouped together to provide superscalar execution of multiple instruction dispatches per clock cycle.

Some instructions are *single instruction group* instructions. These are dispatched by themselves one clock at a time as a single instruction in the group.

Note – Pipeline Recirculation: During recirculation, the recirculation invoking instruction is often re-executed as a single group instruction and often with a helper instruction inserted into the pipeline by the hardware. Even groupable instructions are retried in a single instruction group. See Section 4.3 “Pipeline Recirculation” on page 41 for details.

4.2.1.2 Dispatch Dependencies

Instructions can be held at the R-stage for many different reasons, including:

- Working register operand is not available
- Functional Unit is not available
- Store-load sequence is in progress (atomic operation)

When instructions are held at the dispatch stage, the upper pipeline continues to operate until the instruction buffer is full. At that point, the upper pipeline stalls.

During recirculation, the recirculation invoking instruction is held at the dispatch stage until its execution dependency is resolved.

4.2.1.3 Execution Dependencies

The pipeline assumes all load instructions will hit in a primary cache, allowing the pipeline to operate at full speed. There are two occurences that will recirculate the pipeline:

- D-cache Miss
- Load requires data to be bypassed from an earlier store that has not completed and does not meet the criteria for read-after-write data bypassing

4.2.2 Instruction-Fetch Stages

The instruction-fetch pipeline stages A, P, F, and B are described below.

4.2.2.1 A-stage (Address Generation)

The address stage generates and selects the fetch address to be used by the Instruction Cache (I-cache) in the next cycle. The address that can be selected in this stage for instruction fetching comes from several sources including:

- Sequential PC
- Branch target (from B-stage)
- Trap target
- Interrupt
- Predicted return target
- Jmpl target
- Resolved branch/Jmpl target from execution pipeline

4.2.2.2 P-stage (Preliminary Fetch)

The preliminary fetch stage starts fetching four instructions from the I-cache. Since the I-cache has a two-cycle latency, the P-stage and the F-stage are both used to complete an I-cache access. Although the I-cache has a two-cycle latency, it is pipelined and can access a new set of up to four instructions every cycle. The address used to start an I-cache access is generated in the previous cycle.

The P-stage also accesses the Branch Predictor (BP), which is a small, single-cycle access SRAM whose output is latched at the end of the P-stage. The BP predicts the direction of all conditional branches, based on the PC of the branch and the direction history of the most recent conditional branches.

4.2.2.3 F-stage (Fetch)

The F-stage is used for the second half of the I-cache access. At the end of this stage, up to four instructions from an I-cache line (32-bytes) are latched for decode. An I-cache fetch group is not permitted to cross an I-cache line (32-byte boundary).

4.2.2.4 B-stage (Branch Target Computation)

The B-stage is the final stage of the instruction-fetch pipeline, A-P-F-B. In this stage, the four fetched instructions are first available in a register. The processor analyzes the instructions, looking for Delayed Control Transfer Instructions (`DCTI`) that can alter the path of execution. It finds the first `DCTI`, if any, among the four instructions and computes (if `PC` relative) or predicts (if register based) its target address. If this `DCTI` is predicted taken, the target address is passed to the A-stage to begin fetching from that stream; if predicted not taken, the target is passed on to the CTI queue for use in case of mispredict. Also in the B-stage, the computation of the hit or miss status of the instruction fetch is performed, so that the validity of the four instructions can be reported to the instruction queue.

In the case of an I-cache miss, a request is issued to the L2-cache and all the way out to memory if needed to get the required line. The processor includes an optimization, where along with the line being fetched, the subsequent line (32-bytes) is also returned and placed into the instruction prefetch buffer. A subsequent miss that can get its instructions from the instruction prefetch buffer will behave like a fast miss.

4.2.3 Instruction Issue and Queue Stages

The I-stage and J-stage correspond to the enqueueing and dequeuing of instructions from the instruction queue. The R-stage is where instruction dependencies are resolved.

4.2.3.1 I-stage (Instruction Group Formation)

In the I-stage, the instructions fetched from the I-cache are entered as a group into the instruction queue. The instruction queue is four instructions wide by four instruction groups deep. The instruction may wait in the queue for an arbitrary period of time until all earlier instructions are removed from the queue.

The instructions are grouped to use up to four of the execution pipelines, shown in TABLE 4-2.

TABLE 4-2 Execution Pipelines

Pipeline	Description
A0	Integer ALU pipeline 0
A1	Integer ALU pipeline 1
BR	Branch pipeline
MS	Memory/Special pipeline
FGM	Floating-point/VIS multiply pipeline (with divide/square root pathway)
FGA	Floating-point/VIS add ALU pipeline

4.2.3.2 J-stage (Instruction Group Staging)

In the J-stage, a group of instructions are dequeued from the instruction queue and prepared for being sent to the R-stage. If the R-stage is expected to be empty at the end of the current cycle, the group is sent to the R-stage.

4.2.3.3 R-stage (Dispatch and Register Access)

The integer working register file is accessed during the R-stage for the operands of the instructions (up to three) that have been steered to the A0, A1, and MS pipelines. At the end of the R-stage, results from previous instructions are bypassed in place of the register file operands, if required.

Up to two floating-point or VIS instructions are sent to the Floating-Point/VIS Unit in this stage.

The register and pipeline dependencies between the instructions in the group and the instructions in the execution pipelines are calculated concurrently with the register file access. If a dependency is found, the dependent instruction and any older instruction in the group is held in the R-stage until the dependency is resolved.

4.2.3.4 S-stage (Normally Bypassed)

The S-stage provides a 1-entry buffer per pipeline in cases when the R-stage is not able to take a new instruction.

4.2.4 Execution Pipeline

The execution pipeline contains the E, C, M, W, and X stages.

4.2.4.1 Integer Instruction Execution: E-stage (Execute)

The E-stage is the first stage of the execution pipelines. Different actions are performed in each pipeline.

Integer instructions in the A0 and A1 pipelines compute their results in the E-stage. The instructions include most arithmetic, all shift, and all logical instructions. Their results are available for bypassing to dependent instructions that are in the R-stage, resulting in single-cycle execution for most integer instructions. The A0 and A1 pipelines are the only two sources of bypass results in the E-stage.

Other integer instructions are steered to the MS pipeline and, if necessary, are sent with their operands to the special execution unit in this stage. They can start their execution during the E-stage, but will not produce any results to be bypassed until the C-stage or the M-stage.

Load instructions steered to the MS pipeline start accessing the D-cache or P-cache during the E-stage. The D-cache features Sum Addressed Memory (SAM) decode logic that combines the arithmetic calculation for the virtual address with the row decode of the memory array to reduce look-up time. The virtual address is computed in the E-stage for translation lookaside buffer (TLB) access and possible access to the P-cache.

Floating-point and VIS instructions access the floating-point register file in the E-stage to obtain their operands. At the end of the E-stage, the results from previous completing floating-point/VIS instructions can be bypassed to the E-stage instructions.

Conditional branch instructions in the BR pipeline resolve their directions in the E-stage. Based on their original predicted direction, a *mispredict* signal is computed and sent to the A-stage for possible refetching of the correct instruction stream.

`JMPL` and `RETURN` instructions compute their target addresses in the E-stage of the MS pipeline. The results are sent to the A-stage to start fetching instructions from the target stream.

4.2.4.2 C-stage (Cache)

The D-cache delivers results for doubleword (64-bit) and unsigned word (32-bit) integer loads in the C-stage. The D-TLB access is initiated in the C-stage and proceeds in parallel with the D-cache access. For floating-point loads, the P-cache access is initiated in the C-stage. The results of the D-TLB access and P-cache access are available in the M-stage.

Special instruction unit results are produced at the end of this stage and can be bypassed to waiting dependent instructions in the R-stage—minimum two-cycle latency for SIU instructions. The integer pipelines, A0 and A1, write their results back to the working register file in the C-stage.

The C-stage is the first stage of execution for floating-point and VIS instructions in the FGA and FGM pipelines.

4.2.4.3 M-stage (Miss)

D-cache misses are determined in the M-stage by a comparison of the physical address from the D-TLB to the physical address in the D-cache tags. If the load requires additional alignment or sign extension (such as signed word, all halfword, and all byte loads), it is carried out in this stage, resulting in a three-cycle latency for those load operations. This stage is used for the second execution cycle of floating-point and VIS instructions. Load data is available to the floating-point pipelines in the M-stage.

4.2.4.4 W-stage (Write)

In the W-stage, the MS integer pipeline results are written into the working register file. The W-stage is also used as the third execution cycle for floating-point and VIS instructions. The results of the D-cache miss are available in this stage and the requests are sent to the L2-cache if needed.

4.2.4.5 X-stage (Extend)

The X-stage is the last execution stage for most floating-point operations (except divide and square root) and for all VIS instructions. Floating-point results from this stage are available for bypass to dependent instructions that will be entering the C-stage in the next cycle.

4.2.5 Trap and Done Stages

This section describes the stages that interrupt or complete instruction execution.

The results of operations are bypassed and sent to the working register file. If no traps are generated, then they are successfully pipelined down to the architectural register file and committed. If a trap or recirculation occurs, then the architectural register file (contains committed data) is copied to the working register in preparation for the instructions to be re-executed.

4.2.5.1 T-stage (Trap)

Traps, including floating-point and integer traps, are signalled in this stage. The trapping instruction, and all instructions younger than the trapping instruction must invalidate their results before reaching the D-stage to prevent their results from being erroneously written into the architectural or floating-point register files.

4.2.5.2 D-stage (Done)

Integer results are written into the architectural register file in this stage. At this point, they are fully committed and are visible to any traps generated from younger instructions in the pipeline.

Floating-point results are written into the floating-point register file in this stage. These results are visible to any traps generated from younger instructions.

4.3 Pipeline Recirculation

When a dependency is encountered *in* or *before* the dispatch R-stage, then the pipeline is stalled. Most dependencies, like register or FV dependencies are resolved in the R-stage. When a dependency is encountered *after* the dispatch R-stage, then the pipeline is *recirculated*. Recirculation involves resetting the PC back to the recirculation invoking instruction. Instructions older than the dependent instruction continue to execute. The offending instructions and all younger instructions are recirculated. The offending instruction is retried and goes through the entire pipeline again.

Upon recirculation, the instruction responsible for the recirculation becomes a single-group instruction that is held in the R-stage until the dependency is resolved.

Load Instruction Dependency

In the case of a load instruction miss in a primary cache, the pipeline recirculates and the load instruction waits in the R-stage. When the data is returned in the D-cache *fill buffer*, the load instruction is dispatched again and the data is provided to the load instruction from the fill buffer. The pipeline logic inserts two helpers behind the load instruction to move the data in the fill buffer to the D-cache. The instruction in the instruction fetch stream, after the load instruction, follows the helpers and will re-group with younger instructions, if possible.

4.4 Grouping Rules

Grouping rules are made before going into R-stage. A *group* is a collection of instructions with no resource constraints that will limit them from being executed in parallel.

Instruction grouping rules are necessary for the following reasons:

- Maintain the instruction execution order
- Each pipeline runs a subset of instructions
- Resource dependencies, data dependencies, and multicycle instructions require helpers (NOPs) to maintain the pipelines

Before continuing, the following terms that apply to instructions are defined as:

break-before: The instruction will always be the first instruction of a group.

break-after: The instruction will always be the last instruction of a group.

single-instruction group (SIG): The instruction will not be issued with any other instructions in the group. (SIG is sometimes shortened herein to “single-group.”)

instruction latency: The number of processor cycles after dispatching an instruction from the R-stage that a following data-dependent instruction can dispatch from the R-stage.

blocking, multicycle: The instruction reserves one or more of the execution pipelines for more than one cycle. The reserved pipelines are not available for other instructions to issue into until the blocking, multicycle instruction completes.

4.4.1 Execution Order

Rule: Within the R-stage, some of the instructions can be dispatched and others cannot. If an instruction is younger than an instruction that is not able to dispatch, then the younger instruction will not be dispatched.

“Younger” and “older” refer to instruction order within the program. The instruction that comes first in the program order is the older instruction.

4.4.2 Integer Register Dependencies to Instructions in the MS Pipeline

Rule: If a source register operand of an instruction in the R-stage matches the destination register of an instruction in the MS pipeline’s E-stage, then the instruction in the R-stage may not proceed.

The MS pipeline has no E-stage bypass.

If an operand of an instruction in the R-stage matches the destination register of an instruction in the MS pipeline’s C-stage, then the instruction in the R-stage may not proceed if the instruction in the MS pipeline’s C-stage does not generate its data until the M-stage. For example, LDSB does not have the load data until the M-stage, but LDX has its data in the C-stage. Thus, LDX would not cause an interlock, but LDSB would.

Most instructions in the MS pipeline have their data by the M-stage, so there is no dependency check on the MS pipeline’s M-stage destination register. In the case of multicycle MS instructions, the data is always available by the M-stage as the last of the instructions passes through the pipeline.

4.4.2.1 Helpers

Sometimes an instruction, as part of its operation, requires multiple flows in the pipeline. These extra flows after the initial instruction flow are called *helper cycles*. The only pipeline that executes such instructions is the MS pipeline. If an instruction requires a helper, that helper is generated in the R-stage. The help generation logic generates as many helpers as the instruction requires.

Most of the time the logic determines the number of helpers by examining the opcode. However, some recirculate cases run the recirculated instruction differently than the original flow down the pipeline, and some instructions, like integer multiply and divide, require variable numbers of helpers. Some helper counts are determined by I/O and memory controllers and system devices. For example, the D-cache unit requires helpers as it completes an atomic memory instruction.

Rule: Instructions requiring helpers are always break-after.

There can be no instruction in a group that is younger than an instruction that requires helpers. Another way of saying this is “an instruction that requires helpers will be the youngest in its group.” This rule preserves the in-order execution of the integer instructions.

Rule: Helpers block the pipeline.

Helpers block the pipeline from executing other instructions; thus, instructions with helpers are blocking.

Rule: Helpers are always single-group.

A helper cycle is always alone in a group. No other instruction will ever be dispatched from the R-stage if there is a helper cycle in the R-stage.

4.4.3 Integer Instructions Within a Group

Rule: Integer instructions within a group are not allowed to write the same destination register.

By not writing the same destination register at the same time, the bypass logic is simplified as well as the register file write-enable determination and potential Write After Write (WAW) errors. The instructions are break-before second destination is written.

This rule applies only to integer instructions writing integer registers. Floating-point instructions and floating-point loads (done in the integer A0, A1, and MS pipelines) can be grouped so that two or more instructions in the same group can write the same floating-point destination register. Instruction age is associated with each instruction. The write from an older instruction is not visible, but the execution of the instruction might still cause a trap and set condition codes.

There are no special rules concerning integer instructions that set condition codes and integer branch instructions.

Integer instructions that set condition codes can be grouped in any way with integer branches. In fact, any number instructions that set condition codes are allowed in any order relative to the branch, provided that they do not violate any other rules. No special rules apply to this specific case. Integer instructions that set condition codes in the A1 and A0 pipelines can compute a taken/untaken result in the E-stage, which is the same stage in which the branch is evaluating the correctness of its prediction. The control logic guarantees that the correct condition codes are used in the evaluation.

4.4.4 Same-Group Bypass

Rule: Same-group bypass is disallowed, except store instructions.

The group bypass rule states that no instruction can bypass its result to another instruction in the same group. The one exception to this rule is *store*. A store instruction can get its store data (`rd`), but not its address operands (`rs1`, `rs2`), from an instruction in the same group.

4.4.5 Floating-Point Unit Operand Dependencies

4.4.5.1 Latency and Destination Register Addresses

Floating-point operations have longer latencies than most integer instructions. Moreover, floating-point square root and divide instructions have varying latencies depending on whether the operands are single precision or double precision. All the floating-point instruction latencies are four clock cycles (except for floating-point divide and square root and `PDIST` → `PDIST`).

The operands for floating-point operations can either be single precision (32-bit) or double precision (64-bit). Sixteen of the double precision registers are each made up of two single precision registers. An operation using one of these double precision registers as a source operand may be dependent on an earlier single precision operation producing part of the register value. Similarly, an operation using one of the single precision registers as a source operands may be dependent on an earlier double precision operation, a part of which may produce the single precision register value.

4.4.5.2 Grouping Rules for Floating-Point Instructions

Rule: Floating-point divide/square root is busy.

The floating-point divide/square root unit is a non-pipelined unit. The Integer Execution Unit sets a busy bit for each of the two stages of the divide/square root and depends on the FGU to clear them. Only the first part of the divide/square root is considered to have a busy unit; therefore, once the first part is complete, a new floating-point divide/square root operation can be started.

Rule: Floating-point divide/square root needs a write slot in the FGM pipeline.

In the stage in which a divide/square root is moved from the first part to the last part, instructions must not be issued to the FGM pipeline. This constraint provides the write slot in the FGM pipeline so the divide/square root can write the floating-point register file.

Rule: Floating-point store is dependent on floating-point divide/square root.

The floating-point divide/square root unit has a latency longer than the normal pipeline. As a result, if a floating-point store depends on the result of a floating-point divide/square root, then the floating-point store instruction may not be dispatched until the floating-point divide/square root instruction has completed.

4.4.5.3 Grouping Rules for VIS Instructions

Rule: Graphics Status Register (GSR) Write instructions are break-after.

The `SIAM`, `BMASK`, and `FALIGNADDR` instructions write the `GSR`. The `BSHUFFLE` and `FALIGNDATA` instructions read the `GSR` in their operation. Because of the `GSR` write latency, a `GSR` reader cannot be in the same group as a `GSR` writer unless the `GSR` reader is older than the `GSR` writer. The simplest solution to this dependency is to make all `GSR` write instructions break-after.

Note – The `WRGSR` instruction is not included in this rule as a special case. The `WRGSR` instruction is already break-after by virtue of being a `WRASR` instruction.

4.4.5.4 PDIST Special Cases

`PDIST`-to-dependent-`PDIST` is handled as a special case with one-cycle latency. `PDIST` latency to any other dependent operation is a four-cycle latency. In addition, a `PDIST` cannot be issued if there is `ST`, block store (`BST`), or partial store instruction in the M-stage of the pipeline. `PDIST` issue is delayed if there is a store type instruction two groups ahead of it.

4.4.6 Grouping Rules for Register-Window Management Instructions

Rule: Window changing instructions are single-group.

The window changing instructions SAVE, RESTORE, and RETURN are all single-group instructions. These instructions are never grouped with any other instruction. This rule greatly simplifies the tracking of register file addresses.

Rule: Window changing instructions force bubbles after.

The window changing instructions SAVE, RESTORE, and RETURN also force a subsequent pipeline bubble. A bubble is distinct from a helper cycle in that there is nothing valid in the pipeline within a bubble. During the bubble, control logic transfers the new window from the Architectural Register File (ARF) to the Working Register File (WRF).

Rule: FLUSHW is single-group.

To simplify the Integer Execution Unit's handling of the register file window flush, the FLUSHW instruction is single-group.

Rule: SAVED and RESTORED are single-group.

To simplify the Integer Execution Unit's window tracking, SAVED and RESTORED are single-group instructions.

4.4.7 Grouping Rules for Reads and Writes of the ASRs

Rule: Write ASR and Write PR instructions are single-group.

WRASR and WRPR are always the youngest instructions in a group. This case prevents problems with an instruction being dependent on the result of the write, which occurs late in the pipeline.

Rule: Write ASR and Write PR force seven bubbles after.

To guarantee that any instruction that starts in the R-stage is started with the most up-to-date status registers, WRASR and WRPR force bubbles after they are dispatched. Thus, if a WRASR or a WRPR instruction is in the pipeline anywhere from the E-stage to the T-stage, no instructions are dispatched from the R-stage (bubbles are forced in).

Rule: Read ASR and Read PR force up to six bubbles before (break-before multicycle).

Many instructions can update the ASRs and PRs. Therefore, if an RDASR or RDPR instruction is in the R-stage and any valid instruction is in the integer pipelines from the E-stage to the X-stage, the UltraSPARC IIIi processor does not allow the RDASR and RDPR instructions to be dispatched. Instead, all pipeline states must wait to write the ASRs and privileged registers and then read them.

4.4.8 Grouping Rules for Other Instructions

Rule: Block Load (BLD) and Block Store (BST) are single-group and multicycle.

For simplicity in the Integer Execution Unit and memory system, BLD and BST are single-group instructions with helpers.

Rule: `FLUSH` is single-group and seven bubbles after.

To simplify the Instruction Issue Unit and Integer Execution Unit, the `FLUSH` instruction is single-group. This makes instruction cancellation and issue easier. `FLUSH` is held in the R-stage until the store queue and the pipeline from E-stage through D-stage is empty.

Rule: `MEMBAR` (`#Sync`, `#Lookaside`, `#StoreLoad`, `#Memissue`) is single-group.

To simplify the Integer Execution Unit and memory system, `MEMBAR` is a single-group instruction. `MEMBAR` will not dispatch until the memory system has completed necessary transactions.

Rule: Software-initiated reset (`SIR`) is single-group.

For simplicity, `SIR` is a single-group instruction.

Rule: Load FSR (`LDFSR`) is single-group and forces seven bubbles after.

For simplicity, `LDFSR` is a single-group instruction.

Rule: `DONE` and `RETRY` are single-group.

`DONE` and `RETRY` instructions are dispatched as a single-group.

Rule: `DONE` and `RETRY` force seven bubbles after.

`DONE` and `RETRY` are typically used to return from traps or interrupts and are known as *trap exit instructions.*

It takes a few cycles to properly restore the pre-trap state and the working register file from the architectural register file, so bubbles are forced after the trap exit instructions to provide the cycles to do it all. A new instruction is not accepted until the trap exit instruction leaves the pipeline (also known as D + 1).

4.5 Conditional Moves

The compiler needs to have a detailed model of the implementation of the various conditional moves so it can optimally schedule code. TABLE 4-3 describes the implementation of the five classes of SPARC-V9 conditional moves in the pipeline. FADD and ADD instructions (shaded rows) are also described as a reference for comparison with the conditional move instructions.

TABLE 4-3 SPARC-V9 Conditional Moves

Instruction	RD Latency	Pipelines Used	Busy Cycles	Groupable	Dependency
FMOVicc	3 cycles	FGA and BR	1	Yes	icc - 0
FMOVfcc	3 cycles	FGA and BR	1	Yes	fcc - 0
FMOVr	3 cycles	FGA and MS	1	Yes	N/A
FADD	4 cycles	FGA	1	Yes	N/A
ADD	1 cycle	A0 or A1	1	Yes	N/A
MOVcc	2 cycles	MS and BR	1	Yes	icc - 0
MOVR	2 cycles	MS and BR	1	Yes	N/A

Where:

RD Latency — The number of processor cycles until the destination register is available for bypassing to a dependent instruction.

Pipes Used — The pipeline that the instruction uses when it is issued. The pipelines are shown in TABLE 4-2.

Busy Cycles — The number of cycles that the pipelines are not available for other instructions to be issued. A value of one signifies a fully pipelined instruction.

Groupable — Whether instructions using pipelines, other than those used by the conditional move, can be issued in the same cycle as the conditional move.

{i,f}CC Dependency — The number of cycles that a CC setting instruction must be scheduled ahead of the conditional move in order to avoid incurring pipeline stall cycles.

4.6 Instruction Latencies and Dispatching Properties

In this section, a machine description is given in the form of a table (TABLE 4-5 on page 50) dealing with dispatching properties and latencies of operations. The static or nominal properties are modelled in the following terms (columns in TABLE 4-5 on page 50), which are discussed below:

- Latencies
- Blocking properties in dispatching
- Pipeline resources (A0, A1, FGA, FGM, MS, BR)
- Break rules in grouping (before, after, single-group)

The pipeline assumes the primary cache will be accessed. The dynamic properties, such as the effect of a cache miss and other conditions, are not described here.

4.6.1 Latency

In the Latency column of TABLE 4-5 on page 50, latencies are minimum cycles at which a dependent operation (consumer) can be dispatched, relative to the producer operation, without causing a dependency stall or instructions to hold back in the R-stage to execute.

Operations like `ADDcc` produce two results, one in the destination register and another in the condition codes. For such operations, latencies are stated as a pair x,y, where x is for the destination register dependence and y is for the condition code.

A zero latency implies that the producer and consumer operations may be grouped together in a single group, as in `{SUBcc, BE %icc}`.

Operations like `UMUL` have different latencies, depending on operand values. These are given as a range, min–max, for example, 6 – 8 in `UMUL`. Operations like `LDFSR` involve waiting for a specified condition. Such cases are described by footnotes and a notation like 32+ for `CASA` (meaning at least 32 cycles).

Cycles for branch operations (like `BPcc`) give the dispatching cycle of the retiring target operation relative to the branch. A pair of numbers, for example 0, 8, is given, depending on the outcome of a branch prediction, where 0 means a correct branch prediction and 8 means a mispredicted case.

Special cases, such as `FCMP(s,d)`, in which latencies depend on the type of consuming operations, are described in footnotes (bracketed, for example, [1]).

4.6.2 Blocking

The Blocking column of TABLE 4-5 gives the number of clock cycles that the dispatch unit waits before issuing another group of instructions. Operations like FDIVd (MS pipeline) have limited blocking property; that is, the blocking is limited to the time before another instruction that uses MS pipeline can be dispatched. Such cases are noted with footnotes. All pipelines block instruction dispatch when an instruction is targeted to them, but they are not ready for another instruction to be pipelined-in.

4.6.3 Pipeline

The Pipeline column of TABLE 4-5 specifies the resource usage. Operations like MOVcc require more than one resource, as designated by the notation MS and BR. The operation LDF can dispatch to either MS, A0, or A1 as indicated.

4.6.4 Break and SIG

Grouping properties are given in columns Break and SIG (single-instruction group). In the Break column an entry can be "Before," meaning that this operation causes a break in a group so that the operation starts a new group. Operations like RDCCR require dispatching to be stalled until all operations in flight are completed (reach D-stage); in such cases, details are provided in a footnote reference in the Break column.

Operations like ALIGNADDR must be the last in an instruction group, causing a break in the group of type "After."

Certain operations are not groupable and therefore are issued in single-instruction groups. A break "before" and "after" are implied for non-groupable instructions.

TABLE 4-5 UltraSPARC IIIi Processor Instruction Latencies and Dispatching Properties *(1 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
ADD	1		A0 or A1		
ADDcc	1, 0 [1]		A0 or A1		
ADDC	5	4	MS		Yes
ADDCcc	6, 5 [2]	5	MS		Yes
ALIGNADDR	2		MS	After	
ALIGNADDRL	2		MS	After	
AND	1		A0 or A1		

TABLE 4-5 UltraSPARC IIIi Processor Instruction Latencies and Dispatching Properties *(2 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
ANDcc	1, 0 [1]		A0 or A1		
ANDN	1		A0 or A1		
ANDNcc	1, 0 [1]		A0 or A1		
ARRAY(8,16,32)	2		MS		
Bicc[D]	0, 8 [3]	0, 5 [4]	BP		
BMASK	2		MS	After	
BPcc	0, 8 [3]	0, 5 [4]	BP		
BPR	0, 8 [3]	0, 5 [4]	BP and MS		
BSHUFFLE	3		FGA		Yes
CALL *label*	0-3 [5]		BP and MS		
CASA	32+	31+	MS	After	
CASXA	32+	31+	MS	After	
DONE[P]	7	Yes	BP and MS		Yes
EDGE(8,16,32){L}	5	4	MS		Yes
EDGE(8,16,32)N	2		MS		
EDGE(8,16,32)LN	2		MS		
FABS(s,d)	3		FGA		
FADD(s,d)	4		FGA		
FALIGNDATA	3		FGA		
FANDNOT1{s}	3		FGA		
FANDNOT2{s}	3		FGA		
FAND{s}	3		FGA		
FBPfcc			BP		
FBfcc[D]			BP		
FCMP(s,d)	1, 5 [6]		FGA		
FCMPE(s,d)	1, 5 [6]		FGA		
FCMPEQ(16,32)	4		MS and FGA		
FCMPGT(16,32)	4		MS and FGA		
FCMPLE(16,32)	4		MS and FGA		
FCMPNE(16,32)	4		MS and FGA		
FDIVd	20 (14) [6]	17 (11) [7]	FGM		
FDIVs	17 (14) [6]	14 (11) [7]	FGM		
FEXPAND	3		FGA		
FiTO(s,d)	4		FGA		

TABLE 4-5 UltraSPARC IIIi Processor Instruction Latencies and Dispatching Properties *(3 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
FLUSH	8	7	BP and MS	Before [8]	Yes
FLUSHW		Yes	MS		Yes
FMOV(s,d)	3		FGA		
FMOV(s,d)cc	3		FGA and BP		
FMOV(s,d)r	3		FGA and MS		
FMUL(s,d)	4		FGM		
FMUL8(,SU,UL)x16	4		FGM		
FMUL8x16(AL,AU)	4		FGM		
FMULD8(SU,UL)x16	4		FGM		
FNAND{s}	3		FGA		
FNEG(s,d)	3		FGA		
FNOR{s}	3		FGA		
FNOT(1,2){s}	3		FGA		
FONE{s}	3		FGA		
FORNOT(1,2){s}	3		FGA		
FOR{s}	3		FGA		
FPACK(FIX, 16,32)	4		FGM		
FPADD(16, 16s, 32, 32s)	3		FGA		
FPMERGE	3		FGA		
FPSUB(16, 16s, 32, 32s)	3		FGA		
FsMULd	4		FGM		
FSQRTd	29 (14) [6]	26 (11) [7]	FGM		
FSQRTs	23 (14) [6]	20 (11) [7]	FGM		
FSRC(1,2){s}	3		FGA		
F(s,d)TO(d,s)	4		FGA		
F(s,d)TOi	4		FGA		
F(s,d)TOx	4		FGA		
FSUB(s,d)	4		FGA		
FXNOR	3		FGA		
FXOR{s}	3		FGA		
FxTO(s,d)	4		FGA		
FZERO{s}	3		FGA		
ILLTRAP			MS		
JMPL *reg*,%o7	0-4, 9-10 [9]	0-3, 8-9	MS and BP		

TABLE 4-5 UltraSPARC IIIi Processor Instruction Latencies and Dispatching Properties *(4 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
JMPL %i7+8,%g0	3-5, 10-12 [10]	2-4, 9-11	MS and BP		
JMPL %o7+8, %g0	0-4, 9 [11]	0-3, 8	MS and BP		
LDD^{D}	2	Yes	MS	After	
LDDA^{D}	2	Yes	MS	After	
LDDF{A}	3		MS, A0, or A1		
LDF{A}	3		MS, A0, or A1		
LDFSR^{D}	[22]	Yes	MS		Yes
LDSB{A}	3		MS		
LDSH{A}	3		MS		
LDSTUB{A}	31+	30+	MS	After	
LDSW{A}	3		MS		
LDUB{A}	3		MS		
LDUH{A}	3		MS		
LDUW{A}	2		MS		
LDX{A}	2		MS		
LDXFSR	[22]	Yes	MS		Yes
MEMBAR #LoadLoad		[12]	MS		Yes
MEMBAR #LoadStore		[12]	MS		Yes
MEMBAR #Lookaside		[13]	MS		Yes
MEMBAR #MemIssue		[13]	MS		Yes
MEMBAR #StoreLoad		[13]	MS		Yes
MEMBAR #StoreStore		[12]	MS		Yes
MEMBAR #Sync		[14]	MS		Yes
MOVcc	2		MS and BP		
MOVfcc	2		MS and BP		
MOVr	2		MS		
MULScc	6, 5 [2]	5	MS		Yes
MULX	6-9	5-8	MS	After	
NOP	na		MS		
OR	1		A0 or A1		
ORcc	1, 0 [1]		A0 or A1		
ORN	1		A0 or A1		
ORNcc	1, 0 [1]		A0 or A1		
PDIST	4		FGM		

TABLE 4-5 UltraSPARC IIIi Processor Instruction Latencies and Dispatching Properties *(5 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
POPC	emulated				
PREFETCH{A}			MS		
RDASI	4		MS	Before [15]	
RDASR	4		MS	Before [15]	
RDCCR	4		MS	Before [15]	
RDDCR^{P}					
RDFPRS	4		MS	Before [15]	
RDPC	4		MS	Before [15]	
RDPR	4		MS	Before [15]	
RDSOFTINT^{P}					
RDTICK	4		MS	Before [15]	
RDY^{D}	4		MS	Before [15]	
RESTORE	2	1	MS	Before [16]	Yes
RESTORED^{P}			MS		Yes
RETRY^{P}	2	Yes	MS and BP	After	
RETURN	2, 9 [17]	1, 8	MS and BP	Before [18]	Yes
SAVE	2	1	MS	Before [19]	Yes
SAVED^{P}	2	Yes	MS		Yes
SDIV	39	38	MS	After	
SDIV{cc}^{D}	40, 39 [2]	39	MS	After	
SDIVX	71	70	MS	After	
SETHI	1		A0 or A1		
SHUTDOWN	[23]	NOP	MS	NOP	
SIAM		Yes	MS		Yes
SIR		Yes	BP and MS		Yes
SLL{X}	1		A0 or A1		
SMUL^{D}	6-7	5-6	MS	After	
SMULcc^{D}	7-8, -6-7 [2]	6-8	MS	After	
SRA{X}	1		A0 or A1		
SRL{X}	1		A0 or A1		
STB{A}			MS		
STBAR^{D}	[20]		MS		Yes
STD{A}^{D}		2	MS		Yes
STDF{A}			MS		

TABLE 4-5 UltraSPARC IIIi Processor Instruction Latencies and Dispatching Properties *(6 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
STF{A}			MS		
STFSR[P]		9	MS	Before [21]	Yes
ST(H,W,X){A}			MS		
STXFSR		9	MS	Before [21]	Yes
SUB	1		A0 or A1		
SUBcc	1, 0 [1]		A0 or A1		
SUBC	5	4	MS		Yes
SUBCcc	6, 5 [2]	5	MS		Yes
SWAP{A}	31+	30+	MS	After	
TADDcc	5	Yes	MS		Yes
TSUBcc	5	Yes	MS		Yes
Tcc			BR and MS		
UDIV[D]	40	39	MS	After	
UDIVcc[D]	41, 40 [2]	40	MS	After	
UDIVX	71	70	MS	After	
UMUL[D]	6-8	5-7	MS	After	
UMULcc[D]	7-8, 6-7 [2]	6-8	MS	After	
WRASI		16	BR and MS		Yes
WRASR		7	BR and MS		Yes
WRCCR		7	BR and MS		Yes
WRFPRS		7	BR and MS		Yes
WRPR[P]		7	BR and MS		Yes
WRY[D]		7	BR and MS		Yes
XNOR	1		A0 or A1		
XNORcc	1, 0 [1]		A0 or A1		
XOR	1		A0 or A1		
XORcc	1, 0 [1]		A0 or A1		

1. These operations produce two results: destination register and condition code (%icc, %xcc). The latency is one in the former case and zero in the latter case. For example, SUBcc and BE %icc are grouped together (zero latency).
2. These operations produce two results: destination register and condition code (%icc, %xcc). The latency is given as a pair of numbers —*m*, *n* — for the register and condition code, respectively. When latencies vary in a range, such as in UMULcc, this range is indicated by pair– pair.
3. Latency is x, y for correct, incorrect branch prediction. It is measured as the difference in the dispatching cycle of the retiring target instruction and that of the branch.

4. Blocking cycles are x,y for correct, incorrect branch prediction. They are measured as the difference in the dispatching cycle of instruction in the delay slot (or target, if annulled) that retires and that of the branch.
5. Native `Call` and `Link` with immediate target address (label).
6. Latency in parentheses applies when operands involve IEEE special values (`NaN`, `INF`), including zero and illegal values.
7. Blocking is limited to another `FD` operation in succession; otherwise, it is unblocking. Blocking cycles in parentheses apply when operands involve special holding and illegal values.
8. Dispatching stall (7+ cycles) until all stores in flight retire.
9. 0–4 if predicted true; 9–10 if mispredicted.
10. Latency is taken to be the difference in dispatching cycles from `jmpl` to target operation, including the effect of an operation in the delay slot. Blocking cycles thus may include cycles due to restore in the delay slot. In a given pair x,y, x applies when predicted correctly and y when predicted incorrectly. Each x or y may be a range of values.
11. 0–4 if predicted true; 9 if mispredicted.
12. This `MEMBAR` has NOP semantics, since the ordering specified is implicitly done by processor (memory model is TSO).
13. All operations in flight complete as in `MEMBAR #Sync`.
14. All operations in flight complete.
15. Issue stalls a minimum of 7 cycles until all operations in flight are done (get to D-stage).
16. Dispatching stalls until previous save in flight, if any, reaches D-stage.
17. 2 if predicted correctly, 9 otherwise. Similarly for blocking cycles.
18. Dispatching stalls until previous restore in flight, if any, reaches D-stage.
19. Dispatching stalls until previous restore in flight, if any, reaches D-stage.
20. Same as `MEMBAR #StoreStore`, which is NOP.
21. Dispatching stalls until all FP operations in flight are done.
22. Wait for completion of all FP operations in flight.
23. The Shutdown instruction is not implemented. The instruction is neutralized and appears as a NOP to software (no visible effects.

SECTION III

Execution Environment

CHAPTER 5

Data Formats

The processor recognizes the following fundamental data types:

- Signed integer: 8, 16, 32, and 64 bits
- Unsigned integer: 8, 16, 32, and 64 bits
- VIS Instruction data formats: pixel (32 bits), fixed16 (64 bits), and fixed32 (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; deprecated)
- Doubleword: 64 bits (deprecated in favor of Extended word)
- 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. In tagged words, the least significant two bits are treated as a tag; the remaining 30 bits are treated as a signed integer.

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

- Signed Integer Double
- Unsigned Integer Double
- Floating-Point, Double-Precision
- Floating-Point, Quad-Precision

5.1 Integer Data Formats

The processor supports the following integer data formats:

- Signed integer
- Unsigned integer
- Tagged integer word

5.1.1 Integer Data Value Range

TABLE 5-1 describes the width and ranges of the signed, unsigned, and tagged integer data formats.

TABLE 5-1 Signed Integer, Unsigned Integer, and Tagged Integer Format Ranges

Data Type	Width (bits)	Range	
		Lower	Upper
Signed integer byte	8	-2^7	$2^7 - 1$
Signed integer halfword	16	-2^{15}	$2^{15} - 1$
Signed integer word	32	-2^{31}	$2^{31} - 1$
Signed integer tagged word	32	-2^{29}	$2^{29} - 1$
Signed integer double word	64	-2^{63}	$2^{63} - 1$
Signed extended integer	64	-2^{63}	$2^{63} - 1$
Unsigned integer byte	8	0	$2^8 - 1$
Unsigned integer halfword	16	0	$2^{16} - 1$
Unsigned integer word	32	0	$2^{32} - 1$
Unsigned integer tagged word	32	0	$2^{30} - 1$
Unsigned integer double word	64	0	$2^{64} - 1$
Unsigned extended integer	64	0	$2^{64} - 1$

5.1.2 Integer Data Alignment

TABLE 5-2 describes the memory and register alignment for integer data.

TABLE 5-2 Integer Data Alignment

Subformat Type	Width	Subformat Field	Required Address Alignment	Memory Address (Big-endian)	Register Number Alignment	Register Number
SB	*B (byte)*	`signed_byte_integer`<7:0>	None	*n*	Any	*r*
UB		`unsigned_byte_integer`<7:0>				
SH	*H (halfword)*	`signed_halfwd_integer`<7:0>	0 **mod** 2	*n*	Any	*r*
UH		`unsigned_halfwd_integer`<7:0>				
SW	*W (word)*	`signed_word_integer`<7:0>	0 **mod** 4	*n*	Any	*r*
UW		`unsigned_word_integer`<7:0>				
SD-0	*D (double word)*	`signed_dbl_integer`<63:32>	0 **mod** 8	*n*	0 **mod** 2	*r*
UD-0		`unsigned_dbl_integer`<63:32>				
SD-1		`signed_dbl_integer`<31:0>	4 **mod** 8	*n + 4*	1 **mod** 2	*r + 1*
UD-1		`unsigned_dbl_integer`<31:0>				
SX	*X (extended word)*	`signed_ext_integer`<63:0>	0 **mod** 8	*n*	—	*r*
UX		`unsigned_ext_integer`<63:0>				

The data types are illustrated in the following subsections.

5.1.3 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 extended integer

5.1.3.1 Signed Integer Byte

FIGURE 5-1 illustrates the signed integer byte data format.

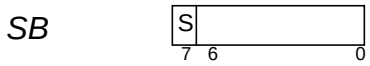


FIGURE 5-1 Signed Integer Byte Data Format

5.1.3.2 Signed Integer Halfword

FIGURE 5-2 illustrates the signed integer halfword data format.

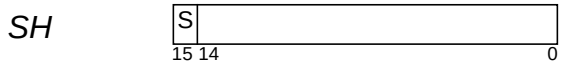


FIGURE 5-2 Signed Integer Halfword Data Format

5.1.3.3 Signed Integer Word

FIGURE 5-3 illustrates the signed integer word data format.

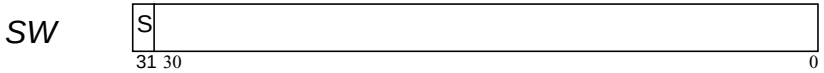


FIGURE 5-3 Signed Integer Word Data Format

5.1.3.4 Signed Integer Double

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

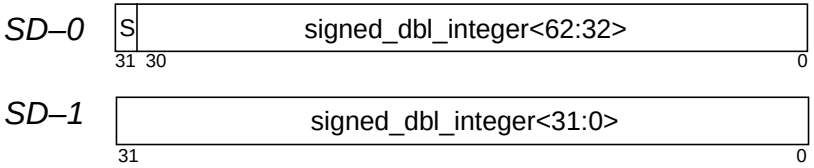


FIGURE 5-4 Signed Integer Double Data Format

5.1.3.5 Signed Extended Integer

FIGURE 5-5 illustrates the signed extended integer (SX) data format.

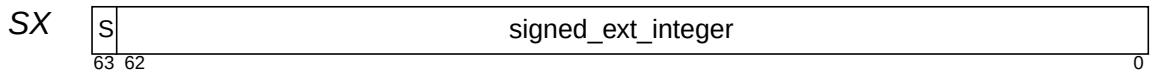


FIGURE 5-5 Signed Extended Integer Data Format

5.1.4 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 extended integer

5.1.4.1 Unsigned Integer Byte

FIGURE 5-6 illustrates the unsigned integer byte data format.

FIGURE 5-6 Unsigned Integer Byte Data Format

5.1.4.2 Unsigned Integer Halfword

FIGURE 5-7 illustrates the unsigned integer halfword data format.

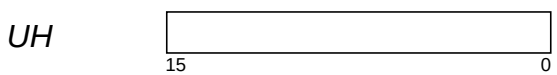


FIGURE 5-7 Unsigned Integer Halfword Data Format

5.1.4.3 Unsigned Integer Word

FIGURE 5-8 illustrates the unsigned integer word data format.

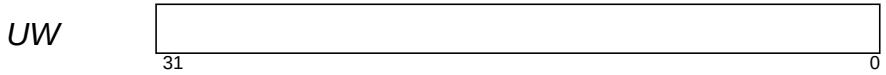


FIGURE 5-8 Unsigned Integer Word Data Format

5.1.4.4 Unsigned Integer Double

FIGURE 5-9 illustrates both components (UD-0 and UD-1) of the unsigned integer double data format.

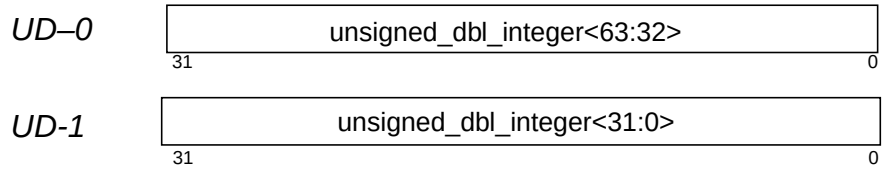


FIGURE 5-9 Unsigned Integer Double Data Format

5.1.4.5 Unsigned Extended Integer

FIGURE 5-10 illustrates the unsigned extended integer (UX) data format.

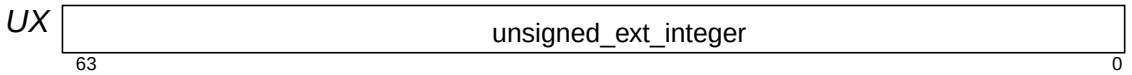


FIGURE 5-10 Unsigned Extended Integer Data Format

5.1.5 Tagged Word

The Tagged word data format is similar to the unsigned word format except for a 2-bit field in the two LSB positions. Bit 31 is the overflow bit.

FIGURE 5-11 illustrates the tagged word data format.

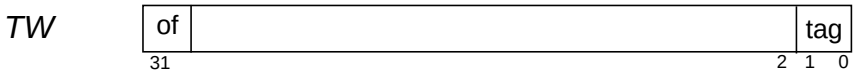


FIGURE 5-11 Tagged Word Data Format

5.2 Floating-Point Data Formats

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

- Single-precision floating-point (32-bit)
- Double-precision floating-point (64-bit)
- Quad-precision floating-point (128-bit)

5.2.1 Floating-Point Data Value Range

The value range for each format is included with the format and description of each format.

5.2.2 Floating-Point Data Alignment

TABLE 5-3 describes the address and memory alignment for floating-point data.

TABLE 5-3 Floating-Point Doubleword and Quadword Alignment

Subformat Name	Subformat Field	Required Address Alignment	Memory Address (Big-endian)*	Register Number Alignment	Available Registers
FS	`s:exp`<7:0>`:fraction`<22:0>	0 **mod** 4 †	*n*	Any	*f0, f1,... f31*
FD-0	`s:exp`<10:0>`:fraction`<51:32>	0 **mod** 4 †	*n*	0 **mod** 2	*f0, f2,... f62*
FD-1	`fraction`<31:0>	0 **mod** 4 †	*n* + 4	1 **mod** 2	*f1, f3,... f63*
FX-0	—	0 **mod** 4 †	n	0 **mod** 4	*f0, f4,... f60*
FX-1	—	0 **mod** 4 †	n	0 **mod** 4	*f2, f6,... f62*
FQ-0	`s:exp`<14:0>:`fraction`<111:96>	0 **mod** 4 ‡	*n*	0 **mod** 4	*f0, f4,... f60*
FQ-1	`fraction`<95:64>	0 **mod** 4 ‡	*n* + 4	1 **mod** 4	*f1, f5,... f61*

TABLE 5-3 Floating-Point Doubleword and Quadword Alignment *(Continued)*

Subformat Name	Subformat Field	Required Address Alignment	Memory Address (Big-endian)*	Register Number Alignment	Available Registers
FQ-2	`fraction`<63:32>	0 **mod** 4 ‡	*n* + 8	2 **mod** 4	*f2, f6,... f62*
FQ-3	`fraction`<31:0>	0 **mod** 4 ‡	*n* + 12	3 **mod** 4	
FX	—	0 **mod** 4 †	n	0 **mod** 4	*f3, f7,... f63*

* 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 single word 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).

5.2.3 Floating-Point, Single-Precision

FIGURE 5-12 illustrates the floating-point single-precision data format, and TABLE 5-4 describes the formats.

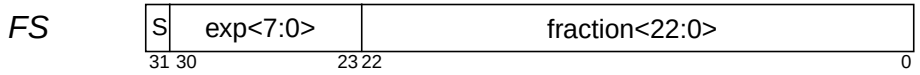


FIGURE 5-12 Floating-Point Single-Precision Data Format

TABLE 5-4 Floating-Point Single-Precision Format Definitions

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)	$(-1)^s \times 0$
Signalling NaN	s = *u*; e = 255 (max); f = .0*uu*--*uu* (At least one bit of the fraction must be nonzero)
Quiet NaN	s = *u*; e = 255 (max); f = .1*uu*--*uu*
− ∞ (negative infinity)	s = 1; e = 255 (max); f = .000--00
+ ∞ (positive infinity)	s = 0; e = 255 (max); f = .000--00

5.2.4 Floating-Point, Double-Precision

FIGURE 5-13 illustrates both components (FD-0 and FD-1) of the floating-point double-precision data format when two 32-bit registers are used. FIGURE 5-14 illustrates a double-precision data format using one 64-bit register.

TABLE 5-5 describes the data formats.

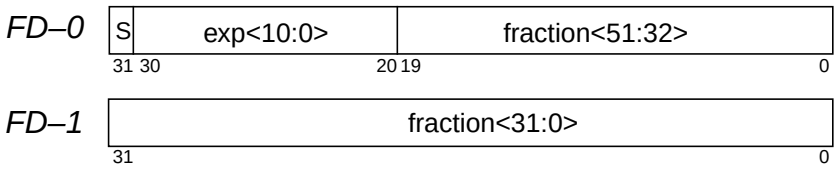


FIGURE 5-13 Floating-Point Double-Precision Double Word Data Format

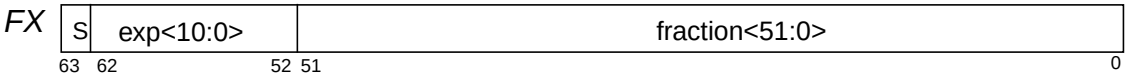


FIGURE 5-14 Floating-Point Double-Precision Extended Word Data Format

TABLE 5-5 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)	$(-1)^s \times 0$
Signalling NaN	s = *u*; e = 2047 (max); f = .0*uu--uu* (At least one bit of the fraction must be nonzero)
Quiet NaN	s = *u*; e = 2047 (max); f = .1*uu--uu*
− ∞ (negative infinity)	s = 1; e = 2047 (max); f = .000--00
+ ∞ (positive infinity)	s = 0; e = 2047 (max); f = .000--00

5.2.5 Floating-Point, Quad-Precision

FIGURE 5-15 illustrates all four components (FQ-0 through FQ-3) of the floating-point quad-precision data format, and TABLE 5-6 describes the formats.

Compatibility Note – Floating-point quad is not implemented in the processor. Quad-precision operations are emulated in the OS kernel.

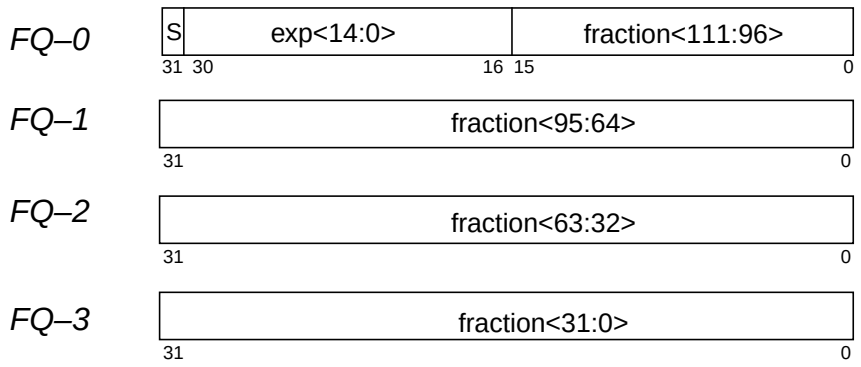


FIGURE 5-15 Floating-Point Quad-Precision Data Format

TABLE 5-6 Floating-Point Quad-Precision Format Definitions

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)	$(-1)^s \times 0$
Signalling NaN	s = *u*; e = 32767 (max); f = .0*uu*--*uu* (At least one bit of the fraction must be nonzero.)
Quiet NaN	s = *u*; e = 32767 (max); f = .1*uu*--*uu*
− ∞ (negative infinity)	s = 1; e = 32767 (max); f = .000--00
+ ∞ (positive infinity)	s = 0; e = 32767 (max); f = .000--00

5.3 VIS Execution Unit Data Formats

VIS instructions are optimized for short integer arithmetic, where the overhead of converting to and from floating point is significant. Data components can be 8 or 16 bits; intermediate results are 16 or 32 bits.

There are two VIS data formats:

- Pixel Data
- Fixed-point Data

Data Conversions

Conversion from pixel data to fixed data occurs through pixel multiplications. Conversion from fixed data to pixel data is done with the pack instructions, which clip and truncate to an 8-bit unsigned value. Conversion from 32-bit fixed to 16-bit fixed is also supported with the `FPACKFIX` instruction.

Rounding

Rounding can be performed by adding one to the round bit position. Complex calculations needing more dynamic range or precision should be performed using floating-point data.

Range

The range of values that each format supports is described below.

Data Alignment

The data in memory is expected to be aligned according to TABLE 5-7. If the address does not properly align, then an exception is generated and the load/store operation fails.

TABLE 5-7 Pixel, Fixed16, and Fixed32 Data Alignment

VIS Data Format Type	Width	VIS Data Format Name	Required Address Alignment	Memory Address (big-endian)	Register Number Alignment	Register Number
Pixel 8	32	Pixel Data Format	0 **mod** 4	*n*	r	*r*
Fixed16	64	Fixed16 Data Format	0 **mod** 8	*n*	0 **mod** 2	*r*
Fixed32	64	Fixed32 Data Format	0 **mod** 8	*n*	0 **mod** 2	*r*

5.3.1 Pixel Data Format

The Fixed 8-bit data format consists of four unsigned 8-bit integers contained in a 32-bit word (see FIGURE 5-16).

One common use is to represent intensity values for the color components of an image. For example, R, G, B, and α are used as color components and are positioned as shown:

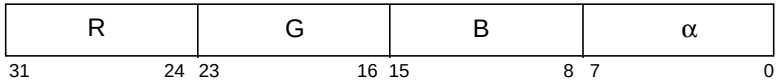


FIGURE 5-16 Pixel Data Format with Band Sequential Ordering Shown

The fixed 8-bit data format can represent two types of pixel data:

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

5.3.2 Fixed-Point Data Formats

The fixed 16-bit data format consists of four 16-bit signed fixed-point values contained in a 64-bit word. The fixed 32-bit format consists of two 32-bit signed fixed-point values contained in a 64-bit word. Fixed-point data values provide an intermediate format with enough precision and dynamic range for filtering and simple image computations on pixel values.

5.3.2.1 Fixed16 Data Format

Fixed data values provide an intermediate format with enough precision and dynamic range for filtering and simple image computations on pixel values.

Perform rounding by adding one to the round bit position. Perform complex calculations needing more dynamic range or precision by means of floating-point data.

The fixed 16-bit data format consists of four 16-bit, signed, fixed-point values contained in a 64-bit word. FIGURE 5-17 illustrates the Fixed16 VIS data format.

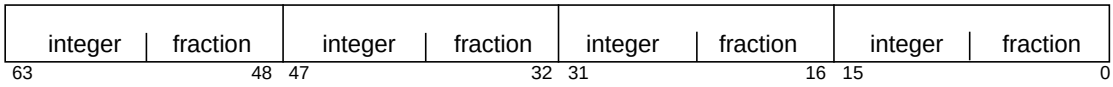


FIGURE 5-17 Fixed16 VIS Data Format

5.3.2.2 Fixed32 Data Format

The fixed 32-bit format consists of two 32-bit, signed, fixed-point values contained in a 64-bit word. FIGURE 5-18 illustrates the Fixed32 VIS data format.

FIGURE 5-18 Fixed32 VIS Data Format

CHAPTER 6

Registers

The topics covered in this chapter are discussed in the following sections:

6.1 Introduction

The processor consists of many types of registers that serve various purposes and accessed in many different ways.

There are separate working registers for the integer and floating-point units (FPUs). Both of the these register sets have been expanded over the evolution of the SPARC processor. The integer unit registers are shadowed using windowing and selection methods. The registers in the floating-point register set (also used for VIS and block load store instructions) are combined in specific ways to support data sizes up to 128 bits. All integer registers and the upper floating-point registers are 64 bits wide.

The processor also has a vast array of control, status, state, and diagnostic registers that are used to setup, control, and operate the processor. The two main operating modes of the processor, privileged and non-privileged mode, have a profound effect on which of the control and status registers are available to the software.

The majority of the control and status registers are 64 bits wide and are accessed using the privileged register access instructions, state register access instructions, and load/store with ASI access instructions. For convenience, some registers in this chapter are illustrated as fewer than 64 bits wide. Any bits not shown are reserved for future extensions to the architecture. Such reserved bits are read as zeroes and when written by software, should be written with the values of those bits previously read from that register or with zeroes.

- Integer Unit Working Registers (includes `r` and global)
- Floating-point Unit Working Registers
- Privileged Registers
- State and Ancillary State Registers (includes ASRs)
- Floating-point Status Register (FSR)
- ASI Mapped Registers (CSRs)

Some of the figures and tables in this chapter are reproduced from *The SPARC Architecture Manual-Version 9* and other sources. Many diagrams and tables appear here for the first time.

6.1.1 Document Notes

Contents of this chapter apply to non-privileged mode unless stated otherwise.

6.2 Integer Unit General-Purpose `r` Registers

An UltraSPARC IIIi processor contains 160 general-purpose 64-bit `r` registers. They are windowed into 32 registers addressable by Integer Unit Instructions.

The `r` registers are partitioned into eight addressable global registers and 24 addressable `windowed` registers. There are four global register sets: normal, MMU, Interrupt, and Alternate. The windowed registers point to eight working register sets that are windowed into `r`[8] to `r`[31], as one full register set (eight `local`s and eight `in`s) and a half register set (eight `out`s) belonging to the next higher state.

In summary, the `r` registers consist of eight `in` registers, eight `local` registers, eight `out` registers, and the selected eight global registers.

The current window pointer (CWP) register selects the in/local/out windowed registers. SAVE and RESTORE instructions modify the CWP register.

The PSTATE.AG, PSTATE.IG, and PSTATE.MG fields select the global register set. Processor exceptions modify the PSTATE register fields to select the global register set.

PSTATE and CWP registers are accessible using privileged instructions.

At any moment, general-purpose registers appear in non-privileged mode as shown in TABLE 6-1.

TABLE 6-1 Integer Unit General-Purpose Registers

Windowed Register Name	r Register Address	Source
in[7]	r[31]	Current Register Set
in[6]	r[30]	Current Register Set
in[5]	r[29]	Current Register Set
in[4]	r[28]	Current Register Set
in[3]	r[27]	Current Register Set
in[2]	r[26]	Current Register Set
in[1]	r[25]	Current Register Set
in[0]	r[24]	Current Register Set
local[7]	r[23]	Current Register Set
local[6]	r[22]	Current Register Set
local[5]	r[21]	Current Register Set
local[4]	r[20]	Current Register Set
local[3]	r[19]	Current Register Set
local[2]	r[18]	Current Register Set
local[1]	r[17]	Current Register Set
local[0]	r[16]	Current Register Set
out[7]	r[15]	Next higher level Register Set (see footnote 1)
out[6]	r[14]	Next higher level Register Set
out[5]	r[13]	Next higher level Register Set
out[4]	r[12]	Next higher level Register Set
out[3]	r[11]	Next higher level Register Set
out[2]	r[10]	Next higher level Register Set
out[1]	r[9]	Next higher level Register Set
out[0]	r[8]	Next higher level Register Set
global[7]	r[7]	Global[7]
global[6]	r[6]	Global[6]
global[5]	r[5]	Global[5]
global[4]	r[4]	Global[4]
global[3]	r[3]	Global[3]
global[2]	r[2]	Global[2]
global[1]	r[1]	Global[1]
global[0]	r[0]	Global[0] (value(r[0]) always 0)

1. The CALL instruction writes its own address into the r[15] register (*out*[7]).

6.2.1 Windowed (*in/local/out*) r Registers

At any time, an integer unit instruction can access a 24-register *window* into the register sets. A register window comprises of the eight in and eight local registers (a complete register set) together with the eight in registers (upper half of the next higher register set).

6.2.1.1 Predefined r Register Usages

Two of the r registers have a specific usage:

- 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).

6.2.1.2 128-bit Operand Considerations

LDD, LDDA, STD, and STDA instructions access 128-bit data associated with adjacent r registers and require even-odd register alignment. An attempt to execute a LDD, LDDA, STD, or STDA instruction that refers to a misaligned (odd) destination register number causes an *illegal_instruction* trap.

6.2.2 Global r Register Sets

Registers r[0]–r[7] refer to a set of eight global registers (g0–g7). At any time, one of four sets of eight global register sets is selected and can be accessed as the current global register set. The currently enabled set of global registers is selected by the Alternate Global (AG), Interrupt Global (IG), and MMU Global (MG) fields in the PSTATE register. See Section 6.8.3 “Processor State (PSTATE) Privileged Register 6” on page 6-107 for a description of the AG, IG, and MG fields.

Global register zero (g0) always reads as zero; writes to it have no program-visible effect.

FIGURE 6-1 illustrates the current IU registers.

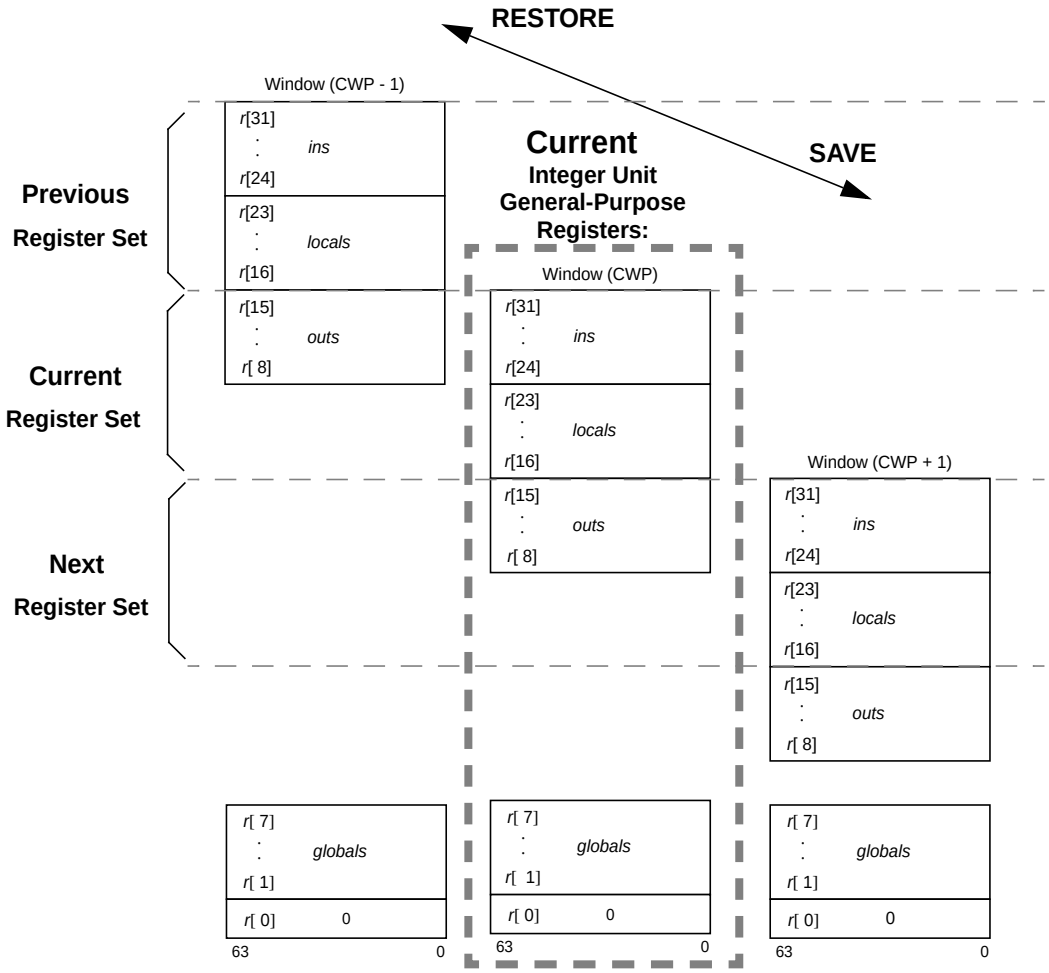


FIGURE 6-1 Three Overlapping Windows and the Eight Global Registers

---

Compatibility Note – Since the PSTATE register is writable only by privileged software, existing non-privileged SPARC-V8 software operates correctly on a processor if Supervisor Software ensures that User Software sees a consistent set of global registers.

In summary, the processor has eight windows or register sets (NWINDOWS = 8). The total number of r registers in the processor is 160: eight normal global registers, eight alternate global registers, eight interrupt global registers, eight MMU global registers, plus the number of register sets (eight) times 16 registers/set.

6.2.2.1 Overlapping Windows

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

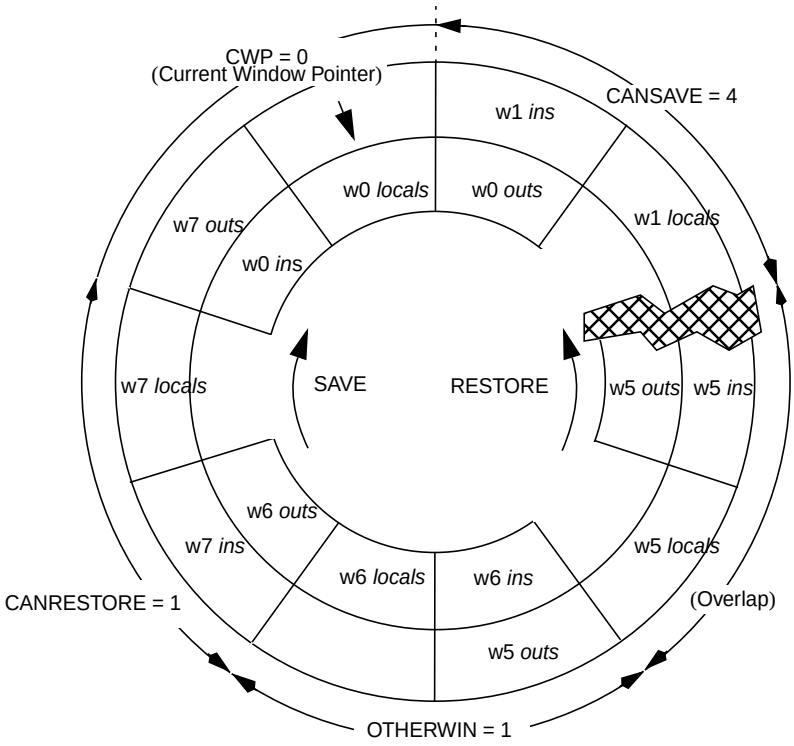
An *outs* register with address *o*, where $8 \le o \le 15$, refers to exactly the same register as $(o + 16)$ does after the CWP is incremented by one (modulo NWINDOWS). Likewise, an *in* register with address *i*, where $24 \le i \le 31$, refers to exactly the same register as address $(i - 16)$ does after the CWP is decremented by one (modulo NWINDOWS). See FIGURE 6-1 and FIGURE 6-2 for additional information.

Since CWP arithmetic is performed modulo NWINDOWS, the highest-numbered implemented window (window 7) overlaps with window 0. The outs of window NWINDOWS – 1 are the *ins* of window 0. Implemented windows are numbered contiguously from 0 through NWINDOWS – 1.

6.3 Register Window Management

The current window in the windowed portion of r registers is given by the CWP register. The CWP is decremented by the RESTORE instruction and incremented by the SAVE instruction. 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.

Programming Note – Because the windows overlap, the number of windows available to software is one less than the number of implemented windows; that is, 7 (NWINDOWS – 1).




```
CANSAVE + CANRESTORE + OTHERWIN = NWINDOWS – 2
```

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

NWINDOWS = 8, 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.

FIGURE 6-2 Windowed r Registers for NWINDOWS = 8

6.3.1 CALL and JMPL Instructions

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

6.3.2 Circular Windowing

Programming Note – When the register file is full, the *outs* of the newest window are the *ins* of the oldest window, which still contains valid data.

6.3.3 Clean Window with RESTORE and SAVE Instructions

Programming Note – The `local` and `out` registers of a register window are guaranteed to contain either zeroes or an old value that belongs to the current context upon reentering the window through a SAVE instruction. If a program executes a `RESTORE` followed by a SAVE, then the resulting window's *locals* and *outs* may not be valid after the SAVE, since a trap may have occurred between the `RESTORE` and the SAVE.

6.4 Floating-Point General-Purpose Registers

The floating-point register file contains addressable registers for the following:

- Floating-point Instructions
- `VIS` instructions
- `Block` load and `store` instructions
- `FSR` load and store instructions

The registers have various widths and assigned addresses as follows:

- 32 32-bit (single-precision) floating-point registers, `f`[0], `f`[1], … `f`[31]
- 32 64-bit (double-precision) floating-point registers, `f`[0], `f`[2], … `f`[62]
- 16 128-bit (quad-precision) floating-point registers, `f`[0], `f`[4], … `f`[60]

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 is shown in TABLE 6-2, TABLE 6-3, and TABLE 6-4. 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 FPop (`FPop1`/`FPop2` format) instructions, load/store single/double/quad floating-point instructions, and block load and block store instructions.

TABLE 6-2 32-bit Floating-Point Registers with Aliasing

Operand Register and Field		From Register	Operand Register and Field		From Register
f31	<31:0>	f31<31:0>	f15	<31:0>	f15<31:0>
f30	<31:0>	f30<31:0>	f14	<31:0>	f14<31:0>
f29	<31:0>	f29<31:0>	f13	<31:0>	f13<31:0>
f28	<31:0>	f28<31:0>	f12	<31:0>	f12<31:0>
f27	<31:0>	f27<31:0>	f11	<31:0>	f11<31:0>
f26	<31:0>	f26<31:0>	f10	<31:0>	f10<31:0>
f25	<31:0>	f25<31:0>	f9	<31:0>	f9<31:0>
f24	<31:0>	f24<31:0>	f8	<31:0>	f8<31:0>
f23	<31:0>	f23<31:0>	f7	<31:0>	f7<31:0>
f22	<31:0>	f22<31:0>	f6	<31:0>	f6<31:0>
f21	<31:0>	f21<31:0>	f5	<31:0>	f5<31:0>
f20	<31:0>	f20<31:0>	f4	<31:0>	f4<31:0>
f19	<31:0>	f19<31:0>	f3	<31:0>	f3<31:0>
f18	<31:0>	f18<31:0>	f2	<31:0>	f2<31:0>
f17	<31:0>	f17<31:0>	f1	<31:0>	f1<31:0>
f16	<31:0>	f16<31:0>	f0	<31:0>	f0<31:0>

TABLE 6-3 64-bit Floating-Point Registers with Aliasing

Operand Register and Field		From Register	Operand Register and Field		From Register
f62	<63:0>	f62<63:0>	f30	<63:0>	f30<31:0>:f31<31:0>
f60	<63:0>	f60<63:0>	f28	<63:0>	f28<31:0>:f29<31:0>
f58	<63:0>	f58<63:0>	f26	<63:0>	f26<31:0>:f27<31:0>
f56	<63:0>	f56<63:0>	f24	<63:0>	f24<31:0>:f25<31:0>
f54	<63:0>	f54<63:0>	f22	<63:0>	f22<31:0>:f23<31:0>
f52	<63:0>	f52<63:0>	f20	<63:0>	f20<31:0>:f21<31:0>
f50	<63:0>	f50<63:0>	f18	<63:0>	f18<31:0>:f19<31:0>
f48	<63:0>	f48<63:0>	f16	<63:0>	f16<31:0>:f17<31:0>
f46	<63:0>	f46<63:0>	f14	<63:0>	f14<31:0>:f15<31:0>
f44	<63:0>	f44<63:0>	f12	<63:0>	f12<31:0>:f13<31:0>
f42	<63:0>	f42<63:0>	f10	<63:0>	f10<31:0>:f11<31:0>
f40	<63:0>	f40<63:0>	f8	<63:0>	f8<31:0>:f9<31:0>
f38	<63:0>	f38<63:0>	f6	<63:0>	f6<31:0>:f7<31:0>
f36	<63:0>	f36<63:0>	f4	<63:0>	f4<31:0>:f5<31:0>
f34	<63:0>	f34<63:0>	f2	<63:0>	f2<31:0>:f3<31:0>
f32	<63:0>	f32<63:0>	f0	<63:0>	f0<31:0>:f1<31:0>

TABLE 6-4 128-bit Floating-Point Registers with Aliasing

Operand Register and Field		From Register
f60	<127:0>	f60<63:0>:f62<63:0>
f56	<127:0>	f56<63:0>:f58<63:0>
f52	<127:0>	f52<63:0>:f54<63:0>
f48	<127:0>	f48<63:0>:f50<63:0>
f44	<127:0>	f44<63:0>:f46<63:0>
f40	<127:0>	f40<63:0>:f42<63:0>
f36	<127:0>	f36<63:0>:f38<63:0>
f32	<127:0>	f32<63:0>:f34<63:0>
f28	<127:0>	f28<31:0>:f29<31:0>:f30<31:0>:f31<31:0>
f24	<127:0>	f24<31:0>:f25<31:0>:f26<31:0>:f27<31:0>
f20	<127:0>	f20<31:0>:f21<31:0>:f22<31:0>:f23<31:0>
f16	<127:0>	f16<31:0>:f17<31:0>:f18<31:0>:f19<31:0>
f12	<127:0>	f12<31:0>:f13<31:0>:f14<31:0>:f15<31:0>
f8	<127:0>	f8<31:0>:f9<31:0>:f10<31:0>:f11<31:0>
f4	<127:0>	f4<31:0>:f5<31:0>:f6<31:0>:f7<31:0>
f0	<127:0>	f0<31:0>:f1<31:0>:f2<31:0>:f3<31:0>

6.4.1 Floating-Point Register Number Encoding

The floating-point register number encoding in the instruction field depends on the width of register being addressed. The encoding for the 5-bit instruction field (labeled b<4>–b<0>, where b<4> is the most significant bit of the register number), is given in TABLE 6-5.

TABLE 6-5 Floating-Point Register Number Encoding

Register Operand Type	6-bit Register Number, fn						Encoding in a 5-bit Register Field in an Instruction, rd/rs				
32-bit (single)	0	b<4>	b<3>	b<2>	b<1>	b<0>	b<4>	b<3>	b<2>	b<1>	b<0>
64-bit (double)	b<5>	b<4>	b<3>	b<2>	b<1>	0	b<4>	b<3>	b<2>	b<1>	b<5>
128-bit (quad)	b<5>	b<4>	b<3>	b<2>	0	0	b<4>	b<3>	b<2>	0	b<5>

Compatibility Note – In SPARC-V8, bit 0 of 64- and 128-bit register numbers encoded in instruction fields was required to be zero. Therefore, all SPARC-V8 floating-point instructions can run unchanged on an UltraSPARC IIIi processor, using the encoding in TABLE 6-5.

6.4.2 Double and Quad Floating-Point Operands

A 32-bit `f` register can hold one single-precision operand; a 64-bit (double-precision) operand requires an aligned pair of `f` registers, and a 128-bit (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.

See FIGURE 6-3, TABLE 6-2, TABLE 6-3, and TABLE 6-4 for illustrative formats.

Programming Note – Data to be loaded into a floating-point double or quad register that is not doubleword aligned in memory must be loaded into the lower 16 double registers (8 quad registers) by means of single-precision `LDF` instructions. If desired, the data can then be copied into the upper 16 double registers (8 quad registers).

An attempt to execute an instruction that refers to a misaligned floating-point register operand (that is, a quad-precision operand in a register whose 6-bit register number is not 0 **mod** 4) shall cause a *fp_exception_other* trap, with `FSR.ftt` = 6 (*invalid_fp_register*).

Given the encoding in TABLE 6-5, it is impossible to specify a double-precision register with a misaligned register number.

Note – The processor does not implement quad-precision operations in hardware. All floating-point quad (including load and store) operations trap to the OS kernel and are emulated. Since the processor does not implement quad floating-point arithmetic operations in hardware, the *fp_exception_other* trap with `FSR.ftt` = 6 (*invalid_fp_register*) does not occur in processors.

6.5 Control and Status Register Summary

This section presents a summary of control and status registers.

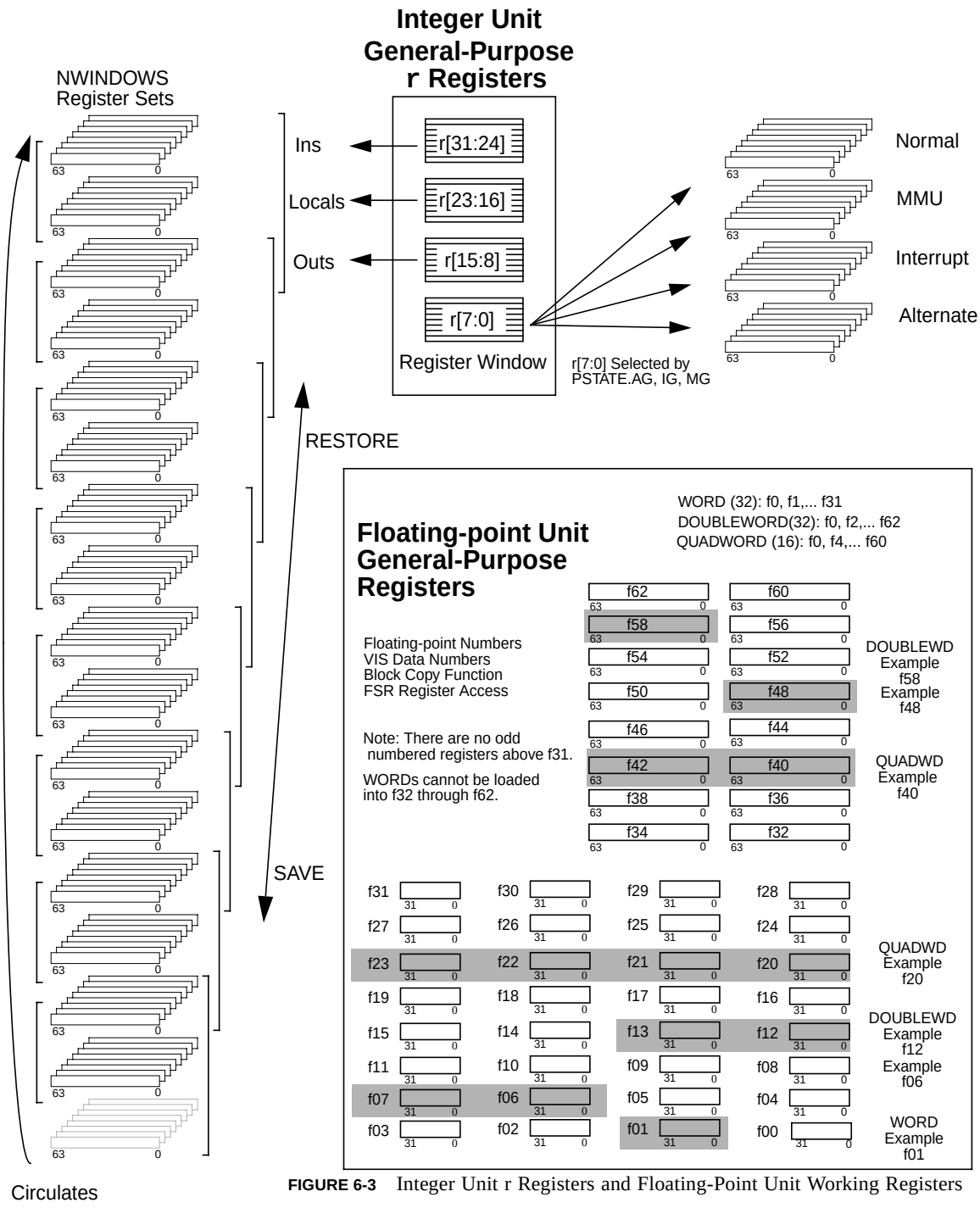


FIGURE 6-3 Integer Unit r Registers and Floating-Point Unit Working Registers

6.5.1 State and Ancillary State Register Summary

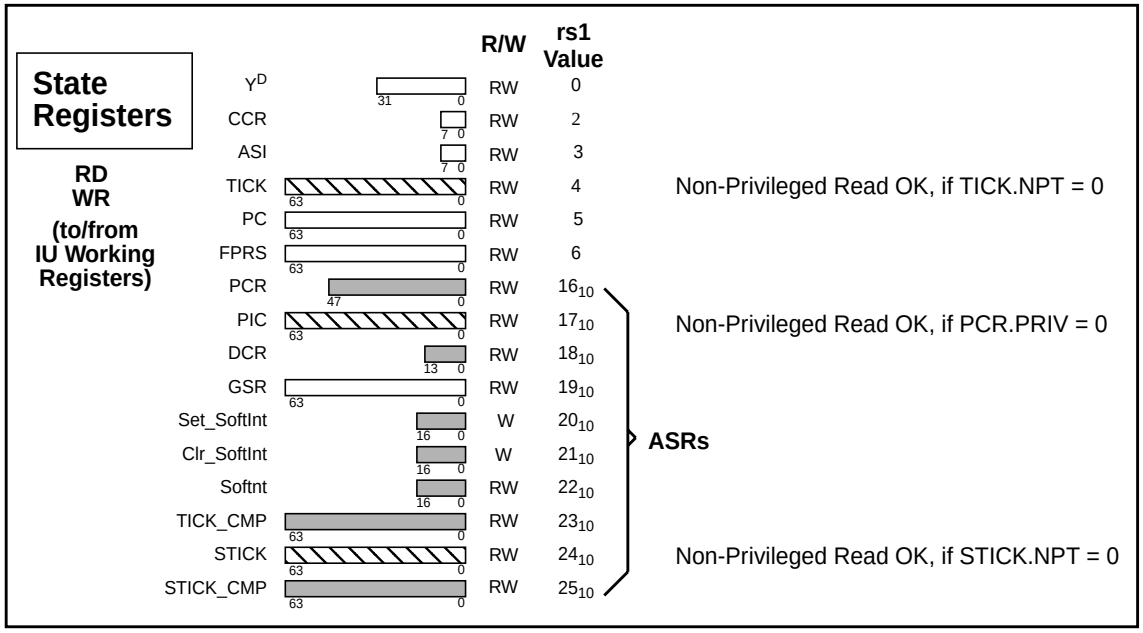


FIGURE 6-4 State and Ancillary State Registers

TABLE 6-6 State and Ancillary State Registers

State Register Number (base 10 used)	Access Restriction	R/W	Abbreviation	Description	Reference Section	Notes
0	None	RW	Y^D Register	32-bit Multiply/Divide (deprecated)		
1			*Reserved*			
2	None	RW	CCR	Condition Code		
3	None	RW	ASI	Address Space Identifier	Section 6.6.3	
4	Depends	R	TICK	TICK register for Processor Timer, also accessible as a privileged register	Section 6.7.4	1
5	None	R	PC	Program Counter	Section 6.6.5	
6	None	RW	FPRS	Floating-Point Registers State		
ASR 7 - 15			*Reserved*	Reserved for future use, do not reference by software.		

TABLE 6-6 State and Ancillary State Registers *(Continued)*

State Register Number (base 10 used)	Access Restriction	R/W	Abbreviation	Description	Reference Section	Notes
ASR 16	Privileged	RW	PCR	Performance Instrumentation	Chapter 11 "Performance Instrumentation"	2
ASR 17	Depends	RW	PIC			3
ASR 18	Privileged	RW	DCR	Dispatch Control Register	Section 6.7.1	
ASR 19	None	RW	GSR	Graphics (VIS) Status Register	Section 6.7.2	
ASR 20	Privileged	W	SET_SOFTINT	Software Interrupts	Section 6.7.3	
ASR 21	Privileged	W	CLR_SOFTINT			
ASR 22	Privileged	RW	SOFTINT_REG			
ASR 23	Privileged	RW	TICK_CMP	Processor and System Timer Registers	Section 6.7.4	
ASR 24	Depends	RW	STICK			4
ASR 25	Privileged	RW	STICK_CMP			
ASR 26 - 31			*Reserved*	Reserved for future use, do not reference by software.		

1. Writes are always privileged; reads are privileged if `TICK.NPT` = 1. Otherwise, reads are non-privileged.
2. If `PCR.NC` = 0, access is always privileged. If `PCR.NC` ≠ 0 and `PCR.PRIV` = 0, access is non-privileged; otherwise, access is privileged.
3. All accesses are privileged if `PCR.PRIV` = 1; otherwise, all accesses are non-privileged.
4. Writes are always privileged; reads are privileged if `STICK.NPT` = 1. Otherwise, reads are non-privileged.

6.5.2 Privileged Register Summary

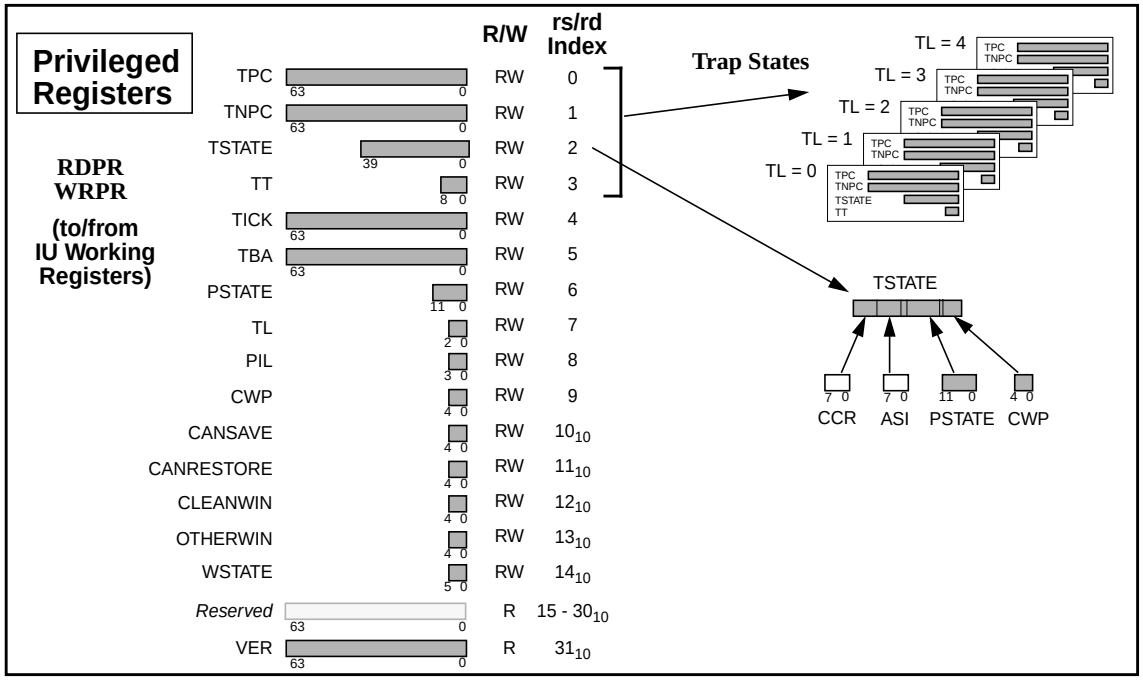


FIGURE 6-5 Privileged Registers

TABLE 6-7 Privileged Registers

Privileged Register Number (base 10 used)	Access Restriction	R/W	Abbreviation	Description	Reference Section	Notes
0	Privileged	RW	TPC	Trap stage program counter	Section 6.8.1	
1	Privileged	RW	TNPC	Trap state next program counter		
2	Privileged	RW	TSTATE	Trap state register		
3	Privileged	RW	TT	Trap type register		
4	Privileged	RW	TICK	Processor TICK timer register, also accessible as a state register	Section 6.7.4	
5	Privileged	RW	TBA	Trap base address register	Section 6.8.2	
6	Privileged	RW	PSTATE	Processor state register	Section 6.8.3	
7	Privileged	RW	TL	Trap level register	Section 6.8.4	
8	Privileged	RW	PIL	Processor Interrupt Level register	Section 6.8.5	
9	Privileged	RW	CWP	Current window pointer	Section 6.8.6	
10	Privileged	RW	CANSAVE	Savable register sets		
11	Privileged	RW	CANRESTORE	Restorable register sets		
12	Privileged	RW	CLEANWIN	Clean register sets		
13	Privileged	RW	OTHERWIN	Other register sets susceptible to spill/fill		
14	Privileged	RW	WSTATE	Window state register for traps due to spills and fills	Section 6.8.7	
15 - 30	Privileged		*Reserved*			
31	Privileged	R	VER	Processor version register	Section 6.8.8	

6.5.3 ASI and Specially Accessed Register Summary

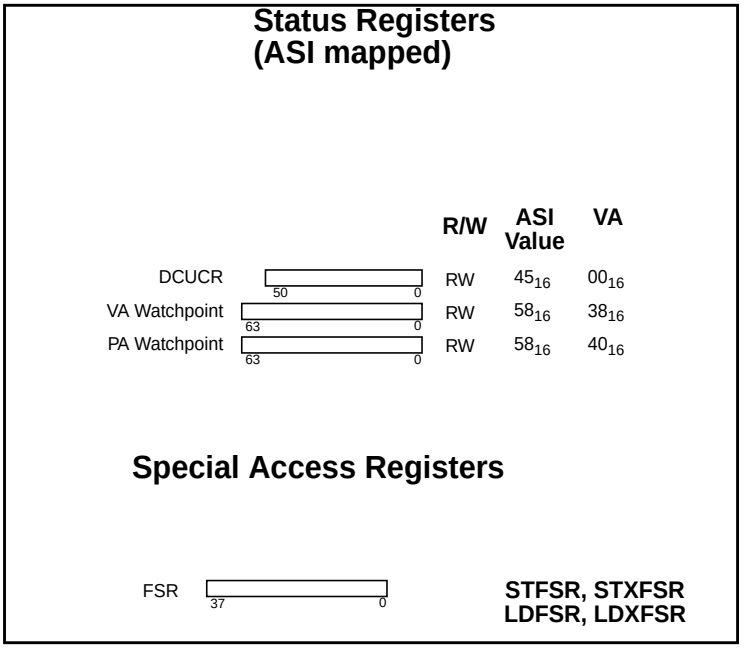


FIGURE 6-6 ASI and Specially Accessed Registers

TABLE 6-8 ASI and Specially Accessed Registers

Type	Abbreviation	Description	Reference Section
ASI	DCUCR	Data Cache Unit Control Register	Section 6.10.1
ASI 58_{16}	PA WATCHPOINT	Watchpoint for physical addresses	Section 6.10.2
	VA WATCHPOINT	Watchpoint for virtual addresses	
LD/ST floating-point Opcode	Load/Store FSR	Access the Floating-point Status Register	

6.6 State Registers

The state registers provide control and status to the Integer Execution Unit.

The type and accessibility of the registers (privileged vs. non-privileged mode) are summarized in FIGURE 6-4.

The SPARC-V9 architecture provides for up to 31 state registers, 24 of which are classified as ancillary state registers (ASRs), numbered from 7 through 31. The eight State Registers, 0 through 7, are defined by SPARC-V9.

6.6.1 32-bit Multiply/Divide (Y^D) State Register 0

The Y register is deprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC-V9 software. It is recommended that all instructions that reference the Y register (that is, $SMUL^D$, $SMULcc^D$, $UMUL^D$, $UMULcc^D$, $MULScc^D$, $SDIV^D$, $SDIVcc^D$, $UDIV^D$, $UDIVcc^D$, RDY^D, and WRY^D) be avoided.

The low-order 32 bits of the Y register, illustrated in FIGURE 6-7, contain the more significant word of the 64-bit product of an integer multiplication, as a result of either a 32-bit integer multiply ($SMUL^D$, $SMULcc^D$, $UMUL^D$, $UMULcc^D$) 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 ($SDIV^D$, $SDIVcc^D$, $UDIV^D$, $UDIVcc^D$) instruction.

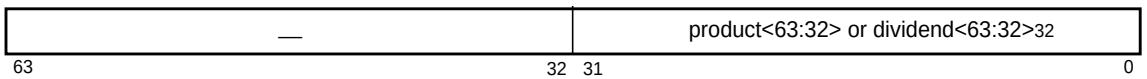


FIGURE 6-7 Y Register

Although Y is a 64-bit register, its high-order 32 bits are reserved and always read as zero. The Y register is read and written with the RDY^D and WRY^D instructions, respectively.

6.6.2 Integer Unit Condition Codes State Register 2 (CCR)

The Condition Codes Register (CCR), shown in FIGURE 6-8, holds the integer condition codes.

The CCR is accessible using Read and Write State Register instructions (RDCCR and WRCCR) in non-privileged or privileged mode.

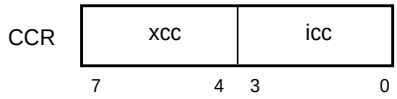


FIGURE 6-8 Condition Codes Register

6.6.2.1 CCR Condition Code Fields (xcc and 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 one) but the 64-bit result is nonnegative (xcc.*N* is set to zero).

Each of the 4-bit condition code fields is composed of four 1-bit subfields, as shown in FIGURE 6-9.

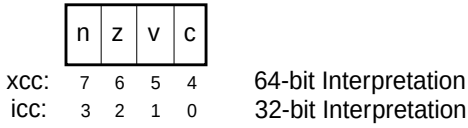


FIGURE 6-9 Integer Condition Codes (CCR_icc and CCR_xcc)

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 = nonnegative.

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 two'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 = carry, 0 = no carry.

Condition Codes

These bits are modified by the 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 CCR field of the TSTATE register. The BPcc and Tcc instructions may cause a transfer of control based on the values of these bits. The MOVcc instruction can conditionally move the contents of an integer register based on the state of these bits. The FMOVcc instruction can conditionally move the contents of a floating-point register according to the state of these bits.

CCR_extended_integer_cond_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.

CCR_integer_cond_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. In addition to the BPcc and Tcc instructions, the Bicc instruction may also cause a transfer of control based on the values of these bits.

6.6.3 Address Space Identifier (ASI) Register ASR 3

The ASI Register, shown in FIGURE 6-10, specifies the ASI to be used for load and store alternate instructions that use the “rs1 + simm13” addressing form.

Non-privileged (user-mode) software may write any value into the ASI register; however, values with bit 7 = 0 select restricted ASIs. When a non-privileged instruction makes an access that uses an ASI with bit 7 = 0, a *privileged_action* exception is generated.

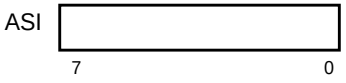


FIGURE 6-10 Address Space Identifier Register

6.6.4 TICK Register (TICK) ASR4

See Section 6.7.4 “Timer State Registers: ASRs 4, 23, 24, 25” on page 6-101 for more details.

6.6.5 Program Counters State Register 5

The program counter (PC) contains the address of the instruction currently being executed. The next program counter (nPC) holds the address of the next instruction to be executed if a trap does not occur. The low-order two bits of PC and nPC always contain zero.

For a delayed control transfer, the instruction that immediately follows the transfer instruction is known as the delay instruction. This delay instruction is executed (unless the control transfer instruction annuls it) before control is transferred to the target. During execution of the delay instruction, the nPC points to the target of the control transfer instruction, and the PC points to the delay instruction. See Chapter 7 “Instruction Types” for more details.

The PC is used implicitly as a destination register by CALL, Bicc, BPcc, BPr, FBfcc, FBPfcc, JMPL, and RETURN instructions. It can be read directly by a RDPC instruction.

6.6.6 Floating-Point Registers State (FPRS) Register 6

The Floating-Point Registers State (FPRS) Register, shown in FIGURE 6-11, holds control information for the floating-point register file. Mode and status information about the Floating-point Unit is presented in Section 6.9.1 “Floating-Point Status Register (FSR)” on page 6-117.

This register is readable and writable using the read and write state register instructions RDFPRS and WRFPRS when the processor is in non-privileged or privileged mode.

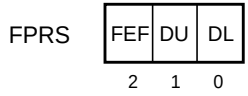


FIGURE 6-11 Floating-Point Registers State Register

6.6.6.1 FPRS_enable_fp (FEF)

Bit 2, FEF, determines whether the FPU is enabled. If this bit is set but the PSTATE.PEF bit is not set, then executing a floating-point instruction causes a *fp_disabled* trap; that is, both FPRS.FEF and PSTATE.PEF must be set to enable floating-point operations. If it is disabled, executing a floating-point instruction causes a *fp_disabled* trap.

6.6.6.2 FPRS_dirty_upper (DU)

Bit 1 is the "dirty" bit for the upper half of the floating-point registers; that is, f32–f62. It is set whenever any of the upper floating-point registers is modified. The processor may set the bit whenever a floating-point instruction is issued, even though that instruction never completes and no output register is modified. The dirty bit may be set by instructions that the processor executes but does not complete due to wrong branch prediction. The DU bit is cleared only by software.

6.6.6.3 FPRS_dirty_lower (DL)

Bit 0 is the "dirty" bit for the lower 32 floating-point registers; that is, f0–f31. It is set whenever any of the lower floating-point registers is modified. The processor may set the bit whenever a floating-point instruction is issued, even though that instruction never completes and no output register is modified. The DL bit is cleared only by software.

6.7 Ancillary State Registers: ASRs 16-25

The SPARC-V9 architecture provides for optional *ancillary* state registers (ASRs) in addition to the six state registers defined for all SPARC-V9 processors and already described.

An ASR is read and written with the RDASR and WRASR instructions, respectively. Access to a particular ASR may be privileged or non-privileged. A RDASR or WRASR instruction is privileged if the accessed register is privileged.

All the state and ancillary state registers are summarized in TABLE 6-6. Some of the registers descriptions are presented below.

Note – PCR (ASR 16) and PIC (ASR 17) are discussed in detail in Chapter 11 "Performance Instrumentation."

6.7.1 Dispatch Control Register (DCR) ASR 18

The DCR provides control over the dispatch unit and branch prediction logic. The DCR also provides factory test equipment with access to internal logic states using the OBSDATA bus interface.

The DCR is a read/write register. Unused bits are read as zero and should be written only with zero or values previously read from them. The DCR is a privileged register; attempted access by non-privileged (user) code causes a *privileged_opcode* trap. POR value is $xxxx.xx0x_2$.

The DCR is illustrated in FIGURE 6-12 and described in TABLE 6-9.

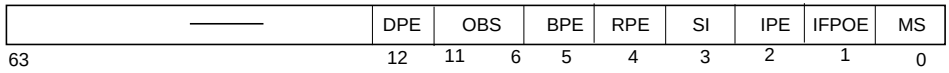


FIGURE 6-12 Dispatch Control Register (ASR 0x12)

TABLE 6-9 DCR Bit Description

Bit	Field	Type	Description	
63:13	-		*Reserved*	
12	DPE		*Data Cache Parity Error Enable*. If cleared, no parity checking at the Data Cache SRAM arrays (Data, Physical Tag, and Snoop Tag arrays) will be done. It also implies no *dcache_parity_error* trap (TT 0x071) will ever be generated. However, parity bits are still generated and written to the D-cache Parity SRAM. Therefore, when DPE is set, valid D-cache lines will automatically have correct parity bits.	
13:6	OBSDATA		These bits are used to select the set of signals driven on the OBSDATA<9:0> pins of the processor for factory test purposes.	
Branch and Return Control				
5	BPE		*Branch Prediction Enable*. When BPE = 1, conditional branches are predicted through internal hardware. When BPE = 0, all branches are predicted not taken. After *Power-On Reset* initialization, this bit is set to zero. This bit is also automatically set to zero on any trap causing RED_state entry (but not cleared when privileged code enters RED_state by setting the RED bit in PSTATE).	

TABLE 6-9 DCR Bit Description *(Continued)*

Bit	Field	Type	Description	
4	RPE		*Return Address Prediction Enable*. When RPE = 0, the return address prediction stack is disabled. Even when encountering a JMPL instruction, instruction fetch will continue on a sequential path until the return address is generated and a mispredict is signalled. When RPE = 1, the processor may attempt to predict the target address of JMPL instructions and prefetch subsequent instructions accordingly. After *Power-On Reset* initialization, this bit is set to zero. This bit is also automatically set to zero on any trap causing a RED_state entry (but left unchanged when privileged code enters RED_state by setting PSTATE.RED).	
Instruction Dispatch Control				
3	SI		*Single Issue Disable*. When SI = 0, only one instruction will be outstanding at a time. Superscalar instruction dispatch is disabled, and only one instruction is executed at a time. When SI = 1, normal pipelining is enabled. The processor can issue new instructions prior to the completion of previously issued instructions. After *Power-On Reset* initialization, this bit is set to zero. This bit is also automatically set to zero on any trap causing RED_state entry (but left unchanged when privileged code enters RED_state by setting PSTATE.RED).	
2	IPE		*Instruction Cache Parity Error Enable*. If cleared, no parity checking at the Instruction Cache SRAM arrays (Data, Physical Tag, and Snoop Tag arrays) will be done. It also implies no *Icache_Parity_error* trap (TT 0x072) will ever be generated. However, parity bits are still generated and written to the I-cache Parity SRAM. Therefore, when IPE is set, valid I-cache lines will automatically have correct parity bits.	
1	IFPOE		*Interrupt Floating-Point Operation Enable*. The IFPOE bit enables system software to take interrupts on floating-point instructions. When set, the processor forces a *fp_disabled* trap when an interrupt occurs on floating-point code.	
0	MS		*Multiscalar dispatch enable*. When MS = 0, the processor operates in scalar mode, issuing and executing one instruction at a time. Pipelined operation is still controlled by the SI bit. MS = 1 enables superscalar (normal) instruction issue. After *Power-On Reset* initialization, this bit is set to zero. The bit is also automatically set to zero on any trap causing RED_state entry (but left unchanged when privileged code enters RED_state by setting PSTATE.RED).	

Interrupt Floating-Point Operation Enable (Bit 1)

The IFPOE bit enables system software to take interrupts on floating-point instructions. This enable bit is cleared by hardware at power-on. System software must set the bit as needed. When this bit is enabled, the UltraSPARC IIIi processor forces an *fp_disabled* trap when an

interrupt occurs on FP-only code. The trap handler is then responsible for checking whether the floating-pint is indeed disabled. If it is not, the trap handler then enables interrupts to take the pending interrupt.

Note – This behavior deviates from SPARC-V9 trap priorities in that interrupts are of lower priorities than the other two types of floating-point exceptions (*fp_exception_ieee_754*, *fp_exception_other*).

- This mechanism is triggered for an floating-point instruction only if none of the approximately twelve preceding instructions across the two integer, load/store, and branch pipelines are valid, under the assumption that they are better suited to take the interrupt (only one trap entry/exit).
- Upon entry, the handler must check both `TSTATE.PEF` and `FPRS.FEF` bits. If `TSTATE.PEF` = 1 and `FPRF.FEF` = 1, the handler has been entered because of an interrupt, either *interrupt_vector* or *interrupt_level*. In such a case:
  - The *fp_disabled* handler should enable interrupts (that is, set `PSTATE.IE` = 1), then issue an integer instruction (for example, `add %g0,%g0,%g0`). An interrupt is triggered on this instruction.
  - The processor then enters the appropriate interrupt handler (`PSTATE.IE` is turned off here) for the type of interrupt.
  - At the end of the handler, the interrupted instruction is a `RETRY` after returning from the interrupt. The `add %g0,%g0,%g0` is a `RETRY`.
  - The *fp_disabled* handler then returns to the original process with a `RETRY`.
  - The “interrupted” FPop is then retried (taking a *fp_exception_ieee_754* or *fp_exception_other* at this time if needed).

6.7.2 Graphics Status Register (GSR) ASR 19

The `GSR` is used with the VIS Instruction Set.

The `GSR` is accessible in non-privileged mode. It can be read and written using the `RDASR` and `WRASR` state register instructions.

TABLE 6-10 GSR Opcodes

Opcode	Op3	Reg Field	Operation
RDASR	101000	*rs1* == 0x13	Read GSR
WRASR	110000	*rd* == 0x13	Write GSR

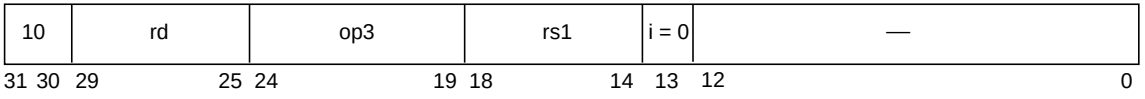


FIGURE 6-13 RDASR format

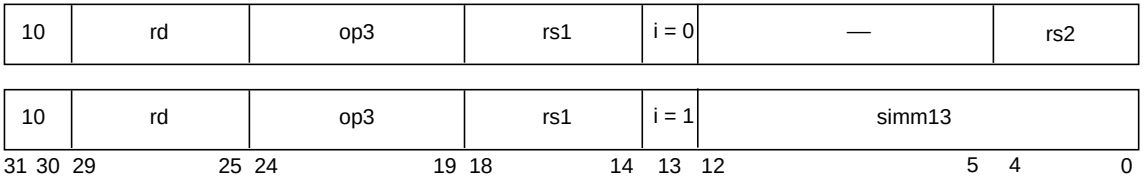


FIGURE 6-14 WRASR format

Suggested Assembly Language Syntax	
rd	%gsr, reg_{rd}
wr	reg_{rs1}, *reg_or_imm*, %gsr

Accesses to this register cause an *fp_disabled* trap if PSTATE.PEF or FPRS.FEF are zero.

The format of the GSR is:

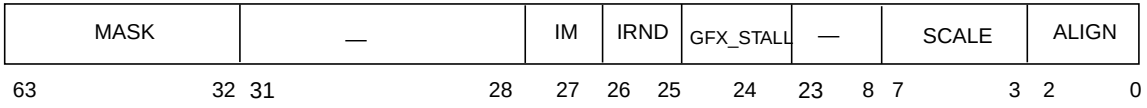


FIGURE 6-15 GSR Format (ASR 0x13)

TABLE 6-11 GSR Bit Description

Bit	Field	Description
63:32	MASK<31:0>	This 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: When IM = 1, the values in FSR.RD and FSR.NS are ignored; the processor operates as if FSR.NS = 0 and rounds floating-point results according to GSR.IRND.

TABLE 6-11 GSR Bit Description *(Continued)*

<table>
<tr><th>Bit</th><th>Field</th><th>Description</th></tr>
<tr><td>26:25</td><td>IRND<1:0></td><td>IEEE Std 754-1985 rounding direction to use in Interval Mode (GSR.IM = 1), as follows:
<table><tr><th>IRND</th><th>Round toward</th></tr><tr><td>0</td><td>Nearest (even if tie)</td></tr><tr><td>1</td><td>0</td></tr><tr><td>2</td><td>+ ∞</td></tr><tr><td>3</td><td>− ∞</td></tr></table>When GSR.IM = 1, the value in GSR.IRND overrides the value in FSR.RD.</td></tr>
<tr><td>24</td><td>GFX_STALL</td><td>This field is for the flow control signal from the graphics devices that indicates the status of their input command queues, that could be read by user software without having a load go to the bus. (read-only)

This has a big benefit in keeping a sustained pipeline of stores from the processor to the graphics devices, since you don’t have to wait for stores to drain, in order to get the load to complete.

This pin is inverted polarity compared to the external pin (i.e., 0 = stall, 1 = do not stall)</td></tr>
<tr><td>23:8</td><td>Reserved</td><td></td></tr>
<tr><td>7:3</td><td>SCALE<4:0></td><td>Shift count in the range 0–31, used by the PACK instructions for formatting.</td></tr>
<tr><td>2:0</td><td>ALIGN<2:0></td><td>Least three significant bits of the address computed by the last executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction.</td></tr>
</table>

6.7.3 Software Interrupt State Registers: ASRs 20, 21, and 22

Three registers are used to control software interrupts: SOFTINT, SET_SOFTINT, and CLR_SOFTINT. Bits written to the SOFTINT register will cause traps to the level the trap is enabled. The SOFTINT register can be written to directly using ASR 22, or indirectly using the SET_SOFTINT and CLR_SOFTINT registers as described in this section.

All three registers are accessible only in privileged mode. The SOFTINT register is accessed using the RD and WR state register access instructions. The SET_SOFTINT and CLR_SOFTINT registers are written using the WR state register access instruction. See TABLE 6-12 and FIGURE 6-16 for more details.

TABLE 6-12 Register-window State Registers

Soft Interrupt Register	ASR #	Name and Description	Privileged Access Instructions
SOFTINT	22	Software Interrupt Register	RDSOFTINIT WRSOFTINT
SET_SOFTINT	20	Sets Software Interrupt register bits.	WRSOFTINIT_SET
CLR_SOFTINT	21	Clears Software Interrupt register bits.	WRSOFTINIT_CLR

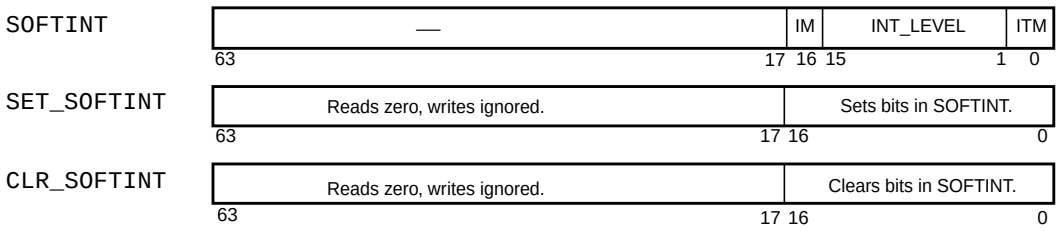


FIGURE 6-16 SOFTINT, SET_SOFTINT, and CLR_SOFTINT Register Formats

SOFTINT Register

The operating system uses the SOFTINT to schedule interrupts. The field definitions are described in TABLE 6-13.

TABLE 6-13 SOFTINT Bit Descriptions

Bit	Field	Description
16	SM (STICK_INT)	When the STICK_COMPARE.INT_DIS bit is zero (system tick compare is *enabled*) and its STICK_CMPR field matches the value in the STICK register, then the SM field in SOFTINT is set to one and a Level-14 interrupt is generated. See Section 6.7.4 “Timer State Registers: ASRs 4, 23, 24, 25” on page 6-101 for details.
15:1	INT_LEVEL	When a bit is set within this field (bits 15:1), an interrupt is caused at the corresponding interrupt level. Note that INT_LEVEL<15> is shared by Level-15 interrupt and PIC overflow interrupt.
0	TM (TICK_INT)	When the TICK_COMPARE.INT_DIS bit is zero (that is, tick compare is *enabled*) and its TICK_CMPR field matches the value in the TICK register, then the TM field in the SOFTINT register is set to one and a Level-14 interrupt is generated. See Section “TICK_COMPARE Register” on page 6-102 for details.

SET_SOFTINT Register

The SET_SOFTINT register is written to set bits in the SOFTINT register to set a bit in that register. When a bit in the SET_SOFTINT register is set to a one, the corresponding bit in the SOFTINT is set.

CLR_SOFTINT Register

The CLR_SOFTINT register is written in privileged mode using the WR write state register instruction to clear bits in the SOFTINT register. When a bit in the CLR_SOFTINT register is set to a one, the corresponding bit in the SOFTINT register is cleared.

6.7.4 Timer State Registers: ASRs 4, 23, 24, 25

The processor has two timers. The TICK timer is driven by the processor clock. The STICK timer is driven by the system clock. Four registers are used to implement the timer and support the timer interrupts. Timer state registers are described in TABLE 6-14.

TABLE 6-14 Timer State Registers

Soft Interrupt Register	ASR # (base 10)	Name and Description	Access Instructions
TICK	4	TICK register	Depends
TICK_COMPARE	23	TICK Compare register	State Register Instructions in privileged mode
STICK	24	STICK register	Depends
STICK_COMPARE	25	STICK Compare register	State Register Instructions in privileged mode

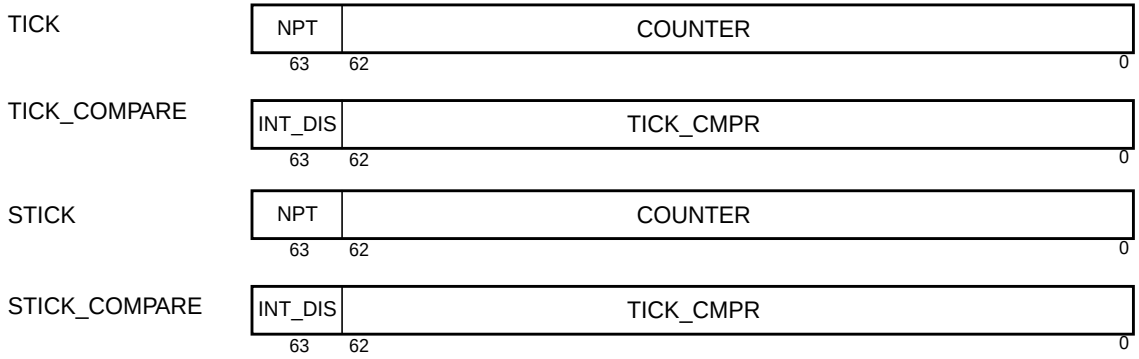


FIGURE 6-17 Timer State Registers

TICK Register

The TICK register is a 63-bit counter that counts processor clock cycles.

In privileged mode, the TICK register is always readable using either the RDPR (privileged read) or RDTICK (state register read) instructions. The TICK register is always write-able in privileged mode using the WRPR (privileged write) instruction; there is no WRTICK (state register write) instruction.

The TICK.NPT bit (bit 63) selects the non-privileged mode readability. If TICK.NPT = 0, then the TICK register is readable in non-privileged mode using the RDTICK state register read instruction. When TICK.NPT = 1, an attempt by software to read the TICK register in non-privileged mode causes a *privileged_action* exception. Software operating in non-privileged mode can never write to the TICK register.

The TICK.NPT is set to one by a *Power-On Reset* trap. The value of TICK.COUNTER is reset after a *Power-On Reset* trap.

After the TICK register is written, reading the TICK register returns a value incremented (by one or more) from the last value written, rather than from some previous value of the counter. The number of counts between a write and a subsequent read does not accurately reflect the number of processor cycles between the write and the read. Software may rely only on read-to-read counts of the TICK register for accurate timing, not on write-to-read counts.

Note – The TICK register is unaffected by any reset other than a *Power-On Reset*.

Programming Note – TICK.NPT may be used by a secure operating system to control access by user software to high-accuracy timing information. The operation of the timer might be emulated by the trap handler, which could read TICK.counter and change the value to lower its accuracy.

TICK_COMPARE Register

The TICK_COMPARE register causes the processor to generate a trap when the TICK register reaches the value in the TICK_COMPARE register and the INT_DIS bit is zero. If the INT_DIS bit is one, then no interrupt is generated.

When the TICK_CMPR field exactly matches the TICK.COUNTER field and INT_DIS = 0, then a TICK_INT is posted in the SOFTINT register. This has the effect of posting a Level-14 interrupt to the processor when the processor has PIL register value less than fourteen and PSTATE.IE register field 1.

Programming Note – The Level-14 interrupt handler must check the SOFTINT<14>, TM (TICK_INT), and SM (STICK_INT) fields of the SOFTINT register to determine the source or sources of the Level-14 interrupt.

In privileged mode, the TICK_COMPARE register is always accessible using the state register read and write instructions. The TICK_COMPARE register is not accessible in non-privileged mode. Non-privileged accesses to this register causes a *privileged_opcode* trap.

STICK Register

The STICK register is a 63-bit counter that increments at a rate determined by the system clock.

The STICK register is always accessible in privileged mode using the RDSTICK and WRSTICK state register instructions.

The STICK.NPT bit (bit 63) selects the non-privileged mode readability. If STICK.NPT = 0, then the STICK register is readable in non-privileged mode using the RDSTICK state register read instruction. When STICK.NPT = 1, an attempt by software to read the STICK register in non-privileged mode causes a *privileged_action* exception. Software operating in non-privileged mode can never write to the STICK register.

The STICK.NPT bit is set to one by a *Power-On Reset* trap. The value of STICK.COUNTER is cleared after a *Power-On Reset* trap.

After the STICK register is written, reading the STICK register returns a value incremented (by one or more) from the last value written, rather than from some previous value of the counter.

Note – The STICK register is unaffected by any reset other than a *Power-On Reset*.

STICK_COMPARE Register

The STICK_COMPARE register causes the processor to generate a trap when the STICK register reaches the value in the STICK_COMPARE register and the INT_DIS bit is zero. If the INT_DIS bit is one, then no interrupt is generated.

The STICK_COMPARE is only accessible in privileged mode. Accesses to this register in non-privileged mode causes a *privileged_opcode* trap.

When STICK_CMPR field exactly matches STICK.COUNTER field and INT_DIS = 0, then a TICK_INT is posted in the SOFTINT register. This has the effect of posting a Level-14 interrupt to the processor when the processor has PIL register value less than fourteen and PSTATE.IE register field 1.

Programming Note – The Level-14 interrupt handler must check SOFTINT<14>, TICK_INT, and STICK_INT to determine the source of the Level-14 interrupt.

After a *Power-On Reset* trap, the INT_DIS bit is set to one (disabling system tick compare interrupts), and the STICK_CMPR value is set to zero.

6.8 Privileged Registers

The privileged registers are described in this section. The privileged registers are visible only to software running in privileged mode (PSTATE.PRIV = 1). Privileged registers are written with the WRPR instruction and read with the RDPR instruction.

Refer to FIGURE 6-5 on page 6-87 for more details.

6.8.1 Trap Stack Privileged Registers 0 through 3

The four trap stack registers (TPC, TNPC, TSTATE, and TT) form a group of registers that are shadowed for each of the five trap levels. Each instance of the registers save the state of key integer unit parameters at each trap level. FIGURE 6-18 shows the format for this register group. This figure is followed by a description of each register. FIGURE 6-19 shows how the register stack responds to an event example.

The group of trap stack registers contain state information from the previous trap level. The registers include values from the program counter (PC), the next program counter (nPC), the trap state (TSTATE) register (a group of fields comprising the contents of the CCR, ASI, CWP, and PSTATE registers), and the trap type (TT) register containing the value of the trap that caused entry into the current trap level.

6.8.1.1 Common Attributes

There are MAXTL = 5 instances of the trap control registers, but only one group is accessible at any time. The current value in the TL register determines which instance of the trap control registers are accessible.

All trap control registers are accessible in privileged mode. An attempt to read or write any of these registers in non-privileged mode causes a *privileged_opcode* exception.

An attempt to read or write any of these registers when TL = 0 causes an *illegal_instruction* exception.

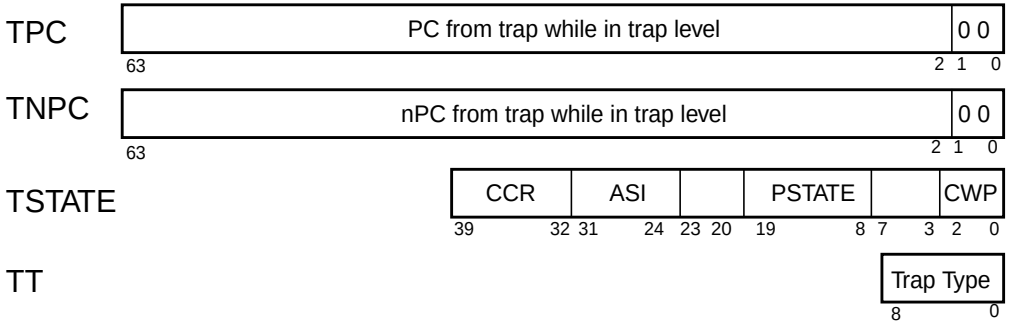


FIGURE 6-18 Trap State Register Format

Trap Program Counter

The Trap Program Counter (TPC) contains the PC from the previous trap level.

Trap Next Program Counter

The Trap Next Program Counter (TNPC) register is the nPC from the previous trap level.

Trap State Register

The Trap State (TSTATE) Register contains the state from the previous trap level, comprising the contents of the CCR, ASI, CWP, and PSTATE registers from the previous trap level.

Trap Type

The Trap Type (TT) register normally contains the trap type of the trap that caused entry to the current trap level.

6.8.1.2 Trap Stack Operation

The trap stack and an event example are illustrated in FIGURE 6-19.

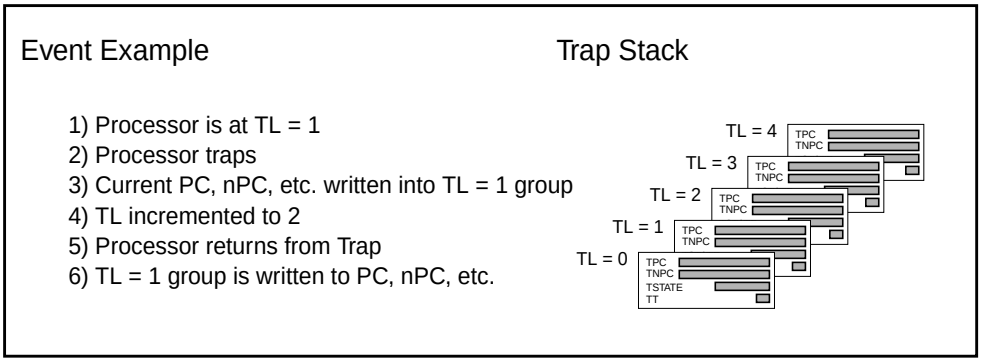


FIGURE 6-19 Trap Stack and Event Example

6.8.1.3 Effects of Reset and Normal Operation

The effects of reset on each register are shown in TABLE 6-15. During normal operation, the trap stack register values defined for the trap levels above the current one are undefined.

TABLE 6-15 Trap Stack Register Power-on and Normal Operation

Trap Control Register	After Power-On Reset	During Normal Operation, for *n* greater than the current trap level (n > TL)
TPC	TPC[0] = TPC[1] to TPC[5] are undefined	TPC[*n*] is undefined
TNPC	TPC[0] = TNPC[1] to TNPC[5] are undefined	TNPC[*n*] is undefined
TSTATE	TPC[0] = TSTATE[1] to TSTATE[5] are undefined	TSTATE[*n*] is undefined
TT	TPC[0] = Reset Trap Type TT[1] to TT[4] are undefined TT[5] = 001_{16}	TT[*n*] is undefined

6.8.2 Trap Base Address (TBA) Privileged Register 5

The TBA register, shown in FIGURE 6-20, provides the upper 49 bits of the address used to select the trap vector for a trap. The TBA register is accessible using read and write privileged register instructions. The lower 15 bits of the TBA always read as zero, and writes to them are ignored.

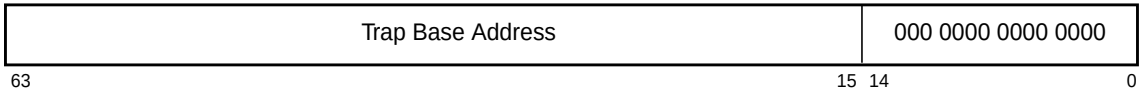


FIGURE 6-20 Trap Base Address Register

The full address for a trap vector is specified by the contents in the TBA, TL, and TT[TL] registers at the time the trap is taken, as shown in FIGURE 6-21.

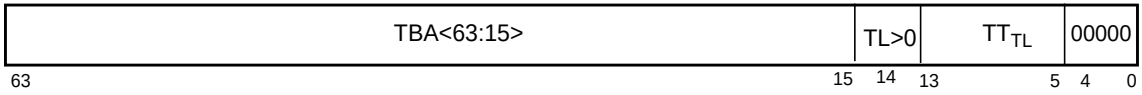


FIGURE 6-21 Trap Vector Address Format

TL>0 bit

The "TL > 0" bit is zero if TL = 0 when the trap was taken, and one if TL > 0 when the trap was taken. This implies that there are two trap tables: one for traps from TL = 0 and one for traps from TL > 0.

TT_{TL} field

The TT_{TL} field is written with the contents of the TT register representing the new trap level that is being taken.

6.8.3 Processor State (PSTATE) Privileged Register 6

The PSTATE register, shown in FIGURE 6-22, holds the current state of the processor. There is only one instance of the PSTATE register. The PSTATE register is copied to a 12-bit field in the TSTATE register of the trap stack.

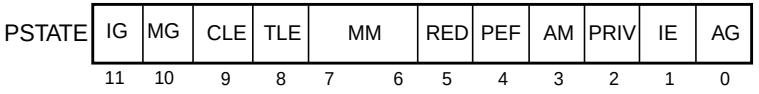


FIGURE 6-22 PSTATE Fields

Writing PSTATE is 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 all the bits in the PSTATE, respectively.

Subsections on page 108 through page 110 describe the fields contained in the PSTATE register.

6.8.3.1 Global Register Set Selection - IG, MG, AG bits

The UltraSPARC IIIi processor provides Interrupt and MMU Global Register sets in addition to the two global register sets (normal and alternate) specified by SPARC-V9. The currently active set of global registers is specified by the AG, IG, and MG bits and are set and cleared according to the events listed in TABLE 6-16.

Note – The IG, MG, and AG fields are saved on the trap stack along with the rest of the PSTATE Register.

TABLE 6-16 PSTATE Global Register Selection Events

Event	Globals selected for use	PSTATE settings		
		AG	IG	MG
DONE, RETRY [1]	Global Registers encoded in TSTATE register (Previous Global Registers before most recent trap)	0	0	0
fast_instruction_access_MMU_miss, fast_data_access__MMU_miss, fast_data_access_protection, data_access_exception, instruction_access_exception	MMU Global registers	0	0	1
interrupt_vector_trap	Interrupt Global registers	0	1	0
	Reserved [2]	0	1	1
Write to privileged register (WPR) that modifies AG, IG, or MG bits in PSTATE register	Any Global Register	x	x	x

TABLE 6-16 PSTATE Global Register Selection Events

Event	Globals selected for use	PSTATE settings		
		AG	IG	MG
Any trap other than those listed above	Alternate Global registers	1	0	0
Reserved	*Reserved*	1	0	1
Reserved	*Reserved*	1	1	0
Reserved	*Reserved*	1	1	1

1. Since PSTATE is preserved in the TSTATE register when a trap occurs, the previous value of these bits are normally restored upon return from a trap (via DONE or RETRY instruction).
2. A WRPR to PSTATE, using a reserved combination of AG, IG, and MG bit values, causes an *illegal_instruction* exception.

Executing a DONE or RETRY instruction restores the previous {AG, IG, MG} state before the trap is taken. Programmers can also set or clear these three bits by writing to the PSTATE register with a WRPR instruction.

Note – Attempting to use the "`wrpr %pstate`" instruction to set a reserved encoding for IG, MG, and AG (more than one of these bits set) results in an *illegal_instruction* exception. However, the processor does not check for a reserved encoding when writing directly to the TSTATE register. Hence, executing a DONE or RETRY with an invalid AG, IG, MG bit combination may result in an undefined behavior of the processor.

Compatibility Note – The UltraSPARC IIIi processor support two more sets (privileged only) of eight 64-bit global registers compared to the UltraSPARC II family: interrupt globals and MMU globals. These additional registers are called the *trap globals*. Two 1-bit fields, PSTATE.IG and PSTATE.MG, were added to the PSTATE register to select which set of global registers to use.

PSTATE_interrupt_globals (IG)

When PSTATE.IG = 1, the processor interprets integer register numbers in the range 0 – 7 as referring to the interrupt global register set. See the **Note** on page 109. When an *interrupt_vector* trap (trap type = 60_{16}) is taken, the processor sets IG and clears AG and MG.

PSTATE_MMU_globals (MG)

When PSTATE.MG = 1, the processor interprets integer register numbers in the range 0 – 7 as referring to the MMU global register set.

The processor sets PSTATE.MG and clears PSTATE.IG and PSTATE.AG when any of the following traps are taken:

- *fast_instruction_access_MMU_miss* trap (trap type = 64_{16}–67_{16})
- *fast_data_access_MMU_miss* trap (trap type = 68_{16}–$6B_{16}$)
- *fast_data_access_protection* trap (trap type = $6C_{16}$–$6F_{16}$)
- *data_access_exception* trap (trap type = 30_{16})
- *instruction_access_exception* trap (trap type = 08_{16})

PSTATE_alternate_globals (AG)

When PSTATE.AG = 1, the processor interprets integer register numbers in the range 0 – 7 as referring to the alternate global register set.

If an exception is taken and it does not set another global bit, then the processor defaults to the Alternate Global register set by setting PSTATE.AG and clearing PSTATE.IG and PSTATE.MG.

6.8.3.2 PSTATE_current_little_endian (CLE)

When PSTATE.CLE = 1, all data reads and writes using an implicit ASI are performed in little-endian byte order with an ASI of ASI_PRIMARY_LITTLE. When PSTATE.CLE = 0, all data reads and writes using an implicit ASI are performed in big-endian byte order with an ASI of ASI_PRIMARY. Instruction accesses are always big-endian.

6.8.3.3 PSTATE_trap_little_endian (TLE)

When a trap is taken, the current PSTATE register is pushed onto the trap stack and 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 one, 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.

6.8.3.4 PSTATE_mem_model (MM)

The processor supports Total Store Order (TSO) only. The 2-bit field in the PSTATE.MM is hardwired to 00 indicating TSO mode. See TABLE 6-17 for MM Encodings.

TABLE 6-17 MM Encodings

MM Value	SPARC-V9
00	Total Store Order (TSO)
01	*Reserved*
10	*Reserved*
11	*Reserved*

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 and stores. Programs that execute correctly in either PSO or RMO will execute correctly in the TSO model.

6.8.3.5 PSTATE_RED_state (RED)

PSTATE.RED (**R**eset, **E**rror, and **D**ebug state) is set whenever the UltraSPARC IIIi processor takes a RED state disrupting or nondisrupting trap. The IU sets PSTATE.RED when any hardware reset occurs. It also sets PSTATE.RED when a trap is taken while TL = (MAXTL − 1). Software can exit RED_state by executing a DONE or RETRY instruction, which restores the stacked copy of PSTATE and clears PSTATE.RED if it was zero in the stacked copy.

Note – Software can also exit the RED_state by writing a zero to PSTATE.RED with a WRPR instruction. However, this method is not recommended due to potential side-effects and unpredictable behavior.

6.8.3.6 PSTATE_enable_floating-point (PEF)

When set to one, the PSTATE.PEF bit enables the FPU, which allows privileged software to manage the FPU. For the FPU to be usable, both PSTATE.PEF and FPRS.FEF must be set. Otherwise, any floating-point instruction that tries to reference the FPU causes a *fp_disabled* trap.

6.8.3.7 PSTATE_address_mask (AM)

When PSTATE.AM = 1, the high-order 32 bits of any virtual addresses for instruction and data are cleared to zero in the following cases:

- Before data addresses are sent out of the processor
- Before addresses are sent to the MMU
- For instruction accesses to all caches
- Before being stored to a general-purpose register for CALL, JMPL, and RDPC instructions
- Before being stored to TPC[*n*] and TNPC[*n*] when a trap occurs

When an ASI_PHYS_* ASI is used in a load or store instruction, the setting of PSTATE.AM is ignored and the full 64-bit address is used. (See ASI 14_{16}, ASI_PHYS_USE_EC, for an example).

When PSTATE.AM = 1, the processor writes the full 64-bit program counter value (upper 32 bits are forced to be zero) to the destination register of a CALL, JMPL, or RDPC instruction.

When PSTATE.AM = 1 and a trap occurs, the processor writes the full 64-bit program counter value to TPC[TL].

When PSTATE.AM = 1 and a synchronous exception occurs, the processor writes the full 64-bit address to the Data Synchronous Fault Address Register.

When PSTATE.AM = 1 and an asynchronous exception occurs, the processor writes the full 64-bit address to the Data Asynchronous Fault Address Register.

The PSTATE.AM bit must be set when 32-bit software is executed.

6.8.3.8 PSTATE_privileged_mode (PRIV)

When PSTATE.PRIV = 1, the processor is in privileged mode. This bit is controlled by events in the processor and can be explicitly set.

6.8.3.9 PSTATE_interrupt_enable (IE)

When PSTATE.IE = 1, the processor can accept interrupts.

6.8.4 Trap Level (TL) Privileged Register 7

The trap level register, shown in FIGURE 6-23, 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. The maximum valid value that the TL register may contain is MAXTL = 5, which is always equal to the number of supported trap levels beyond Level-0.

FIGURE 6-23 Trap Level Register

Programming Note – Writing to the `TL` register with a value greater than `MAXTL` (five for the UltraSPARC IIIi processor) causes the value `MAXTL` to be written.

Writing the `TL` register with a wrpr `%tl` instruction does not alter any other processor state; that is, it is not equivalent to taking or returning from a trap.

6.8.5 Processor Interrupt Level (PIL) Privileged Register 8

The processor interrupt level (`PIL`), illustrated in FIGURE 6-24, is the interrupt level above which the processor will accept an interrupt. Interrupt priorities are mapped so that interrupt Level-2 has greater priority than interrupt Level-1, and so on.

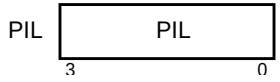


FIGURE 6-24 Processor Interrupt Level Register

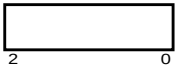
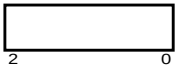
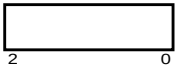
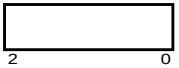
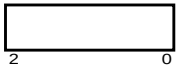
Compatibility Note – On SPARC-V8 processors, the Level-15 interrupt is considered to be nonmaskable, so it has different semantics from other interrupt levels. SPARC-V9 processors do not treat Level-15 interrupts differently from other interrupt levels.

6.8.6 Register-Window State Privileged Registers 9 through 13

The state of the register window is determined by a set of privileged registers that are read and written by privileged mode software using the `RDPR` and `WRPR` instructions, respectively. In addition, these privileged registers are modified by instructions related to register windowing and are used to generate traps that allow supervisor software to spill, fill, and clean the register window sets. TABLE 6-18 describes the register-window state privileged registers.

Register-window management is described in a separate chapter.

TABLE 6-18 Register-Window State Privileged Registers

Register-window State Registers	Value Range	Description
Current Window Pointer CWP 	0 to 7	**State Register 9:** The CWP register is a counter that identifies the current window into the set of integer registers.
Savable Window Sets CANSAVE 	0 to 6	**State Register 10:** The CANSAVE register contains the number of register sets following CWP that are not in use and are available to be allocated by a SAVE instruction without generating a window spill exception.
Restorable Window Sets CANRESTORE 	0 to 7	**State Register 11:** The CANRESTORE register contains the number of register sets preceding CWP that are in use by the current program and can be restored (by the RESTORE instruction) without generating a window fill exception.
Clean Window Sets CLEANWIN 	0 to 6	**State Register 12:** The CLEANWIN register contains the number of windows that can be used by the SAVE instruction without causing a *clean_window* exception.
Other Window Sets OTHERWIN 	0 to 7	**State Register 13:** The OTHERWIN register contains the count of register sets that will be spilled/filled by a separate set of trap vectors based on the contents of WSTATE_OTHER. If OTHERWIN is zero, register sets 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 sets among different address spaces and handle spill/fill traps efficiently by use of separate spill/fill vectors.

Note – The CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN registers contain values in the range 0 to 7 or 0 to 6 as indicated in TABLE 6-18. The effect of writing a value greater than indicated to any of these registers is undefined. The values programmed into these registers must combine into a consistent set of numbers that will work.

Note – The most significant 61 bits of all these registers are set to zero. When any are written, the most significant 61 bits are ignored.

Compatibility Note – The following differences between SPARC-V8 and SPARC-V9 are visible only to privileged software; they are invisible to non-privileged software.

1. In SPARC-V9, SAVE increments CWP and RESTORE decrements CWP. In SPARC-V8, the opposite is true: SAVE decrements PSR.CWP and RESTORE increments PSR.CWP.

2. PSR.CWP in SPARC-V8 is changed on each trap. In SPARC-V9, CWP is affected only by a trap caused by a window fill or spill exception.

Clean Windows (CLEANWIN) Register Note

The CLEANWIN register counts the number of register window sets that are "clean" with respect to the current program, that is, register sets 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 sets that can be restored (the value in the CANRESTORE register) and the register sets 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.

Programming Note – CLEANWIN must never be set to a value greater than six. Setting CLEANWIN greater than six would violate the register window state definition. Notice that the hardware does not enforce this restriction; it is up to Supervisor software to keep the window state consistent.

6.8.7 Window State (WSTATE) Privileged Register 14

The WSTATE register, shown in FIGURE 6-25, specifies bits that are inserted into TT_{TL}<4:2> on traps caused by window spill and fill exceptions.

This register is read/write by using the RDPR and WRPR privileged instructions.

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] field of the Trap Vector Address. Otherwise, the WSTATE.OTHER bits are inserted into TT[TL].

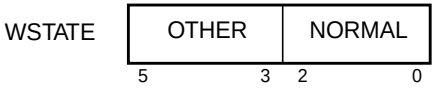


FIGURE 6-25 WSTATE Register

6.8.8 Version (VER) Privileged Register 31

The version register, shown in FIGURE 6-26, specifies the fixed parameters pertaining to a particular processor implementation and mask set.

The VER register is read-only, readable by the RDPR privileged instruction.

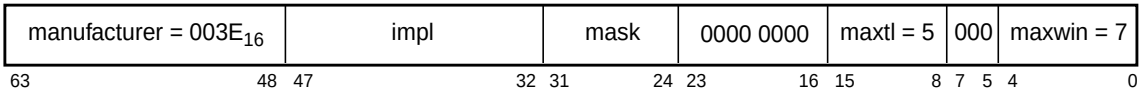


FIGURE 6-26 Version Register

VER.manuf field

The VER.manuf field contains Sun's 16-bit manufacturer code, $003E_{16}$, which is Sun's JEDEC semiconductor manufacturer code.

VER.impl field

The VER.impl field uniquely identifies the processor implementation or class of software-compatible implementations of the architecture. TABLE 6-19 shows the processor implementation codes.

TABLE 6-19 Processor Implementation Codes

Processor	VER.impl
UltraSPARC I	0010_{16}
UltraSPARC II	0011_{16}
UltraSPARC IIi	0012_{16}
UltraSPARC IIe	0013_{16}
UltraSPARC IIIi	0015_{16}

VER.mask field

The VER.mask specifies the current mask set revision and is chosen by the implementor. It generally increases numerically with successive releases of the processor but does not necessarily increase by one for consecutive releases. TABLE 6-20 shows the UltraSPARC IIIi Processor Mask Version.

TABLE 6-20 UltraSPARC IIIi Processor Mask Version Codes

Mask Version	VER.mask
TO_1.x	4'h1
TO_2.x	4'h2

VER.maxtl field

The VER.maxtl value, 5, is the maximum number of trap levels supported by the processor.

VER.maxwin field

The VER.maxwin value, 7, is the maximum number of Integer Unit register windows that access the NWINDOWS = 8 window register sets.

6.9 Special Access Register

6.9.1 Floating-Point Status Register (FSR)

The FSR register fields, illustrated in FIGURE 6-26, contain FPU mode and status information. State information about the FPU is presented in section Section 6.6.6 "Floating-Point Registers State (FPRS) Register 6" on page 6-93.

The FSR is accessible using special load and store opcodes. They work in privileged and non-privileged mode. The lower 32 bits of the FSR are read and written by the $STFSR^{D}$ and $LDFSR^{D}$ floating-point instructions; all 64 bits of the FSR are read and written by the STXFSR and LDXFSR floating-point instructions, respectively. FIGURE 6-27 illustrates the FSR fields.

The ver, ftt, and reserved ("—") fields are not modified by LDFSR or LDXFSR, they are read-only fields.

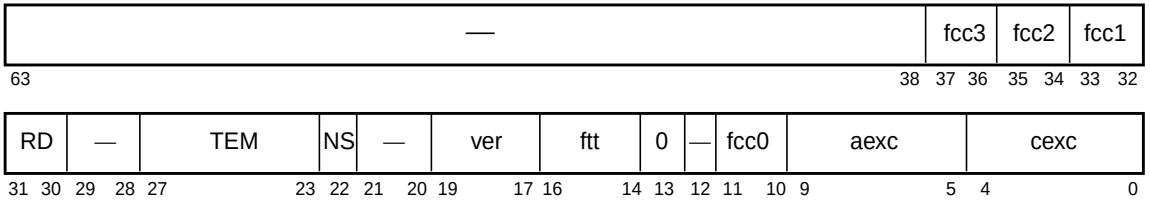


FIGURE 6-27 FSR Fields

Reserved Bits

Bits 63–38, 29–28, 21–20, and 12 are reserved. When read by a STXFSR instruction, these bits will read as zero. Software should issue LDXFSR instructions only with zero values in these bits, unless the values of these bits are exactly those derived from a previous STXFSR.

The subsections on pages page 118 through page 126 describe the remaining fields in the FSR.

6.9.1.1 FSR_fp_condition_codes (fcc0, fcc1, fcc2, fcc3)

The four sets of floating-point condition code fields are labeled `fcc0`, `fcc1`, `fcc2`, and `fcc3`.

Compatibility Note – `fcc0` defined in SPARC-V9 is the same as `fcc` defined in SPARC-V8.

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

In TABLE 6-21, f_{rs1} and f_{rs2} correspond to the single, double, or quad values in the floating-point registers specified by a floating-point compare instruction's `rs1` and `rs2` fields. The question mark (?) indicates an unordered relation, which is true if either f_{rs1} or f_{rs2} is a signalling NaN or a quiet NaN. If `FCMP` or `FCMPE` generates an *fp_exception_ieee_754* exception, then `fccn` is unchanged. TABLE 6-21 shows the floating-point condition codes.

TABLE 6-21 Floating-Point Condition Codes (`fcc`*n*) Fields of `FSR`

Content of *fccn*	Indicated Relation
0	$f_{rs1} = f_{rs2}$
1	$f_{rs1} < f_{rs2}$
2	$f_{rs1} > f_{rs2}$
3	f_{rs1} ? f_{rs2} (*unordered*)

6.9.1.2 FSR_rounding_direction (RD)

Bits 31 and 30 select the rounding direction for floating-point results according to IEEE Std 754-1985. TABLE 6-22 shows the rounding direction fields.

TABLE 6-22 Rounding Direction (`RD`) Field of `FSR`

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

If `GSR.IM` = 1, then the value of `FSR.RD` is ignored and floating-point results are instead rounded according to `GSR.IRND`.

6.9.1.3 FSR_nonstandard_fp (NS)

The `NS` bit allows the processor to flush a subnormal floating-point value to zero. If a floating-point add/subtract operation results in a subnormal value and `FSR.NS` = 1, the value is replaced by a floating-point zero value of the same sign. This replacement is usually performed in hardware. However, for the following cases when a subnormal value is generated in the course of the instruction and `FSR.NS` = 1, an *fp_exception_other* exception with `FSR.ftt` = 2 (*unfinished_FPop*) is taken and trap handler software is expected to replace the subnormal value with a zero value of the appropriate sign:

- `fadd` of numbers with opposite signs

- `fsub` of numbers with the same signs
- `fdtos`

The effects of `FSR.NS` = 1 are as follows:

- If a floating-point source operand is subnormal, it is replaced by a floating-point zero value of the same sign (instead of causing an exception).
- If a floating-point operation generates a subnormal value, the value is replaced with a floating-point zero value of the same sign.
- This is implemented by performing the replacement in hardware, and sometimes cause a *fp_exception_other* exception with `FSR.ftt` = 2 (*unfinished_FPop*) so that trap handler software can perform the replacement.

If `GSR.IM` = 1, then the value of `FSR.NS` is ignored and the processor operates as if `FSR.NS` = 0.

6.9.1.4 FSR_version (*ver*)

Version number 7 is reserved to indicate that no hardware floating-point controller is present.

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

6.9.1.5 FSR_floating-point_trap_type (*ftt*)

When a floating-point exception trap occurs, `ftt` (bits 16 through 14 of the `FSR`) identifies the cause of the exception, the "floating-point trap type." Several conditions can cause a floating-point exception trap. After a floating-point exception occurs, the `ftt` field encodes the type of the floating-point exception until a `STFSR` or FPop is executed.

The `ftt` field can be read by the `LDFSR` and `LDXFSR` instructions. The `STFSR` and `STXFSR` instructions do not affect `ftt` because this field is read-only.

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

Programming Note – Neither LDFSR nor LDXFSR can be used for the purpose of clearing ftt, since both leave ftt unchanged. However, executing a non-trapping FPop such as "fmovs %f0,%f0" prior to returning to non-privileged mode will zero ftt. The ftt remains valid until the next FPop instruction completes execution.

The ftt field encodes the floating-point trap type according to TABLE 6-23. **Note**: The value "7" is reserved for future expansion.

TABLE 6-23 Floating-Point Trap Type (ftt) Field of FSR)

ftt	Trap Type	Trap Vector
0	None	No trap taken
1	*IEEE_754_exception*	*fp_exception_ieee_754*
2	*unfinished_FPop*	*fp_exception_other*
3	*unimplemented_FPop*	*fp_exception_other*
4	*sequence_error*	*Reserved*, Unimplemented
5	*hardware_error*	*Reserved*, Unimplemented
6	*nvalid_fp_register*	*Reserved*, Unimplemented
7	*Reserved*	*Reserved*, Unimplemented

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

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

1. The value of aexc is unchanged. See Section 6.9.1.6 for details of aexc.
2. The value of cexc is unchanged, except for an *IEEE_754_exception*, where a bit corresponding to the trapping exception is set. The *unfinished_FPop*, *unimplemented_FPop*, *sequence_error*, and *invalid_fp_register* floating-point trap types do not affect cexc. See Section 6.9.1.6 for details of cexc.
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 *IEEE_754_exception* or after recovery from an *unfinished_FPop* or *unimplemented_FPop*. In either case, cexc as seen by the trap handler reflects the exception causing the trap.

In the cases of *fp_exception_other* exceptions with *unfinished_FPop* or *unimplemented_FPop* trap types that do not subsequently generate IEEE traps, the recovery software should define cexc, aexc, and the destination registers or fccs, as appropriate.

ftt = 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 exception type is encoded in the `cexc` field.

The `aexc` and `fcc`*s* fields and the destination `f` register are not affected by an *IEEE_754_exception* trap.

ftt = unfinished_FPop. The *unfinished_FPop* floating-point trap type indicates that the processor was unable to generate correct results or that exceptions as defined by IEEE Std 754-1985 have occurred. Where exceptions have occurred, the `cexc` field is unchanged.

The conditions under which a *fp_exception_other* exception with floating-point trap type of *unfinished_FPop* can occur are implementation dependent. The recommended set of conditions is shown in TABLE 6-24. An implementation may cause *fp_exception_other* with *unfinished_FPop* under a different (but specified) set of conditions.

TABLE 6-24 Standard Conditions Under Which *unfinished_FPop* Trap Type Can Occur

FPU Operation	1 subnormal (SBN) operand IM = 1 or NS=0	2 subnormal (SBN) operands IM = 1 or NS = 0	Result/Non-SBN Operand IM = 1 or NS = 0
fadds	Unfinished trap	Unfinished trap	fi fv, fu, sbn (IM = NS = x) NaN (either operand)
fsubs	Unfinished trap	Unfinished trap	fi fv, fu, sbn (IM = NS = x) NaN (either operand)
faddd	Unfinished trap	Unfinished trap	fi fv, fu, sbn (IM = NS = x) NaN (either operand)
fsubd	Unfinished trap	Unfinished trap	fi fv, fu, sbn (IM = NS = x) NaN (either operand)
fmuls	Unfinished trap if - result not zero	Unfinished trap if - result not zero	-25 < Er <= 1
fdivs	Unfinished trap	Unfinished trap	-25 < Er <= 1
fsmuld	Unfinished trap	Unfinished trap	None
fmuld	Unfinished trap if - result not zero	Unfinished trap if - result not zero	-54 < Er <= 1
fdivd	Unfinished trap	Unfinished trap	-54 < Er <= 1
fsqrts	Unfinished trap	N/A	None
fsqrtd	Unfinished trap	N/A	None
fstoi	Unfinished trap	N/A	$-2^{31} <= res < 2^{31}$, Infinity, NaN
fdtoi	Unfinished trap	N/A	$-2^{31} <= res < 2^{31}$, Infinity, NaN
fstox	Unfinished trap	N/A	$\|result\| >= -2^{52}$, Infinity, NaN
fdtox	Unfinished trap	N/A	$\|result\| >= -2^{52}$, Infinity, NaN

TABLE 6-24 Standard Conditions Under Which *unfinished_FPop* Trap Type Can Occur *(Continued)*

FPU Operation	1 subnormal (SBN) operand IM = 1 or NS=0	2 subnormal (SBN) operands IM = 1 or NS = 0	Result/Non-SBN Operand IM = 1 or NS = 0
fitos	N/A	N/A	$-2^{22} \leq$ operand $< 2^{22}$
fxtos	N/A	N/A	$-2^{22} \leq$ operand $< 2^{22}$
fitod	N/A	N/A	None
fxtod	N/A	N/A	$-2^{51} \leq$ operand $< 2^{51}$
fstod	Unfinished trap	N/A	NaN
fdtos	Unfinished trap	N/A	fi fv, fu, sbn (IM = NS = x), NaN

Note:
Er ← Biased Exponent of the result before rounding
Ei ← Biased Exponent of input operand
fi ← Invalid(Infinity – Infinity, Infinity*0, 0/0, Infinity/Infinity)
fv ← OverflowEr >= 2047(DP) or 255(SP) but not exact infinity
fu ← Underflow0 < |result| < 2^{-1022}(DP) or 2^{-126}(SP)
sbnormal(sbn): |number| = 2^{-1022} * (significand x 2^{-52}) (DP) or 2^{-126} * (significand x 2^{-23}) (SP)
{-54 < Er < 1 (DP) or -25 < Er < 1 (SP)}

ftt = unimplemented_FPop. The *unimplemented_FPop* floating-point trap type indicates that the processor decoded an FPop that it does not implement. In this case, the `cexc` field is unchanged.

All quad `FPop`s variations set `ftt` = *unimplemented_FPop*.

6.9.1.6 Floating-Point Exceptions Control and Status

There are three FSR register fields used to control and status the events associated with floating-point exceptions.

FSR_trap_enable_mask (TEM)

Bits 27 through 23 are enable bits for each of the five IEEE-754 floating-point exceptions that can be indicated in the current_exception field (`cexc`). See FIGURE 6-28 for an illustration. If a floating-point operate instruction generates one or more exceptions and the `TEM` bit corresponding to any of the exceptions is one, then this condition causes a *fp_exception_ieee_754* trap. A `TEM` bit value of zero prevents the corresponding exception type from generating a trap.

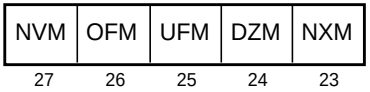


FIGURE 6-28 Trap Enable Mask (TEM) Fields of FSR

FSR_accrued_exception (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 6-29 for an illustration. After an FPop completes with ftt = 0, the TEM and cexc fields are logically ANDed together. If the result is nonzero, aexc is left unchanged and a *fp_exception_ieee_754* trap is generated; otherwise, the new cexc field is ORed into the aexc field and no trap is generated. Thus, while (and only while) traps are masked, exceptions are accumulated in the aexc field.

This field is also written with the appropriate value when an LDFSR or LDXFSR instruction is executed.

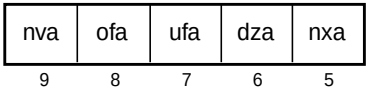


FIGURE 6-29 Accrued Exception Bits (aexc) Fields of FSR

FSR_current_exception (cexc)

Bits 4 through 0 indicate that 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. See FIGURE 6-30 for an illustration.

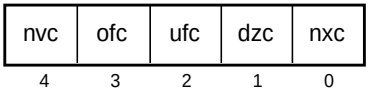


FIGURE 6-30 Current Exception Bits (cexc) Fields of FSR

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 non-privileged program.

The *cexc* bits are set as described in Section 6.9.1.7, “Floating-Point Exception Fields”,” by the execution of an FPop that either does not cause a trap or causes a *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 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:

- An IEEE 754 overflow condition (*of*) occurs:
  - If `OFM` = 0 and `NXM` = 0, the `cexc.ofc` and `cexc.nxc` bits are both set to one, the other three bits of `cexc` are set to zero, and a *fp_exception_ieee_754* trap does *not* occur.
  - If `OFM` = 0 and `NXM` = 1,the `cexc.nxc` bit is set to one, the other four bits of `cexc` are set to zero, and a *fp_exception_ieee_754* trap *does* occur.
  - If `OFM` = 1, the `cexc.ofc` bit is set to one, the other four bits of `cexc` are set to zero, and a *fp_exception_ieee_754* trap *does* occur.
- An IEEE 754 underflow condition (*uf*) occurs:
  - If `UFM` = 0 and `NXM` = 0, the `cexc.ufc` and `cexc.nxc` bits are both set to one, the other three bits of `cexc` are set to zero, and a *fp_exception_ieee_754* trap does *not* occur.
  - If `UFM` = 0 and `NXM` = 1, the `cexc.nxc` bit is set to one, the other four bits of `cexc` are set to zero, and a *fp_exception_ieee_754* trap *does* occur.
  - If `UFM` = 1, the `cexc.ufc` bit is set to one, the other four bits of `cexc` are set to zero, and a *fp_exception_ieee_754* trap *does* occur.

The behavior is summarized in TABLE 6-25 (where “x” indicates “don’t care”):

TABLE 6-25 Setting of FSR.cexc bits

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**	**Notes**
-	-	-	x	x	x	No	0	0	0	
-	-	1	x	x	0	No	0	0	1	
-	1	1	x	0	0	No	0	1	1	(1)
1	-	1	0	x	0	No	1	0	1	(2)

Notes:

(1) When the underflow trap is disabled (UFM = 0), underflow is always accompanied by inexact.
(2) Overflow is always accompanied by inexact.

TABLE 6-25 Setting of FSR.cexc bits *(Continued)*

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**	**Notes**
-	-	1	x	x	1	Yes	0	0	1	
-	1	1	x	0	1	Yes	0	0	1	
-	1	-	x	1	x	Yes	0	1	0	
-	1	1	x	1	x	Yes	0	0	0	
1	-	1	1	x	x	Yes	1	0	0	(2)
1	-	1	0	x	1	Yes	0	0	1	(2)

Notes:

(1) When the underflow trap is disabled (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.

6.9.1.7 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_invalid (nvc, nva)

An operand is improper for the operation to be performed. For example, $0.0 \div 0.0$ and $\infty - \infty$ are invalid; 1 = invalid operand(s), 0 = valid operand(s).

FSR_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.

FSR_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 zero. Otherwise:

- If UFM = 0, underflow occurs if a nonzero result is tiny and a loss of accuracy occurs.
- If UFM = 1, underflow occurs if a nonzero result is tiny.

SPARC-V9 allows underflow to be detected either before or after rounding. The UltraSPARC IIIi processor detects underflow before rounding.

FSR_division-by-zero (dzc, dza)

X ÷ 0.0, where X is subnormal or normalized; 1 = division by zero, 0 = no division by zero.

Note – 0.0 ÷ 0.0 *does not* set the dzc or dza bits.

FSR_inexact (nxc, nxa)

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

Programming Note – Software must be capable of simulating the operation of the FPU in order to properly handle the *unimplemented_FPop*, *unfinished_FPop*, and *IEEE_754_exception* floating-point trap types. Thus, a user application program always sees a FSR that is fully compliant with IEEE Std 754-1985.

6.10 ASI Mapped Registers

In this section, the Data Cache Unit Control Register and Data Watchpoint registers (virtual address data watchpoint and physical address data watchpoint) are described.

6.10.1 Data Cache Unit Control Register (DCUCR)

ASI 45_{16} (ASI_DCU_CONTROL_REGISTER), VA = 0_{16}

The DCUCR contains fields that control several memory-related hardware functions. The functions include instruction, prefetch, write and data caches, MMUs, and watchpoint setting.

After a *Power-On Reset* (POR), all fields of DCUCR are set to zero. After a WDR, XIR, or SIR, all fields of DCUCR defined in this section are set to zero.

The DCUCR is illustrated in FIGURE 6-31 and described in TABLE 6-26. In the table, the field definitions and bits are grouped by function rather than by a strict bit sequence.

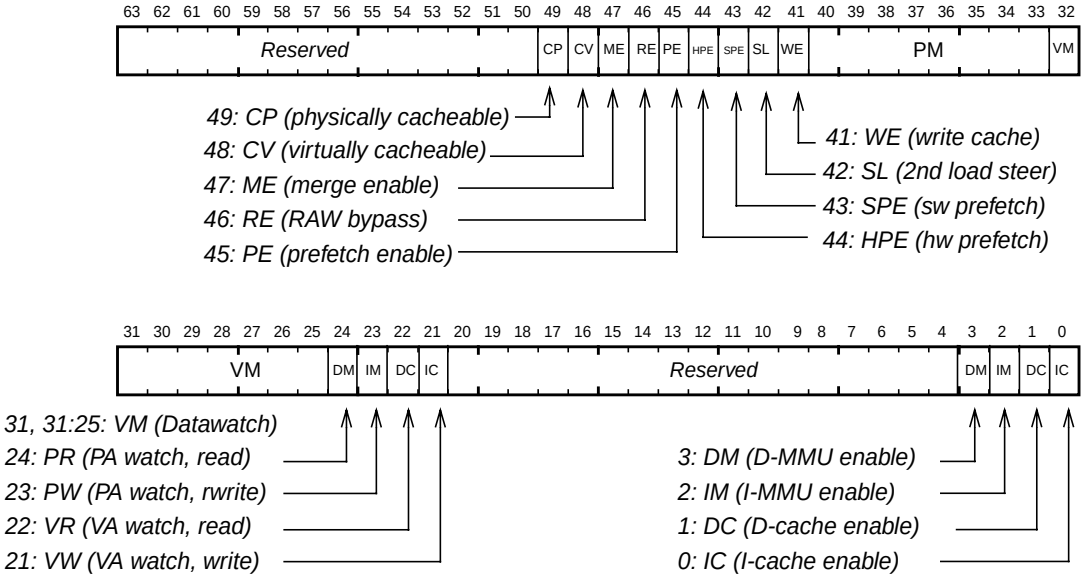


FIGURE 6-31 DCU Control Register Access Data Format (ASI 45_{16})

TABLE 6-26 DCUCR Bit Field Descriptions *(1 of 4)*

Bits	Field	Type	Description	Note
63:50, 20:4	*Reserved*	RW		
MMU Control				
49	CP	RW	*Cacheability of PA.* CP determines the physical cacheability of memory accesses when the I-MMU or D-MMU is disabled (IM = 0 or DM = 0). The TTE.E (side-effect) bit is set to the complement of CP when MMUs are enabled; 1 = cacheable, 0 = non-cacheable.	1

TABLE 6-26 DCUCR Bit Field Descriptions *(2 of 4)*

Bits	Field	Type	Description	Note
48	CV	RW	*Cacheability of VA*. CV determines the virtual cacheability of memory accesses when the D-MMU is disabled (DM = 0); 1 = cacheable, 0 = non-cacheable.	
3	DM		*D-MMU Enable*. If DM = 0, the D-MMU is disabled (pass-through mode). **Note:** When the MMU/TLB is disabled, a virtual address is passed through as a physical address.	
2	IM		*I-MMU Enable*. If IM = 0, the I-MMU is disabled (pass-through mode).	
Store Queue Control				
47	ME	RW	*Non-cacheable Store Merging Enable*. If cleared, no merging of non-cacheable, non-side-effect store data will occur. Each non-cacheable store will generate a system bus transaction.	
46	RE		*RAW Bypass Enable*. If cleared, no bypassing of data from the store queue to a dependent load instruction will occur. All load instructions will have their RAW predict field cleared.	
Prefetch Control				2
45	PE		*Prefetch Cache Enable*. If prefetch is disabled by clearing the PE bit, all references to the P-cache are handled as P-cache misses. If cleared, the P-cache does not generate any hardware prefetch requests to the L2-cache. Software prefetch instructions are not affected by this bit.	
44	HPE		*Prefetch Cache Hardware Prefetch Enable.*	*3*
43	SPE		*Software Prefetch Enable*. Clear to disable prefetch instructions. When disabled, software prefetch instructions do no generate a request to the L2-cache or the system interface. They will continue to be issued to the pipeline, where they will be treated as NOPs.	
Second Load Control				
42	SL		*Second Load Steering Enable*. If cleared, all load type instructions will be steered to the MS pipeline and no floating-point load type instructions will be issued to the A0 or A1 pipelines.	
I-cache, D-cache, and W-cache Control				
41	WE		*Write Cache Enable*. If zero, all W-cache references will be handled as W-cache misses. Each store queue entry will perform a RMW transaction to the L2-cache, and the W-cache will be maintained in a clean state. Software is required to flush the W-cache (force it to a clean state) before setting this bit to zero.	

TABLE 6-26 DCUCR Bit Field Descriptions *(3 of 4)*

Bits	Field	Type	Description	Note
1	DC		*Data Cache Enable*. The DC is used to enable/disable the operation of the data cache closest to the processor (D-cache); DC = 1 enables the D-cache and DC = 0 disables it. When DC = 0, memory accesses (loads, stores, atomic load-stores) are satisfied by caches lower in the cache hierarchy. When the D-cache is disabled, its contents are not updated. When the D-cache is reenabled, any D-cache lines still marked as "valid" may be inconsistent with the state of memory or other caches. In that case, software must handle any inconsistencies by flushing the inconsistent lines from the D-cache.	
0	IC		*Instruction Cache Enable*. The IC is used to enable/disable the operation of the instruction cache closest to the processor (I-cache); IC = 1 enables the I-cache and IC = 0 disables it. When IC = 0, instruction fetches are satisfied by caches lower in the cache hierarchy. When the I-cache is disabled, its contents are not updated. When the I-cache is reenabled, any I-cache lines still marked as "valid" may be inconsistent with the state of memory or other caches. In that case, software must handle any inconsistencies by invalidating the inconsistent lines in the I-cache.	
Watchpoint Control				
40:33	PM<7:0>		*DCU Physical Address Data Watchpoint Mask*. The Physical Address Data Watchpoint Register contains the physical address of a 64-bit word to be watched. The 8-bit Physical Address Data Watch Point Mask controls which byte(s) within the 64-bit word should be watched. If all eight bits are cleared, the physical watchpoint is disabled. If the watchpoint is enabled and a data reference overlaps any of the watched bytes in the watchpoint mask, then a physical watchpoint trap is generated. Watchpoint behavior for a Partial Store instruction may differ. Please see the VM field description in the table.	4

TABLE 6-26 DCUCR Bit Field Descriptions *(4 of 4)*

<table>
<tr><th>Bits</th><th>Field</th><th>Type</th><th>Description</th><th>Note</th></tr>
<tr><td>32:25</td><td>VM<7:0></td><td></td><td>DCU Virtual Address Data Watchpoint Mask. The Virtual Address Data Watchpoint Register contains the virtual address of a 64-bit word to be watched. This 8-bit mask controls which byte(s) within the 64-bit word should be watched. If all eight bits are cleared, then the virtual watchpoint is disabled. If watchpoint is enabled and a data reference overlaps any of the watched bytes in the watchpoint mask, then a virtual watchpoint trap is generated.
VA/PA data watchpoint byte mask examples are shown below.
<table><tr><th>Watchpoint Mask (PM and VM)</th><th>Least Significant 3 Bits of Address of Bytes Watched 7654 3210</th></tr><tr><td>00_{16}</td><td>Watchpoint disabled</td></tr><tr><td>01_{16}</td><td>0000 0001</td></tr><tr><td>32_{16}</td><td>0011 0010</td></tr><tr><td>FF_{16}</td><td>1111 1111</td></tr></table></td><td>4</td></tr>
<tr><td>24, 23</td><td>PR, PW</td><td></td><td>DCU Physical Address Data Watchpoint Enable. If PR (PW) is one, then a data read (write) that matches the range of addresses in the Physical Watchpoint Register causes a watchpoint trap. If both PR and PW are set, a watchpoint trap will occur on either a read or write access.</td><td></td></tr>
<tr><td>22, 21</td><td>VR, VW</td><td></td><td>DCU Virtual Address Data Watchpoint Enable. If VR (VW) is one, then a data read (write) that matches the range of addresses in the Virtual Watchpoint Register causes a watchpoint trap. If both VR and VW are set, a watchpoint trap will occur on either a read or write access.</td><td></td></tr>
</table>

1. The CP and CV bits of DCUCR must be changed with care. It is recommended that a MEMBAR #Sync be executed before and after CP or CV is changed. Also, software must manage cache states to be consistent before and after CP or CV is changed.
2. Prefetch is enabled in the UltraSPARC IIIi processor. Both hardware prefetch and software prefetch for data to the P-cache are valid only for floating-point load instructions and are not valid for integer load instructions.
3. Both Hardware prefetch and second load unit may not be enabled at the same time. Enabling both may cause incorrect program behavior.
4. Watchpoint exceptions on Partial Store instruction occur conservatively. The DCUCR.VM masks are only checked for nonzero value (watchpoint disabled). 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).

6.10.2 Data Watchpoint Registers

The UltraSPARC IIIi processor implements “break before” watchpoint traps. When the address of a data access matches a preset physical or virtual watchpoint address, instruction execution is stopped immediately before the watched memory location is accessed. TABLE 6-27 lists ASIs that are affected by the two watchpoint traps.

TABLE 6-27 ASIs Affected by Watchpoint Traps

ASI Type	ASI Range	Data MMU	Watchpoint If Matching VA	Watchpoint If Matching PA
Translating ASIs	04_{16}–11_{16}, 18_{16}–19_{16}, 24_{16}–$2C_{16}$, 70_{16}–71_{16}, 78_{16}–79_{16}, 80_{16}–FF_{16}	On Off	Y N	Y Y
Bypass ASIs	14_{16}–15_{16}, $1C_{16}$–$1D_{16}$	—	N	Y
Non-translating ASIs	30_{16}–$6F_{16}$, 72_{16}–77_{16}, $7A_{16}$–$7F_{16}$	—	N	N

For 128-bit (quad) atomic load and 64-byte block load and store instructions, a watchpoint trap is generated only if the watchpoint overlaps the lowest-address eight bytes of the access.

To avoid trapping infinitely, software should emulate the instruction that caused the trap and return from the trap by using a DONE instruction or turn off the watchpoint before returning from a watchpoint trap handler.

Two 64-bit data watchpoint registers provide the means to monitor data accesses during program execution. When Virtual/Physical Data Watchpoint is enabled, the virtual/physical addresses of all data references are compared against the content of the corresponding watchpoint register. If a match occurs, a *VA_watchpoint* or *PA_watchpoint* trap is signalled before the data reference instruction is completed. The virtual address watchpoint trap has higher priority than the physical address watchpoint trap.

Separate 8-bit byte masks allow watchpoints to be set for a range of addresses. Each zero bit in the byte mask causes the comparison to ignore the corresponding byte in the address. These watchpoint byte masks and the watchpoint enable bits reside in the DCUCR.

Virtual Address Data Watchpoint Register

ASI 58_{16}, VA = 38_{16}

Name: VA Data Watchpoint Register

FIGURE 6-32 illustrates the Virtual Address Watchpoint Register. **DB_VA** is the most significant 61 bits of the 64-bit virtual data watchpoint address.

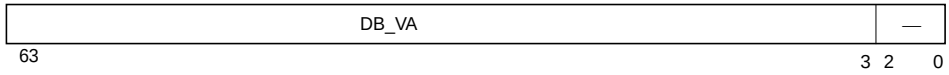


FIGURE 6-32 VA Data Watchpoint Register Format

Physical Address Data Watchpoint Register

ASI 58_{16}, VA=40_{16}

Name: PA Data Watchpoint Register

FIGURE 6-33 illustrates the PA Data Watchpoint Register. **DB_PA** is the most significant 61 bits of the physical data watchpoint address. The width of an UltraSPARC IIIi processor physical address is 43 bits.

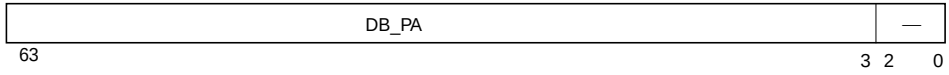


FIGURE 6-33 PA Data Watchpoint Register Format

Compatibility Note – The UltraSPARC IIIi processor supports a 43-bit physical address space. Software is responsible for writing a zero-extended 64-bit address into the `PA Data Watchpoint` register.

Data Watchpoint Reliability

The processor supports watchpoint comparison on the MS (memory) pipeline; any second-issue (Ax pipeline) floating-point loads will not trigger a watchpoint. For reliable use of the watchpoint mechanism, the second floating-point load feature must be disabled using `DCUCR.SL`.

CHAPTER 7

Instruction Types

Instructions are accessed by the processor from memory and are executed, annulled, or trapped. Instructions are discussed in seven general categories. The processor instructions are described in the following sections:

Learning the Instructions

- Section 7.1, “Introduction”
- Section 7.2, “Memory Addressing for Load and Store Instructions”
- Section 7.3, “Integer Execution Environment”
- Section 7.4, “Floating-Point Execution Environment”
- Section 7.5, “VIS Execution Environment”
- Section 7.6, “Data Coherency Instructions”
- Section 7.7, “Register Window Management Instructions”
- Section 7.8, “Program Control Transfer Instructions”
- Section 7.9, “Prefetch Instructions”

Reference Sections

- Section 7.10, “Instruction Summary Table by Category”
- Section 7.10.5, “Integer Execution Environment Instructions”
- Section 7.10.6, “Floating-Point Execution Environment Instructions”
- Section 7.10.7, “VIS Execution Environment Instructions”
- Section 7.10.8, “Data Coherency Instructions”
- Section 7.10.9, “Register-window Management Instructions”
- Section 7.10.10, “Program Control Transfer Instructions”
- Section 7.10.11, “Data Prefetch Instructions”

- Section 7.11, “Instruction Formats and Fields”
- Section 7.12, “Reserved Opcodes and Instruction Fields”
- Section 7.13, “Big/Little-Endian Addressing”

7.1 Introduction

The processor’s RISC architecture is defined primarily by the SPARC-V9 architecture. The UltraSPARC II processors were the first to extend the SPARC-V9 architecture with new instructions and additional logic units. The UltraSPARC IIIi processor further extends this instruction execution environment.

The UltraSPARC IIIi processor provides backward compatibility for SPARC application programs. Upgraded system software is required. Noteworthy enhancements to the processor include greater capability in the execution units to improve instruction scheduling, new VIS instructions to reduce the length of code sequences, and data prefetch instructions to provide the compiler with ways to improve cache hit rates.

Our compiler and other software development tools take advantage of the new instruction features to increase parallel execution, reduce code size, and achieve shorter instruction execution latencies.

7.2 Memory Addressing for Load and Store Instructions

SPARC-V9 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. SPARC-V9 also can support 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 unit being accessed.

7.2.1 Integer Unit Memory Alignment Requirements

Halfword accesses are aligned on 2-byte boundaries; word accesses (which include instruction fetches) are aligned on 4-byte boundaries; extended-word and doubleword accesses are aligned on 8-byte boundaries. An improperly aligned address in a load, store, or load-store instruction causes a trap to occur, with possible exceptions.

Programming Note – By setting `i` = 1 and `rs1` = 0, you can access any location in the lowest or highest 4 KB of an address space without using a register to hold part of the address.

7.2.2 FP/VIS Memory Alignment Requirements

Extended word and doubleword (64-bit) accesses must be aligned on 8-byte boundaries, quadword accesses must be aligned on 16-byte boundaries, and block load (BLD) and block store (BST) accesses must be aligned on 64-byte boundaries.

All references are 32, 64, or 128 bits. They must be naturally aligned to their data width in memory except for double-precision floating-point (FP) values, which may be aligned on word boundaries. However, if so aligned, doubleword loads/stores may not be used to access them, resulting in less efficient and nonatomic accesses.

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

- An `LDDF` or `LDDFA` instruction accessing an address that is word aligned but not doubleword aligned causes an *LDDF_mem_address_not_aligned* exception.
- An `STDF` or `STDFA` instruction accessing an address that is word aligned but not doubleword aligned causes an *STDF_mem_address_not_aligned* exception.

7.2.3 Byte Order Addressing Conventions (Endianess)

The processor uses big-endian byte order for all instruction accesses and, by default, for data accesses. It is possible to access data in little-endian format by using load and store alternate instructions that support little-endian data structures. It is also possible to change the default byte order for implicit data accesses.

See Section 7.13, “Big/Little-Endian Addressing” for details.

7.2.4 Address Space Identifiers (ASIs)

Versions of load/store instructions, the *load and store alternate* instructions, can specify an 8-bit address space identifier (ASI) to go along with the load/store data instruction.

The load and store alternate instructions have the following three sources of ASIs:

- Explicit immediate of instruction
- ASI Register reference
- Hardcode to the instruction

Supervisor software (privileged mode) uses ASIs to access special, protected registers, such as MMU, cache control, and processor state registers, and other processor- or system-dependent values.

ASIs are also used to modify the function of many instructions. This overloading of load/store instructions provide partial store, block load/store, and atomic memory access operations.

Implicit ASI Value

Load and store instructions provide an implicit ASI value of `ASI_PRIMARY`, `ASI_PRIMARY_LITTLE`, `ASI_NUCLEUS`, or `ASI_NUCLEUS_LITTLE`. Load and store alternate instructions provide an explicit ASI, specified by the `imm_asi` instruction field when `i` = 0, or the contents of the ASI register when `i` = 1.

Privileged and Non-Privileged ASIs

ASIs 00_{16} through $7F_{16}$ are restricted; only privileged software is allowed to access them. An attempt to access a restricted ASI by non-privileged software results in a *privileged_action* exception. ASIs 80_{16} through FF_{16} are unrestricted; software is allowed to access them whether the processor is operating in privileged or non-privileged mode.

Compatibility Note – The SPARC-V9 architecture provides the basic framework and defines the required ASIs for the processor. Other ASIs are defined (and sometimes re-defined) for a specific processor or family of processors as allowed by the SPARC-V9 architecture.

Implementation Note – The processor decodes all eight bits of each ASI specifier. In addition, the processors redefine certain ASIs as appropriate for a specific processor.

7.2.5 Maintaining Data Coherency

The processor's memory architecture requires some software intervention to provide data coherency during program execution. These requirements are discussed in Chapter 8 "Memory Models" using the `FLUSH` and `MEMBAR` instructions described in Section 7.6, "Data Coherency Instructions."

The two types of data coherency instructions are needed to flush the cache for self-modifying code and to write data buffers out to memory.

7.3 Integer Execution Environment

7.3.1 IU Data Access Instructions

Load, store, and atomic instructions are the only instructions that access memory. All the IU data access instructions, except the compare and store (CASx) use either two `r` registers or SIMM13, a signed 13-bit immediate value, to calculate a 64-bit, byte-aligned memory address. Compare and Swap uses a single `r` register to specify a 64-bit memory address. Floating-point register load and store instructions are discussed in Section 7.4.2, "FPU/VIS Data Access Instructions."

The processor appends an ASI to the 64-bit address used with all the data access instructions.

Note – In addition to the large physical main memory, the processor has many memory mapped control, status, and diagnostic registers that are accessed using load and store instructions with an appropriate ASI value.

The destination field of the data access instruction specifies an `r` or `f` (single, double/ extended, or quadword) register that supplies the data for a store or that receives the data from a load.

7.3.1.1 Load and Store Instructions

Integer load and store instructions support byte, halfword (16-bit), word (32-bit), and doubleword (64-bit) accesses. Some versions of integer load instructions perform sign extension on 8-, 16-, and 32-bit values as they are loaded into a 64-bit destination register.

7.3.1.2 Move Instruction

There is no explicit integer move instruction. A move instruction can be easily synthesized by adding, subtracting or OR-ing a zero with a register and pointing the result to another register. The zero can come as a register input (such as `%r0` that has a value zero in SPARC-V9) or as an immediate input to the instruction.

7.3.1.3 Conditional Move Instructions

Based on Integer (icc/xcc) and Floating-Point (fcc) Condition Codes

This subsection describes two instructions that copy the contents of one register to another register within the same register file: one instruction for moving within the integer register file and another for moving within the floating-point register file.

- `MOVcc` Instruction

If a specified `icc`/`xcc` or `fcc` condition is satisfied, then the `MOVcc` instruction copies the contents of any integer to a destination integer register.

- `FMOVcc` Instruction

If a specified `icc`/`xcc` or `fcc` condition is satisfied, then the `FMOVcc` instruction copies the contents of any floating-point register to a destination floating-point register.

(A similar set of conditional move instructions are based on an integer register value. These conditional move instructions are described in Section 7.4, “Floating-Point Execution Environment.”)

The condition code to test is specified in the instruction and may be any of the conditions allowed in conditional delayed control transfer instructions. This condition is tested against 1 of the 6 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 situations, branches will take more clock cycles than the `MOVcc` or `FMOVcc` instructions.

For example, the following 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, %g0,1, %i3  ! overwrite X with 1 if A > B
```

which eliminates the need for a branch.

Based on Integer Register Value

There are separate versions for the IU and floating-point unit (FPU) register files:

- `MOVr` Instruction

If the contents of an integer register satisfy a specified condition, then the `MOVr` instruction copies the contents of any integer register to a destination integer register.

- `FMOVr` Instruction

If the contents of an integer register satisfy a specified condition, then the `FMOVr` instruction copies the contents of any floating-point register to a destination floating-point register.

The conditions to test are enumerated in TABLE 7-1.

TABLE 7-1 `MOVr` and `FMOVr` Test Conditions

Condition	Symbol	Description
NZ	≠ 0	Nonzero
Z	= 0	Zero
LZ	< 0	Less than zero
LEZ	≤ 0	Less than or equal to zero
GZ	> 0	Greater than zero
GEZ	≥ 0	Greater than or equal to zero

Any of the integer registers 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 to emulate multiple unsigned condition codes by using an integer register to hold the result of a comparison.

7.3.1.4 Atomic Instructions

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

The SWAP and LDSTUB instructions can optionally access alternate space. (The CASA instruction always accesses alternate memory spaces.) If the ASI specified for any alternate form of these instructions is a privileged ASI (value 80_{16}), then the processor must be in privileged mode to access it.

Atomic Quad Load Instruction (LDDA with ASI xx)

The atomic quad load instruction supplies an indivisible quadword (16-byte) load that is important in system software programs.

Compare and Swap Atomic Instruction (CASA)

An r register specifies the value that is compared with the value in memory at the computed address. CASA accesses words, and CASXA accesses doublewords.

If the values are equal (memory location and r register), 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 was to receive the value at the addressed memory location; in this case, the addressed memory location remains unchanged.

Swap Atomic Instruction (SWAP^D)

The destination register identifies the r register to be exchanged atomically with the calculated memory location. SWAP accesses words.

Load-Store Unsigned Byte (LDSTUB)

The LDSTUB instruction reads a byte from memory and writes ones to the location read. LDSTUB accesses bytes.

7.3.2 IU Arithmetic Instructions

The integer arithmetic instructions are generally triadic-register-address instructions that compute a result of a function of two source operands. They either write the result into the destination register `r[rd]` or discard it. One of the source operands is always `r[rs1]`. The other source operand depends on the `i` bit in the instruction. If `i` = 0, then the operand is `r[rs2]`. If `i` = 1, then the operand is the immediate constant `simm10`, `simm11`, or `simm13` sign-extended to 64 bits.

The arithmetic/logical/shift instructions perform arithmetic, tagged arithmetic, logical, and shift operations. One exception is the `SETHI` instruction that can be used in combination with another arithmetic or logical instruction to create a 32-bit constant in an `r` register.

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.

7.3.2.1 Integer Add and Subtract Instructions

Sixty-four bit arithmetic is performed on two `r` registers to generate a 64-bit result. The `icc` and `xcc` condition codes can be optionally set.

7.3.2.2 Tagged Integer Add and Subtract Instructions

The tagged arithmetic instructions assume that the least-significant two bits of each operand are a data-type tag. These instructions set the integer condition code (`icc`) and extended integer condition code (`xcc`) overflow bits on 32-bit (`icc`) or 64-bit (`xcc`) arithmetic overflow.

The tagged instructions are described in Appendix A “Instruction Definitions.”

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 `xcc` overflow bit is not affected by the tag bits.

The trapping versions (`TADDccTV`, `TSUBccTV`) are deprecated. See Section A.70.16, “Tagged Add and Trap on Overflow” and Section A.70.17, “Tagged Subtract and Trap on Overflow” for details.

7.3.2.3 Integer Multiply and Divide Instructions

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, $32 \times 32 \rightarrow 64$-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.

Some versions of the 32-bit multiply and divide instructions set the condition codes.

7.3.2.4 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. It is primarily used to construct constants in registers.

7.3.2.5 Integer Shift Instructions

Shift logical instructions (`SLL`, `SRL`) shift an `r` register left or right by an immediate constant in the instruction or by the amount pre-loaded in an `r` register.

7.3.3 IU Logic Instructions

7.3.3.1 `ADD`, `ANDN`, `OR`, `ORN`, `XOR`, `XNOR` Instructions

These are standard logic operations that work on all 64 bits of the register. The instructions can optionally set the integer condition codes (`icc/xcc`).

7.3.4 IU Compare Instructions

A special comparison instruction for integer values is not needed since it is easily synthesized with the "subtract and set condition codes" (`SUBcc`) instruction.

7.3.5 IU Miscellaneous Instructions

7.3.5.1 Interval Arithmetic Mode Instruction (SIAM) (VIS II)

The Set Interval Arithmetic Mode (SIAM) instruction sets the interval arithmetic mode fields in the graphics status register (GSR).

7.3.5.2 Align Address Instruction

The ALIGNADDR instruction takes two r registers and adds them together. The three least significant bits are forced to zero.

The ALIGNADDRL instruction supports little-endian data structures by taking the two r registers, adding them together, and placing the two's-complement of the three least significant bits of the result and storing them in the 3-bit GSR.ALIGN field.

7.3.5.3 Population of Ones Count

A population opcode is defined but not implemented in hardware; instead, a trap is generated.

7.3.5.4 Privileged Register Access Instructions

The privileged register access instructions read and write another group of state and status registers called privileged registers. These registers are visible only to privileged software. The read privileged register instruction moves the privileged register contents into an r register. The write privileged register instruction moves the contents of an r register into the selected privileged register.

7.3.5.5 State Register Access Instructions

The state register instructions access program-visible state and status registers. The read state register instruction moves the state register contents into an r register. The write state register instruction moves the contents of an r register into the selected state register.

Some state registers can only be accessed in privileged mode, others in either privileged or non-privileged mode. Some registers have access bits to restrict their availability as desired by the privileged software.

7.4 Floating-Point Execution Environment

The floating-point and VIS execution unit includes the floating-point register file for floating-point and fixed-point data formats and the execution pipelines for floating-point and VIS instructions.

This execution unit is a single unit that may be referred to any one of the following, depending on the textual context:

- Floating-point Unit (FPU)
- Floating-point and Graphics Unit (FGU)
- VIS Execution Unit (VIS)
- FPU/VIS

Note – The instructions associated with the FPU/VIS execution unit are divided between floating-point and VIS execution environments, but otherwise use the same hardware pipelines.

7.4.1 Floating-Point Operate Instructions

Floating-point operate (FPop) instructions perform all floating-point calculations; they are register-to-register instructions that operate on the floating-point registers. Like arithmetic, logical, and shift instructions, FPops compute a result that is a function of one or two source operands. Specific floating-point operations are selected by a subfield of the `FPop1`/`FPop2` instruction formats.

FPops are generally triadic-register-address instructions. They compute a result that is a function of one or two source operands and place the result in one or more destination `f` registers, with two exceptions:

- Floating-point convert operations, which use one source and one destination operand
- Floating-point compare operations, which do not write to an `f` register but update one of the `fcc`*n* fields of the `FSR` instead

The term “FPop” refers to those instructions encoded by the `FPop1` and `FPop2` opcodes and does *not* include branches based on the floating-point condition codes (`FBfcc`^{D} and `FBPfcc`) or the load/store floating-point instructions.

If `PSTATE.PEF` = 0 or `FPRS.FEF` = 0, then any instruction, including an FPop instruction, that attempts to access a `FPU` register generates a *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; therefore, they clear `cexc` and leave `aexc` unchanged.

Note – The processor may indicate that a floating-point instruction did not produce a correct IEEE Standard 754-1985 result by generating a *fp_exception_other* exception with `FSR.ftt` = *unfinished_FPop* or unimplemented FPop. In this case, privileged software must emulate any functionality not present in the hardware.

The processor does not implement quad-precision floating-point operations in hardware. Instead, these operations cause a *fp_exception_other* trap with `FSR.ftt` = *unimplemented_FPop*, and the system software emulates quad operations.

7.4.2 FPU/VIS Data Access Instructions

Floating-point load and store instructions support word, doubleword, and quadword memory accesses.

There are no move instructions to move data directly between the integer and floating-point register files.

7.4.2.1 Load Instructions

Byte, halfword, word, and double/extended word data widths are supported with access to alternate address spaces. Data loaded into a register that is not 64 bits is filled with zeroes in the high-order bits.

7.4.2.2 Store Instructions

Byte, halfword, word, and double/extended word data widths are supported with access to alternate address spaces.

7.4.2.3 Block Load and Store Instructions

Block load and store access eight consecutive doublewords. The LDDFA instruction is used with the various ASIs to specify a type of block transaction. The LDDFA instruction is specified with ASIs 70, 71, 78, 79, F0, F1, F8, F9, E0, and E1 to select between primary and secondary D-MMU contexts, little- and big-endian, privileged and non-privileged, and a set of block commit store ASIs.

7.4.2.4 Conditional Move Instructions

The FP/VIS conditional move instructions are described with the IU conditional move instructions, Section 7.3.1.3.

7.4.3 Floating-Point Arithmetic Instructions

Single-precision and double-precision FP is executed in hardware. Quad precision (128-bit) instructions are recognized by the processor and trapped so they can be emulated in software.

7.4.3.1 Absolute Value and Negate Instructions

These instructions modify the sign of the floating-point operand.

7.4.3.2 Add and Subtract Instructions

These instructions use standard IEEE operation.

7.4.3.3 Multiply Instructions

These instructions use standard IEEE operation with some exceptions.

7.4.3.4 Square Root and Divide Instructions

The square root and divide instructions begin their execution in the FGM pipeline and block new instructions from entering until the result is nearly ready to leave the pipeline and be written to the register file.

7.4.4 Floating-Point Conversion Instructions

The following FP conversions are supported. Conversions do not generate `fcc` condition codes.

7.4.4.1 Floating-Point to Integer

All floating-point precision to word and double/extended word integer conversions are supported.

7.4.4.2 Integer to Floating-Point

Word and double/extended word integer to all floating-point precision number conversions are supported.

7.4.4.3 Floating-Point to Floating-Point

All floating-point precision to all floating-point precision number conversions are supported.

7.4.5 Floating-Point Compare Instructions

The same precision operands are compared and the `fcc` condition codes are set.

7.4.6 Floating-Point Miscellaneous Instructions

7.4.6.1 Load and Store FSR Register

The FSR register is accessed by load and store instructions into and out of the floating-point register file.

7.4.6.2 Data Alignment Instruction

The data alignment instruction `FALIGNDATA` concatenates two registers (16 bytes) and stores a contiguous block of eight of these bytes starting at the offset stored in the `GSR.ALIGN` field.

7.5 VIS Execution Environment

The floating-point and VIS execution unit includes the floating-point register file for floating-point and fixed-point data formats and the execution pipelines for floating-point and VIS instructions.

This execution unit is a single unit that may be referred to any one of the following, depending on the textual context:

- Floating-point Unit (FPU)
- Floating-point and Graphics Unit (FGU)
- VIS Execution Unit (VIS)
- FPU/VIS

Note – The instructions associated with the FPU/VIS execution unit are divided between floating-point and VIS execution environments, but otherwise use the same hardware pipelines.

7.5.1 VIS Pixel Data Instructions

7.5.1.1 Array Instruction

These instructions convert three-dimensional (3D) fixed-point addresses to a blocked-byte address.

7.5.1.2 Byte Mask and Shuffle Instructions

Byte Mask instruction adds two integer registers and stores the result in the integer register. The least significant 32 bits of the result are stored in a special field.

Byte Shuffle concatenates the two 64-bit floating-point registers to form a 16-byte value. Bytes in the concatenated value are numbered from most significant to least significant, with the most significant byte being byte 0.

7.5.1.3 Edge Handling Instructions

These instructions handle the boundary conditions for parallel pixel scan line loops, where the address of the next pixel to render and the address of the last pixel in the scan line are provided.

7.5.1.4 Pixel Packing Instructions

These instructions convert multiple values in a source register to a lower-precision fixed or pixel format and store the resulting values in the destination register. Input values are clipped to the dynamic range of the output format. Packing applies a scale factor to allow flexible positioning of the binary point.

7.5.1.5 Expand and Merge Instructions

Expand takes four 8-bit unsigned integers, 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.

Merge interleaves four corresponding 8-bit unsigned values to produce a 64-bit value in the 64-bit floating-point destination register. This instruction converts from packed to planar representation when it is applied twice in succession.

7.5.1.6 Pixel Distance Instruction

Eight unsigned 8-bit values are contained in the 64-bit floating-point source registers. The corresponding 8-bit values in the source registers are subtracted. The sum of the absolute value of each difference is added to the integer in the 64-bit floating-point destination register. The result is stored in the destination register. Typically, this instruction is used for motion estimation in video compression algorithms.

7.5.2 VIS Fixed-Point 16-bit and 32-bit Data Instructions

7.5.2.1 Partitioned Add and Subtract Instructions

The standard versions of these instructions perform four 16-bit or two 32-bit partitioned adds or subtracts between the corresponding fixed-point values contained in the source operands.

The single-precision versions of these instructions perform two 16-bit or one 32-bit partitioned add(s) or subtract(s); only the low 32 bits of the destination register are affected.

7.5.2.2 Partitioned Multiply Instructions

These instructions multiply signed and unsigned registers of different sizes and place the results in different types of destination registers.

7.5.2.3 Pixel Compare Instruction

Either four 16-bit or two 32-bit fixed-point values in the 64-bit floating-point source registers are compared. The 4-bit or 2-bit results are stored in the least significant bits in the integer destination register. Signed comparisons are used.

7.5.3 VIS Logic Instructions

7.5.3.1 Fill with Ones and Zeroes Instruction

These instructions perform a zero fill or a one fill.

7.5.3.2 Source Copy

These instructions perform a source copy.

7.5.3.3 AND, OR, NAND, NOR, and XNOR Instructions

These instructions perform the logical operations.

7.6 Data Coherency Instructions

The processor implements a Total Store Ordering (TSO) that provides the majority of data coherency support in hardware. Two instructions are used with this model to synchronize the data for memory operations to insure the latest data is accessed for load instructions and DMA activity.

Chapter 8 “Memory Models” discusses TSO in detail.

7.6.1 FLUSH Instruction Cache Instruction

The FLUSH instruction is used to flush the caches out to main memory. The MEMBAR instruction is used to flush the various data buffers in the processor out to data coherent domain.

Self-modifying code (storable in the unified L2-cache) requires the use of the FLUSH instruction.

Note – The FLUSHW instruction flushes the Window-registers and is not related to the FLUSH command for the I-cache.

7.6.2 MEMBAR (Memory Synchronization) Instruction

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

7.6.3 Store Barrier Instruction

Note – STBAR^{P} is also supported, but this instruction is deprecated and should not be used in newly developed software.

7.7 Register Window Management Instructions

Register window instructions manage the register windows. SAVE and RESTORE are non-privileged and cause a register window to be pushed or popped. FLUSHW is non-privileged 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.

The instructions that manage register windows include SAVE, RESTORE, SAVED^{P}, RESTORE^{P}, and FLUSHW.

SAVE Instruction

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

RESTORE Instruction

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

SAVED^P Instruction

The SAVED instruction is used by a spill trap handler to indicate that a window spill has completed successfully. It increments CANSAVE.

RESTORED^P Instruction

The RESTORED instruction is used by a fill trap handler to indicate that a window has been filled successfully. It increments CANRESTORE.

Flush Register Windows Instruction

The FLUSHW instruction cleans register windows of the data from other processes to insure a secure execution environment.

7.8 Program Control Transfer Instructions

Control transfer instructions (CTIs) include PC-relative branches and calls, register-indirect jumps, and conditional traps. Most of the CTIs are delayed; that is, the instruction immediately following a CTI 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 CTI in memory.

The instruction following a delayed CTI is called a *delay* instruction. A bit in a delayed CTI (the *annul bit*) can cause the delay instruction to be annulled (that is, to have no effect) if the branch is not taken (or in the "branch always" case if the branch is taken).

Compatibility Note – SPARC V8 specified that the delay instruction was always fetched, even if annulled, and an annulled instruction could not cause any traps. SPARC-V9 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 MB; the "branch on condition codes with prediction" instruction provides a displacement of ±1 MB; the "branch on register contents" instruction provides a displacement of ±128 KB; and the CALL instruction's 30-bit word displacement allows a control transfer to any address within ±2 GB ($\pm 2^{31}$ bytes).

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

7.8.1 Control Transfer Instructions (CTIs)

The following are the basic CTI types:

- Conditional branch (Bicc^{D}, BPcc, BPr, FBfcc^{D}, FBPfcc)
- Unconditional branch
- Call and link (CALL)
- Jump and link (JMPL, RETURN)
- Return from trap (DONE^{P}, RETRY^{P})
- Trap (Tcc, ILLTRAP)
- No Operation (NOP, SIR when in non-privileged mode)

A CTI functions by changing the value of the next program counter (nPC) or by changing the value of both the program counter (PC) and the nPC. When only the next program counter, nPC, is changed, the effect of the transfer of control is delayed by one instruction. Most control transfers are of the delayed variety. The instruction following a delayed CTI is said to be in the *delay slot* of the CTI. Some CTI (branches) can be optionally annul, that is, not execute, the instruction in the delay slot, depending upon whether the transfer is taken or not taken. Annulled instructions have no effect upon the program-visible state, nor can they cause a trap.

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. The UltraSPARC IIIi processor is a superscalar SPARC implementation in which 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 7-2 defines the value of the `PC` and the value of the `nPC` 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` (same as `Bicc`), and branches that are unconditional, that is, always or never taken, represented in the table by `B`. The effect of an annulled branch is shown in the table through explicit transfers of control, rather than fetching and annulling the instruction.

TABLE 7-2 Control Transfer Characteristics

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
B	PC-relative	Yes	Yes	0	nPC	EA
B	PC-relative	Yes	No	0	nPC	nPC + 4
B	PC-relative	Yes	Yes	1	EA	EA + 4
B	PC-relative	Yes	No	1	nPC + 4	nPC + 8
CALL	PC-relative	Yes	—	—	nPC	EA
JMPL, RETURN	Register-indirect	Yes	—	—	nPC	EA
DONE	Trap state	No	—	—	TNPC[TL]	TNPC[TL] + 4
RETRY	Trap state	No	—	—	TPC[TL]	TNPC[TL]
Tcc	Trap vector	No	Yes	—	EA	EA + 4
Tcc	Trap vector	No	No	—	nPC	nPC + 4

The effective address (EA) in TABLE 7-2 specifies the target of the control transfer instruction. The EA is computed in different ways, depending on the particular instruction:

- **PC-relative effective address** — A PC-relative EA is computed by sign extending the instruction's immediate field to 64 bits, left-shifting the word displacement by two bits to create a byte displacement, and adding the result to the contents of the PC.
- **Register-indirect effective address** — A register-indirect EA computes its target address as either r[rs1] + r[rs2] if i = 0, or r[rs1] + sign_ext(simm13) if i = 1.
- **Trap vector effective address** — A trap vector EA first computes the software trap number as the least significant 7 bits of r[rs1] + r[rs2] if i = 0, or as the least significant 7 bits of r[rs1] + *sw_trap#* if i = 1. The trap level, TL, is incremented. The hardware trap type is computed as 256 + *sw_trap#* and stored in TT[TL]. The EA is generated by concatenation of the contents of the TBA register, the "TL > 0" bit, and the contents of TT[TL].
- **Trap state effective address** — A trap state EA is not computed but is taken directly from either TPC[TL] or TNPC[TL].

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

SPARC V8 left undefined the result of executing a delayed conditional branch that had a delayed control transfer in its delay slot. For this reason, programmers should avoid such constructs when backward compatibility is an issue.

7.8.1.1 Conditional Branches

A conditional branch transfers control if the specified condition is true. If the annul bit is zero, the instruction in the delay slot is always executed. If the annul bit is one, the instruction in the delay slot is *not* executed *unless* the conditional branch is taken.

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

7.8.1.2 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 zero, then the instruction in the delay slot is always executed. If the annul bit is one, then the instruction in the delay slot is *never* executed.

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

7.8.1.3 CALL/JMPL and RETURN Instructions

CALL

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.

When PSTATE.AM = 1, the value of the high-order 32 bits is transmitted to r[15] by the CALL instruction.

Jump and Link

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[rd] by the JMPL instruction is zero.

RETURN

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

7.8.1.4 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.

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

7.8.1.5 Trap Instruction (Tcc)

The Tcc instruction initiates a trap if the condition specified by its cond field matches the current state of the condition code register specified by 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 the trap table pointed to by TBA.

A Tcc instruction can specify 1 of 128 software trap types. When a Tcc is taken, 256 plus the seven least significant bits of the sum of the Tcc's source operands is 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.

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

7.8.1.6 ILLTRAP

The `ILLTRAP` instruction causes an *illegal_instruction* exception.

7.8.1.7 NOP

A `NOP` instruction occupies the entire (single) instruction group and performs no visible work.

There are other instructions that also result in an operation that has no visible effect:

- `SIR` instruction executed in non-privileged mode
- `SHUTDOWN` instruction executed in privileged mode

There are other instructions that appear to be a NOP as long as they do not affect the condition codes.

7.9 Prefetch Instructions

The prefetch instruction is used to request that data be fetched from memory and put into the cache(s) if not already there for use in the floating-point and VIS execution environment. A subsequent load, if properly scheduled, can expect the data to more likely be in the cache, reducing the number of times the pipeline must recycle and thus improving performance.

The destination field of a `PREFETCH` instruction (`fcn`) is used to encode the prefetch type. The `PREFETCHA` instruction supports accesses to alternate space.

`PREFETCH` accesses at least 64 bytes.

7.10 Instruction Summary Table by Category

A summary of instructions are categorized in TABLE 7-3.

7.10.1 Instruction Superscripts

`INSTRUCTION`^{p} - Instruction must execute in privileged mode.

`INSTRUCTION` - Instruction can execute in privileged or non-privileged mode.

7.10.2 Instruction Mnemonics Expansion

`INSTRUCTION{_A}` - means INSTRUCTION, INSTRUCTION_A

`INSTRUCTION_(A,B,C)` - means INSTRUCTION_A, INSTRUCTION_B, and `INSTRUCTION_C`

7.10.3 Instruction Grouping Rules

Instruction grouping rules are explained in detail in Chapter 4 “Instruction Execution.”

Execution Latency

All instructions execute within the pipeline except the following:

- `FSQRT` (floating-point square root)
- `FPDIVx` (floating-point divide)

The latency of these instructions depend on the precision of the floating-point values. Some instructions execute early in the pipeline and have special bypass abilities. The details of the execution latencies are explained in Chapter 4 “Instruction Execution.”

7.10.4 Table Organization

The Instruction Summary Table has the following main sections:

- Integer Execution Environment (TABLE 7-3)
  - Data access, Arithmetic, Logic, Compare, Miscellaneous instructions
- Floating-point Execution Environment (TABLE 7-4)
  - FP/VIS data access, FP arithmetic/logic/compare/miscellaneous
- VIS Execution Environment (TABLE 7-5)
  - VIS pixel and fixed-point arithmetic/logic
- Data Coherency Instructions (TABLE 7-6)
- Register-window Management Instructions (TABLE 7-7)

- Program Control Transfer Instructions (TABLE 7-8)
- Prefetch Instructions (TABLE 7-9)

Shaded areas indicate instructions that are completely deprecated (entire row) or always privileged (cell holding instruction name). Deprecated and privilege status is identified with a ^{D} or ^{P} superscript, respectively.

7.10.5 Integer Execution Environment Instructions

TABLE 7-3 Instruction Summary for the Integer Execution Environment *(1 of 3)*

Instruction	Description		Notes
Integer Execution Environment			
IU Data Access Instructions B= byte; H= halfword; W=word;		**ASI Load (hex)**	
LDD^{D}	Load integer double word	No	
LDDA$^{D, PASI}$	Load integer double word from alternate space		
LDDA^{PASI}	Atomic quad load	24, 2C	
LDS$_{(B,H,W)}$	Load signed extended byte, halfword, or word: Memory → IU register	No	
LDX	Load extended (double) word	No	
LDXA^{PASI}	Load extended (double) word from alternate space		
LDS$_{(B,H,W)}$A^{PASI}	Load signed extended byte, halfword, or word from alternate space		
LDSTUB	Load-store (atomic) unsigned byte: Memory → IU register & Compare logic; IU register → Memory (conditional)	No	
LDSTUBA^{PASI}	Load-store (atomic) unsigned byte (see LDSTUB) in alternate space		
LDU$_{(B,H,W)}$	Load unsigned byte, halfword, word: Memory → IU register		
LDU$_{(B,H,W)}$A^{PASI}	Load unsigned byte, halfword, word from alternate space		
ST$_{(B,H,W,D^{D},X)}$	Store byte, halfword, word, double, or extended word: IU register → Memory		
ST$_{(B,H,W,D^{D},X)}$A^{PASI}	Store byte, halfword, word, double, or extended word in alternate space		
MOVcc	Conditional move based on icc/fcc: IU register → IU register		1
MOVr	Conditional move based on IU register value: IU register → IU register		2
CASA^{PASI}, CASXA^{PASI}	Atomic Compare and Swap word/double word in alternate space: Memory → Compare logic Memory ↔ (conditional) Working register		3, 4, 5
SWAP^{D}{A$^{D, PASI}$}	Atomically swap optionally with alternate space: IU register ↔ Memory		

TABLE 7-3 Instruction Summary for the Integer Execution Environment *(2 of 3)*

Instruction	Description		Notes
IU Arithmetic Instructions S= signed; U= unsigned; X= 64-bit (otherwise 32)			
ADD{cc}	Integer add		
ADDC{cc}	Integer add with carry		
SUB{cc}	Integer subtract, optionally setting `icc`/`xcc`		
SUBC{cc}	Integer subtract with carry, optionally setting icc/xcc		
MULX	Signed or unsigned 64-bit multiply		
(S,U)MUL{cc}^{D}	Signed/unsigned integer multiply optionally setting icc/xcc		
UDIVX	Unsigned 64-bit integer divide		
SDIVX	Signed 64-bit integer divide		
(S,U)DIV{cc}^{D}	Signed/unsigned 32-bit integer divide, optionally setting icc/xcc		
SETHI	Modify highest 22 bits of low word in IU register: Immediate → IU register (partial)		
SLL{X}	Shift left logical (32/64-bit)		
SRL{X}	Shift right logical (32/64-bit)		
SRA{X}	Shift right arithmetic (32/64-bit)		
TADDcc{TV^{D}}	Tagged add and modify `icc`, optionally trap on overflow		
TSUBcc{TV^{D}}	Tagged subtract and modify icc, optionally trap on overflow		
IU Logic Instructions			
AND{cc}	Logical AND, optionally setting `icc`/`xcc`		
ANDN{cc}	Logical AND-not, optionally setting `icc`/`xcc`		
OR{cc}	Logical OR, optionally setting `icc`/`xcc`		
ORN{cc}	Logical OR-not, optionally setting `icc`/`xcc`		
XOR{cc}	Logical XOR, optionally setting `icc`/`xcc`		
XNOR{cc}	Logical XNOR, optionally setting `icc`/`xcc`		
IU Miscellaneous Instructions			
SIAM			
ALIGNADDRESS{_LITTLE}	Calculates aligned address		
POPC	Defined to count the number of ones in register, unimplemented (causes an illegal instruction execution which traps to software for emulation)		

TABLE 7-3 Instruction Summary for the Integer Execution Environment *(3 of 3)*

Instruction	Description		Notes
RDPR[P]	Read privileged register		
WRPR[P]	Write privileged register		
RDASR[PASR]	Read ancillary state register (ASR) - see below. Privileged mode required for privileged ASRs.		
RDY[D], RDCCR, RDASI, RDPC, RDFPRS, RDPCR[P], RDPIC[PPCR.PRIV], RDDCR[P], RDGSR, RDSOFTINT[P], RDTICK[PNPT], RDSTICK[PNPT], RDTICK_CMPR[P], RDSTICK_CMPR[P]	Read state and ancillary state registers: - If `PCR.PRIV` field is one, then PIC register access requires privileged mode. - If {`TICK`|`STICK`}.`NPT` field is zero, then `TICK`/`STICK` register reads require privileged mode.		
WRASR[PASR]	Write ancillary state register (ASR); Privileged mode required for privileged ASRs.		
WRY[D], WRCCR, WRASI, WRFPRS, WRPCR[P], WRPIC[PPCR.PRIV], WRDCR[P], WRGSR, WRSOFTINT[P], WRSOFTINT_CLR[P], WRSOFTINT_SET[P], WRSTICK[PNPT], WRTICK_CMPR[P], WRSTICK_CMPR[P]	Read state and ancillary state registers: - If `PCR.PRIV` field is one, then `PIC` register access requires privileged mode. - If `STICK.NPT` field is zero, then `STICK` register writes require privileged mode.		

1. A simple register-to-register move is accomplished by using the `OR` instruction with `r[0]`.
2. Load (`LD`) and store (`ST`) instructions are provided with many size formats (byte, word, double word, etc.) and most can be specified with an alternate space identifier (ASI).
3. The “r” refers to value in `r` registers.
4. The cc refers to settings of the integer condition codes.
5. The conditional move instructions (integer and floating-point) are influenced by the condition codes of either execution unit to facilitate moves in one type of execution unit based on the condition codes of the other or of those within the execution unit.

7.10.6 Floating-Point Execution Environment Instructions

TABLE 7-4 Instruction Summary for the Floating-point Execution Environment

Instruction	Description		Reference Pages	Notes
FP/VIS Data Access Instruction s= 32-bit; d= 64-bit; q= 128-bit (q is trapped)		**ASI Load (hex)**		
LD{D}F	Load word (or double word): Memory → FPU register	No		
LD{D}FA^{PASI}	Load word (or double word) from alternate space: Memory → FPU register			
LDDFA	Block load 64 bytes: Memory → FPU registers			
LDDFA	Load short: Memory → FPU register			
LDQF	Load quadword: Memory → FPU register	No		
LDQFA^{PASI}	Load quadword from alternate space: Memory → FPU register	No		
ST(F,DF,QF)	Store word, double, or quad word to memory: FPU register → Memory	No		
ST(F,DF,QF)A^{PASI}	Store word, double, or quad word to memory using alternate memory space.			
STDFA	Block store 64 bytes: uses ASIs	70, 71, 78, 79, F0, F1, F8, F9, E0, E1		
STDFA	Short FP store: uses ASIs $D(0:3)_{16}$, $D(8:B)_{16}$			
STDFA	Partial store FPU: uses ASIs $C(0:5)_{16}$, $C(8:D)_{16}$			
FMOV(s,d,q)	FPU → FPU register	No		
FMOV(s,d,q)cc	Conditional move, IU or FPU condition codes: FPU → FPU register	No		
FMOV(s,d,q)r	Conditional move, IU or FPU register value: FPU → FPU register	No		
FP Arithmetic Instructions s= 32-bit; d= 64-bit; q= 128-bit (q is trapped)				
FABS(s,d,q)	FP absolute value			
FNEG(s,d,q)	Change FP sign			
FADD(s,d,q)	FP add			
FSUB(s,d,q)	FP subtract			

TABLE 7-4 Instruction Summary for the Floating-point Execution Environment *(Continued)*

Instruction	Description		Reference Pages	Notes
FMUL(s,d,q)	FP multiply			
FdMULq	FP multiple doubles to quadword			
FsMULd	FP multiple singles to doubleword			
FDIV(s,d,q)	FP division			
FSQRT(s,d,q)	FP square root			
FP Conversion Instructions s= 32-bit; d= 64-bit; q= 128-bit (q is trapped); i= integer word; x= double (or extended) word				
F(s,d,q)TOi	Floating-point to integer word			
F(s,d,q)TOx	Floating-point to integer double word			
F(s,d,q)TO(s,d,q)	Floating-point to floating-point			
FiTO(s,d,q)	Integer word to floating-point			
FxTO(s,d,q)	Integer double (or extended) word to floating-point			
FP Compare Instructions				
FCMP(s,d,q)	FP compare of like precision, sets fcc condition codes			
FCMPE(s,d,q)	Same as `FCMP`, but an exception is generated if unordered			
FP Miscellaneous Instructions				
LDFSR^{D}	Load FSR into FP reg file: FSR → FPU register (lower 32-bit)			
LDXFSR	Load FSR into FP reg file: FSR → FPU register (64-bit)			
STFSR^{D}	Store FSR register: FPU (lower 32-bit) → FSR register			
STXFSR	Store FSR register: FPU → FSR register			
FALIGNDATA	Concatenates two 64-bit registers into one based on GSR.`ALIGN`			

7.10.7 VIS Execution Environment Instructions

TABLE 7-5 Instruction Summary for the VIS Execution Environment

Instruction	Description		Reference Pages	Notes
VIS Data Access Instructions				
Refer to Section 7.10.6, “Floating-Point Execution Environment Instructions” of the Instruction Summary Table.				
VIS Pixel Data Instructions L= little-endian; N= fcc not modified; S= 32-bit (otherwise 64-bit);				
ARRAY(8,16,32)	3D-array addressing			
BMASK	Writes the GSR.MASK field			
BSHUFFLE	Permute bytes as specified by GSR.MASK field.			
EDGE(8,16,32) (L,N,LN)	Edge handling instructions			
FEXPAND	Pixel data expansion			
FPMERGE	Pixel merge			
FPACK(16,32,FIX)	Pixel packing			
PDIST	Pixel component distance			
VIS Fixed-point 16/32-bit Data Instructions				
FPADD(16,32){S}	Fixed-point add, 16- or 32-bit operands, 32/64-bit register			
FPSUB(16,32){S}	Fixed-point subtract, 16- or 32-bit operands, 32/64-bit register			
FMUL8x16	8x16 partitioned multiply			
FMUL8x16(AU,AL)	8x16 Upper/Lower α partitioned multiply			
FMUL8(SU,SL)x16	8x16 Upper/Lower partitioned multiply			
FMULD8(SU,SL)x16	8x16 Upper/Lower partitioned multiply			
FCMP(GT,LE,NE,EQ)(16,32)	Fixed-point compare (also known as “pixel compare”)			
VIS Logic Instructions S= 32-bit (otherwise 64-bit)				
FSRC(1,2){S}	Copy source			
FONE{S}	Fill with ones (32/64-bit)			
FZERO{S}	Fill with zeroes (32/64-bit)			
FAND{S}	Logical AND (32/64-bit)			
FANDNOT(1,2){S}	Logical AND with a src inverted (32/64-bit)			
FOR{S}	Logical OR (32/64-bit)			
FNAND{S}	Logical NAND (32/64-bit)			
FNOR{S}	Logical NOR (32/64-bit)			

TABLE 7-5 Instruction Summary for the VIS Execution Environment *(Continued)*

Instruction	Description		Reference Pages	Notes
FORNOT(1,2){S}	Logical OR with a source inverted (32/64-bit)			
FNOT(1,2){S}	Logical inversion of source bits (32/64-bit)			
FXNOR{S}	Logical XNOR (32/64-bit)			
FXOR{S}	Logical XOR (32/64-bit)			

7.10.8 Data Coherency Instructions

TABLE 7-6 Instruction Summary for Data Coherency

Instruction	Description		Reference Pages	Notes
Data Coherency Instructions				
FLUSH	Flush I-cache			
MEMBAR	Memory barrier			
STBAR[D]	Store barrier			

7.10.9 Register-window Management Instructions

TABLE 7-7 Instruction Summary for Register-window Management

Instruction	Description		Reference Pages	Notes
Register-Window Management Instructions				
SAVE	Save caller's window			
SAVED[P]	Window has been saved			
RESTORE	Restore caller's window			
RESTORED[P]	Window has been restored			
FLUSHW	Flush register windows			

7.10.10 Program Control Transfer Instructions

TABLE 7-8 Instruction Summary for Program Control Transfer

Instruction	Description		Reference Pages	Notes
Program Control Transfer Instructions icc/xcc= integer condition codes (32/64-bit); fcc= FP condition codes				
Bicc[D]	Conditional branch on icc/xcc			
BPcc	Conditional branch on icc/xcc with branch prediction			
BPr	Conditional branch on IU reg value with branch prediction			
CALL	Call and link			
DONE[P]	Return from Trap			

TABLE 7-8 Instruction Summary for Program Control Transfer *(Continued)*

Instruction	Description		Reference Pages	Notes
FBfcc[D]	Conditional branch on fcc			
FBPfcc	Conditional branch on fcc with branch prediction			
ILLTRAP	Causes *illegal_instruction* trap			
JMPL	Jump and link			
NOP	No operation			
RETRY[P]	Return from trap entry			
RETURN	Return (jump and link)			
SHUTDOWN[P]	Intended for Low Power, but is a NOP in the processor			
SIR[PNOP]	Software initiated reset: a NOP when executed in non-privileged mode			
Tcc	Trap on `icc`/`xcc`			

7.10.11 Data Prefetch Instructions

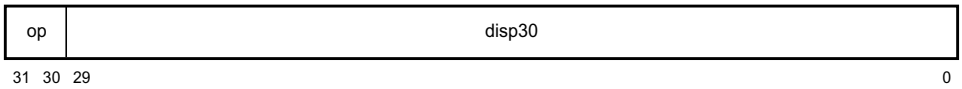
TABLE 7-9 Instruction Summary Table

Instruction	Description		Reference Pages	Notes
Prefetch Instructions				
PREFETCH	Instructs processor to fetch data			
PREFETCHA[PASI]	Instructs processor to fetch data from alternate memory space			

7.11 Instruction Formats and Fields

Instructions are encoded in four major 32-bit formats and several minor formats, as shown in FIGURE 7-1, FIGURE 7-2, and FIGURE 7-3.

Format 1 (op = 1): CALL



Format 2 (op = 0): SETHI *and Branches* (Bicc, BPcc, BPr, FBfcc, FBPfcc)

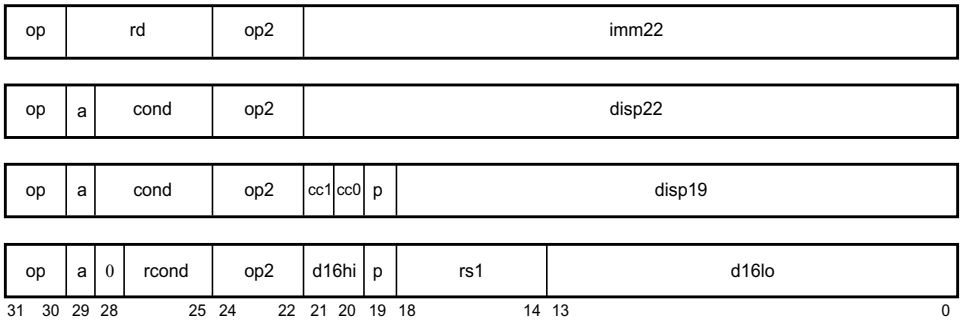


FIGURE 7-1 Summary of Instruction Formats: Formats 1 and 2

Format 3 (op = 2 or 3): Arithmetic, Logical, `MOVr`, `MEMBAR`, Prefetch, Load, *and* Store

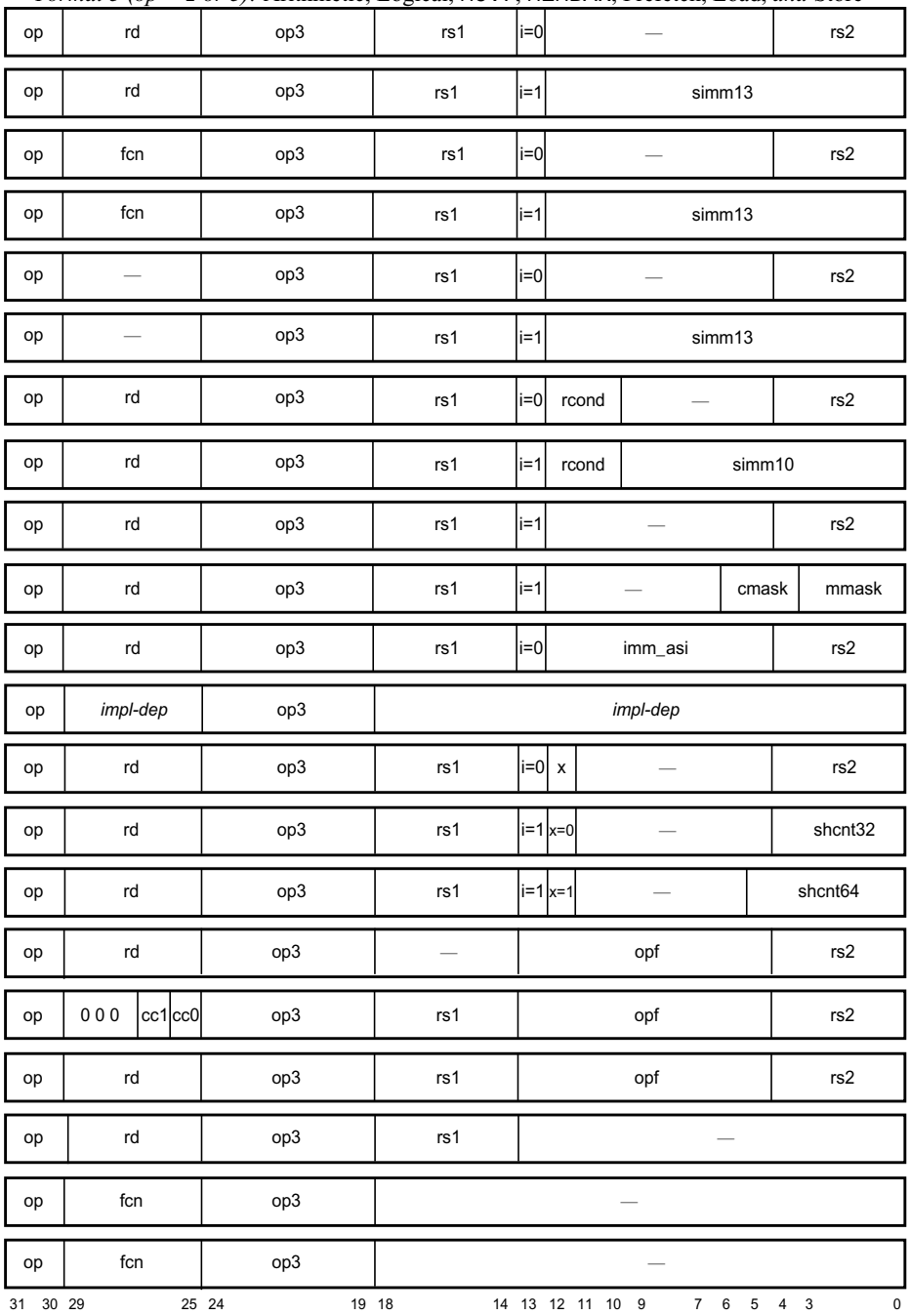


FIGURE 7-2 Summary of Instruction Formats: Format 3

Format 4 (op = 2): MOVcc, FMOVr, FMOVcc, *and* Tcc

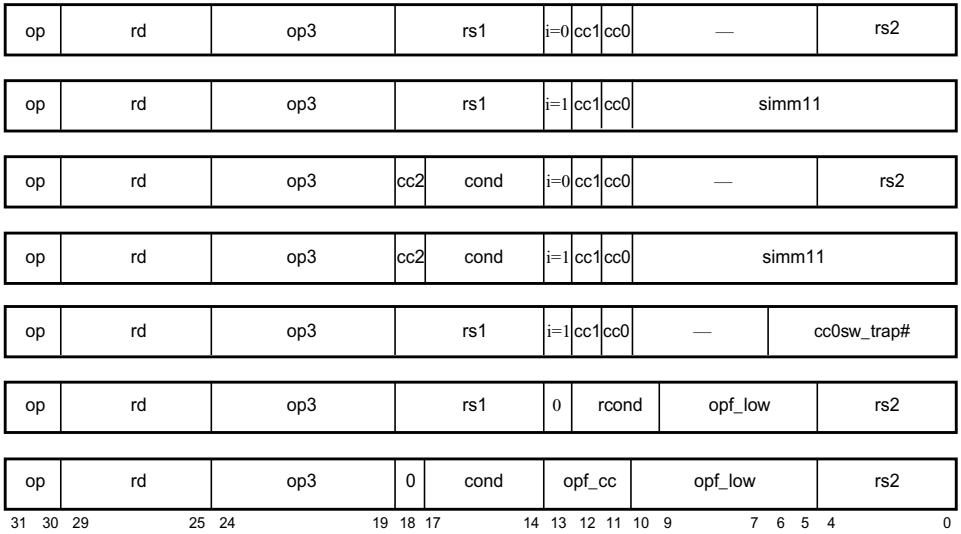


FIGURE 7-3 Summary of Instruction Formats: Format 4

The instruction fields are interpreted as described in TABLE 7-10.

TABLE 7-10 Instruction Field Interpretation *(1 of 3)*

Field	Description
a	The a bit annuls the execution of the following instruction if the branch is conditional and not taken, or if it is unconditional and taken.
cc2, cc1, cc0	cc2, cc1, and cc0 specify the condition codes (icc, xcc, fcc0, fcc1, fcc2, fcc3) to be used in the following instructions: • Branch on Floating-Point Condition Codes with Prediction Instructions (FBPfcc) • Branch on Integer Condition Codes with Prediction (BPcc) • Floating-Point Compare Instructions (FCMP and FCMPE) • Move Integer Register If Condition Is Satisfied (MOVcc) • Move Floating-Point Register If Condition Is Satisfied (FMOVcc) • Trap on Integer Condition Codes (Tcc) In instructions such as Tcc that do not contain the cc2 bit, the missing cc2 bit takes on a default value.
cmask	This 3-bit field specifies sequencing constraints on the order of memory references and the processing of instructions before and after a MEMBAR instruction.
cond	This 4-bit field selects the condition tested by a branch instruction.
d16hi, d16lo	These 2-bit and 14-bit fields together comprise a word-aligned, sign-extended, PC-relative displacement for a branch-on-register-contents with prediction (BPr) instruction.
disp19	This 19-bit field is a word-aligned, sign-extended, PC-relative displacement for an integer branch-with-prediction (BPcc) instruction or a floating-point branch-with-prediction (FBPfcc) instruction.

TABLE 7-10 Instruction Field Interpretation *(2 of 3)*

Field	Description
disp22, disp30	These 22-bit and 30-bit fields are word-aligned, sign-extended, PC-relative displacements for a branch or call, respectively.
fcn	This 5-bit field provides additional opcode bits to encode the DONE, RETRY, and PREFETCH(A) instructions.
i	The i bit selects the second operand for integer arithmetic and load/store instructions. If i = 0, then the operand is r[rs2]. If i = 1, then the operand is simm10, simm11, or simm13, depending on the instruction, sign-extended to 64 bits.
imm22	This 22-bit field is a constant that SETHI places in bits 31:10 of a destination register.
imm_asi	This 8-bit field is the ASI in instructions that access alternate space.
mmask	This 4-bit field imposes order constraints on memory references appearing before and after a MEMBAR instruction.
op, op2	These 2-bit and 3-bit fields encode the three major formats and the Format 2 instructions.
op3	This 6-bit field (together with one bit from op) encodes the Format 3 instructions.
opf	This 9-bit field encodes the operation for a floating-point operate (FPop) instruction.
opf_cc	Specifies the condition codes to be used in FMOVcc instructions. See field cc0, cc1, and cc2 for details.
opf_low	This 6-bit field encodes the specific operation for a Move Floating-Point Register if condition is satisfied (FMOVcc) or Move Floating-Point Register if contents of integer register match condition (FMOVr) instruction.
p	This 1-bit field encodes static prediction for BPcc and FBPfcc instructions; branch prediction bit (p) encodings are shown below.
rcond	This 3-bit field selects the register-contents condition to test for a move, based on register contents (MOVr or FMOVr) instruction or a Branch on Register Contents with Prediction (BPr) instruction.
rd	This 5-bit field is the address of the destination (or source) r or f register(s) for a load, arithmetic, or store instruction.
rs1	This 5-bit field is the address of the first r or f register(s) source operand.
rs2	This 5-bit field is the address of the second r or f register(s) source operand with i = 0.
shcnt32	This 5-bit field provides the shift count for 32-bit shift instructions.
shcnt64	This 6-bit field provides the shift count for 64-bit shift instructions.
simm10	This 10-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for a MOVr instruction when i = 1.
simm11	This 11-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for a MOVcc instruction when i = 1.
simm13	This 13-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for an integer arithmetic instruction or for a load/store instruction when i = 1.

p	Branch Prediction
0	Predict that branch will not be taken
1	Predict that branch will be taken

TABLE 7-10 Instruction Field Interpretation *(3 of 3)*

Field	Description
sw_trap#	This 7-bit field is an immediate value that is used as the second ALU operand for a Trap on Condition Code instruction.
x	The x bit selects whether a 32-bit or 64-bit shift will be performed.

7.12 Reserved Opcodes and Instruction Fields

An attempt to execute an opcode to which no instruction is assigned causes a trap, specifically:

- Attempting to execute a reserved FPop (floating-point opcode) causes a *fp_exception_other* exception (with FSR.ftt = *unimplemented_FPop*).
- Attempting to execute any other reserved opcode causes an *illegal_instruction* exception.
- Attempting to execute an FPop with a nonzero value in a reserved instruction field causes a *fp_exception_other* exception (with FSR.ftt = *unimplemented_FPop*).[1]
- Attempting to execute a Tcc instruction with a nonzero value in a reserved instruction field causes an *illegal_instruction* exception.
- Attempting to execute any other instruction with a nonzero value in a reserved instruction field causes an *illegal_instruction* exception.[1]

7.12.1 Summary of Unimplemented Instructions

Certain SPARC-V9 instructions are not implemented in hardware in the processor. Executing any of these instructions results in the behavior described in TABLE 7-11.

TABLE 7-11 Processor Actions on Unimplemented Instructions

Instructions	Trap Taken	Processor-specific Behavior	Operating System Response
Quad FPops (including FdMULq)	*fp_exception_other*	FSR.ftt = *unimplemented_FPop*	Emulates Instruction
POPC	*illegal_instruction*	None	Emulates Instruction
RDPR FQ	*illegal_instruction*	None	Skips Instruction and Returns
LDQF	*illegal_instruction*	None	Emulates Instruction
STQF	*illegal_instruction*	None	Emulates Instruction

1. Although it is recommended that this exception is generated, a JPS1 implementation may ignore the contents of reserved instruction fields (in instructions other than Tcc).

If a trap does not occur and the instruction is not a control transfer, the next program counter (nPC) is copied into the PC, and the nPC is incremented by four (ignoring overflow, if any). If the instruction is a control transfer instruction, the nPC is copied into the PC and the target address is written to nPC. Thus, the two program counters provide for a delayed-branch execution model.

For each instruction access and each normal data access, the IU appends an 8-bit address space identifier (ASI) to the 64-bit memory address. Load/store alternate instructions (see Section 7.2.4, “Address Space Identifiers (ASIs)”) can provide an arbitrary ASI with their data addresses or can use the ASI value currently contained in the ASI register.

7.13 Big/Little-Endian Addressing

The processor uses big-endian byte order for all instruction accesses and, by default, for data accesses.

It is possible to access data in little-endian format by using selected ASIs.

It is also possible to change the default byte order for implicit data accesses.

7.13.1 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 illustrated in FIGURE 7-4 and described below the figure.

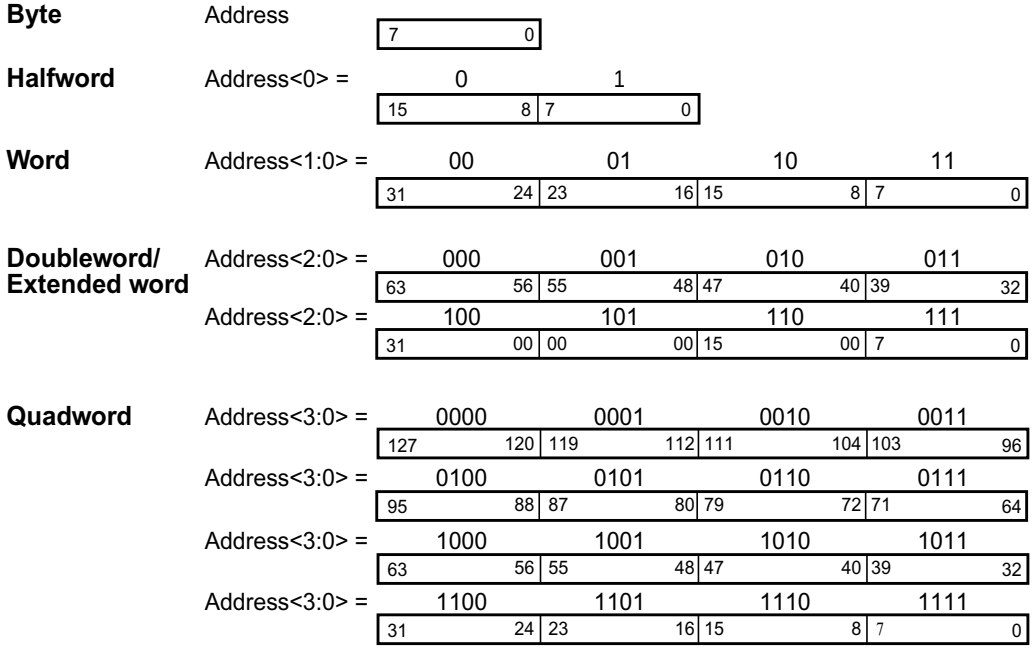


FIGURE 7-4 Big-Endian Addressing Convention

big-endian byte A load/store byte instruction accesses the addressed byte in both big-endian and little-endian modes.

big-endian halfword For a load/store halfword instruction, 2 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.

big-endian word For a load/store word instruction, 4 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.

big-endian doubleword or extended word For a load/store extended or floating-point load/store double instruction, 8 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 double instructions (LDD/STD), 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.

big-endian 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.

7.13.2 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 illustrated in FIGURE 7-5 and defined below the figure.

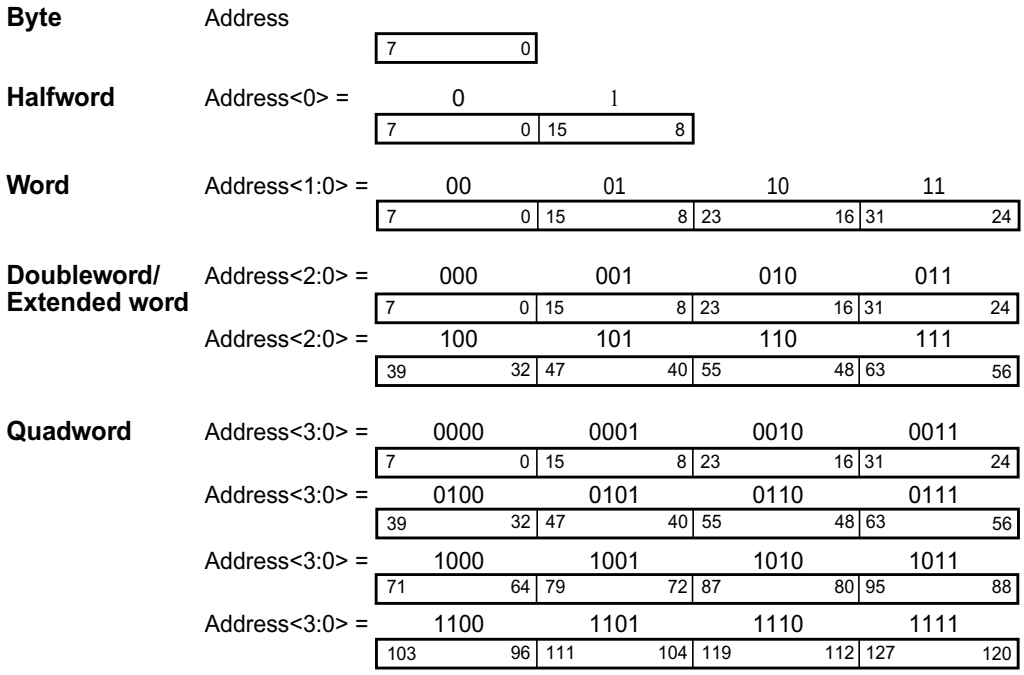


FIGURE 7-5 Little-Endian Addressing Conventions

little-endian byte A load/store byte instruction accesses the addressed byte in both big-endian and little-endian modes.

little-endian halfword For a load/store halfword instruction, 2 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.

little-endian word For a load/store word instruction, 4 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.

little-endian doubleword or extended word For a load/store extended or floating-point load/store double instruction, 8 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 double instructions (LDD/STD), 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 plus four corresponds to the following odd-numbered register. With respect to little-endian memory, an LDD (STD) 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).

little-endian 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.

SECTION IV

Memory and Cache

CHAPTER 8

Memory Models

The SPARC-V9 architecture is a *model* that specifies the behavior observable by software on SPARC-V9 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 in the following:

- Chapter 8 of *The SPARC Architecture Manual, Version 9*
- Appendix D of *The SPARC Architecture Manual, Version 9*

The SPARC-V9 architecture defines three different memory models: *Total Store Order (TSO)*, *Partial Store Order (PSO)*, and *Relaxed Memory Order (RMO)*. The UltraSPARC IIIi processor implements TSO, the strongest of the memory models defined by SPARC-V9. By implementing TSO, software written for any memory model (TSO, PSO, and RMO) executes correctly on the UltraSPARC IIIi processor.

This chapter departs from the organization of the memory models described in *The SPARC Architecture Manual, Version 9*. It describes the characteristics of the memory models for the UltraSPARC IIIi processor in sections organized as follows:

- TSO Behavior
- Memory Location Identification
- Memory Accesses and Cacheability
- Memory Synchronization
- Atomic Operations
- Non-Faulting Load
- Prefetch Instructions
- Block Loads and Stores
- I/O and Accesses with Side-Effects
- Internal ASIs
- Store Compression
- Read After Write (RAW) Bypassing

8.1 TSO Behavior

The UltraSPARC IIIi processor implements the TSO memory model. The current memory model is indicated in the PSTATE.MM field and is set to TSO (PSTATE.MM = 0).

In some cases, the UltraSPARC IIIi processor implements stronger ordering than the TSO requirements. The significant cases are listed below:

- A MEMBAR #Lookaside is not needed between a store and a subsequent load to the same non-cacheable address.
- Accesses with the TTE.E bit set, such as those that have side-effects, are all strongly ordered with respect to one another.
- An L2-cache or W-cache update is delayed on a store hit until all previous stores reach global visibility. For example, a cacheable store following a non-cacheable store will not appear globally visible until the non-cacheable store has become globally visible; there is an implicit MEMBAR #MemIssue between them.

8.2 Memory Location Identification

A memory location is identified by an 8-bit address space identifier (ASI) and a 64-bit (virtual) address. The 8-bit ASI can be obtained from an ASI register or included in a memory access instruction. The ASI distinguishes among and provides an attribute to different 64-bit address spaces. For example, the ASI is used by the MMU and memory access hardware for control of virtual-to-physical address translations, access to implementation-dependent control and data registers, and access protection. Attempts by non-privileged software (PSTATE.PRIV = 0) to access restricted ASIs (ASI<7> = 0) cause a *privileged_action* exception.

8.3 Memory Accesses and Cacheability

Memory is logically divided into real memory (cached) and I/O memory (non-cached with and without side-effects) spaces. Real memory spaces can be accessed without side-effects. For example, a read from real memory space returns the information most recently written. In addition, an access to real memory space does not result in program-visible side-effects. In contrast, a read from I/O space may not return the most recently written information and may result in program-visible side-effects.

8.3.1 Coherence Domains

The two types of memory operations supported in the UltraSPARC IIIi processor are cacheable and non-cacheable accesses, as indicated by the page translation (TTE.CP, TTE.CV) of the MMU or by an ASI override.

SPARC-V9 does not specify memory ordering between cacheable and non-cacheable accesses. The UltraSPARC IIIi processor maintains TSO ordering between memory references regardless of their cacheability.

8.3.1.1 Cacheable Accesses

Accesses within the coherence domain are called cacheable accesses. They have the following properties:

- Data reside in real memory locations.
- Accesses observe supported cache coherency protocol(s).
- The unit of coherence is 64 bytes.

8.3.1.2 Non-Cacheable and Side-Effect Accesses

Accesses outside of the coherence domain are called non-cacheable accesses. Some of these memory-mapped locations may have side-effects when accessed. 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, such as those in a UART.
- Accesses do not observe supported cache coherency protocol(s).
- The smallest unit in each transaction is a single byte.

Non-cacheable accesses with the TTE.E bit set (those having side-effects) are all strongly ordered with respect to other non-cacheable accesses with the E bit set. In addition, store compression is disabled for these accesses. Speculative loads with the E bit set cause a *data_access_exception* trap (with SFSR.FT = 2, speculative load to page marked with E bit).

Note – TTE.E bit comes from the page translation of the MMU or an ASI override.

Non-cacheable accesses with the TTE.E bit cleared (non-side-effect accesses) are processor consistent and obey TSO memory ordering. In particular, processor consistency ensures that a non-cacheable load that references the same location as a previous non-cacheable store will load the data of the previous store. Store compression is supported. See Section 8.11, “Store Compression” for details.

Note – Side-effect, as indicated in TTE.E, does not imply non-cacheability.

8.3.2 Global Visibility

A memory access is considered globally visible when the transaction request is issued on JBUS.

8.3.3 Memory Ordering

To ensure the correct ordering between cacheable and non-cacheable domains, explicit memory synchronization is needed in the form of MEMBAR instructions. CODE EXAMPLE 8-1 illustrates the issues involved in mixing cacheable and non-cacheable accesses.

CODE EXAMPLE 8-1 Memory Ordering and MEMBAR Examples

```
Assume that all accesses go to non-side-effect memory locations.
Process A:
While (1)
{
    Store D1:data produced
1   MEMBAR #StoreStore (needed in PSO, RMO for SPARC-V9 compliance)
    Store F1:set flag
    While F1 is set (spin on flag)
    Load F1
2   MEMBAR #LoadLoad, #LoadStore (needed in RMO for SPARC-V9
compliance)
    Load D2
}

Process B:
While (1)
{
    While F1 is cleared (spin on flag)
    Load F1
```

CODE EXAMPLE 8-1 Memory Ordering and MEMBAR Examples *(Continued)*

```
2   MEMBAR #LoadLoad, #LoadStore (needed in RMO for SPARC-V9
compliance)
    Load D1
    Store D2
1   MEMBAR #StoreStore (needed in PSO, RMO for SPARC-V9 compliance)
    Store F1:clear flag
}
```

8.4 Memory Synchronization

Normal loads and stores by an UltraSPARC IIIi processor are performed in order. TSO defines how other processors may see the ordering of the loads and stores of a particular processor. Memory synchronizations are used to force the ordering that other processors see beyond the rules of TSO.

In some cases, memory synchronizations are required for deterministic behavior, even with respect to the program's own operations. This applies to memory operations outside of normal cacheable loads and stores.

The UltraSPARC IIIi processor achieves memory synchronization through MEMBAR and FLUSH. It provides MEMBAR (STBAR in SPARC-V8) and FLUSH instructions for explicit control of memory ordering in program execution. MEMBAR has several variations. All MEMBARs are implemented in one of two ways in the UltraSPARC IIIi processor:

- As a NOP
- With MEMBAR #Sync semantics

Since the processor always executes with TSO memory ordering semantics, three of the ordering MEMBARs are implemented as NOPs. TABLE 8-1 lists the MEMBAR implementations.

TABLE 8-1 MEMBAR Semantics

MEMBAR	Semantics
#LoadLoad	NOP. All loads wait for completion of all previous loads.
#LoadStore	NOP. All stores wait for completion of all previous loads.
#Lookaside	#Sync. Wait until store buffer is empty.
#StoreStore, STBAR	NOP. All stores wait for completion of all previous stores.
#StoreLoad	#Sync. All loads wait for completion of all previous stores.
#MemIssue	#Sync. Wait until all outstanding memory accesses complete.
#Sync	#Sync. Wait for all outstanding instructions and all deferred errors.

8.4.1 MEMBAR #Sync

MEMBAR #Sync forces all outstanding instructions and all deferred errors to be completed before any instructions after the MEMBAR are issued.

8.4.2 MEMBAR Rules

TABLE 8-2 and TABLE 8-3 summarize the cases where the programmer must insert a MEMBAR to ensure *ordering* between two memory operations on the UltraSPARC IIIi processor. Use TABLE 8-2 and TABLE 8-3 for ordering purposes only. Be sure not to confuse memory operation ordering with processor consistency or deterministic operation; MEMBARs are required for deterministic operation of certain ASI register updates.

Caution – The MEMBAR requirements for the UltraSPARC IIIi processor are less stringent than the requirements of SPARC-V9. To ensure code portability across systems, use the stronger of the MEMBAR requirements of SPARC-V9.

Read the tables as follows: Read from row to column; the first memory operation in program order in a row is followed by the memory operation found in the column. Two symbols are used as table entries:

- # — No intervening operation is required because Fireplane-compliant systems automatically order R before C.
- M — MEMBAR #Sync or MEMBAR #MemIssue or MEMBAR #StoreLoad

For VA<12:5> of a column operation not matching with VA<2:5> of a row operation while a strong ordering is desired, the MEMBAR rules summarized in TABLE 8-2 reflect the UltraSPARC IIIi processor hardware implementation.

TABLE 8-2 MEMBAR Rules for Column VA <12:5> ≠ Row VA <12:5> While Desiring Strong Ordering

	To Column Operation C:													
From Row Operation R:	**load**	**load from internal ASI**	**store**	**store to internal ASI**	**atomic**	**load_nc_e**	**store_nc_e**	**load_nc_ne**	**store_nc_ne**	**bload**	**bstore**	**bstore_commit**	**bload_nc**	**bstore_nc**
load	#	#	#	#	#	#	#	#	#	M	M	#	M	M
load from internal ASI	#	#	#	#	#	#	#	#	#	#	#	#	#	#
store	M	#	#	#	#	M	#	M	#	M	M	#	M	M
store to internal ASI	#	M	#	#	#	#	#	#	#	M	#	#	M	M
atomic	#	#	#	#	#	#	#	#	#	M	M	#	M	M
load_nc_e	#	#	#	#	#	#	#	#	#	M	M	#	M	M
store_nc_e	M	#	#	#	#	#	#	M	#	M	M	#	M	M
load_nc_ne	#	#	#	#	#	#	#	#	#	M	M	#	M	M
store_nc_ne	M	#	#	#	#	M	#	M	#	M	M	#	M	M
bload	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bstore	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bstore_commit	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bload_nc	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bstore_nc	M	#	M	#	M	M	M	M	M	M	M	#	M	M

When VA<12:5> of a column operation matches VA<12:5> of a row operation, the MEMBAR rules summarized in TABLE 8-3 reflect the UltraSPARC IIIi processor hardware implementation.

TABLE 8-3 MEMBAR Rules for Column VA<12:5> = Row VA<12:5> While Desiring Strong Ordering

		To Column Operation C:												
From Row Operation R:	**load**	**load from internal ASI**	**store**	**store to internal ASI**	**atomic**	**load_nc_e**	**store_nc_e**	**load_nc_ne**	**store_nc_ne**	**bload**	**bstore**	**bstore_commit**	**bload_nc**	**bstore_nc**
load	#	#	#	#	#	#	#	#	#	#	#	#	#	#
load from internal ASI	#	#	#	#	#	#	#	#	#	#	#	#	#	#
store	#	#	#	#	#	#	#	#	#	M	#	#	#	#
store to internal ASI	#	M	#	#	#	#	#	#	#	M	#	#	M	M
atomic	#	#	#	#	#	#	#	#	#	#	#	#	#	#
load_nc_e	#	#	#	#	#	#	#	#	#	#	#	#	#	#
store_nc_e	#	#	#	#	#	#	#	#	#	M	#	#	M	#
load_nc_ne	#	#	#	#	#	#	#	#	#	#	#	#	#	#
store_nc_ne	#	#	#	#	#	#	#	#	#	M	#	#	M	#
bload	#	#	#	#	#	#	#	#	#	#	#	#	#	#
bstore	#	#	#	#	#	#	#	#	#	M	#	#	#	#
bstore_commit	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bload_nc	#	#	#	#	#	#	#	#	#	#	#	#	#	#
bstore_nc	#	#	#	#	#	#	#	#	#	#	#	#	M	#

8.4.3 FLUSH

FLUSH behaves like a MEMBAR with further restrictions. MEMBAR blocks execution of subsequent instructions until all memory operations and errors are resolved. FLUSH is similar with further behavior in that all instruction fetch and instruction buffering operations are also blocked.

8.5 Atomic Operations

SPARC-V9 provides three atomic instructions to support mutual exclusion, including:

- **SWAP** — Atomically exchanges the lower 32 bits in an integer register with a word in memory. This instruction is issued only after store buffers are empty. Subsequent loads interlock on earlier SWAPs.
- If a page is marked as virtually non-cacheable but physically cacheable (TTE.CV = 0 and TTE.CP = 1), allocation is done to the L2-cache and W-cache only. This includes all of the atomic-access instructions.
- **LDSTUB** — Behaves like a SWAP except that it loads a byte from memory into an integer register and atomically writes all 1's (FF_{16}) into the addressed byte.
- **Compare and Swap (CAS(X)A)** — Combines a load, compare, and store into a single atomic instruction. It compares the value in an integer register to a value in memory. If they are equal, the value in memory is swapped with the contents of a second integer register. If they are not equal, the value in memory is still swapped with the contents of the second integer register, but is not stored. The L2-cache will still go into M-state, even if there is no store.

  All of these operations are carried out atomically; in other words, no other memory operation can be applied to the addressed memory location until the entire compare-and-swap sequence is completed.

These instructions behave like both a load and store access, but the operation is carried out indivisibly. These instructions can be used only in the cacheable domain (not in non-cacheable I/O addresses).

These atomic instructions can be used with the ASIs listed in TABLE 8-4. Access with a restricted ASI in unprivileged mode (PSTATE.PRIV = 0) results in a *privileged_action* trap. Atomic accesses with non-cacheable addresses cause a *data_access_exception* trap (with SFSR.FT = 4, atomic to page marked non-cacheable). Atomic accesses with unsupported ASIs cause a *data_access_exception* trap (with SFSR.FT = 8, illegal ASI value or virtual address).

TABLE 8-4 ASIs That Support SWAP, LDSTUB, and CAS

ASI Name	Access
ASI_NUCLEUS (LITTLE)	Restricted
ASI_AS_IF_USER_PRIMARY (LITTLE)	Restricted
ASI_AS_IF_USER_SECONDARY (LITTLE)	Restricted
ASI_PRIMARY (LITTLE)	Unrestricted
ASI_SECONDARY (LITTLE)	Unrestricted
ASI_PHYS_USE_EC (LITTLE)	Restricted

Note – Atomic accesses with non-faulting ASIs are not allowed, because the latter have the load-only attribute.

8.6 Non-Faulting Load

A non-faulting load behaves like a normal load, with the following exceptions:

- It does not allow side-effect access. An access with the TTE.E bit set causes a *data_access_exception* trap (with SFSR.FT = 2, speculative load to page marked E bit).
- It can be applied to a page with the TTE.NFO (non-fault access only) bit set; other types of accesses cause a *data_access_exception* trap (with SFSR.FT = 10_{16}, normal access to page marked NFO).

These loads are issued with ASI_PRIMARY_NO_FAULT{_LITTLE} or ASI_SECONDARY_NO_FAULT{_LITTLE}. A store with a NO_FAULT ASI causes a *data_access_exception* trap (with SFSR.FT = 8, illegal RW).

When a non-faulting load encounters a TLB miss, the operating system should attempt to translate the page. If the translation results in an error, then zero is returned and the load completes silently.

Typically, optimizers use non-faulting 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 code scheduling and improves performance in certain algorithms by removing address checking from the critical code path.

For example, when following a linked list, non-faulting 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 NFO bit in the MMU marks pages that are mapped for safe access by non-faulting 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 code—while benefiting from the acceleration of non-faulting access in debugged library routines.

8.7 Prefetch Instructions

The UltraSPARC IIIi processor implements all SPARC-V9 prefetch instructions except for prefetch page. All prefetches check the L2-cache before issuing a system request for the requested data. Prefetch instructions are a performance feature. Prefetch instructions do not change the underlying memory model and do not have any effect from an architectural standpoint.

TABLE 8-5 describes prefetch instructions.

TABLE 8-5 Types of Software Prefetch Instructions

fcn Value (hex)	Instruction Type	Prefetch (64 bytes of data) into:	Instruction Strength	Request Exclusive Ownership
00	Prefetch read many	P-cache and L2-cache	Weak	No
01	Prefetch read once	P-cache only	Weak	No
02	Prefetch write many	L2-cache only	Weak	Yes
03	Prefetch write once[1]	L2-cache only	Weak	No
04	*Reserved*	Undefined		
05 - 0F	*Reserved*	Undefined		
10	Prefetch invalidate	Invalidates a P-cache line, no data is prefetched.		N/A
11 - 13	*Reserved*	Undefined		
14	Same as `fcn` = 00		Weak[2]	No
15	Same as `fcn` = 01		Weak[2]	No
16	Same as `fcn` = 02		Weak[2]	Yes
17	Same as `fcn` = 03		Weak[2]	No
18 - 1F	*Reserved*	Undefined		

1. Although the name is “prefetch write once,” the actual use is prefetch to L2-cache for a future read.
2. These weak instructions may be implemented as strong in future implementations.

8.8 Block Loads and Stores

Block load and store instructions work like normal floating-point load and store instructions, except that the data size (granularity) is 64 bytes per transfer.

Block loads and stores do not obey TSO. They do not even obey the processor's consistency rules without the correct use of MEMBAR. Section A.4 "Block Load and Block Store (VIS I)" on page A-274 discusses block loads and stores in detail.

8.9 I/O and Accesses with Side-Effects

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.

The UltraSPARC IIIi MMU includes an attribute bit in each page translation, TTE.E, which when set signifies that this page has side-effects. Accesses other than block loads or stores to pages that have this bit set exhibit the following behavior:

- Non-cacheable accesses are strongly ordered with respect to each other.
- Non-cacheable loads with the E bit set will not be issued to the system until all previous control transfers are resolved.
- Non-cacheable store compression is disabled for E bit accesses.
- *Exactly* those E bit accesses implied by the program are made in program order.
- Non-faulting loads are not allowed and cause a *data_access_exception* (with SFSR.FT = 2, speculative load to page marked E bit).
- For portability across SPARC-V9 processors, a MEMBAR may be needed between side-effect and non-side-effect accesses while in PSO and RMO modes, as well as in some cases of TSO.

8.9.1 Instruction Prefetch to Side-Effect Locations

The processor does instruction prefetching and follows branches that it predicts are taken. Addresses mapped by the I-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 by the I-MMU; therefore, prefetching will not cause problems.

When running with the I-MMU disabled, software must avoid placing data in the path of a control transfer instruction target or sequentially following a trap or conditional branch instruction. Data can be placed sequentially following the delay slot of a `BA`, `BPA` (`p` = 1), `CALL`, or `JMPL` instruction. Instructions should not be placed closer than 256 bytes to locations with side-effects.

8.9.2 Instruction Prefetch Exiting Red State

Exiting `RED_state` by writing zero to `PSTATE.RED` in the delay slot of a `JMPL` instruction is not recommended. A non-cacheable instruction prefetch may be made to the `JMPL` target, which may be in a cacheable memory area. This situation can result in a bus error on some systems and can cause an instruction access error trap. Programmers can mask the trap by setting the `NCEEN` bit in the L2-cache Error Enable Register to zero, but doing so will mask all non-correctable error checking. Exiting `RED_state` with `DONE`, `RETRY`, or with the destination of the `JMPL` non-cacheable will avoid the problem.

8.10 Internal ASIs

ASIs in the ranges 30_{16}–$6F_{16}$ and 72_{16}–$7F_{16}$ are used for accessing internal states. Stores to these ASIs do not follow the normal memory-model ordering rules. Correct operation can be assured by adhering to the following requirements:

- A `MEMBAR #Sync` is needed after a store to an internal ASI other than MMU ASIs before the point that side-effects must be visible. This `MEMBAR` must precede the next load or non-internal store. To avoid data corruption, the `MEMBAR` must also occur before the delay slot of a delayed control transfer instruction of any type.
- Alternatively, a `MEMBAR #Sync` could be inserted at the beginning of any vulnerable trap handler. “Vulnerable” trap handlers are those which contain one or more LDXAs from any internal ASI (ASIs 0x30-0x6F, 0x72-0x77, and 0x7A-0x7F). However, this may cause an unacceptable performance reduction in some trap handlers, so this is not the preferred alternative.

- A FLUSH, DONE, or RETRY is needed after a store to an internal I-MMU ASI (ASI 50_{16}–52_{16}, 54_{16}–$5F_{16}$), an I-cache ASI (66_{16}–$6F_{16}$), or the IC bit in the DCU Control Register, prior to the point that side-effects must be visible. A store to D-MMU registers other than the context ASIs can use a MEMBAR #Sync. To avoid data corruption, the MEMBAR must also occur before the delay slot of a delayed control transfer instruction of any type.
- If the store is to an I-MMU state register (ASI = 50_{16}, virtual address = 18_{16}), then the FLUSH, DONE, or RETRY must *immediately* follow the store. Furthermore, one of the following must be true, to prevent an intervening I-TLB miss from causing stale data to be stored:
  - The code must be locked down in the I-TLB, or
  - The store and the subsequent FLUSH, DONE, or RETRY should be kept on the same 8 KB page of instruction memory.

8.11 Store Compression

Consecutive non-side-effect, non-cacheable stores can be combined into aligned 16-byte entries in the store buffer to improve store bandwidth. Cacheable stores will naturally coalesce in the W-cache rather than be compressed in the store buffer. Non-cacheable stores can be compressed only with adjacent non-cacheable stores. To maintain strong ordering for I/O accesses, stores with the side-effect attribute (E bit set) cannot be combined with any other stores.

A 16-byte non-cacheable merge buffer is used to coalesce adjacent non-cacheable stores. Non-cacheable stores will continue to coalesce into the 16-byte buffer until one of the following conditions occurs:

- The data is pulled from the non-cacheable merge buffer by the target device.
- The store overwrites a previously written entry (a valid bit is kept for each of the 16 bytes).

Caution – This behavior is unique to the UltraSPARC IIIi processor and differs from previous UltraSPARC processor implementations.

- The store is not within the current address range of the merge buffer (within the 16-byte aligned merge region).
- The store is a cacheable store.
- The store is to a side-effect page.
- MEMBAR #Sync

8.12 Read After Write (RAW) Bypassing

Load data can be bypassed from previous stores before they become globally visible (data for load from the store queue). This is specifically allowed by the TSO memory model. Data for all types of loads cannot be bypassed from all types of stores.

All types of load instructions can get data from the store queue, *except* the following load instructions:

- Signed loads (`ldsb`, `ldsh`, `ldsw`)
- Atomics
- Load double to integer register file (`ldd`)
- Quad loads to integer register file
- Load from `FSR` register
- Block loads
- Short floating-point loads
- Loads from internal ASIs

All types of store instructions can give data to a load, *except* the following store instructions:

- Floating-point partial stores
- Store double from integer register file (`std`)
- Store part of atomic
- Short FP stores
- Stores to pages with side-effect bit set
- Stores to non-cacheable pages

8.12.1 RAW Bypassing Algorithm

The algorithm used in the UltraSPARC IIIi processor for RAW bypassing is as follows:

```
if ( (Load/store access the same physical address)                  and
                        (Load/store endianness is the same)         and
                        (Load/store size is the same)               and
                        (Load data can get its data from store queue) and
                        (Store data in store can give its data to a load) and
                        (Load hits in either D-cache or P-cache)
                    )
then
                    Load will get its data from store queue
```

```
else
                    Load will get its data from the memory system
endif
```

8.12.2 RAW Detection Algorithm

When data for a load cannot be bypassed from previous stores before they become globally visible (store data is not yet retired from the store queue), the load is recirculated after the RAW hazard is removed. The following conditions can cause this recirculation:

- Load data can be bypassed from more than one store in the store queue.
- The load's VA<12:0> overlaps a store's VA<12:0> and store data cannot be bypassed from the store queue.
- The load's VA<12:5> matches a store's VA<12:5> and the load misses the D-cache.
- Load is from side-effect page (page attribute E = 1) when the store queue contains one or more stores to side-effect pages.

CHAPTER 9

Caches and Coherency

This chapter describes the use of caches and TLBs, and contains the following sections:

- Cache Organization
- Cache Flushing
- Controlling P-Cache
- Translation Lookaside Buffers (TLBs)

9.1 Cache Organization

9.1.1 Virtually Indexed, Physically Tagged Caches (VIPT)

The D-cache is Virtually Indexed, Physically Tagged (VIPT). Virtual addresses are used to index into the cache tag and data arrays while accessing the D-MMU (that is, D-TLBs). The resulting tag is compared against the translated physical address to determine a cache hit.

A side-effect inherent in a virtual-indexed cache is *address aliasing*. This issue is addressed in Section 9.2.1 “Address Aliasing Flushing” on page 206.

9.1.1.1 Data Cache (D-Cache)

The Data Cache is a write-through, non-allocating on a write miss, 64 KB, pseudo-4-way associative cache with a 32-byte line.

Data accesses bypass the data cache when:

- The Data Cache enable (`DC`) bit in the Data Cache Unit Control Register (`DCUCR`) is clear, or

- The D-MMU Enable (DCUCR.DM) bit and the virtual cacheability (DCUCR.CV) bit are clear, or
- The access is mapped by the D-MMU as non-virtual-cacheable

Note – A non-virtual-cacheable access may access data in the Data Cache from an earlier cacheable access to the same physical block, unless the Data Cache is disabled. Software must flush the Data Cache when changing a physical page from cacheable to non-cacheable (see Section 9.2 “Cache Flushing” on page 205).

9.1.2 Bypassing the D-Cache

D-cache can return stale data if CP == 1, CV == 0 is used to bypass the cache, after use of CP==1 and CV==1, for loads and stores to a particular address.

D-cache should be flushed, after mixing use of any CP/CV settings for a physical address, including cacheable (DRAM) and non-cacheable (I/O) physical addresses.

The term “virtually non-cacheable” refers to the “non-D-cacheable” CP == 1, CV == 0 case, as opposed to the more common use of “non-cacheable” to describe I/O or graphics related physical addresses.

- CP == 1, CV == 1: Cacheable, Virtually-cacheable
- CP == 1, CV == 0: Cacheable, Virtually-non-cacheable (ASI_PHYS_USE_EC has this effect)
- CP == 0, CV == 1: P-cacheable
- CP == 0, CV == 0: Non-cacheable

Only two indexes in the D-cache need to be flushed for each 32-byte aligned physical address:

- {VA[13] == 0,PA[12:5]} and
- {VA[13] == 1,PA[12:5]}

9.1.2.1 Special Case 1

When performing a load with a physical address, using ASI = 0x14 (ASI_PHYS_USE_EC), causing CP == 1 and CV == 0, and the address hits in the D-cache, the following describes how the data comes from D-cache instead of L2-cache:

If CP == 0 and CV == 0, which indicates a “non-cacheable” access, and the address is in the D-cache, data can be returned from the D-cache.

The address should be flushed from the D-cache before changing its mapping.

Similarly, if CP == 1, and CV == 0, and the data is in the D-cache, data may be returned from the D-cache. However, there are corner cases where it may not be the case.

For instance, with ASI_PHYS_USE_EC, the physical PA[13] is used to index the D-cache, where VA[13] would ordinarily be used. Therefore, the data might not be correctly returned if the real data was in VA[13] == 0, but PA[13] == 1. Ordinarily the rest of the PA bits will have a difference, therefore, it will miss in the D-cache, and go to the L2-cache correctly. This takes advantage of knowing that a valid PA can only exist in one VA[13] mapping at a time in the D-cache.

This depends on how the addresses were mapped earlier, when the line was installed in the D-cache.

This ASI_PHYS_USE_EC load hitting on the D-cache behavior is not defined or tested, so software should not rely on it.

9.1.2.2 Special Case 2

When performing a store with a physical address, using ASI=0x14 (ASI_PHYS_USE_EC), causing CP == 1 and CV == 0, and the address hits in the D-cache, the following describes how the D-cache gets updated:

The software should make sure the physical address is not in the D-cache, before accessing that address using CP == 1, CV == 0, whether by a TLB mapping, or using one of the special ASIs.

9.1.3 Physically-Indexed, Physically-Tagged Caches (PIPT)

9.1.3.1 Instruction Cache (I-Cache)

The Instruction Cache is a 32KB pseudo 4-way, set-associative, write-invalidate cache with 32-byte lines. Instruction fetches bypass the Instruction Cache when:

- The Instruction Cache enable (DCUCR.IC) is clear, or
- The I-MMU enable (DCUCR.IM) bit and the physical cacheability (DCUCR.CP) bit are clear, or
- The processor is in RED_state, or
- The fetch is mapped by the I-MMU as nonphysical-cacheable.

The Instruction Cache snoops stores from other processors or DMA transfers, as well as stores in the same processor and block commit store.

The FLUSH instruction is not required to maintain coherency. Stores and block store commits invalidate the Instruction Cache, but do not flush instructions that have already been prefetched into the pipeline. A FLUSH, DONE, or RETRY instruction can be used to flush the pipeline.

If a program changes I-cache mode to I-cache-ON from I-cache-OFF, then the next instruction fetching always causes an I-cache miss even if it is supposed to hit. This rule applies even when the DONE instruction turns on the I-cache by changing its status from RED_state to normal mode. For example,

(in RED_state)

```
setx            0x37e0000000007, %g1, %g2
stxa            %g2,[%g0]0x45         // Turn on I-cache when
processor
                                      // returns normal mode.
done                                  // Escape from RED_state.
```

(back to normal mode)

```
nop             // 1st instruction; this always causes an I-cache
                   miss.
```

9.1.3.2 Prefetch Cache (P-Cache)

The P-cache is a write-invalidate, 2 KB, 4-way associative cache with a 64-byte line and two 32-byte sub-blocks. It is physically-indexed and physically-tagged and never contains modified data. The P-cache only needs to be flushed for error handling.

The “PREFETCH fcn=16” instruction can be used to invalidate, or flush a P-cache entry, and to prefetch non-cacheable data, after the data is loaded into registers from the P-cache.

The cache line size is 64 bytes with 32-byte subblocks. The P-cache is globally invalidated on context changes and MMU updates, individual lines are invalidated on store hits.

The P-cache is globally invalidated if any of the following conditions occur:

- Context registers are written.
- Demap operation in the D-MMU
- D-MMU is turned on or off.

Individual lines are invalidated on any of the following conditions:

- A store hits
- An external snoop hit
- Use of software prefetch invalidate function. (PREFETCH with fcn = 16)

The P-cache is used for software prefetch instructions as well as a autonomous hardware prefetch from the L2-cache. This cache never needs to be flushed (not even for address aliases).

9.1.4 Second Level and Write Caches (L2-Cache, W-Cache)

The on-chip L2-cache[1] and the W-cache—are physically-indexed, physically-tagged (PIPT). These caches have no references to virtual address and context information. The operating system needs no knowledge of such caches after initialization, except for stable storage management and error handling.

The L2-Cache is a 1 MB unified, write-back, write-allocate, 4-way set associative cache with 64-byte lines. The L2-cache does not include the contents of the Instruction Cache, Prefetch Cache and Data Cache. The replacement policy is pseudo-random. The L2-cache cannot be disabled by software.

It is necessary to flush the L2-cache for stable storage.

Instruction fetches bypass the L2-cache when the following occurs:

- I-MMU is disabled AND when the CP bit in the Data Cache Unit Control Register is not set.
- The processor is in `RED_state`.
- Access is mapped by the I-MMU as nonphysical cacheable.

Data accesses bypass the L2-cache if the D-MMU enable bit in the DCU Control Register is clear, or if the access is mapped by the D-MMU as non-physical-cacheable (unless `ASI_PHYS_USE_EC` is used).

The system must provide a non-cacheable, scratch memory for booting code use until the MMUs are enabled.

Block loads and block stores, which load or store a 64-byte block of data from memory to the floating-point register file, do not allocate into the L2-cache, in order to avoid pollution. Prefetch Read Once instructions, which load a 64-byte block of data into the P-cache, do not allocate into the L2-cache.

The W-cache is a 2 KB, 4-way associative, with 64 bytes per line and 32-byte sub-blocks. The W-cache is included in the L2-cache, and flushing the L2-cache ensures that the W-cache has also been flushed.

1. L2-cache and Embedded Cache (E-cache) are used interchangeably.

9.1.5 L2-Cache Replacement Policy

The selection is more complicated when some of the ways are blocked using EC_block. That is not shown here. The victim way is determined by a 5-bit Linear Feedback Shift Register (LFSR,) which is described in the following code. Note that the code reflects the algorithm when all 4 ways are active.

CODE EXAMPLE 9-1 reflects the cache replacement algorithm when all four ways of the L2-cache are active.

CODE EXAMPLE 9-1 L2-Cache Replacement Policy

```
module lfsr (rand_out, event_in, reset, clk);

output  [3:0]   rand_out;
input   event_in;
input   reset;
input   clk;

wire [4:0] lfsr_reg;

dffe #(5) ff_lfsr (lfsr_reg, lfsr_in, ~reset, event_in, clk);

// 01010 is the non-reachable state for this implementation.
wire [4:0] lfsr_in = {~lfsr_reg[0],
                      lfsr_reg[0] ^ lfsr_reg[4],
                      lfsr_reg[3],
                      lfsr_reg[0] ^ lfsr_reg[2],
                      lfsr_reg[0] ^ lfsr_reg[1]};


// update on reads that miss the L2-cache
assign event_in = ec_lt_cs_r_d1 & ~ec_lt_we_r_d1 &
~lt_ec_hit_miss_d1;


dffire #(5) f_lfsr (lfsr_reg, lfsr_in, reset, event_in, clk);

assign rand_out = { lfsr_reg[1] & lfsr_reg[0],
                    lfsr_reg[1] & ~lfsr_reg[0],
                   ~lfsr_reg[1] & lfsr_reg[0],
                   ~lfsr_reg[1] & ~lfsr_reg[0]};

endmodule
```

9.1.6 L2-Cache Locking

Networking applications get performance boost if the interrupt code is in the L2-cache. Therefore, software can have guaranteed latency to certain critical data and instructions. The UltraSPARC IIIi processor supports way blocking, that is, software can enable/disable a way to take part in replacement strategy. Software could initialize a way with L2-cache diagnostic writes and then prohibit this way from the replacement algorithm.

Software flushes a particular line in L2-cache even if it is locked, if it desires to do so by issuing the `ASI_ECACHE_FLUSH` instruction.

Note – If software blocks all four ways of the L2-cache, then the ECU will behave as if only way 0 is blocked.

9.2 Cache Flushing

Data in the write-invalidate or write-through caches can be flushed by invalidating the entry in the cache. Modified data in the L2-cache and W-cache must be written back to memory when flushed.

Cache flushing is required in the following cases:

- A D-cache flush is needed when a physical page is changed from (virtually) cacheable to (virtually) non-cacheable, or an illegal address aliasing is created (see Section 9.2.1 “Address Aliasing Flushing” on page 206). This is done using ASI 0x42, `ASI_DCACHE_INVALIDATE`, which specifies a physical address to flush, like for a system bus snoop.
- L2-cache flush is needed for stable storage. This is done with either a `ASI_ECACHE_FLUSH` or a store with `ASI_BLK_COMMIT`. Flushing the L2-cache will flush the corresponding blocks from the W-cache. See Section 9.2.2 “Committing Block Store Flushing” on page 206.
- L2-cache, D-cache, prefetch cache, and I-cache flushes may be required when an ECC error occurs on a read from the memory or the L2-cache. When an ECC error occurs, invalid data may be written into one of the caches and the cache lines must be flushed to prevent further corruption of data.

Note – When flushing a single 64-byte line, with a given PA, there are sixteen locations that must be flushed in the D-cache. This is because it has 32-byte lines (two places), one VA index bit (two places), and the PA can simultaneously exist in all four ways of a set (four places).

9.2.1 Address Aliasing Flushing

A side-effect inherent in a virtual-indexed cache is *illegal address aliasing*. Aliasing occurs when multiple virtual addresses map to the same physical address.

Caution – Since the D-cache is indexed with the virtual address bits and is larger than the minimum page size, it is possible for the different aliased virtual addresses to end up in different cache blocks. Such aliases are illegal because updates to one cache block will not be reflected in aliased cache blocks. (There are corner cases where the same cache block can end up in different ways, within the same set (index); the hardware will update all ways within a set that have the line.)

Normally, software avoids illegal aliasing by forcing aliases to have the same address bits (*virtual color*) up to an *alias boundary*. The minimum alias boundary is 16 KB.

When the alias boundary is violated, software must flush the D-cache if the page was virtually cacheable. In this case, only one mapping of the physical page can be allowed in the D-MMU at a time.

Alternatively, software can turn off the virtual caching of illegally aliased pages. This allows multiple mapping of the alias to be in the D-MMU and avoids flushing the D-cache each time a different mapping is referenced.

Note – A change in virtual color when allocating a free page does not require a D-cache flush, because the D-cache is write through.

9.2.2 Committing Block Store Flushing

Stable storage must be implemented by software cache flush. Examples of stable storage are battery-backed memory and a transaction log. Data which is present and modified in the L2-cache or the W-cache must be written back to the stable storage.

Two ASIs (`ASI_BLK_COMMIT_PRIMARY` and `ASI_BLK_COMMIT_SECONDARY`) perform these write backs efficiently when software can ensure exclusive write access to the block being flushed. These ASIs write back the data from the floating-point registers to memory and invalidate the entry in the cache. The data in the floating-point registers must first be loaded by a block load instruction. A `MEMBAR #Sync` instruction can be used to ensure that the flush is complete.

9.2.3 L2-Cache Flushing

L2-cache flushing may also be accomplished by ASI loads (ASI_ECACHE_FLUSH). This is done by reading a range of addresses that map to the corresponding cache line in a particular way being flushed, forcing out modified entries in the local cache. The load ASI physical address will be the same as its virtual address, and will cause a miss if the line it is intended to replace is in a valid state (M/O/E/S) in the L2-cache. If the line is modified (M/O), the data will also be forced out to memory. The hardware will guarantee a read miss to the way accessed by the ASI even if there is a hit in any of the other ways. The fetched line will be installed in the Invalid state (I) in the L2-cache.

Note – Diagnostic ASI accesses to the L2-cache can be used to invalidate a line, but they are not an alternative to above type of flushing. Modified data in the L2-cache will not be written back to memory using these Diagnostic ASI accesses (these are destructive flushes).

L2-cache flush operation is performed by accessing ASI 0x4E (ASI_ECACHE_FLUSH). This ASI can be accessed only by a privileged instruction. A privileged action trap if PSTATE.PRIV not set. The L2-cache flush ASI format is illustrated in FIGURE 9-1 and described in TABLE 9-1.

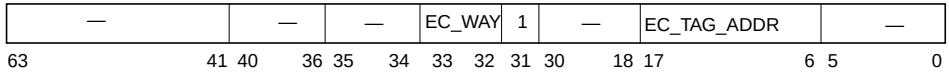


FIGURE 9-1 L2-Cache Flush ASI Format

TABLE 9-1 L2-Cache Flush ASI Format

Bit	Field	Description
63:43	—	*Reserved*. Set to 0.
42:41	—	*Reserved*. Set to 0. Makes sure that the victimizing read is treated as a cacheable space.
40:36	—	*Reserved*
35:34	—	*Reserved*.
33:32	EC_WAY	L2 Way Selection
31	—	*Reserved*. Set to 1.
30:18	—	*Reserved*. Set to 0.
17:6	EC_TAG_ADDR	Index into the L2-cache
5:0	—	*Reserved*. Set to 0.

A load using the L2-cache Flush ASI can be used to flush a L2-cache line with EC_TAG_ADDR supplying the index and EC_WAY providing the required way.

The loads will not generate a miss in L2-cache if there is no dirty data in the associated set/way. However, they will cause a miss if there is dirty data to be flushed (the W-cache data will be merged with L2-cache data if needed). The returned data for this load miss will be installed in an invalid state. A store to this ASI will execute like a NOP.

Clean (S or E) lines are invalidated immediately. There is no JBUS read.

The VA<42:0> is used directly to create the PA<42:0> used for the read that goes out to JBUS (as an RDS).

PA<33:0> is used for the DRAM at each memory controller. PA<33:32> is used for the Chip Select decode, and not all encodings may point to a DIMM in a system. Therefore, it is not possible to create an address that will definitely read from a DRAM.

The read will receive AFSR.JETO if a nonexistent port is used in the address, causing a fatal error (system reset).

The read will not receive AFSR.TO if the DRAM does not exist on a valid port. Flush completes normally. Unknown data is installed in the invalid state.

It is possible to log UE/FRU/RUE or CE/FRC/RCE due to the DRAM read, if DRAM exists at the address created by hardware. (A read is done to create a displacement flush.) If this happens, the processor traps like a normal read that triggered these errors.

In a multiprocessor system, the target address must point to your own ID, because as a destination, the UltraSPARC IIIi processor cannot tolerate having to return multiple read error packets to different masters around the same time (the system will hang). By pointing to your own ID, a JBUS read error packet is not used. However, note that the address does not need to point to valid DRAM.

It is possible that the JBUS read address may actually be in another processor's cache. The data will be correctly returned from that cache. Since a JBUS RDS is used, any write permission will be removed at that cache (M to O). If the line was E, it will be reduced to S state in other caches. It is possible that such a cache read could cause an L2-cache error to be logged by that other processor.

Note – Since the I-cache, D-cache, and P-cache are non-inclusive, flushing the L2-cache has no affect on them, and they may need to be flushed separately. The W-cache is inclusive, and gets flushed with the L2-cache, if necessary.

9.3 Controlling P-Cache

This section clarifies the use of DCUCR.PE, DCUCR.HPE, and DCUCR.SPE bits.

Note – Block loads do not cause installs into the P-cache. They are also not allowed to hit on the P-cache and, therefore, never triggers hardware prefetch.

Non-cacheable address space never installs in P-cache or L2-cache, unless a software prefetch is done specifically to the non-cacheable address (should be followed by a prefetch invalidate to that address, after using the data).

TABLE 9-2 Explanation of P-cache control bits

DCUCR. PE	DCUCR. HPE	DCUCR. SPE	Hardware Prefetch Enabled?	Software Prefetch Enabled?	FP load miss (32B) installed in the P-Cache?	FP loads checked for P-Cache hit/miss?
0	X	X	no	no	no	no
1	0	0	no	no	no	yes
1	0	1	no	yes	no	yes
1	1	0	yes	no	yes	yes
1	1	1	yes	yes	yes	yes

9.4 Translation Lookaside Buffers (TLBs)

The Instruction TLB has a 16-entry, fully-associative TLB to hold entries for 64 KB, 512 KB, 4 MB pages, and all locked pages of any size, and a 128-entry, 2-way associative TLB is used for the unlocked 8 KB pages.

The Data TLB has a 16-entry, fully-associative TLB to hold entries for unlocked 8 KB, 64 KB, 512 KB, 4 MB pages, and all locked pages, and two 512-entry, 2-way associative TLBs used for unlocked 8 KB, 64 KB, 512 KB, or 4 MB pages.

9.4.1 TLB Flushing

A demap-all operation that removes all unlocked TTEs has been added to both the I-TLBs and D-TLBs.

9.4.2 TTE Format

The UltraSPARC IIIi processor now has the additional elements in the TTE format:

- Physical Address field was expanded from 28 bits (PA<40:13>, TTE<40:13>) to 30 bits (PA<42:13>, TTE<42:13>)
- A snoop bit was added to mark a page as outside the coherence domain (TTE<47>)

9.4.3 Synchronous Fault Status Register (SFSR) Extensions

One status bit has been added to the I/D-TLB SFSRs:

- NF — Set to indicate the faulting operation was a speculative load instruction

A new fault type was added to the FT field of the SFSR to indicate an I/D-TLB miss.

9.4.4 I/D Translation Storage Buffer Register

Three new register extensions of the I/D-TSB register have been added to the UltraSPARC IIIi processor. These registers allow a different TSB virtual address base to be used for each of the three virtual address spaces (Primary, Secondary, Nucleus) in the D-TLB and two virtual address spaces (Primary, Nucleus) in the I-TLB. On an I/D-TLB miss it selects which TSB Extension Register to use to form the TSB base address based on the virtual space accessed by the faulting instruction.

9.4.5 TLB Data Access Register

The access address for the TLB Data Access Register has been expanded to enable access to three TLBs each with up to 512 entries.

Warning – Under some circumstances a diagnostic read from the fully associative TLBs (ASI_DTLB_DATA_ACCESS_REG (ASI = 0x5D) and ASI_ITLB_DATA_ACCESS_REG (ASI = 0x55) will return wrong data. Software should read the fully associative TLB Entry twice, back-to-back. The second access will return correct data.

9.4.5.1 Special Case for Data TLBs

If after any memory access instruction that misses TLB is followed by a read (`LDXA` from `ASI_DTLB_DATA_ACCESS_REG`, that is, `ASI` = 0x5d) access from fully associative TLBs and the accessed TTE has page size set to 64KB/512KB/4MB then data returned from TLB will be wrong.

9.4.5.2 Special Case for Instruction TLBs

If after any instruction that misses instruction TLB is followed by a read (`LDXA` from `ASI_ITLB_DATA_ACCESS_REG`, that is, `ASI`=0x55) access from fully associative TLBs and the accessed TTE has page size set to 64KB/512KB/4MB then data returned from TLB will be wrong.

9.4.6 TLB Diagnostic Register

This is a new register to replace the function of the diagnostic bits in the TTE.

SECTION V

Supervisor Programming

CHAPTER 10

Interrupt Handling

Processors and I/O devices can interrupt a selected processor by assembling and sending an interrupt packet consisting of eight 64-bit words of interrupt vector data. The contents of these data are defined by software convention. Thus, hardware interrupts and cross-calls can have the same hardware mechanism for interrupt delivery and can share a common software interface for processing.

The interrupt requesting/receiving mechanism is a two-step process: the sending of an interrupt request on a vector data register to the target and the scheduling of the received interrupt request on the target upon receipt.

An interrupt request packet is sent by processors or I/O devices through the interrupt vector *dispatch* mechanism and is received by the specified target through the interrupt vector *receive* mechanism. Upon receipt of an interrupt request packet, a special trap is invoked on the target processor. The trap handler software invoked in the target processor then schedules the interrupt request to itself by posting the interrupt into `SOFTINT` register at the desired interrupt level.

Note that the processor may not send an interrupt request packet to itself through the interrupt dispatch mechanism. Separate sets of dispatch (outgoing) and receive (incoming) interrupt data registers allow simultaneous interrupt dispatching and receiving.

Different aspects of interrupt handling are described in the following sections:

- Interrupt Vector Dispatch
- Interrupt Vector Receive
- Interrupt Global Registers
- Interrupt ASI Registers
- Software Interrupt Register (SOFTINT)

10.1 Interrupt Vector Dispatch

To dispatch an interrupt or cross-call, a processor or I/O device first writes to the outgoing Interrupt Vector Data Registers according to an established software convention, described below. A subsequent write to the Interrupt Vector Dispatch Register triggers the interrupt delivery. The status of the interrupt dispatch can be read by polling the ASI_INTR_DISPATCH_STATUS BUSY and NACK bits. A MEMBAR #Sync should be used before polling begins to ensure that earlier stores are completed. CODE EXAMPLE 10-1 shows the pseudo-code sequence that sends an interrupt.

BUSY and NACK bits of the Interrupt Vector Dispatch Status Register, listed in TABLE 10-1, indicate the status of the interrupt dispatched.

TABLE 10-1 BUSY and NACK Bits of Interrupt Vector Dispatch Register

BUSY	NACK	Status
0	0	Interrupt dispatch successful
1	0	Interrupt dispatch pending
0	1	Interrupt dispatch failed

The ASI_INTR_DISPATCH_STATUS Register contains four pairs of BUSY/NACK bit pairs enabling interrupts to be pipelined. Specifying a unique pair of BUSY/NACK bits used for each interrupt when writing, the Interrupt Dispatch Register enables up to four interrupts to be outstanding at one time.

Note – The processor may not send an interrupt vector to itself through outgoing interrupt vector data registers. Doing so causes undefined interrupt vector data to be returned.

CODE EXAMPLE 10-1 Code Sequence for Interrupt Dispatch

```
Read state of ASI_INTR_DISPATCH_STATUS; Error if BUSY
<no pending interrupt dispatch packet>
Repeat
    Begin atomic sequence(PSTATE.IE ← 0)
    Store to IV data reg 0 at ASI_INTR_W, VA=0x40 (optional)
    Store to IV data reg 1 at ASI_INTR_W, VA=0x48 (optional)
    Store to IV data reg 2 at ASI_INTR_W, VA=0x50 (optional)
    Store to IV data reg 3 at ASI_INTR_W, VA=0x58 (optional)
    Store to IV data reg 4 at ASI_INTR_W, VA=0x60 (optional)
    Store to IV data reg 5 at ASI_INTR_W, VA=0x68 (optional)
    Store to IV data reg 6 at ASI_INTR_W, VA=0x80 (optional)
```

CODE EXAMPLE 10-1 Code Sequence for Interrupt Dispatch *(Continued)*

```
    Store to IV data reg 7 at ASI_INTR_W, VA=0x88 (optional)
    Store to IV dispatch at ASI_INTR_W, VA<63:29>=0,
        VA<28:24>=BUSY/NACK bit #,VA<23:14>=ITID,
        VA<13:0>=0x70 initiates interrupt delivery
    Membar #Sync (wait for stores to finish)
    Poll state of ASI_INTR_DISPATCH_STATUS (BUSY, NACK)
        Loop if BUSY
    End atomic sequence(PSTATE.IE ← 1)
   DONE if !NACK
    (Retry after random delay if NACKED)
Until DONE
```

Note – To avoid deadlocks, enable interrupts for some period before retrying the atomic sequence. Alternatively, implement the atomic sequence with locks without disabling interrupts.

10.2 Interrupt Vector Receive

When an interrupt is received, all eight Interrupt Data Registers are updated, regardless of which are being used by software. This update is done in conjunction with the setting of the BUSY bit in the ASI_INTR_RECEIVE register. At this point, the processor inhibits further interrupt packets from the system bus. If interrupts are enabled (PSTATE.IE = 1), then an interrupt trap (trap type 60_{16}) is generated. Software reads the ASI_INTR_RECEIVE register and Incoming Interrupt Data Registers to determine the entry point of the appropriate trap handler. All of the external interrupt packets are processed at the highest interrupt priority level and are then reprioritized as lower-priority interrupts in the software handler. CODE EXAMPLE 10-2 illustrates interrupt receive handling.

CODE EXAMPLE 10-2 Code Sequence for an Interrupt Receive

```
Read state of ASI_INTR_RECEIVE; Error if !BUSY
Read from IV data reg 0 at ASI_SDB_INTR_R, VA=0x40 (optional)
Read from IV data reg 1 at ASI_SDB_INTR_R, VA=0x48 (optional)
Read from IV data reg 2 at ASI_SDB_INTR_R, VA=0x50 (optional)
Read from IV data reg 3 at ASI_SDB_INTR_R, VA=0x58 (optional)
Read from IV data reg 4 at ASI_SDB_INTR_R, VA=0x60 (optional)
Read from IV data reg 5 at ASI_SDB_INTR_R, VA=0x68 (optional)
```

CODE EXAMPLE 10-2 Code Sequence for an Interrupt Receive *(Continued)*

```
Read from IV data reg 6 at ASI_SDB_INTR_R, VA=0x80 (optional)
Read from IV data reg 7 at ASI_SDB_INTR_R, VA=0x88 (optional)
Determine the appropriate handler
Handle interrupt or reprioritize this trap and
    set the SOFTINT register
Store zero to ASI_INTR_RECEIVE to clear the BUSY bit
```

10.3 Interrupt Global Registers

A separate set of global registers is implemented to expedite interrupt processing. As described in Section 10.2, “Interrupt Vector Receive”, the processor takes an interrupt trap after receiving an interrupt packet. Software uses a number of scratch registers while determining the appropriate handler and constructing the interrupt state.

A separate set of eight Interrupt Global Registers (IGRs) replaces the eight programmer-visible global registers during interrupt processing. After an interrupt trap is dispatched, the hardware selects the interrupt global registers by setting the PSTATE.IG field. The previous value of PSTATE is restored from the trap stack by a DONE or RETRY instruction on exit from the interrupt handler.

10.4 Interrupt ASI Registers

MEMBAR #Sync is generally needed after stores to interrupt ASI registers, which avoids unnecessary effects caused by possible prefetches to the locations with side effect.

10.4.1 Outgoing Interrupt Vector Data<7:0> Register

ASI_INTR_DATA0_W (data 0): ASI = 77_{16}, VA<63:0> = 40_{16}
ASI_INTR_DATA1_W (data 1): ASI = 77_{16}, VA<63:0> = 48_{16}
ASI_INTR_DATA2_W (data 2): ASI = 77_{16}, VA<63:0> = 50_{16}
ASI_INTR_DATA3_W (data 3): ASI = 77_{16}, VA<63:0> = 58_{16}
ASI_INTR_DATA4_W (data 4): ASI = 77_{16}, VA<63:0> = 60_{16}
ASI_INTR_DATA5_W (data 5): ASI = 77_{16}, VA<63:0> = 68_{16}
ASI_INTR_DATA6_W (data 6): ASI = 77_{16}, VA<63:0> = 80_{16}
ASI_INTR_DATA7_W (data 7): ASI = 77_{16}, VA<63:0> = 88_{16}

Name: ASI_INTR_DATA_W: Outgoing Interrupt Vector Data Registers (Privileged, Write-only)

TABLE 10-2 describes the register field of the eight Outgoing Interrupt Vector Data Registers.

TABLE 10-2 Outgoing Interrupt Vector Data Register Format

Bits	Field	Type	Description
63:0	Data	W	Interrupt data

A write to these eight registers modifies the outgoing Interrupt Dispatch Data Registers.

Non-privileged access to this register causes a *privileged_action* trap. An attempt to read this register causes a *data_access_exception* trap.

10.4.2 Interrupt Vector Dispatch Register

ASI 77_{16}

VA<63:19> = 0
VA<18:14> = Target Processor ID
VA<13:0> = 70_{16}

Name: ASI_INTR_W (Interrupt dispatch, Privileged, Write-only)

TABLE 10-3 describes the fields of the Interrupt Vector Dispatch Register.

TABLE 10-3 Interrupt Vector Dispatch Register Format

Bits	Field	Type	Description
VA<18:14>	ITID	W	*Interrupt Target ID*. Specifies the interrupt target processor using the BUSY/NACK bit pair BN, along with the contents of the eight Interrupt Vector Data Registers. VA<15:14> specifies which of the BUSY/NACK bit pairs to use for the interrupt (the lower two bits of Agent/Target ID are direct mapped to BN#). • 0x0 in this field selects BUSY/NACK bits ASI_INTR_DISPATCH_STATUS<1:0>. • 0x1 in this field selects BUSY/NACK bits ASI_INTR_DISPATCH_STATUS<3:2>. • 0x2 in this field selects BUSY/NACK bits ASI_INTR_DISPATCH_STATUS<5:4>. • 0x3 in this field selects BUSY/NACK bits ASI_INTR_DISPATCH_STATUS<7:6>. If there are more than four processors in the system, software must take care of aliasing caused by direct mapping of the lower two bits of AGENT IDs.

A write to this ASI triggers an interrupt vector dispatch to the target processor identified with Interrupt Target ID (ITID), using BUSY/NACK bit pair BN along with the contents of the eight Interrupt Vector Data Registers. Note that the write acts as a trigger; however, the data for the write is ignored.

A read from the Interrupt Vector Dispatch Register causes a *data_access_exception* trap. Non-privileged access to this register causes a *privileged_action* trap.

10.4.3 Interrupt Vector Dispatch Status Register

ASI 48_{16}

VA<63:0> = 0

Name: ASI_INTR_DISPATCH_STATUS (Privileged, Read-only)

TABLE 10-4 describes the fields of the Interrupt Vector Dispatch Status Register.

TABLE 10-4 Interrupt Dispatch Status Register Format

Bits	Field	Type	Description
<63:8>	- -		*Reserved*, read as 0.
1,3,5,7	NACK	R	Set if interrupt dispatch has failed. Cleared at the start of every interrupt dispatch attempt; set when a dispatch has failed.
0,2,4,6	BUSY	R	Set when there is an outstanding dispatch.

In the UltraSPARC IIIi processor, four BUSY/NACK pairs are implemented in the Interrupt Vector Dispatch Status Register.

The status of up to four outgoing interrupts can be read from ASI_INTR_DISPATCH_STATUS BUSY/NACK bits. This register contains up to 4 pairs of BUSY/NACK bit pairs: the pairs at <1:0>, <3:2>, <5:4>, and <7:6> are referred to as pair 0, pair 1, pair 2, and pair 3, respectively.

The VA<15:14> field of the Interrupt Dispatch Register specifies which BUSY/NACK bit pair will be used for the interrupt.

Writes to this ASI cause a *data_access_exception* trap. Non-privileged access to this register causes a *privileged_action* trap.

10.4.4 Incoming Interrupt Vector Data<7:0>

ASI_INTR_R (data 0): ASI = $7F_{16}$, VA<63:0> = 40_{16}
ASI_INTR_R (data 1): ASI = $7F_{16}$, VA<63:0> = 48_{16}
ASI_INTR_R (data 2): ASI = $7F_{16}$, VA<63:0> = 50_{16}
ASI_INTR_R (data 3): ASI = $7F_{16}$, VA<63:0> = 58_{16}
ASI_INTR_R (data 4): ASI = $7F_{16}$, VA<63:0> = 60_{16}
ASI_INTR_R (data 5): ASI = $7F_{16}$, VA<63:0> = 68_{16}
ASI_INTR_R (data 6): ASI = $7F_{16}$, VA<63:0> = 80_{16}
ASI_INTR_R (data 7): ASI = $7F_{16}$, VA<63:0> = 88_{16}

Name: ASI_INTR_R (Privileged, Read-only)

TABLE 10-5 describes the register field of the eight Incoming Interrupt Vector Data Registers.

TABLE 10-5 Incoming Interrupt Vector Data Register Format

Bits	Field	Type	Description
63:0	Data	R	Interrupt data

A read from these registers returns incoming interrupt information from the incoming Interrupt Receive Data Registers.

Non-privileged access to this register causes a *privileged_action* trap.

10.4.5 Interrupt Vector Receive Register

ASI 49_{16}

VA<63:0> = 0

Name: ASI_INTR_RECEIVE (Privileged)

TABLE 10-6 describes the fields of the Interrupt Receive Register.

TABLE 10-6 Interrupt Receive Register Format

Bits	Field	Type	Description
63:6	--	R	*Reserved*. Read as 0.
5	BUSY	RW	Set when an interrupt vector is received. The BUSY bit must be cleared by software writing zero.
4:0	SOURCE	R	Source ID of Interrupter. Accurate when BUSY is set. Source ID is the AID field of the interrupting agent.

The status of an incoming interrupt can be read from ASI_INTR_RECEIVE. The BUSY bit is cleared by writing zero to this register. BUSY bit is also cleared during Power-on Reset.

Non-privileged access to the Interrupt Vector Receive Register causes a *privileged_action* trap.

10.5 Software Interrupt Register (SOFTINT)

To schedule interrupt vectors for processing at a later time, each processor can send itself signals by setting bits in the SOFTINT register.

The SOFTINT register (ASR 16_{16}), described in TABLE 10-7, is used for communication from nucleus (TL > 0) code to kernel (TL = 0) code. 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*>.

TABLE 10-7 SOFTINT Register Format

Bits	Field	Description	RW
<16>	STICK_INT	System Timer interrupt. When the STICK_CMPR INT_DIS field is cleared (that is, STICK interrupt is enabled) and the 63-bit STICK_Compare Register's STICK_CMPR field matches the STICK Register's counter field, the STICK_INT field is set and a software interrupt is generated.	RW
<15:1>	SOFTINT<15:1>	When set, bits<15:1> cause interrupts with each bit corresponding to levels IRL<15:1>, respectively.	RW
<0>	TICK_INT	Timer interrupt. When TICK_CMPR's INT_DIS field is cleared (that is, TICK interrupt is enabled) and the 63-bit TICK_Compare Register's TICK_CMPR field matches the TICK Register's counter field, the TICK_INT field is set and a software interrupt is generated.	RW

Non-privileged access to this register causes a *privileged_opcode* trap.

10.5.1 Setting the Software Interrupt Register

Setting SOFTINT<*n*> is done by a write to the SET_SOFTINT register (ASR 14_{16}), with bit *n* corresponding to the interrupt level set. The value written to the SET_SOFTINT register is effectively ORed into the SOFTINT register. This approach allows the interrupt handler to set one or more bits in the SOFTINT register with a single instruction.

Read accesses to the SET_SOFTINT register cause an *illegal_instruction* trap. Non-privileged accesses to this register cause a *privileged_opcode* trap.

When the nucleus returns, if (PSTATE.IE = 1) and (*n* > PIL), then the processor will receive the highest-priority interrupt IRL<*n*> of the asserted bits in SOFTINT<16:0>. The processor then takes a trap for the interrupt request, and the nucleus sets the return state to the interrupt handler at that PIL and returns to TL = 0. In this manner, the nucleus can schedule services at various priorities and process them according to their priority.

10.5.2 Clearing the Software Interrupt Register

When all interrupts scheduled for service at level *n* have been serviced, the kernel writes to the CLEAR_SOFTINT register (ASR 15_{16}) with bit *n* set, to clear that interrupt. The complement of the value written to the CLEAR_SOFTINT register is effectively ANDed with the SOFTINT register. This approach allows the interrupt handler to clear one or more bits in the SOFTINT register with a single instruction.

Read accesses to the CLEAR_SOFTINT register cause an *illegal_instruction* trap. Non-privileged write accesses to this register cause a *privileged_opcode* trap.

The timer interrupt TICK_INT and system timer interrupt STICK_INT are equivalent to SOFTINT<14> and have the same effect.

Note – To avoid a race condition between the kernel clearing an interrupt and the nucleus setting it, the kernel should examine the queue for any valid entries again after clearing the interrupt bit.

TABLE 10-8 summarizes the SOFTINT ASRs.

TABLE 10-8 SOFTINT ASRs

ASR Value	ASR Name	Type	Description
14_{16}	SET_SOFTINT	W	Sets bit(s) in Soft Interrupt Register.
15_{16}	CLEAR_SOFTINT	W	Clears bit(s) in Soft Interrupt Register.
16_{16}	SOFTINT	RW	Per-processor Soft Interrupt Register.

SECTION VI

Performance Programming

CHAPTER 11

Performance Instrumentation

Performance instrumentation consists of processor event counters that can be used to gather statistics during program execution. Approximately 70 events can be monitored, two at a time, to gain information about the performance of the processor. Cache miss counts and stall times, for example, can be measured using two, 32-bit Performance Instrumentation Counters (PICs). Some event counting can be synthesized from the event counters available to provide additional program execution statistics.

The counters can be monitored during program execution to gather on-going statistics or reconfigure during steady-state program execution to gather statistics for more than two events.

The Performance Control Register (PCR) is used to select the events to monitor and provide control for counting in privileged and/or non-privileged modes.

Each of the two 32-bit performance instrumentation counters (PIC), PICL, and PICU, can accumulate over four billion events before wrapping. Event logging counts can be extended by periodically reading contents of the performance instrumentation counters to detect and avoid an overflow. An interrupt can be enabled on a counter overflow. Additional event or stall cycle statistics can be collected by reading the PIC counts between repeated program executions.

This chapter describes the performance instrumentation features in the following sections:

Supervisor/User Mode

Access to the PCR is restricted to supervisor software. User software accessing the PCR causes a *privileged_opcode* trap.

Supervisor software controls user accessibility to the PIC counters through the PCR.PRIV field. When PCR.PRIV = 1 (supervisor access only), an attempt by user software to access the PIC register causes a *privileged_action* trap. By default, PCR.PRIV = 0. In this default state, the PIC register is accessible to user software.

In Supervisor/User configuration, the mode in which the counters are enabled to count is controlled by setting the PCR.UT (User Trace) and PCR.ST (System Trace) bits.

11.1 Performance Control Register (PCR)

The 64-bit PCR and PIC are accessed through read/write Ancillary State Register (ASR) instructions (RDASR/WRASR). PCR and PIC are located at ASRs 16 (10_{16}) and 17 (11_{16}), respectively.

Two events can simultaneously be measured by setting the PIC_SL and PIC_SU fields. The counters can be enabled separately for Supervisor and User mode using UT and ST fields. The selected statistics are reflected during subsequent accesses to the PICs.

The PCR is a read/write register used to control the counting of performance monitoring events. FIGURE 11-1 shows the details of the PCR and TABLE 11-1 describes the various fields of the PCR. Counts are collected in the PIC register (see Section 11.2 “Performance Instrumentation Counter (PIC) Register” on page 230”).

PCR - Performance Control Register — ASR Register

The PCR selects the events and controls the operating modes of the Performance Instrumentation Counters (PICs).			
ASR 16_{10}	64-bit Read/Write	Privileged Mode, otherwise *privileged_action* trap.	Reset: 0x0000.0000

FIGURE 11-1 Performance Control Register

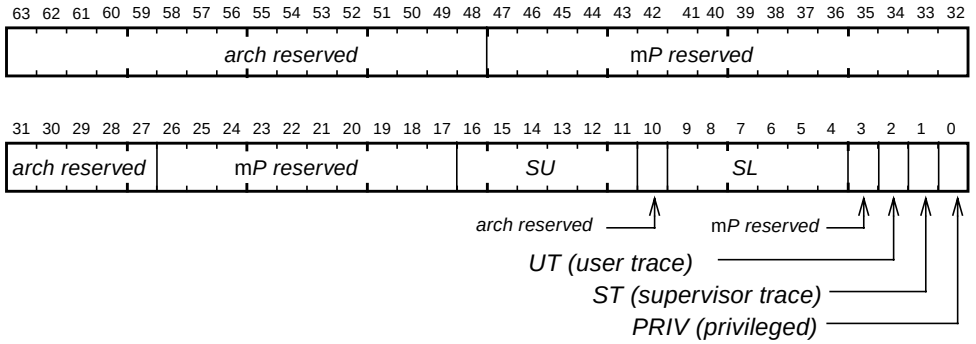


TABLE 11-1 PCR Bit Description

Bit	Field	Description
16:11	SU	Selects 1 of up to 64 counters accessible in the upper half (bits <63:32>) of the PIC register.
9:4	SL	Selects 1 of up to 64 counters accessible in the lower half (bits <31:0>) of the PIC register.
2	UT	User Trace Enable. If set to one, counts events in non-privileged mode (User).
1	ST	System Trace Enable. If set to one, counts events in privileged mode (Supervisor). Notes: If both PCR.UT and PCR.ST are set to one, all selected events are counted. If both PCR.UT and PCR.ST are zero, counting is disabled. PCR.UT and PCR.ST are global fields which apply to both PIC pairs.
0	PRIV	Privileged. If PCR.PRIV = 1, a non-privileged (PSTATE.PRIV = 0) attempt to access PIC (via a RDPIC or WRPIC instruction) will result in a *privileged_action* exception.
63:48 31:27 10	–	Reserved by SPARC architecture. Read zero, Write zero, or Write value read previously.
47:32 26:17 3	–	Unused in the UltraSPARC IIIi processor. Read zero, Write zero, or Write value read previously.

11.2 Performance Instrumentation Counter (PIC) Register

The difference between the values read from the PIC on two reads reflects the number of events that occurred between register reads. Software can only rely on read-to-read PIC accesses to get an accurate count and not a write-to-read of the PIC counters. Every time the select values (PCR.SU or PCR.SL) are changed, the PIC register is reset and starts counting from zero. If there is a context switch, it is the responsibility of software to save the previous PCR and PIC values. FIGURE 11-2 shows the details of the PIC and TABLE 11-2 describes the various fields of the PIC.

PIC - Performance Instrumentation Counter register	ASR Register

The PIC register provides access to the counter values for the two events being monitored.			
ASR 17_{10}	64-bit Read/Write **Note:** Writes are designed for diagnostic and test purposes.	Accessibility depends on *PCR.PRIV* bit: 0 = accessible in any mode 1 = accessible in Supervisor Mode, otherwise *privileged_action* trap	Reset: 0x0000.0000

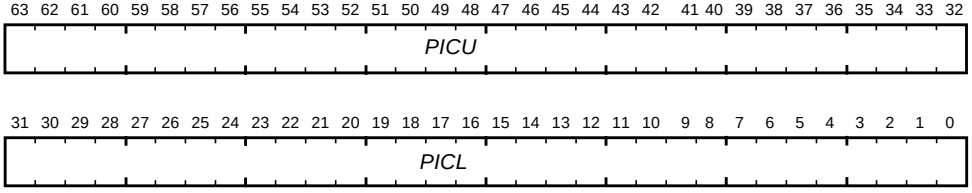


FIGURE 11-2 Performance Instrumentation Counter Register

TABLE 11-2 PIC Register Fields

Bit	Field	Description
63:32	PICU	32-bit field representing the count of an event selected by the SU field of the Performance Control Register (PCR)
31:0	PICL	32-bit field representing the count of an event selected by the SL field of the Performance Control Register (PCR)

11.2.1 PIC Counter Overflow Trap Operation

When a PIC counter overflows, an interrupt is generated as described in TABLE 11-3.

TABLE 11-3 PIC Counter Overflow Processor Compatibility Comparison

Function	Description
PIC Counter Overflow	On overflow, a counter wraps to zero, SOFTINT register bit 15 is set to one, and an *interrupt_level_15* trap (a disrupting trap). The counter overflow trap is triggered on the transition from value FFFF FFFF$_{16}$ to value 0. The point at which the interrupt is delivered may be several instructions after the instruction responsible for the overflow event. This situation is known as a “skid.”

11.3 Performance Instrumentation Operation

shows how an operating system might use the performance instrumentation features to provide event monitoring services. Setup the PCR register as desired to select two events and in which modes data should be collected. The monitoring must consider the real effects of the computer that includes calls to the system and interrupts. When used, the PCR register is considered part of a process state and must be saved and restored when switching process contexts.

Multiple data collection times can be done while the program executes to show on-going statistics.

11.3.1 Gathering Data for More Than Two Events

When more than two events need to be monitored, the program, program sequence, or program loop need to be run again with the new events enabled. It is not possible to monitor more than two events at any given time.

11.3.2 Gathering Data in Privileged and Non-Privileged Modes

The PCR has mode bits to enable the counters in privileged mode, non-privileged mode, or to count when in either mode. The mode setting affects both counters.

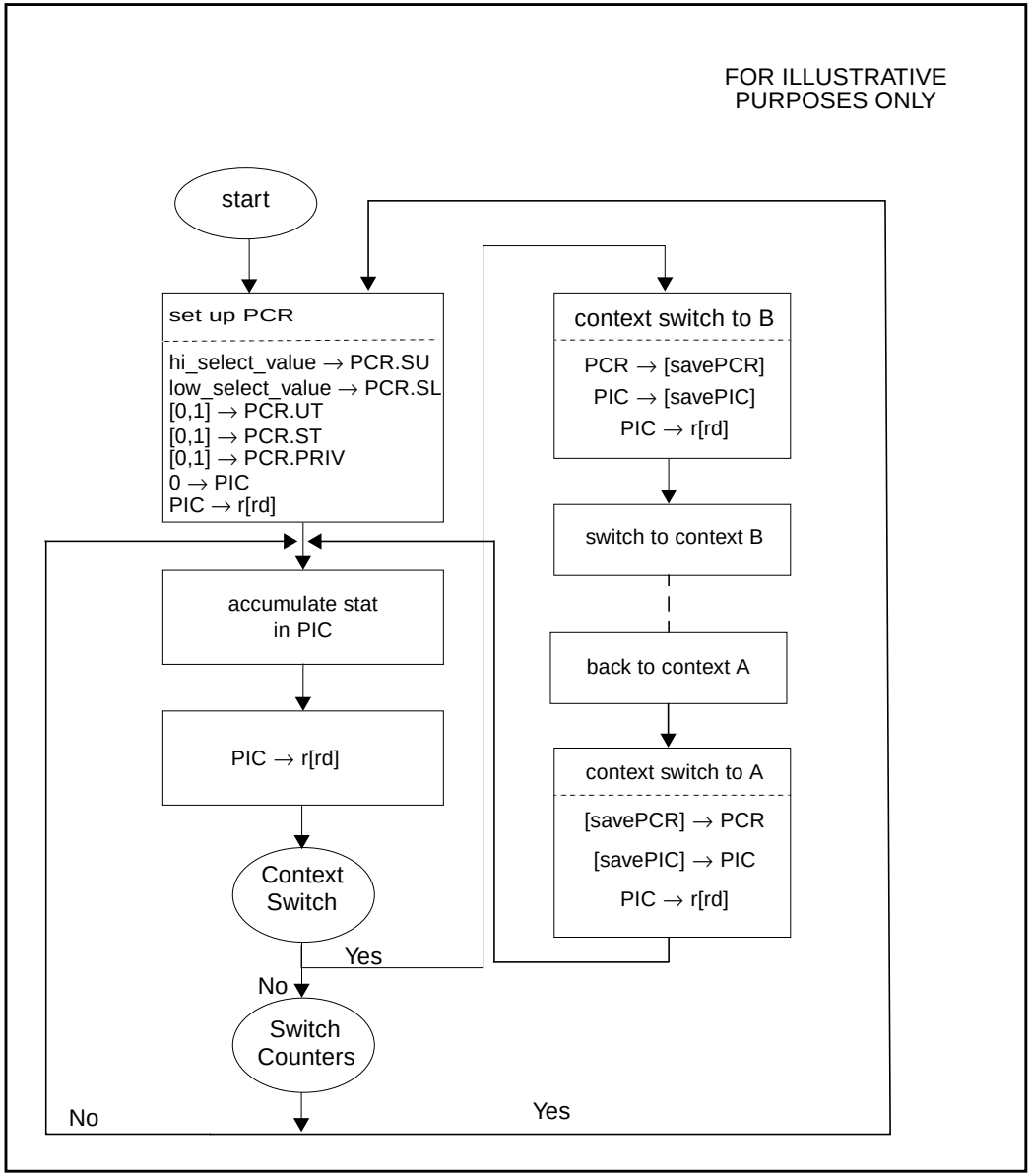


FIGURE 11-3 Operational Flow Diagram for Controlling Event Counters

11.3.3 Performance Instrumentation Implementations

Counting events and cycle stalls are sometimes complex because of the dynamic conditions and cancelled activities.

11.3.4 Performance Instrumentation Accuracy

The performance instrumentation counters are designed to provide reasonable accuracy especially when used to count hundreds or thousands of events or stall cycles or when comparing the PIC counts that have recorded a similar number of events or stall cycles. Accuracy is most challenging when trying to associate an event to an instruction and when comparing PIC counts with one count rarely occurring.

When using the overflow trap, it is sometimes difficult to pinpoint the instruction that is responsible for the overflow because of the way the pipeline is designed. A delay of several instructions is possible before the overflow is able to stop the current instruction flow and fetch the trap vector. This delay is referred to as skid and can occur for dozens of clock cycles. The skid for the load miss detection case is small. The skid value cannot be measured and its length depends on what event or stall cycle is being measured and what other instructions are in the pipeline.

11.4 Pipeline Counters

11.4.1 Instruction Execution and Processor Clock Counts

The instruction execution count monitors are described in TABLE 11-4 for clock and instruction execution counts.

TABLE 11-4 Instruction Execution Clock Cycles and Counts

Counter	Description
`Cycle_cnt`	`[PICL 00.0000 and PICU 00.0000]` Counts clock cycles. This counter increments the same as the SPARC-V9 `TICK` register, except that cycle counting is controlled by the `PCR.UT` and `PCR.ST` fields.
`Instr_cnt`	`[PICL 00.0001 and PICU 00.0001]` Counts the number of instructions completed. Annulled, mispredicted, or trapped instructions are not counted.

Synthesized Clocks Per Instruction (CPI)

The cycle and instruction counts can be used to calculate the average number of instructions completed per cycle: Clock cycles per instruction, `CPI` = `Cycle_cnt` / `Instr_cnt`.

11.4.2 IIU Event Counts

The counters listed in TABLE 11-5 record branch prediction event counts for taken and untaken branches in the Instruction Issue Unit (IIU). A retired branch in the following descriptions refers to a branch that reaches the D-stage without being invalidated.

TABLE 11-5 Counters for Collecting IIU Statistics

Counter	Description
`IU_Stat_Br_miss_taken`	`[PICL 01.0101]` Counts retired branches that were predicted to be taken, but in fact were not taken.
`IU_Stat_Br_miss_untaken`	`[PICU 01.1101]` Counts retired branches that were predicted to be untaken, but in fact were taken.
`IU_Stat_Br_Count_taken`	`[PICL 01.0110]` Counts retired taken branches.
`IU_Stat_Br_Count_untaken`	`[PICU 01.1110]` Counts retired untaken branches.

11.4.3 IIU Dispatch Stall Counts

IIU stall counts, listed in TABLE 11-6 on page 235, are the major cause of pipeline stalls (bubbles) from the instruction fetch and decode pipeline. Stalls are counted for each clock cycle at which the associated condition is true.

FIGURE 11-4 illustrates the first two considerations described in Section 11.4.3.1.

11.4.3.1 Dispatch Counter Considerations

1. Dispatch Counters count when the buffer is empty, regardless of whether the execution pipeline can accept more instructions from the instruction queue.
2. It is difficult to associate an empty queue. Various reasons taken together or separately can cause the instruction queue to be empty. The hardware picks the most recent disruptive event that is in the Fetch Unit to choose a counter to assign the empty queue cycles.

3. Count accuracy is also subject to the conditions described for all counters in the Section 11.3.4 “Performance Instrumentation Accuracy” on page 233.”

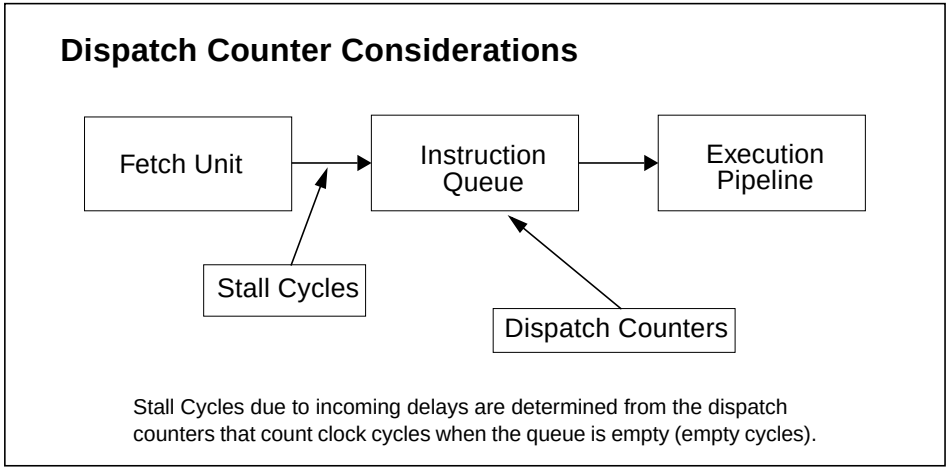


FIGURE 11-4 Dispatch Counters

TABLE 11-6 Counters for IIU Stalls

Counter	Description[1]
`Dispatch0_IC_miss`	`[PICL 00.0010]` Counts the stall cycles due to the event that no instructions are issued because I-queue is empty from instruction cache miss. This count includes L2-cache miss processing if a L2-cache miss also occurs.
`Dispatch0_mispred`	`[PICU 00.0010]` Counts the stall cycles due to the event that no instructions are issued because I-queue is empty due to branch misprediction.
`Dispatch0_br_target`	`[PICL 00.0011]` Counts the stall cycles due to the event that no instructions are issued because I-queue is empty due to a branch target address calculation.
`Dispatch0_2nd_br`	`[PICL 00.0100]` Counts the stall cycles due to the event of having two branch instructions line-up in one 4-instruction group causing the second branch in the group to be refetched, delaying its entrance into the I-queue.
`Dispatch_rs_mispred`	`[PICL 01.0111]` Counts the stall cycles due to the event that no instructions are issued because the I-queue is empty due to a Return Address Stack misprediction.

1. See Section 11.4.3.1 “Dispatch Counter Considerations” on page 234 for important information.

11.4.4 R-stage Stall Counts

Stalls are caused by dependency checks (data not ready for use by the instruction ready for dispatch) and by resources not being available (out-of-pipeline execution units needed, but are in-use).

The counters in TABLE 11-7 count the stall cycles at the R-stage of the pipeline. Stalls are counted for each clock at which the associated condition is true.

TABLE 11-7 Counters for R-stage Stalls

Counter	Description
`Rstall_storeQ`	`[PICL 00.0101]` Counts R-stage stall cycles for a store instruction which is the next instruction to be executed, but is stalled due to the store queue being full, that is, cannot hold additional stores. Up to eight entries can be in the store queue.
`Rstall_FP_use`	`[PICU 00.1011]` Counts R-stage stall cycles due to the event that the next instruction to be executed depends on the result of a preceding floating-point instruction in the pipeline that is not yet available.
`Rstall_IU_use`	`[PICL 00.0110]` Counts R-stage stall cycles due to the event that the next instruction to be executed depends on the result of a preceding integer instruction in the pipeline that is not yet available.

11.4.5 Recirculation Stall Counts

Recirculation instrumentation is implemented through the counters listed in TABLE 11-8.

TABLE 11-8 Counters for Recirculation

Counter	Description
`Re_DC_missovhd`[1]	`[PICU 00.0100]` Counts the stall cycles from when a D-cache load misses (causes a recirculation), but L2-cache hit/miss has not been reported. Counts portion/overhead of stall cycles due to D-cache load miss from the point the load reaches D-stage (about to be recirculated) to the point L2-cache hit/miss for the load is reported.
`Re_endian_miss`	`[NA]` Event counter does not exist in the UltraSPARC IIIi processor.
`Re_RAW_miss`	`[PICU 10.0110]` Counts stall cycles due to recirculation when there is a load in the E-stage which has a non-bypassable read-after-write (RAW) hazard with an earlier store instruction. This condition means that load data are being delayed by completion of an earlier store. See the Section 8.12 “Read After Write (RAW) Bypassing” on page 197” for a description of the RAW hazard and causes of recirculation.
`Re_FPU_bypass`	`[PICU 00.0101]` Counts stall cycles due to recirculation when a FPU bypass condition that does not have a direct bypass path occurs.
`Re_DC_miss`	`[PICU 00.0110]` Counts stall cycles due to loads that miss D-cache and L2-cache and get recirculated. Includes cacheable loads only.

TABLE 11-8 Counters for Recirculation *(Continued)*

Counter	Description
Re_EC_miss	[PICU 00.0111] Counts stall cycles due to loads that miss D-cache and L2-cache and get recirculated. Stall cycles from the point when L2-cache miss is detected to the D-stage of the recirculated flow are counted. Includes cacheable loads only.
Re_PC_miss	[PICU 01.0000] Counts stall cycles due to recirculation when a P-cache miss occurs on a prefetch predicted second load.

1. See Section 11.5.6 “Separating D-cache Stall Cycle Counts” on page 240.

11.5 Cache Access Counters

Instruction cache, data cache, prefetch cache, write cache, and L2-cache access events can be collected through the counters listed in TABLE 11-9. Counts are updated by each cache access, regardless of whether the access will be used.

11.5.1 Instruction Cache Events

TABLE 11-9 Counters for Instruction Cache Events

Counter	Description
IC_ref	[PICL 00.1000] Counts I-cache references. I-cache references are fetches (up to four instructions) from an aligned block of eight instructions. I-cache references are generally speculative and include instructions that are later cancelled due to mis-speculation.
IC_miss	[PICU 00.1000] Counts I-cache misses. Includes fetches from mis-speculated execution paths which are later cancelled.
IC_miss_cancelled	[PICU 00.0011] Counts I-cache misses cancelled due to mis-speculation, recycle, or other events.
ITLB_miss	[PICU 01.0001] Counts I-TLB miss traps taken.

11.5.2 Data Cache Events

TABLE 11-10 describes the counters for D-cache events.

TABLE 11-10 Counters for Data Cache Events

Counter	Description
DC_rd	[PICL 00.1001] Counts D-cache read references (including accesses that subsequently trap). References to pages that are not virtually cacheable (TTE CV bit = 0) are not counted.
DC_rd_miss	[PICU 00.1001] Counts recirculated loads that miss the D-cache. Includes cacheable loads only.
DC_wr	[PICL 00.1010] Counts D-cache cacheable store accesses encountered (including cacheable stores that subsequently trap). Non-cacheable accesses are not counted.
DC_wr_miss	[PICU 00.1010] Counts D-cache cacheable store accesses that miss D-cache. (There is no stall or recirculation on store miss.)
DTLB_miss	[PICU 01.0010] Counts memory reference instructions which trap due to a D-TLB miss.

11.5.3 Write Cache Events

TABLE 11-11 describes the counters for W-cache events.

TABLE 11-11 Counters for Write Cache Events

Counter	Description
WC_miss	[PICU 01.0011] Counts W-cache misses.
WC_snoop_cb	[PICU 01.0100] Counts W-cache copybacks generated by a snoop from a remote processor.
WC_scrubbed	[PICU 01.0101] Counts W-cache hits to clean lines.
WC_wb_wo_read	[PICU 01.0110] Counts W-cache writebacks not requiring a read.

11.5.4 Prefetch Cache Events

TABLE 11-12 describes the counters for P-cache events.

TABLE 11-12 Counters for Prefetch Cache Events

Counter	Description
PC_MS_miss	[PICU 01.1111] Counts FP loads through the MS pipeline that miss P-cache.
PC_soft_hit	[PICU 01.1000] Counts FP loads that hit a P-cache line that was prefetched by a software-prefetch instruction.
PC_hard_hit	[PICU 01.1010] Counts FP loads that hit a P-cache line that was prefetched by a hardware prefetch.
PC_snoop_inv	[PICU 01.1001] Counts P-cache invalidates generated by a snoop from a remote processor and stores by a local processor.
PC_port0_rd	[PICL 01.0000] Counts P-cache cacheable FP loads to the first port (general-purpose load path to D-cache and P-cache via MS pipeline).
PC_port1_rd	[PICU 01.1011] Counts P-cache cacheable FP loads to the second port (memory and out-of-pipeline instruction execution loads via the A0 and A1 pipelines).

11.5.5 L2-Cache Events

The L2-cache write hit count is determined by subtraction of the read hit and the instruction hit count from the total L2-cache hit count. The L2-cache write reference count is determined by subtraction of the D-cache read miss and I-cache misses from the total L2-cache references. Because of write caching, this is not the same as D-cache write misses. TABLE 11-13 describes the counter for L2-cache events.

Note – A block load or store access is counted as 8 references. For atomics, the read and write events are counted individually.

TABLE 11-13 Counters for L2-cache Events

Counter	Description
EC_ref	[PICL 00.1100] Counts L2-cache reference events. A 64-byte request is counted as one reference. Includes speculative D-cache load requests that turn out to be a D-cache hit. Count includes cacheable accesses only.
EC_misses	[PICU 00.1100] Counts L2-cache miss events sent to the System Interface Unit. Includes I-cache, D-cache, P-cache, W-cache exclusive (store), read stream (BLD), write stream (BST) requests that miss L2-cache. Count includes cacheable accesses only.

TABLE 11-13 Counters for L2-cache Events *(Continued)*

Counter	Description
EC_write_hit_RDO	[PICL 00.1101] Counts W-cache exclusive requests that hit L2-cache in S or O state and thus, do a read-to-own (RDO) bus transaction.
EC_wb	[PICU 00.1101] Counts dirty subblocks that produce writebacks due to L2-cache miss events.
EC_snoop_inv	[PICL 00.1110] Counts L2-cache invalidates generated from a snoop by a remote processor.
EC_snoop_cb	[PICU 00.1110] Counts L2-cache copybacks generated from a snoop by a remote processor.
EC_rd_miss	[PICL 00.1111] Counts L2-cache miss events (including atomics) from D-cache requests. Cacheable D-cache loads only.
EC_ic_miss	[PICU 00.1111] Counts L2-cache read misses from I-cache requests. The counter counts all I-cache misses including those for instructions from the mis-speculated execution path. Cacheable requests only.

11.5.6 Separating D-cache Stall Cycle Counts

The D-Cache stall cycle counts can be measured separately for L2-cache hits and misses by using the *Re_DC_missovhd* counter. The *Re_DC_missovhd* stall cycle counter is used with the recirculation and cache access events to separately calculate the D-cache loads that hit and miss the L2-cache. TABLE 11-14 describes the *Re_DC_missovhd* stall cycle counter processor compatibility.

TABLE 11-14 *Re_DC_missovhd* Stall Cycle Counter Processor Compatibility

Function	Description
Miss Overhead Cycle Monitor	The *Re_DC_missovhd* cycle stall counter is defined in TABLE 11-8 and in the equations below.

Synthesizing Individual Hit and Miss Stall Times

To explain the synthesis for L2-cache hit and miss stall times separately, consider the four stall regions A, B, C, and D shown in FIGURE 11-5 and the definitions and calculations that follow.

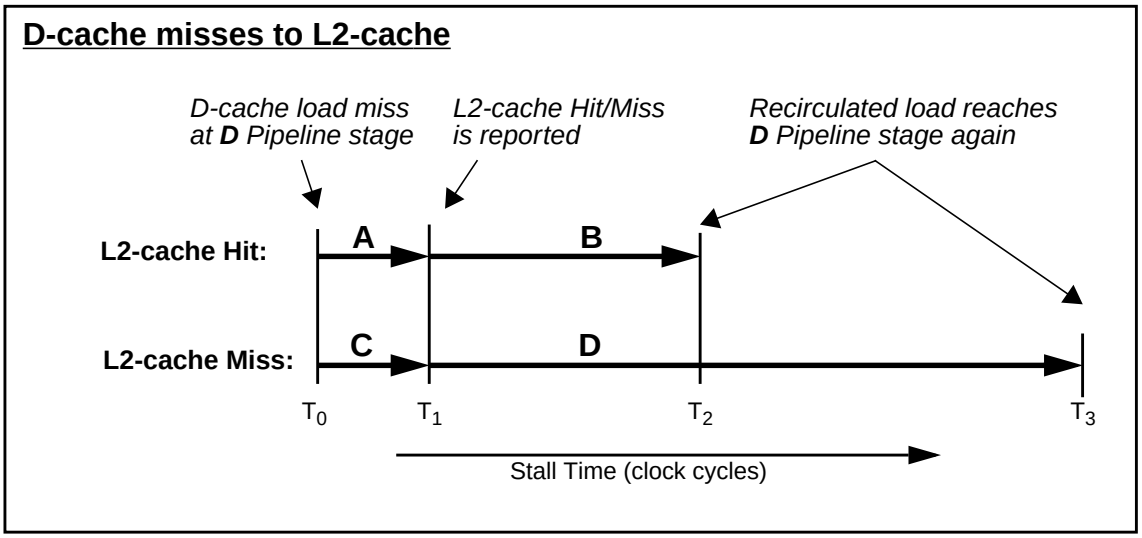


FIGURE 11-5 D-Cache Load Miss Stall Regions

Definitions:

Re_DC_missovhd (stall cycles) = (A + C) stall cycles

Re_EC_miss (stall cycles) = (D) stall cycles

Re_DC_miss (stall cycles) = (A + B + C + D) stall cycles

Fraction of D-cache misses that miss L2-cache $= \frac{\text{miss L2}}{\text{miss D-cache}} = \frac{\textit{EC\_rd\_miss}\ \text{(events)}}{\textit{DC\_rd\_miss}\ \text{(events)}} =$ Miss L2 Ratio

Synthesized Stall Cycle Counts:

(C) Stall Cycles = *Re_DC_missovhd* * Miss L2 Ratio

L2-cache Miss Stall Cycles = (C + D) = (*C)* + *Re_EC_miss*

L2-cache Hit Stall Cycles = (A + B) = *Re_DC_miss* - (C + D)

11.6 Memory Controller Counters

This section describes the memory controller counters in the UltraSPARC IIIi processor. Descriptions of counters for the UltraSPARC IIIi processor memory controller is shown in TABLE 11-15.

TABLE 11-15 Memory Controller Counters

Counter	Description
MC_read_dispatched	[PICL 10.0000] Counts the number of DDR 64-byte reads dispatched by the MIU.
MC_write_dispatched	[PICL 10.0001] Counts the number of DDR 64-byte writes dispatched by the MIU.
MC_read_returned_to_JBU	[PICL 10.0010] Counts the number of 64-byte reads that return data to JBU.
MC_msl_busy_stall	[PICL 10.0011] Counts the number of stall cycles due to msl_busy.
MC_mdb_overflow_stall	[PICL 10.0100] Counts the number of stall cycles due to potential memory data buffer overflow.
MC_miu_spec_request	[PICL 10.0101] Counts the number of speculative requests accepted by MIU.
MC_open_bank_cmds	[PICU 10.0000] Counts the number of open bank commands sent to the DDR SDRAM. With PTB enabled in MCU, this is PTB miss, no entry in PTB.
MC_reads	[PICU 10.0001] Counts the number of DDR 64-byte reads by the MSL.
MC_writes	[PICU 10.0010] Counts the number of DDR 64-byte writes by the MSL.
MC_page_close_stall	[PICU 10.0011] Counts the number of DDR page conflicts. When there is already a Page Tracking Buffer (PTB) entry, and a different page in the same bank needs to be opened, a page close is needed before opening a new page. Always zero when PTB is disabled.

11.7 Miscellaneous Counters

11.7.1 System Interface Events and Clock Cycles

System interface statistics are collected through the counters listed in TABLE 11-16.

TABLE 11-16 Counters for System Interface Statistics

Counter	Description
`SI_snoop`	`[PICL 01.0001]` Counts snoops from remote processor(s) including RDS, RDO.
`SI_ciq_flow`	`[PICL 01.0010]` Counts system clock cycles when the flow control (`DOK/AOK`) is asserted from this processor.
`SI_owned`	`[PICL 010011]` Counts the number of times J_PACK indicating OWNED is asserted on requests.

11.7.2 Software Events

Software statistics are collected through the counters listed in TABLE 11-17.

TABLE 11-17 Counters for Software Statistics

Counter	Description
`SW_count0`	`[PICL 01.0100]` Counts software-generated occurrences of `sethi %hi`(0xfc000), `%g0` instruction.
`SW_count1`	`[PICU 01.1100]` Counts software-generated occurrences of `sethi %hi`(0xfc000), `%g0` instruction.

Note – Both counters measure the same event; thus, the count can be programmed to be read from either the `PICL` or the `PICU` register.

11.7.3 Floating-Point Operation Events

Floating-point operation statistics are collected through the counters listed in TABLE 11-18.

TABLE 11-18 Counters for Floating-Point Operation Statistics

Event Counter	Description
FA_pipe_completion	[PICL 01.1000] Counts instructions that complete execution on the Floating-Point/Graphics ALU pipelines.
FM_pipe_completion	[PICU 10.0111] Counts instructions that complete execution on the Floating-Point/Graphics Multiply pipelines.

11.8 PCR.SL and PCR.SU Encodings

TABLE 11-19 lists PCR.SL and PCR.SL selection bit field encoding. Shaded blocks show SL and SU field duplications.

TABLE 11-19 PIC.SL and PIC.SU Selection Bit Field Encoding

PCR.SL and PCR.SU Encodings	PICL Event Selection	PICU Event Selection
00.0000	Cycle_cnt	Cycle_cnt
00.0001	Instr_cnt	Instr_cnt
00.0010	Dispatch0_IC_miss	Dispatch0_mispred
00.0011	Dispatch0_br_target	IC_miss_cancelled
00.0100	Dispatch0_2nd_br	Re_DC_missovhd
00.0101	Rstall_storeQ	Re_FPU_bypass
00.0110	Rstall_IU_use	Re_DC_miss
00.0111	*Reserved*	Re_EC_miss
00.1000	IC_ref	IC_miss
00.1001	DC_rd	DC_rd_miss
00.1010	DC_wr	DC_wr_miss
00.1011	*Reserved*	Rstall_FP_use
00.1100	EC_ref	EC_misses
00.1101	EC_write_hit_RDO	EC_wb
00.1110	EC_snoop_inv	EC_snoop_cb
00.1111	EC_rd_miss	EC_ic_miss
01.0000	PC_port0_rd	Re_PC_miss
01.0001	SI_snoop	ITLB_miss
01.0010	SI_ciq_flow	DTLB_miss

TABLE 11-19 PIC.SL and PIC.SU Selection Bit Field Encoding *(Continued)*

PCR.SL and PCR.SU Encodings	PICL Event Selection	PICU Event Selection
01.0011	SI_owned	WC_miss
01.0100	SW_count0	WC_snoop_cb
01.0101	IU_Stat_Br_miss_taken	WC_scrubbed
01.0110	IU_Stat_Br_count_taken	WC_wb_wo_read
01.0111	Dispatch_rs_mispred	*Reserved*
01.1000	FA_pipe_completion	PC_soft_hit
01.1001	*Reserved*	PC_snoop_inv
01.1010	*Reserved*	PC_hard_hit
01.1011	*Reserved*	PC_port1_rd
01.1100	*Reserved*	SW_count1
01.1101	*Reserved*	IU_Stat_Br_miss_untaken
01.1110	*Reserved*	IU_Stat_Br_count_untaken
01.1111	*Reserved*	PC_MS_miss
10.0000	MC_read_dispatched	MC_open_bank_cmds
10.0001	MC_write_dispatched	MC_reads
10.0010	MC_read_returned_to_JBU	MC_writes
10.0011	MC_msl_busy_stall	MC_page_close_stall
10.0100	MC_mdb_overflow_stall	*Reserved*
10.0101	MC_miu_spec_request	*Reserved*
10.0110	*Reserved*	Re_RAW_miss
10.0111	*Reserved*	FM_pipe_completion
10.1000	*Reserved*	*Reserved*
10.1001	*Reserved*	*Reserved*
10.1010 - 11.1111	*Reserved*	*Reserved*

SECTION VII

Special Topics

CHAPTER 12

Reset and RED_state

The UltraSPARC IIIi processor can be reset using various mechanisms. This section deals with the reset and RED_state for the UltraSPARC IIIi processor.

12.1 RED_state Characteristics

A processor enters RED_state by one of the two ways:

- Trapping when already at the maximum trap level
- Setting the PSTATE.RED

When the processor enters RED_state, it will clear the DCU Control Register, including enable bits for I-cache, D-cache, I-MMU, D-MMU, and virtual and physical watchpoints.

Note – Exiting RED_state by writing zero to PSTATE.RED in the delay slot of a JMPL is not recommended. A non-cacheable instruction prefetch can be made to the JMPL target, which may be in a cacheable memory area. This condition could result in a bus error on some systems and cause an *instruction_access_error* trap. The trap can be masked by setting the NCEEN bit in the ESTATE_ERR_EN register to zero, but this approach will mask all non-correctable error checking. Exiting RED_state with DONE or RETRY avoids the problem.

12.2 Resets

Reset priorities from highest to lowest are Power-On Reset (POR), System Reset, Externally Initiated Reset (XIR), Watchdog Reset (WDR), and Software-Initiated Reset (SIR).

12.2.1 Power-On Reset

A Power-On Reset (POR) occurs when the J_POR_L and J_RST_L pins are activated and stay asserted until the processor is within its specified operating range. During POR, all other resets and traps are ignored. POR has a trap type of 1 at physical address offset 0x20. Any pending external transactions are canceled.

After POR, software must initialize values of certain registers and state that is unknown after POR. The following bits must be initialized before the caches are enabled:

- In the I-cache, valid bits must be cleared and microtag bits must be set so that each way within a set has a unique microtag value.
- In the D-cache, valid bits must be cleared and microtag bits must be set so that each way within a set has a unique microtag value.
- All L2-cache tags and data.
- The I-MMU and D-MMU TLBs must also be initialized.
- The P-cache valid bits must be initialized before any floating-point loads are executed.

Caution – Executing a `DONE` or `RETRY` instruction when `TSTATE` is uninitialized after a POR can damage the chip. The POR boot code should initialize `TSTATE`<3:0>, using `wrpr` writes, before any `DONE` or `RETRY` instructions are executed.

However, these operations can only be executed in privileged mode. Therefore, user code is not at risk of damaging the chip.

12.2.2 System Reset

A System Reset occurs when the J_RST_L pin is activated without J_POR_L.When this pin is active, all other resets and traps are ignored. System Reset has a trap type of 1 at physical address offset 0x20. Any pending external transactions are cancelled.

After a system reset, software must initialize the following bits as unknown:

In particular,

- The valid and micro-tag bits in the Instruction Cache,
- The valid and micro-tag bits in the D-cache,
- All L2-cache tags and data must be cleared before enabling the caches.
- The I-MMU and D-MMU TLBs must also be initialized.

Memory refresh continues uninterrupted during a System Reset. System interface, L2-cache configuration, memory controller configuration are preserved across a System Reset.

The JBUS clock ratio is unaffected during this reset. Clock PLLs are reset during a Power-On Reset, but not during a System Reset unless the appropriate bit in the CSR is set before the System Reset.

There are bits in JIO that software can write to cause a System Reset, or Power-On Reset at any time. CSRs on the UltraSPARC IIIi processor that change clock ratios generally do not take effect until a System Reset.

12.2.3 Externally Initiated Reset (XIR)

An Externally Initiated Reset (XIR) is sent to all processors through the XIR transaction on the JBUS. It causes an XIR defined in SPARC-V9, which has a trap type 0x3 at physical address offset 0x60. It has higher priority than all other resets except Power-On Reset and System Reset.

This reset (actually a trap) only affects the processors, rather than the entire system. Memory state, cache state and most CSR states remain unchanged.

The saved PC and nPC will only be approximations since the trap is not precise with respect to pipeline state.

Reset due to XIR for the UltraSPARC IIIi processor initiates fetch of instruction code from Boot PROM, and the memory controller continues to perform refresh cycles in order to preserve main memory contents.

12.2.4 Watchdog Reset (WDR) and error_state

The processor enters `error_state` when a trap occurs at `TL` = `MAXTL`.

The processor automatically exits `error_state` using WDR. The processor signals itself internally to take a WDR and sets `TT` = 2. The WDR traps to the address at RSTVaddr + $0x40_{16}$. WDR sets the processor in a state where it is prepared for diagnosis of failures.

WDR affects only one processor, rather than the entire system. `CWP` updates due to window traps that cause watchdog traps are the same as the no watchdog trap case.

12.2.5 Software-Initiated Reset (SIR)

A Software-Initiated Reset (SIR) is initiated by an `SIR` instruction within any processor. This per-processor reset has a trap type 4 at physical address offset 0x80. SIR affects only one processor, rather than the entire system.

12.3 RED_state Trap Vector

When a SPARC-V9 processor processes a reset or trap that enters RED_state, it takes a trap at an offset relative to the *RED_state_trap_vector* base address (`RSTVaddr`). The trap offset depends on the type of RED mode trap and takes the values:

- POR 0x20
- WDR 0x40
- XIR 0x60
- SIR 0x80
- Other 0xA0

In the UltraSPARC IIIi processor, the following is the RSTV base address:

- **Virtual Address:** 0xFFFF FFFF F000 0000
- **Physical Address, PA[42:0]:** 0x7FF F000 0000

The UltraSPARC IIIi processor has a RMTV pin to select a second RSTV to allow use of PC compatible SuperIO chips on a PCI bus. The following is the second RSTV base address:

- **Virtual Address:** 0xFFFF FFFF FFFF 0000
- **Physical Address, PA[42:0]:** 0x7FF FFFF 0000

12.4 Initialization and Use of the Return Address Stack

The need to initialize the various L1-cache and L2-cache states, and MMU states, is well understood, but in the past the need to initialize other caching devices has been overlooked. The Return Address Stack (RAS) is one such device. While it is initialized to zero when RED mode is entered, zeroes may not be an appropriate PA or VA.

Failure Scenario

With the I-MMU off, the RAS can be used to generate a predicated physical address for prefetch. However, the RAS may have a virtual address in it, from execution while the I-MMU was enabled. This virtual address is used as is for instruction prefetch and may cause side-effects at whatever destination it indicates, or other errors.

The UltraSPARC IIIi processor uses the RAS for prediction for CALL, RETURN, DONE, and RETRY. The UltraSPARC IIIi processor considers RETURN to be a JMPL with an %rs1 equal to %o7 (normal subroutine) or %i7 (leaf subroutine).

There are possibly other cases that use RAS for prefetch. For instance, immediately after writing to the LSU control register to enable the I-MMU.

The issue also exists whenever software turns off the I-MMU after executing for a while with the I-MMU enabled. This should only happen due to traps to RED mode, for normal software. There is no problem for the transition of I-MMU off to on, because I-MMU will block the prefetch address if it is an I-MMU miss, and it will get flushed away when the prediction is determined to be wrong.

Software Rules

After any reset, trap to RED mode, or transition of the I-MMU from on to off, the 8-level RAS should be initialized with eight CALL instructions to a valid non-cacheable address before PSTATE.RED turns off. If the I-MMU is enabled before PSTATE.RED turns off, there may be no issue to worry about, if VA == 0x0 is unmapped, the prefetch will be disabled.

The output of the RAS is forced to the Red Mode Trap Vector (RMTV) while PSTATE.RED == 1. However, the RAS is initialized to zeroes, so when PSTATE.RED turns off, the zeroes are used for prediction, and may not be valid addresses (cacheable or non-cacheable).

12.5 Machine States

TABLE 12-1 shows the machine state created as a result of any reset, or after entering RED_state.

TABLE 12-1 Machine State After Reset and in `RED_state` *(1 of 5)*

Name	Fields	Power-On Reset	System Reset	WDR	XIR	SIR	RED_state‡
Integer Registers		Unknown	Unchanged	Unchanged			
Floating-Point Registers		Unknown	Unchanged	Unchanged			
L2-Cache Control Register	EC_MOSI EC_Pwr_Up EC_Act_Way++ EC_Block EC_size++ EC_par_En EC_ECC_en EC_ECC_force EC_check	1 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0	Unchanged			
RSTV Value		If processor pin rmtv = 0 VA=0xffff ffff f000 0000, PA=0x7ff f000 0000 else VA=0xffff ffff ffff 0000, PA = 0x7ff ffff 0000.					
PC nPC		RSTV | 0x20 RSTV | 0x24	RSTV | 0x20 RSTV | 0x24	RSTV | 0x40 RSTV | 0x44	RSTV | 0x60 RSTV | 0x64	RSTV | 0x80 RSTV | 0x84	RSTV | 0xa0 RSTV | 0xa4
PSTATE	MM RED PEF AM PRIV IE AG CLE TLE IG MG	0 (TSO) 1(RED_state) 1 (FPU on) 0 (Full 64-bit address 1 (Privileged mode) 0 (Disable interrupts) 1 (Alternate globals selected) 0 (Current little-endian) 0 (Trap little-endian) 0 (Interrupt globals not selected) 0 (MMU globals not selected)	0 (TSO) 1(RED_state) 1 (FPU on) 0 (Full 64-bit address 1 (Privileged mode) 0 (Disable interrupts) 1 (Alternate globals selected) 0 (current little-endian) 0 (trap little-endian) 0 (Interrupt globals not selected) 0 (MMU globals not selected)	0 (TSO) 1(RED_state) 1 (FPU on) 0 (Full 64-bit address) 1 (Privileged mode) 0 (Disable interrupts) 1 (Alternate globals selected) PSTATE.TLE Unchanged 0 (Interrupt globals not selected) 0 (MMU globals not selected)			
TBA<63:15>		Unknown	Unchanged	Unchanged			
Y		Unknown	Unchanged	Unchanged			
PIL		Unknown	Unchanged	Unchanged			

TABLE 12-1 Machine State After Reset and in `RED_state` *(2 of 5)*

Name	Fields	Power-On Reset	System Reset	WDR	XIR	SIR	RED_state‡
CWP		Unknown	Unchanged	Unchanged except for register window traps			
TT[TL]		1	1	Unchanged	3	4	Trap type
CCR		Unknown	Unchanged	Unchanged			
ASI		Unknown	Unchanged	Unchanged			
TL		MAXTL	MAXTL	**min**(TL+1, MAXTL)			
TPC[TL] TNPC[TL]		Unknown Unknown	Unchanged Unchanged	PC nPC	PC & ~0x1f nPC=PC+4	PC nPC	
TSTATE	CCR ASI PSTATE CWP PC nPC	Unknown Unknown Unknown Unknown Unknown Unknown	Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged	CCR ASI PSTATE CWP PC nPC			
TICK	NPT counter	1 Restart at 0	1 Restart at 0	Unchanged Count	Unchanged Restart at 0	Unchanged Count	
CANSAVE		Unknown	Unchanged	Unchanged			
CANRESTORE		Unknown	Unchanged	Unchanged			
OTHERWIN		Unknown	Unchanged	Unchanged			
CLEANWIN		Unknown	Unchanged	Unchanged			
WSTATE	OTHER NORMAL	Unknown Unknown	Unchanged Unchanged	Unchanged Unchanged			
VER	MANUF IMPL MASK MAXTL MAXWIN	0x003E 0x0016 mask dependent 5 7					
FSR	All	0	0	Unchanged			
FPRS	All	Unknown	Unchanged	Unchanged			
Non-SPARC-V9 ASRs							
SOFTINT		Unknown	Unchanged	Unchanged			
TICK_COMPARE	INT_DIS TICK_CMPR	1 (off) 0	1 (off) 0	Unchanged Unchanged			
STICK	NPT counter	1 0	1 0	Unchanged Count			
STICK_COMPARE	INT_DIS TICK_CMPR	1 (off) 0	1 (off) 0	Unchanged Unchanged			

TABLE 12-1 Machine State After Reset and in `RED_state` *(3 of 5)*

Name	Fields	Power-On Reset	System Reset	WDR	XIR	SIR	RED_state‡
PERF_CONTROL	S1 S0 UT (trace user) ST (trace system) PRIV (priv access)	Unknown Unknown Unknown Unknown Unknown	Unchanged Unchanged Unchanged Unchanged Unchanged	Unchanged Unchanged Unchanged Unchanged Unchanged			
PERF_COUNTER	All	Unknown	Unknown	Unknown			
GSR	IM Others	0 Unknown	0 Unchanged	Unchanged Unchanged			
DISPATCH_CONTROL	MS SI RPE BPE OBS IFPOE	0 0 0 0 0 0	0 0 0 0 0 0	Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged			
Non-SPARC-V9 ASIs							
DCU_CONTROL	WE All others	0(off) 0 (off)	0(off) 0 (off)	Unchanged 0 (off)			
INST_BREAKPOINT	All	0 (off)	0 (off)	Unchanged			
VA_WATCHPOINT		Unknown	Unchanged	Unchanged			
PA_WATCHPOINT		Unknown	Unchanged	Unchanged			
I-& DMMU_SFSR,	ASI FT E CTXT PRIV W OW (overwrite) FV (SFSR valid) NF TM	Unknown Unknown Unknown Unknown Unknown Unknown Unknown 0 Unknown Unknown	Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged 0 Unchanged Unchanged	Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged			
DMMU_SFAR		Unknown	Unchanged	Unchanged			
INTR_DISPATCH	All	0	0	Unchanged			
INTR_RECEIVE	BUSY SOURCE	0 Unknown	0 Unchanged	Unchanged Unchanged			
ESTATE_ERR_EN	All	0 (All off)	0 (All off)	Unchanged			
AFAR	PA	Unknown	Unchanged	Unchanged			
AFSR	All	0	Unchanged	Unchanged			

TABLE 12-1 Machine State After Reset and in `RED_state` *(4 of 5)*

Name	Fields	Power-On Reset	System Reset	WDR	XIR	SIR	RED_state‡
MCU_CTL_REG1	Clk_Update Clk_Stop 30 Remaining bits	Unknown Unknown Unknown 0	0 0 0 Unchanged	Unchanged Unchanged Unchanged Unchanged			
MCU_CTL_REG2	CLK PLL2_M1 PLL2_M2 Remaining bits	2 2 3 0	effect propagated effect propagated effect propagated Unchanged	Unchanged Unchanged Unchanged Unchanged			
MCU_CTL_REG3	All	Unknown	Unchanged	Unchanged			
JBUS_CONFIG	PAR_DLY PORT_LOCN PORT_PRES DBG2 DTL MID MR MT AID{[4:3],[2:0]} SW_JERR E*_CLK SRT TOF TOV DBG1 CLK ARB_MODE	0 0x7f J_PACK6-0<2:0> 0xf {DOWN_25, UP_OPEN} 0x3e 0 0 {00,J_ID<2:0>} 0 0 0 0 0 0x7 0 0	effect propagated effect propagated unchanged effect propagated unchanged unchanged unchanged unchanged effect propagated 0 unchanged effect propagated effect propagated effect propagated effect propagated effect propagated effect propagated	Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged			
JP_IMP_CTL0	All	Varies	Varies	Varies			
JP_IMP_CTL1	All	0	Unchanged	Unchanged			

TABLE 12-1 Machine State After Reset and in `RED_state` *(5 of 5)*

Name	Fields	Power-On Reset	System Reset	WDR	XIR	SIR	RED_state‡
JP_IMP_CTL2	[63:8] [7:0]	0 0	0 Unchanged	Unchanged			
Other Processor-Specific States							
Processor L2-Cache Tags, Micro-tags and Data (Includes Data, Instruction, Prefetch, and Write Caches)		Unknown	Unknown	Unchanged			
Cache Snooping		Enabled					
Instruction Queue		Empty					
Store Queue		Empty	Empty	Unchanged			
I-TLB, D-TLB	Mappings, Valid, Lock, E-bit, NC-bit, Global bit, etc.	Unknown	Unknown	Unchanged			

*This register is read-only from the system.

‡ Processor states are only updated according to the following table if RED_state is entered due to a reset or a trap. If RED_state is entered because the PSTATE.RED bit was explicitly set to 1, then software must create the appropriate states itself.

++ These bits will read as 0 after POR or System Reset, but subsequent to the first write to this register, will read as 1.

Effect propagated: Some CSRs have delayed effects after writes by software. The readable CSR is updated by the software write, and on the next reset, the contents of a shadow register is updated from the CSR, which affects chip behavior from then on. Until the update happens, the shadow register has the old state. If the reset event never happens, it will never have an effect. A Hard POR initializes the shadow register to the same state as the readable CSR.

SECTION VIII

Appendix

CHAPTER A

Instruction Definitions

Related instructions are grouped into subsections. Each subsection consists of the following parts:

1. A table of the opcodes defined in the subsection with the values of the field(s) that uniquely identify the instruction(s).
2. An illustration of the applicable instruction format(s). In these illustrations, a dash (—) indicates that the field is *reserved* for future versions of the architecture and shall be zero in any instance of the instruction. If the processor encounters nonzero values in these fields, its behavior is undefined.
3. A description of the features, restrictions, and exception-causing conditions.
4. A list of exceptions that can occur as a consequence of attempting to execute the instruction(s). Exceptions due to an *instruction_access_error, instruction_access_exception, fast_instruction_access_MMU_miss, fast_ECC_error, ECC_error (corrected ECC_error)*, *WDR*, and interrupts are not listed because they can occur on any instruction. Instructions not implemented in hardware shall generate an *illegal_instruction* exception and therefore will not generate any of the other exceptions listed. The *illegal_instruction* exception is not listed because it can occur on any instruction that triggers an instruction breakpoint or contains an invalid field.

Instruction latencies and execution rates are provided in Chapter 4 “Instruction Execution.”

TABLE A-2 summarizes the instruction set; the instruction definitions follow the table. Within TABLE A-2 and throughout this chapter, certain opcodes are marked with mnemonic superscripts. The superscripts and their meanings are defined in TABLE A-1.

TABLE A-1 Opcode Superscripts

Superscript	Meaning
D	Deprecated instruction
P	Privileged opcode
P_{ASI}	Privileged action if bit 7 of the referenced ASI is zero
P_{ASR}	Privileged opcode if the referenced ASR register is privileged
P_{NPT}	Privileged action if `PSTATE.PRIV` = 0 and (S)`TICK.NPT` = 1
P_{PIC}	Privileged action if `PCR.PRIV` = 1

TABLE A-2 Instruction Set *(1 of 6)*

Operation	Name	Page	V9 extension formats
ADD, ADDcc	Add (and modify condition codes)	268	
ADDC, ADDCcc	Add with carry (and modify condition codes)	268	
ALIGNADDRESS{_LITTLE}	Calculate address for misaligned data	269	3
AND, ANDcc	And (and modify condition codes)	335	
ANDN, ANDNcc	And not (and modify condition codes)	335	
ARRAY(8,16,32)	Three-Dimensional array addressing instructions	271	3
BPcc	Branch on integer condition codes with prediction	288	
Bicc^{D}	Branch on integer condition codes	425	
BMASK	Set the `GSR.MASK` field	282	3
BPr	Branch on contents of integer register with prediction (also known as BRr)	283	
BSHUFFLE	Permute bytes as specified by `GSR.MASK`	282	3
CALL	Call and link	290	
CASA$^{P_{ASI}}$	Compare and swap word in alternate space	291	
CASXA$^{P_{ASI}}$	Compare and swap doubleword in alternate space	291	
DONE^{P}	Return from trap	294	
EDGE(8,16,32){,L,N,LN}	Edge handling instructions	295	3
FABS(s,d,q)	Floating-point absolute value	308	
FADD(s,d,q)	Floating-point add	298	
FALIGNDATA	Perform data alignment for misaligned data	269	3
FAND{S}	Logical AND operation	332	3
FANDNOT(1,2){S}	Logical AND operation with one inverted source	332	3
FBfcc^{D}	Branch on floating-point condition codes	423	

TABLE A-2 Instruction Set *(2 of 6)*

Operation	Name	Page	V9 extension formats
FBPfcc	Branch on floating-point condition codes with prediction	285	
FCMP(s,d,q)	Floating-point compare	300	
FCMPE(s,d,q)	Floating-point compare (exception if unordered)	300	
FCMP(GT,LE,NE,EQ)(16,32)	Pixel compare operations	369	3
FDIV(s,d,q)	Floating-point divide	310	
FdMULq	Floating-point multiply double to quad	310	
FEXPAND	Pixel expansion	377	3
FiTO(s,d,q)	Convert integer to floating-point	306	
FLUSH	Flush instruction memory	313	
FLUSHW	Flush register windows	315	
FMOV(s,d,q)	Floating-point move	308	
FMOV(s,d,q)cc	Move floating-point register if condition is satisfied	343	
FMOV(s,d,q)r	Move floating-point register if integer register contents satisfy condition	349	
FMUL(s,d,q)	Floating-point multiply	310	
FMUL8x16	8x16 partitioned product	364	3
FMUL8x16(AU,AL)	8x16 upper/lower α partitioned product	365	3
FMUL8(SU,UL)x16	8x16 upper/lower partitioned product	366	3
FMULD8(SU,UL)x16	8x16 upper/lower partitioned product	367	3
FNAND{S}	Logical NAND operation	332	3
FNEG(s,d,q)	Floating-point negate	308	
FNOR{S}	Logical NOR operation	332	3
FNOT(1,2){S}	Copy negated source	332	3
FONE{S}	One fill	332	3
FOR{S}	Logical OR operation	332	3
FORNOT(1,2){S}	Logical OR operation with one inverted source	332	3
FPACK(16,32, FIX)	Pixel packing	373, 375, 376	3
FPADD(16,32){S}	Pixel add (single) 16- or 32-bit	361	3
FPMERGE	Pixel merge	378	3
FPSUB(16,32){S}	Pixel subtract (single) 16- or 32-bit	361	3
FsMULd	Floating-point multiply single to double	310	
FSQRT(s,d,q)	Floating-point square root	312	
FSRC(1,2){S}	Copy source	332	3
F(s,d,q)TOi	Convert floating-point to integer	302	
F(s,d,q)TO(s,d,q)	Convert between floating-point formats	304	
F(s,d,q)TOx	Convert floating-point to 64-bit integer	302	

TABLE A-2 Instruction Set *(3 of 6)*

Operation	Name	Page	V9 extension formats
FSUB(s,d,q)	Floating-point subtract	298	
FXNOR{S}	Logical XNOR operation	332	3
FXOR{S}	Logical XOR operation	332	3
FxTO(s,d,q)	Convert 64-bit integer to floating-point	306	
FZERO{S}	Zero fill	332	3
ILLTRAP	Illegal instruction	316	
JMPL	Jump and link	317	
LDD^{D}	Load integer doubleword	433	
LDDA$^{D,\ P_{ASI}}$	Load integer doubleword from alternate space	434	
LDDA ASI_NUCLEUS_QUAD*	Atomic quad load	326	3
LDDF	Load double floating-point	318	
LDDFA$^{P_{ASI}}$	Load double floating-point from alternate space	274	
LDDFA ASI_BLK*	Block loads	274	3
LDDFA ASI_FL*	Short floating-point loads (VIS I)	400	3
LDF	Load floating-point	318	
LDFA$^{P_{ASI}}$	Load floating-point from alternate space	318	
LDFSR^{D}	Load floating-point state register lower	431	
LDQF	Load quad floating-point	318	
LDQFA$^{P_{ASI}}$	Load quad floating-point from alternate space	318	
LDSB	Load signed byte	322	
LDSBA$^{P_{ASI}}$	Load signed byte from alternate space	324	
LDSH	Load signed halfword	322	
LDSHA$^{P_{ASI}}$	Load signed halfword from alternate space	324	
LDSTUB	Load-store unsigned byte	329	
LDSTUBA$^{P_{ASI}}$	Load-store unsigned byte in alternate space	330	
LDSW	Load signed word	322	
LDSWA$^{P_{ASI}}$	Load signed word from alternate space	324	
LDUB	Load unsigned byte	322	
LDUBA$^{P_{ASI}}$	Load unsigned byte from alternate space	324	
LDUH	Load unsigned halfword	322	
LDUHA$^{P_{ASI}}$	Load unsigned halfword from alternate space	324	
LDUW	Load unsigned word	322	
LDUWA$^{P_{ASI}}$	Load unsigned word from alternate space	324	
LDX	Load extended	322	

TABLE A-2 Instruction Set *(4 of 6)*

Operation	Name	Page	V9 extension formats
LDXA$^{P_{ASI}}$	Load extended from alternate space	324	
LDXFSR	Load floating-point state register	318	
MEMBAR	Memory barrier	337	
MOVcc	Move integer register if condition is satisfied	343	
MOVr	Move integer register on contents of integer register	356	
MULScc^{D}	Multiply step (and modify condition codes)	436	
MULX	Multiply 64-bit integers	357	
NOP	No operation	358	
OR, ORcc	Inclusive OR (and modify condition codes)	335	
ORN, ORNcc	Inclusive OR not (and modify condition codes)	335	
PDIST	Pixel component distance	371	3
POPC	Population Count	378	
PREFETCH	Prefetch data	379	
PREFETCHA$^{P_{ASI}}$	Prefetch data from alternate space	379	
RDASI	Read ASI register	388	
RDASR$^{P_{ASR}}$	Read ancillary state register	388	
RDCCR	Read condition codes register	388	
RDDCR^{P}	Read dispatch control register	388	
RDFPRS	Read floating-point registers state register	388	
RDGSR	Read graphic status register	388	
RDPC	Read program counter	388	
RDPCR^{P}	Read performance control register	388	
RDPIC$^{P_{PIC}}$	Read performance instrumentation counters	388	
RDPR^{P}	Read privileged register	385	
RDSOFTINT^{P}	Read per-processor soft interrupt register	388	
RDSTICK$^{P_{NPT}}$	Read system TICK register	388	
RDSTICK_CMPR	Read system TICK compare register	388	
RDTICK$^{P_{NPT}}$	Read TICK register	388	
RDTICK_CMPR^{P}	Read TICK compare register	388	
RDY^{D}	Read Y register	440	
RESTORE	Restore caller's window	392	
RESTORED^{P}	Window has been restored	394	
RETRY^{P}	Return from trap and retry	294	
RETURN	Return	390	

TABLE A-2 Instruction Set *(5 of 6)*

Operation	Name	Page	V9 extension formats
SAVE	Save caller's window	392	
SAVED[P]	Window has been saved	394	
SDIV[D], SDIVcc[D]	32-bit signed integer divide (and modify condition codes)	428	
SDIVX	64-bit signed integer divide	357	
SETHI	Set high 22 bits of low word of integer register	397	
SHUTDOWN	Shut down the processor	402	3
SIAM	Set Interval Arithmetic Mode (VIS II)	395	
SIR	Software-initiated reset	403	
SLL	Shift left logical (IU)	398	
SLLX	Shift left logical, extended (IU)	398	
SMUL[D], SMULcc[D]	Signed integer multiply (and modify condition codes)	436	
SRA	Shift right arithmetic (IU)	398	
SRAX	Shift right arithmetic, extended (IU)	398	
SRL	Shift right logical (IU)	398	
SRLX	Shift right logical, extended (IU)	398	
STB	Store byte (IU)	408	
STBA[PASI]	Store byte into alternate space (IU)	409	
STBAR[D]	Store barrier	441	
STD[D]	Store doubleword	443	
STDA[D, PASI]	Store doubleword into alternate space	445	
STDF	Store double floating-point (FP)	404	
STDFA[PASI]	Store double floating-point into alternate space (FP)	406	
STDFA ASI_BLK*	Block stores	274	3
STDFA ASI_FL*	Short floating-point stores (VIS I)	400	3
STDFA ASI_PST*	Partial Store instructions	359	3
STF	Store floating-point (FP)	404	
STFA[PASI]	Store floating-point into alternate space (FP)	406	
STFSR[D]	Store floating-point state register (FP)	442	
STH	Store halfword (IU)	408	
STHA[PASI]	Store halfword into alternate space (IU)	409	
STQF	Store quad floating-point (FP)	404	
STQFA[PASI]	Store quad floating-point into alternate space (FP)	406	
STW	Store word (IU)	408	
STWA[PASI]	Store word into alternate space (IU)	409	
STX	Store extended (IU)	408	

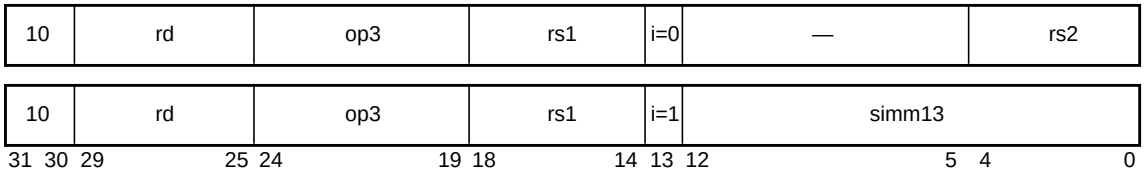
TABLE A-2 Instruction Set *(6 of 6)*

Operation	Name	Page	V9 extension formats
STXA$^{P_{ASI}}$	Store extended into alternate space (IU)	409	
STXFSR	Store extended floating-point state register (MS)	404	
SUB, SUBcc	Subtract (and modify condition codes)	411	
SUBC, SUBCcc	Subtract with carry (and modify condition codes)	411	
SWAP^{D}	Swap integer register with memory	446	
SWAPA$^{D, P_{ASI}}$	Swap integer register with memory in alternate space	448	
TADDcc, TADDccTV^{D}	Tagged add and modify condition codes (trap on overflow)	412, 449	
Tcc	Trap on integer condition codes	415	
TSUBcc, TSUBccTV^{D}	Tagged subtract and modify condition codes (trap on overflow)	413, 450	
UDIV^{D}, UDIVcc^{D}	Unsigned integer divide (and modify condition codes)	428	
UDIVX	64-bit unsigned integer divide	357	
UMUL^{D}, UMULcc^{D}	Unsigned integer multiply (and modify condition codes)	436	
WRASI	Write ASI register	420	
WRASR$^{P_{ASR}}$	Write ancillary state register	420	
WRCCR	Write condition codes register	420	
WRDCR^{P}	Write dispatch control register	420	
WRFPRS	Write floating-point registers state register	420	
WRGSR	Write graphic status register	420	
WRPCR^{P}	Write performance control register	420	
WRPIC$^{P_{PIC}}$	Write performance instrumentation counters register	420	
WRPR^{P}	Write privileged register	417	
WRSOFTINT^{P}	Write per-processor soft interrupt register	420	
WRSOFTINT_CLR^{P}	Clear bits of per-processor soft interrupt register	420	
WRSOFTINT_SET^{P}	Set bits of per-processor soft interrupt register	420	
WRTICK_CMPR^{P}	Write TICK compare register	420	
WRSTICK^{P}	Write System TICK register	420	
WRSTICK_CMPR^{P}	Write System TICK compare register	420	
WRY^{D}	Write Y register	452	
XNOR, XNORcc	Exclusive NOR (and modify condition codes)	335	
XOR, XORcc	Exclusive OR (and modify condition codes)	335	

A.1 Add

Opcode	Op3	Operation
ADD	00 0000	Add
ADDcc	01 0000	Add and modify condition codes
ADDC	00 1000	Add with Carry
ADDCcc	01 1000	Add with Carry and modify condition codes

Format (3)



Assembly Language Syntax	
add	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*
addcc	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*
addc	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*
addccc	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description

ADD and ADDcc compute "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1, and write the sum into r[rd].

ADDC and ADDCcc ("ADD with carry") also add 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 sum into 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.

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).

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

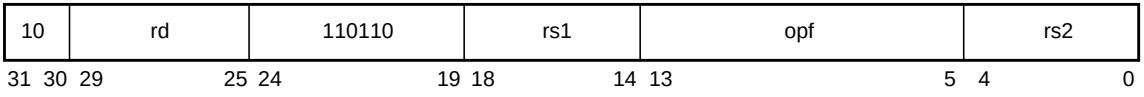
Exceptions

None

A.2 Alignment Instructions (VIS I)

Opcode	opf	Operation
ALIGNADDRESS	0 0001 1000	Calculate address for misaligned data access
ALIGNADDRESS_LITTLE	0 0001 1010	Calculate address for misaligned data access little-endian
FALIGNDATA	0 0100 1000	Perform data alignment for misaligned data

Format (3)



Assembly Language Syntax	
alignaddr	reg_{rs1}, reg_{rs2}, reg_{rd}
alignaddrl	reg_{rs1}, reg_{rs2}, reg_{rd}
faligndata	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

ALIGNADDRESS adds two integer values, r[rs1] and r[rs2], and stores the result (with the least significant three bits forced to zero in the integer register r[rd]. The least significant three 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 – ALIGNADDR_LITTLE generates the opposite-endian byte ordering for a subsequent FALIGNDATA operation.

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, specifying the number of the most significant byte to extract (therefore, the least significant byte extracted from the intermediate value is numbered GSR.align + 7).

A byte-aligned 64-bit load can be performed as shown in CODE EXAMPLE A-1.

CODE EXAMPLE A-1 Byte-Aligned 64-Bit Load

```
alignaddr  Address, Offset, Address

ldd        [Address], %f0

ldd        [Address + 8], %f2

faligndata %f0, %f2, %f4
```

Programming Note – For good performance, the result of FALIGNDATA should not be used as a source operand for a 32-bit FP or VIS instruction in the next three instruction groups.

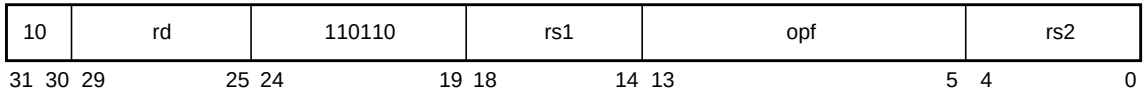
Exceptions

fp_disabled

A.3 Three-Dimensional Array Addressing Instructions (VIS I)

Opcode	opf	Operation
ARRAY8	0 0001 0000	Convert 8-bit 3D address to blocked byte address
ARRAY16	0 0001 0010	Convert 16-bit 3D address to blocked byte address
ARRAY32	0 0001 0100	Convert 32-bit 3D address to blocked byte address

Format (3)



Assembly Language Syntax	
`array8`	*reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$
`array16`	*reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$
`array32`	*reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$

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 L2-cache block reuse, and at the 64 KB 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 `rs2` and their meanings are shown in TABLE A-3. Illegal values produce undefined results in the destination register, `r[rd]`. FIGURE A-1 illustrates a three-dimensional array fixed-point address format.

TABLE A-3 Three-Dimensional `r[rs2]` Array X/Y Dimensions

r[rs2] value	Number of Elements
0	64
1	128
2	256
3	512
4	1024
5	2048

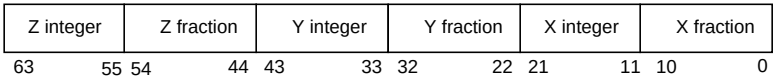


FIGURE A-1 Three-Dimensional Array Fixed-Point Address Format

The integer parts of X, Y, and Z are converted to the following blocked-address formats illustrated in FIGURE A-2, FIGURE A-3, and FIGURE A-4.

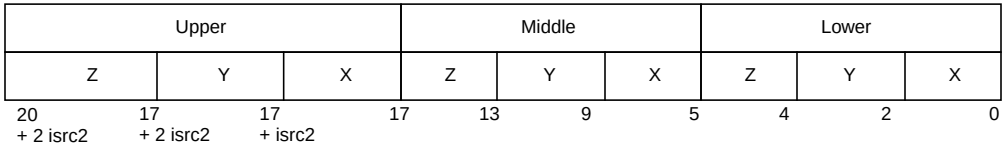


FIGURE A-2 Three-Dimensional Array Blocked-Address Format (`Array8`)

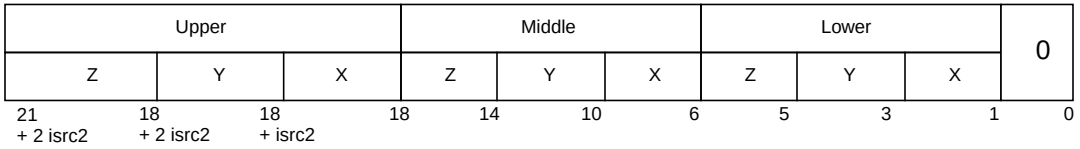


FIGURE A-3 Three-Dimensional Array Blocked-Address Format (`Array16`)

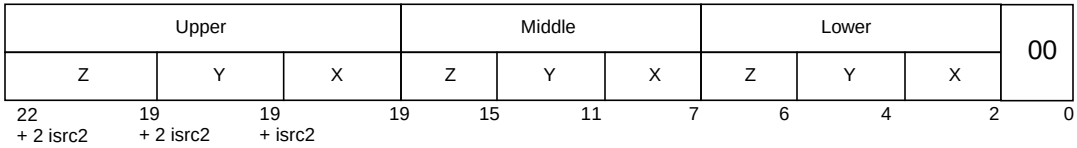


FIGURE A-4 Three-Dimensional Array Blocked-Address Format (`Array32`)

The bits above Z upper are set to zero. 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.

The code fragment in CODE EXAMPLE A-2 shows assembly of components along an interpolated line at the rate of one component per clock.

CODE EXAMPLE A-2 Three-Dimensional Array Addressing Example

```
    add         Addr, DeltaAddr, Addr

    array8      Addr, %g0, bAddr

    ldda        [bAddr] ASI_FL8_PRIMARY, data

    faligndata data, accum, accum
```

Note – To maximize reuse of L2-cache and TLB data, software should block array references of a large image to the 64 KB level. This means processing elements within a 32 x 64 x 64 block.

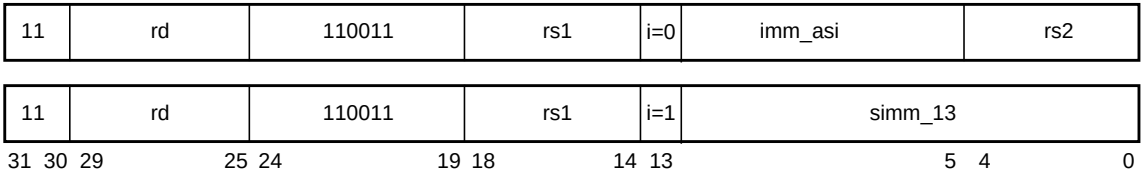
Exceptions

None

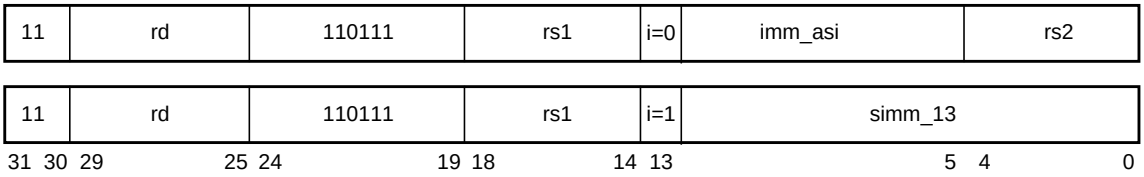
A.4 Block Load and Block Store (VIS I)

Opcode	imm_asi	ASI Value	Operation
LDDFA STDFA	ASI_BLK_AIUP	70_{16}	64-byte block load/store from/to primary address space, privilege mode access only
LDDFA STDFA	ASI_BLK_AIUS	71_{16}	64-byte block load/store from/to secondary address space, privilege mode access only
LDDFA STDFA	ASI_BLK_AIUPL	78_{16}	64-byte block load/store from/to primary address space, little-endian, privilege mode access only
LDDFA STDFA	ASI_BLK_AIUSL	79_{16}	64-byte block load/store from/to secondary address space, little-endian, privilege mode access only
LDDFA STDFA	ASI_BLK_P	$F0_{16}$	64-byte block load/store from/to primary address space
LDDFA STDFA	ASI_BLK_S	$F1_{16}$	64-byte block load/store from/to secondary address space
LDDFA STDFA	ASI_BLK_PL	$F8_{16}$	64-byte block load/store from/to primary address space, little-endian
LDDFA STDFA	ASI_BLK_SL	$F9_{16}$	64-byte block load/store from/to secondary address space, little-endian
STDFA	ASI_BLK_COMMIT_P	$E0_{16}$	64-byte block commit store to primary address space
STDFA	ASI_BLK_COMMIT_S	$E1_{16}$	64-byte block commit store to secondary address space

Format (3) ***LDDFA***



Format (3) STDFA



Assembly Language Syntax	
`ldda`	[*reg_addr*] *imm_asi*, $freg_{rd}$
`ldda`	[*reg_plus_imm*] `%asi`, $freg_{rd}$
`stda`	$freg_{rd}$, [*reg_addr*] *imm_asi*
`stda`	$freg_{rd}$, [*reg_plus_imm*] `%asi`

Description

A block load (BLD) or block store (BST) instruction uses an LDDFA or STDFA instruction combined with a block transfer ASI. Block transfer ASIs allow BLDs and BSTs to be performed accessing the same address space as normal loads and 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 used by the instruction. Endianness does not matter if these instructions are only being used for a block copy operation.

A BST with commit forces the data to be written to memory and invalidates copies in all caches present. As a result, a BST 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 BST with commit is used to write modified instructions, a FLUSH instruction must still be executed to guarantee that the instruction pipeline is flushed.

LDDFA with a block transfer ASI 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 double-precision destination register. An *illegal_instruction* exception occurs if the floating-point registers are not aligned on an eight double-precision register boundary. The least significant six bits of the memory address must be zero or a *mem_address_not_aligned* exception occurs.

STDFA with a block transfer ASI stores data from the eight double-precision floating-point registers specified by `rs1` to a 64-byte-aligned memory area. The lowest-addressed eight bytes in memory are stored from the lowest-numbered double-precision `rd`. An

1. All store instructions in the processor coherently update the instruction cache. In general SPARC-V9 implementations, the store instructions (other than BST with Commit) do not maintain data coherency between instruction and data caches.

illegal_instruction exception occurs if the floating-point registers are not aligned on an eight register boundary. The least significant 6 bits of the memory address must be zero or a *mem_address_not_aligned* exception occurs.

ASIs $E0_{16}$ and $E1_{16}$ are only used for BST with commit operations; they are not used for BLD operations.

Programming Note – In the UltraSPARC IIIi processor, BLD does not offer a performance advantage over normal loads. For high performance, the use of prefetch instructions and 8-byte loads is recommended. BST and BST with commit can offer performance advantage and are used in high performance UltraSPARC IIIi processor libraries.

Programming Note – BLD does not provide register dependency interlocks, as ordinary load instructions do.

Before BLD data can be referenced, a second BLD (to a different set of registers) or a `MEMBAR #Sync` must be performed. If a second BLD is used to synchronize against returning data, the processor will continue execution before all data has been returned. The programmer is then responsible for scheduling instructions so registers are only used when they become valid.

To determine when data is valid, the programmer must count instruction groups containing floating-point (FP) operate instructions (not FP loads or stores). The lowest-numbered destination register of the first BLD may be referenced in the first instruction group following the second BLD, using an FP operate instruction only.

The second-lowest-numbered destination register of the first BLD may be referenced in the second instruction group containing an FP operate instruction, and so on.

If this block-load/block-load synchronization mechanism is used, the initial reference to the BLD data must be an FP operate instruction (not an FP store), and only instruction groups with FP operate instructions are counted when determining BLD data availability.

If these rules are violated, data from before or after the BLD may be returned by a reference to any of the BLD's destination registers.

If a `MEMBAR #Sync` is used to synchronize on BLD data, there are no restrictions on data usage, although performance will be lower than if block-load/block-load synchronization is used. No other `MEMBAR`s can be used to provide data synchronization for BLD.

FP operate instructions can be issued in a single instruction group with FP stores. If block-load/block-load synchronization is used, FP operates and FP stores can be interlaced. This allows an FP operate instruction, such as `FMOVD` or `FALIGNDATA`, to reference the returning data before using the data in any FP store (normal store or BST).

The processor also continues execution, without register interlocks, before all the store data for BSTs are transferred from the register file.

If store source registers are overwritten before the next BST or `MEMBAR #Sync` instruction, then the following rule must be observed: The first register can be overwritten in the same instruction group as the BST, the second register can be overwritten in the instruction group following the BST, and so on. If this rule is violated, the BST may use the old or the new (overwritten) data.

When determining correctness for a code sample, note that the processor may interlock more than what is required above. For example, there may be partial register interlocks, such as on the lowest-number register.

Code that does not meet the above constraints may appear to work on a particular processor. However, to be portable across all processors similar to the UltraSPARC IIIi processor, all of the above rules should be followed.

Rules

Note – These instructions are used for transferring large blocks of data (more than 256 bytes), for example, in C library routines `bcopy()` and `bfill()`. They do not allocate in the data cache or L2-cache on a miss. They update the L2-cache on a hit. One BLD and, in the most extreme cases, up to fifteen (maximum) BSTs can be outstanding on the interconnect at one time.

To simplify the implementation, BLD destination registers may or may not interlock like ordinary load instructions. Before the BLD data is referenced, a second BLD (to a different set of registers) or a `MEMBAR #Sync` must be performed. If a second BLD is used to synchronize with returning data, then it continues execution before all data have been returned. The lowest-number register being loaded can be referenced in the first instruction group following the second BLD, the second lowest number register can be referenced in the second group, and so on. If this rule is violated, data from before or after the load may be returned.

Similarly, BST source data registers are not interlocked against completion of previous load instructions (even if a second BLD has been performed). The previous load data must be referenced by some other intervening instruction, or an intervening `MEMBAR #Sync` must be performed. If the programmer violates these rules, data from before or after the load may be used. The load continues execution before all of the store data have been transferred. If store data registers are overwritten before the next BST or `MEMBAR #Sync` instruction, then the following rule must be observed: The first register can be overwritten in the same instruction group as the BST, the second register can be overwritten in the instruction group following the BST, and so on. If this rule is violated, the store may store correct data or the overwritten data.

There must be a MEMBAR #Sync or a trap following a BST before a DONE, RETRY, or WRPR to PSTATE instruction is executed. If this is rule is violated, instructions after the DONE, RETRY, or WRPR to PSTATE may not see the effects of the updated PSTATE register.

BLD does not follow memory model ordering with respect to stores. In particular, read-after-write and write-after-read hazards to overlapping addresses are not detected. The side-effects bit (TTE.E) associated with the access is ignored. Some ordering considerations are as follows:

- If ordering with respect to earlier stores is important (for example, a BLD that overlaps previous stores), then there must be an intervening MEMBAR #StoreLoad or stronger MEMBAR.
- If ordering with respect to later stores is important (for example, a BLD that overlaps a subsequent store), then there must be an intervening MEMBAR #LoadStore or a reference to the BLD data. This restriction does not apply when a trap is taken; therefore, the trap handler does not have to worry about pending BLDs.
- If the BLD overlaps a previous or later store and there is no intervening MEMBAR, then the trap or data referencing the BLD may return data from before or after the store.

BST does not follow memory model ordering with respect to loads, stores, or flushes. In particular, read-after-write, write-after-write, flush-after-write, and write-after-read hazards to overlapping addresses are not detected. The side-effects bit associated with the access is ignored. Some ordering considerations are as follows:

- If ordering with respect to earlier or later loads or stores is important, then there must be an intervening reference to the load data (for earlier loads) or an appropriate MEMBAR instruction. This restriction does not apply when a trap is taken; therefore, the trap handler does not have to worry about pending BSTs.
- If the BST overlaps a previous load and there is no intervening load data reference or MEMBAR #StoreLoad instruction, then the load may return data from before or after the store and the contents of the block are undefined.
- If the BST overlaps a later load and there is no intervening trap or MEMBAR #LoadStore instruction, then the contents of the block are undefined.
- If the BST overlaps a later store or flush and there is no intervening trap or MEMBAR #Sync instruction, then the contents of the block are undefined.
- If the ordering of two successive BST instructions (overlapping or not) is required, then a MEMBAR #Sync must occur between the BST instructions.

Block operations do not obey the ordering restrictions of the currently selected processor memory model (TSO, PSO, RMO). Block operations always execute under an RMO memory ordering model. Explicit MEMBAR instructions are required to order block operations among themselves or with respect to normal memory operations. In addition, block operations do not conform to dependence order on the issuing processor; that is, no read-after-write, write-after-read, or write-after-write checking occurs between block operations. Explicit MEMBAR #Sync instructions are required to enforce dependence ordering between block operations that reference the same address.

Typically, BLD and BST will be used in loops where software can ensure that the data being loaded and the data being stored do not overlap. The loop will be preceded and followed by the appropriate MEMBARs to ensure that there are no hazards with loads and stores outside the loops. CODE EXAMPLE A-3 demonstrates the loop.

CODE EXAMPLE A-3 Byte-Aligned Block Copy Inner Loop with Block Load/Block Store

Note that the loop must be unrolled two times to achieve maximum performance. All FP registers are double-precision. Eight versions of this loop are needed to handle all the cases of doubleword misalignment between the source and destination.

```
loop:
    faligndata     %f0, %f2, %f34
    faligndata     %f2, %f4, %f36
    faligndata     %f4, %f6, %f38
    faligndata     %f6, %f8, %f40
    faligndata     %f8, %f10, %f42
    faligndata     %f10, %f12, %f44
    faligndata     %f12, %f14, %f46
    addcc          %l0, -1, %l0
    bg,pt          l1
    fmovd          %f14, %f48
                                    ! (end of loop handling)
l1: ldda           [regaddr] ASI_BLK_P, %f0
    stda           %f32, [regaddr] ASI_BLK_P
    faligndata     %f48, %f16, %f32
    faligndata     %f16, %f18, %f34
    faligndata     %f18, %f20, %f36
    faligndata     %f20, %f22, %f38
    faligndata     %f22, %f24, %f40
    faligndata     %f24, %f26, %f42
    faligndata     %f26, %f28, %f44
    faligndata     %f28, %f30, %f46
    addcc          %l0, -1, %l0
    be,pnt         done
    fmovd          %f30, %f48
    ldda           [regaddr] ASI_BLK_P, %f16
    stda           %f32, [regaddr] ASI_BLK_P
```

CODE EXAMPLE A-3 Byte-Aligned Block Copy Inner Loop with Block Load/Block Store

```
    ba              loop
    faligndata      %f48, %f0, %f32
done:               !(end of loop processing)
```

Bcopy Code

To achieve the highest Bcopy bandwidths, use prefetch instructions and floating-point loads instead of BLD instructions. Using prefetch instructions to bring memory data into the prefetch cache hides all of the latency to memory. This allows a Bcopy loop to run at maximum bandwidth. CODE EXAMPLE A-4 shows how to modify the standard UltraSPARC I processor `bcopy()` loop to use PREFETCH and floating-point load instructions instead of BLDs.

CODE EXAMPLE A-4 High-Performance `bcopy()` Preamble Code

```
preamble:
    prefetch        [srcaddr],1
    prefetch        [srcaddr+0x40],1
    prefetch        [srcaddr+0x80],1
    prefetch        [srcaddr+0xc0],1
    lddf            [srcaddr],%f0
    prefetch        [srcaddr+0x100],1
    lddf            [srcaddr+0x8],%f2
    lddf            [srcaddr+0x10],%f4
    faligndata      %f0,%f2,%f32
    lddf            [srcaddr+0x18],%f6
    faligndata      %f2,%f4,%f34
    lddf            [srcaddr+0x20],%f8
    faligndata      %f4,%f6,%f36
    lddf            [srcaddr+0x28],%f10
    faligndata      %f6,%f8,%f38
    lddf            [srcaddr+0x30],%f12
    faligndata      %f8,%f10,%f40
    lddf            [srcaddr+0x38],%f14
    faligndata      %f10,%f12,%f42
    lddf            [srcaddr+0x40],%f16
    subcc           count,0x40,count
    bpe             <exit>
    add             srcaddr,0x40,srcaddr
```

CODE EXAMPLE A-4 High-Performance `bcopy()` Preamble Code *(Continued)*

```
loop:
1   fmovd      %f16,%f0
1   lddf       [srcaddr+0x8],%f2
2   faligndata %f12,%f14,%f44
2   lddf       [srcaddr+0x10],%f4
3   faligndata %f14,%f0,%f46
3   stda       %f32,[dstaddr] ASI_BLK_P
3   lddf       [srcaddr+0x18],%f6
4   faligndata %f0,%f2,%f32
4   lddf       [srcaddr+0x20],%f8
5   faligndata %f2,%f4,%f34
5   lddf       [srcaddr+0x28],%f10
6   faligndata %f4,%f6,%f36
6   lddf       [srcaddr+0x30],%f12
7   faligndata %f6,%f8,%f38
7   lddf       [srcaddr+0x38],%f14
8   faligndata %f8,%f10,%f40
8   lddf       [srcaddr+0x40],%f16
8   prefetch   [srcaddr+0x100],1
9   faligndata %f10,%f12,%f42
9   subcc      count,0x40,count
9   add        dstaddr,0x40,dstaddr
9   bpg        loop
1   add        srcaddr,0x40,srcaddr
```

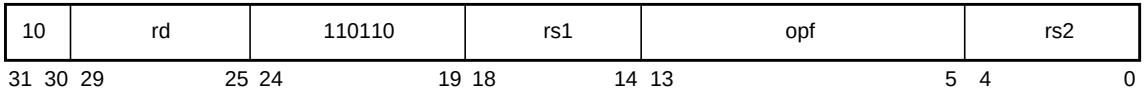
Exceptions

fp_disabled
PA_watchpoint (recognized on only the first 8 bytes of a transfer)
VA_watchpoint (recognized on only the first 8 bytes of a transfer)
illegal_instruction (misaligned `rd`)
mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.5 Byte Mask and Shuffle Instructions (VIS II)

Opcode	opf	Operation
BMASK	0 0001 1001	Set the GSR.MASK field in preparation for a following BSHUFFLE instruction
BSHUFFLE	0 0100 1100	Permute bytes as specified by GSR.MASK

Format (3)



Assembly Language Syntax	
bmask	reg_{rs1}, reg_{rs2}, reg_{rd}
bshuffle	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

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 specified by rs1 (more-significant half) and rs2 (less-significant half) to form a 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 the 16 bytes and stores the result in the 64-bit floating-point register specified by rd. Bytes in the rd register 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 for each byte in rd.

Destination Byte (in r[rd])	Source Byte
0 (Most significant)	(r[rs1] ▯ r[rs2])[GSR.mask<31:28>]
1	(r[rs1] ▯ r[rs2])[GSR.mask<27:24>]
2	(r[rs1] ▯ r[rs2])[GSR.mask<23:20>]
3	(r[rs1] ▯ r[rs2])[GSR.mask<19:16>]
4	(r[rs1] ▯ r[rs2])[GSR.mask<15:12>]
5	(r[rs1] ▯ r[rs2])[GSR.mask<11:8>]
6	(r[rs1] ▯ r[rs2])[GSR.mask<7:4>]
7 (Least significant)	(r[rs1] ▯ r[rs2])[GSR.mask<3:0>]

Note – The BMASK instruction uses the MS pipeline; therefore, it cannot be grouped with a store, non-prefetchable load, or a special instruction. The integer rd register result is available after a two-cycle latency. A younger BMASK can be grouped with an older BSHUFFLE (BMASK is "break-after").

Results have a four-cycle latency to other dependent instructions executed in FGA and FGM pipelines. The FGA pipeline is used to execute BSHUFFLE. The GSR mask must be set at or before the instruction group previous to the BSHUFFLE (GSR.mask dependency). BSHUFFLE is fully pipelined (one per cycle).

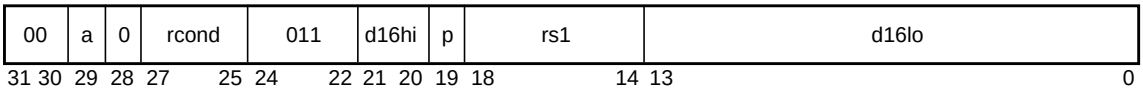
Exceptions

fp_disabled

A.6 Branch on Integer Register with Prediction (BPr)

Opcode	rcond	Operation	Register Contents Test
—	000	*Reserved*	—
BRZ	001	Branch on Register Zero	r[rs1] = 0
BRLEZ	010	Branch on Register Less Than or Equal to Zero	r[rs1] ≤ 0
BRLZ	011	Branch on Register Less Than Zero	r[rs1] < 0
—	100	*Reserved*	—
BRNZ	101	Branch on Register Not Zero	r[rs1] ≠ 0
BRGZ	110	Branch on Register Greater Than Zero	r[rs1] > 0
BRGEZ	111	Branch on Register Greater Than or Equal to Zero	r[rs1] ≥ 0

Format (2)



Assembly Language Syntax	
`brz{,a}{,pt|,pn}`	reg_{rs1}, *label*
`brlez{,a}{,pt|,pn}`	reg_{rs1}, *label*
`brlz{,a}{,pt|,pn}`	reg_{rs1}, *label*
`brnz{,a}{,pt|,pn}`	reg_{rs1}, *label*
`brgz{,a}{,pt|,pn}`	reg_{rs1}, *label*
`brgez{,a}{,pt|,pn}`	reg_{rs1}, *label*

Programming Note – To set the annul 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 shall default to "`,pt`."

Programming Note – Both `BP` and `BR` represent branch on integer register with prediction. They are, in fact, the same instruction.

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 bit. If the branch is not taken and the annul bit (`a`) is one, 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. A one in the `p` bit indicates that the branch is expected to be taken; a zero indicates that the branch is expected not to be taken.

Implementation Note – The UltraSPARC IIIi processor does not implement this instruction by tagging each register value. The UltraSPARC IIIi processor looks at the full 64-bit register to determine a negative or zero.

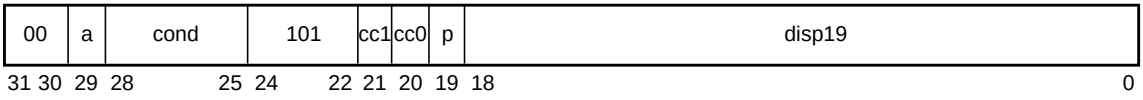
Exceptions

illegal_instruction (if `rcond` = 000_2 or 100_2)

A.7 Branch on Floating-Point Condition Codes with Prediction (FBPfcc)

Opcode	cond	Operation	*fcc* Test
FBPA	1000	Branch Always	1
FBPN	0000	Branch Never	0
FBPU	0111	Branch on Unordered	U
FBPG	0110	Branch on Greater	G
FBPUG	0101	Branch on Unordered or Greater	G **or** U
FBPL	0100	Branch on Less	L
FBPUL	0011	Branch on Unordered or Less	L **or** U
FBPLG	0010	Branch on Less or Greater	L **or** G
FBPNE	0001	Branch on Not Equal	L **or** G **or** U
FBPE	1001	Branch on Equal	E
FBPUE	1010	Branch on Unordered or Equal	E **or** U
FBPGE	1011	Branch on Greater or Equal	E **or** G
FBPUGE	1100	Branch on Unordered or Greater or Equal	E **or** G **or** U
FBPLE	1101	Branch on Less or Equal	E **or** L
FBPULE	1110	Branch on Unordered or Less or Equal	E **or** L **or** U
FBPO	1111	Branch on Ordered	E **or** L **or** G

Format (2)



cc1 cc0	Condition Code
00	fcc*0*
01	fcc*1*
10	fcc*2*
11	fcc*3*

Assembly Language Syntax		
fba{,a}{,pt|,pn}	%fcc*n, label*	
fbn{,a}{,pt|,pn}	%fcc*n, label*	
fbu{,a}{,pt|,pn}	%fcc*n, label*	
fbg{,a}{,pt|,pn}	%fcc*n, label*	
fbug{,a}{,pt|,pn}	%fcc*n, label*	
fbl{,a}{,pt|,pn}	%fcc*n, label*	
fbul{,a}{,pt|,pn}	%fcc*n, label*	
fblg{,a}{,pt|,pn}	%fcc*n, label*	
fbne{,a}{,pt|,pn}	%fcc*n, label*	(*synonym*: fbnz)
fbe{,a}{,pt|,pn}	%fcc*n, label*	(*synonym*: fbz)
fbue{,a}{,pt|,pn}	%fcc*n, label*	
fbge{,a}{,pt|,pn}	%fcc*n, label*	
fbuge{,a}{,pt|,pn}	%fcc*n, label*	
fble{,a}{,pt|,pn}	%fcc*n, label*	
fbule{,a}{,pt|,pn}	%fcc*n, label*	
fbo{,a}{,pt|,pn}	%fcc*n, label*	

Programming Note – To set the annul 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 shall default 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 zero, an FBPN (Floating-Point Branch Never with Prediction) instruction acts like a NOP. If the Branch Never annul field is zero, the following (delay) instruction is executed; if the annul field is one, 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 one, the delay instruction is annulled (not executed). If the annul field is zero, 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 field. If a conditional branch is not taken and the annul field (a) is one, 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 one in the p bit indicates that the branch is expected to be taken. A zero indicates that the branch is expected not to be taken.

If FPRS.FEF = 0 or PSTATE.PEF = 0, or if an FPU is not present, an FBPfcc instruction is not executed and instead, an *fp_disabled* exception is generated.

Compatibility Note – Unlike SPARC-V8, SPARC-V9 does not require an instruction between a floating-point compare operation and a floating-point branch (FBfcc, FBPfcc).

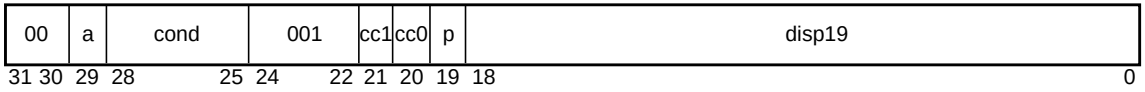
Exceptions

fp_disabled

A.8 Branch on Integer Condition Codes with Prediction (BPcc)

Opcode	cond	Operation	*icc* Test
BPA	1000	Branch Always	1
BPN	0000	Branch Never	0
BPNE	1001	Branch on Not Equal	**not** Z
BPE	0001	Branch on Equal	Z
BPG	1010	Branch on Greater	**not** (Z **or** (N **xor** V))
BPLE	0010	Branch on Less or Equal	Z **or** (N **xor** V)
BPGE	1011	Branch on Greater or Equal	**not** (N **xor** V)
BPL	0011	Branch on Less	N **xor** V
BPGU	1100	Branch on Greater Unsigned	**not** (C **or** Z)
BPLEU	0100	Branch on Less or Equal Unsigned	C **or** Z
BPCC	1101	Branch on Carry Clear (Greater Than or Equal, Unsigned)	**not** C
BPCS	0101	Branch on Carry Set (Less than, Unsigned)	C
BPPOS	1110	Branch on Positive	**not** N
BPNEG	0110	Branch on Negative	N
BPVC	1111	Branch on Overflow Clear	**not** V
BPVS	0111	Branch on Overflow Set	V

Format (2)



cc1 cc0	Condition Code
00	*icc*
01	—
10	*xcc*
11	—

Assembly Language Syntax		
`ba{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bn{,a}{,pt|,pn}`	*i_or_x_cc, label*	(*or:* `iprefetch` label)
`bne{,a}{,pt|,pn}`	*i_or_x_cc, label*	(*synonym*: `bnz`)
`be{,a}{,pt|,pn}`	*i_or_x_cc, label*	(*synonym*: `bz`)
`bg{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`ble{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bge{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bl{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bgu{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bleu{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bcc{,a}{,pt|,pn}`	*i_or_x_cc, label*	(*synonym*: `bgeu`)
`bcs{,a}{,pt|,pn}`	*i_or_x_cc, label*	(*synonym*: `blu`)
`bpos{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bneg{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bvc{,a}{,pt|,pn}`	*i_or_x_cc, label*	
`bvs{,a}{,pt|,pn}`	*i_or_x_cc, label*	

Programming Note – To set the annul 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 shall default 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) is used in SPARC-V9 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 annul field is one, then the following (delay) instruction is annulled (not executed). If the annul field is zero, 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 field of the branch instruction is one, then the delay instruction is annulled (not executed). If the annul field is zero, 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 field. If a conditional branch is not taken and the annul field (a) is one, the delay instruction is annulled (not executed).

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

The predict bit (p) is used to give the hardware a hint about whether the branch is expected to be taken. A one in the p bit indicates that the branch is expected to be taken; a zero indicates that the branch is expected not to be taken.

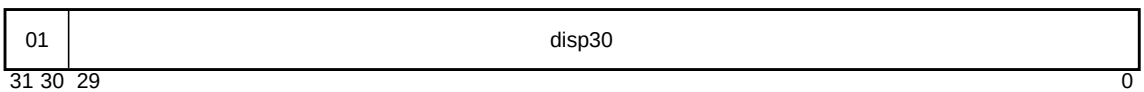
Exceptions

illegal_instruction (*cc1* ▯ *cc0* = 01_2 or 11_2)

A.9 Call and Link

Opcode	op	Operation
CALL	01	Call and Link

Format (1)



Assembly Language Syntax	
`call`	*label*

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).

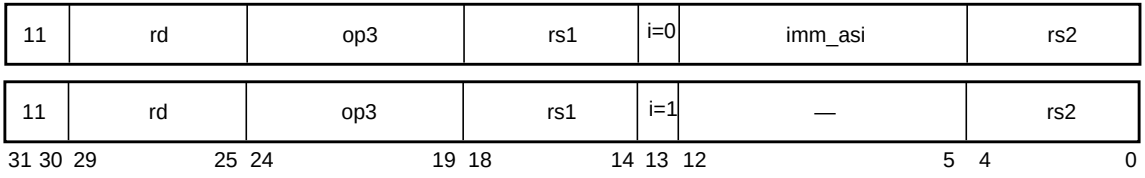
Exceptions

None

A.10 Compare and Swap

Opcode	op3	Operation
CASA$^{P_{ASI}}$	11 1100	Compare and Swap Word from Alternate Space
CASXA$^{P_{ASI}}$	11 1110	Compare and Swap Extended from Alternate Space

Format (3)



Assembly Language Syntax	
`casa`	[reg_{rs1}] *imm_asi*, reg_{rs2}, reg_{rd}
`casa`	[reg_{rs1}] `%asi`, reg_{rs2}, reg_{rd}
`casxa`	[reg_{rs1}] *imm_asi*, reg_{rs2}, reg_{rd}
`casxa`	[reg_{rs1}] `%asi`, reg_{rs2}, reg_{rd}

Description

Concurrent processes use these instructions for synchronization and memory updates. Uses of compare-and-swap include spin-lock operations, updates of shared counters, and updates of linked-list pointers. The last two can use wait-free (non-locking) 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 zero. If the values are not equal, the memory location remains unchanged, but the zero-extended 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 zero.

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

A compare-and-swap operation does *not* imply any memory barrier semantics. When compare-and-swap is used for synchronization, the same consideration should be given to memory barriers as if a load, store, or swap instruction were used.

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

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

A *mem_address_not_aligned* exception is generated if the address in `r[rs1]` is not properly aligned. CASXA and CASA cause a *privileged_action* exception if `PSTATE.PRIV` = 0 and bit 7 of the ASI is zero.

The coherence and atomicity of memory operations between processors and I/O DMA memory accesses is maintained for cacheable memory space.

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.

The compare-and-swap instructions do not affect the condition codes.

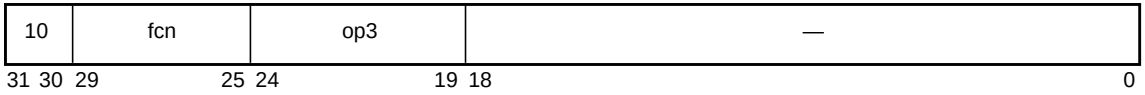
Exceptions

privileged_action
mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.11 DONE and RETRY

Opcode	op3	fcn	Operation
DONE[P]	11 1110	0	Return from Trap (skip trapped instruction)
RETRY[P]	11 1110	1	Return from Trap (retry trapped instruction)
—	11 1110	2–31	*Reserved*

Format (3)



Assembly Language Syntax	
done	
retry	

Description

The DONE and RETRY instructions restore the saved state from TSTATE (CWP, ASI, CCR, and PSTATE), set PC and nPC, and decrement TL.

The RETRY instruction resumes execution with the trapped instruction by setting PC ← TPC[TL] (the saved value of PC on trap) and nPC ← TNPC[TL] (the saved value of nPC on trap).

The DONE instruction skips the trapped instruction by setting PC ← TNPC[TL] and nPC ← TNPC[TL] + 4.

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

Programming Note – Use the `DONE` and `RETRY` instructions to return from privileged trap handlers.

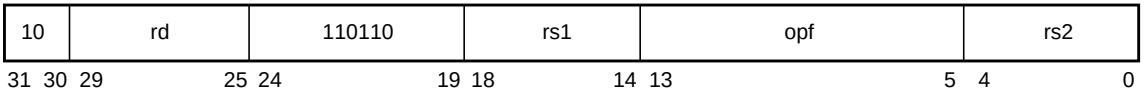
Exceptions

privileged_opcode
illegal_instruction (if TL = 0 or *fcn* = 2–31)

A.12 Edge Handling Instructions (VIS I, VIS II)

Opcode	opf	Operation
`EDGE8`	0 0000 0000	Eight 8-bit edge boundary processing
`EDGE8N`	0 0000 0001	Eight 8-bit edge boundary processing, no condition codes
`EDGE8L`	0 0000 0010	Eight 8-bit edge boundary processing, little-endian
`EDGE8LN`	0 0000 0011	Eight 8-bit edge boundary processing, little-endian, no condition codes
`EDGE16`	0 0000 0100	Four 16-bit edge boundary processing
`EDGE16N`	0 0000 0101	Four 16-bit edge boundary processing, no condition codes
`EDGE16L`	0 0000 0110	Four 16-bit edge boundary processing, little-endian
`EDGE16LN`	0 0000 0111	Four 16-bit edge boundary processing, little-endian, no condition codes
`EDGE32`	0 0000 1000	Two 32-bit edge boundary processing
`EDGE32N`	0 0000 1001	Two 32-bit edge boundary processing, no condition codes
`EDGE32L`	0 0000 1010	Two 32-bit edge boundary processing, little-endian
`EDGE32LN`	0 0000 1011	Two 32-bit edge boundary processing, little-endian, no condition codes

Format (3)



Assembly Language Syntax	
edge8	reg_{rs1}, reg_{rs2}, reg_{rd}
edge8n	reg_{rs1}, reg_{rs2}, reg_{rd}
edge8l	reg_{rs1}, reg_{rs2}, reg_{rd}
edge8ln	reg_{rs1}, reg_{rs2}, reg_{rd}
edge16	reg_{rs1}, reg_{rs2}, reg_{rd}
edge16n	reg_{rs1}, reg_{rs2}, reg_{rd}
edge16l	reg_{rs1}, reg_{rs2}, reg_{rd}
edge16ln	reg_{rs1}, reg_{rs2}, reg_{rd}
edge32	reg_{rs1}, reg_{rs2}, reg_{rd}
edge32n	reg_{rs1}, reg_{rs2}, reg_{rd}
edge32l	reg_{rs1}, reg_{rs2}, reg_{rd}
edge32ln	reg_{rs1}, reg_{rs2}, reg_{rd}

Description

These instructions handle the boundary conditions for parallel pixel scan line loops, where `src1` is the address of the next pixel to render and `src2` is the address of the last pixel in the scan line.

`EDGE8L(N)`, `EDGE16L(N)`, and `EDGE32L(N)` are little-endian versions of `EDGE8(N)`, `EDGE16(N)`, and `EDGE32(N)`. 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 graphics compare operations (see Section A.44, “Pixel Compare (VIS I)”) and with the Partial Store instruction (see Section A.41, “Partial Store (VIS I)”) on little-endian data.

A 2-bit (`EDGE32`), 4-bit (`EDGE16`), or 8-bit (`EDGE8`) 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 three least significant bits (LSBs) of `r[rs1]`, and the right edge mask is computed from the three LSBs of `r[s2]`, according to TABLE A-4 (TABLE A-5 for little-endian byte ordering).

2. If 32-bit address masking is disabled (`PSTATE.AM` = 0, 64-bit addressing) and the upper 61 bits of `r[rs1]` are equal to the corresponding bits in `r[rs2]`, `r[rd]` is set to the right edge mask ANDed with the left edge mask.

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

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

The integer condition codes are set per the rules of the SUBCC instruction with the same operands (see Section A.64, “Subtract”).

The EDGE(8, 16, 32)(L)N instructions do not set the integer condition codes.

Exceptions

None

TABLE A-4 Edge Mask Specification

Edge Size	A2–A0	Left Edge	Right Edge
8	000	1111 1111	1000 0000
8	001	0111 1111	1100 0000
8	010	0011 1111	1110 0000
8	011	0001 1111	1111 0000
8	100	0000 1111	1111 1000
8	101	0000 0111	1111 1100
8	110	0000 0011	1111 1110
8	111	0000 0001	1111 1111
16	00x	1111	1000
16	01x	0111	1100
16	10x	0011	1110
16	11x	0001	1111
32	0xx	11	10
32	1xx	01	11

TABLE A-5 Edge Mask Specification (Little-Endian)

Edge Size	A2–A0	Left Edge	Right Edge
8	000	1111 1111	0000 0001
8	001	1111 1110	0000 0011
8	010	1111 1100	0000 0111
8	011	1111 1000	0000 1111

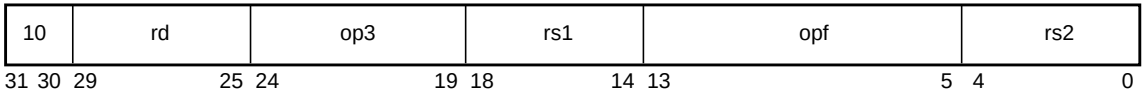
TABLE A-5 Edge Mask Specification (Little-Endian) *(Continued)*

Edge Size	A2–A0	Left Edge	Right Edge
8	100	1111 0000	0001 1111
8	101	1110 0000	0011 1111
8	110	1100 0000	0111 1111
8	111	1000 0000	1111 1111
16	00x	1111	0001
16	01x	1110	0011
16	10x	1100	0111
16	11x	1000	1111
32	0xx	11	01
32	1xx	10	11

A.13 Floating-Point Add and Subtract

Opcode	op3	opf	Operation
FADDs	11 0100	0 0100 0001	Add Single
FADDd	11 0100	0 0100 0010	Add Double
FADDq	11 0100	0 0100 0011	Add Quad
FSUBs	11 0100	0 0100 0101	Subtract Single
FSUBd	11 0100	0 0100 0110	Subtract Double
FSUBq	11 0100	0 0100 0111	Subtract Quad

Format (3)



Assembly Language Syntax	
fadds	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
faddd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
faddq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fsubs	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fsubd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fsubq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

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.

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 the `FSR.RD` field.

Compatibility Note – When `FSR.NS` = 0, the processor operates in standard floating-point mode. `FADD` or `FSUB` with a subnormal result causes an *fp_exception_other* exception with `FSR.ftt` = *unfinished_FPop*, system software emulates the instruction, and the correct numerical result is calculated.

When `FSR.NS` = 1, the processor operates in “nonstandard” floating-point mode. When `FSR.NS` = 1, and `FADD` or `FSUB` produces a subnormal result on an UltraSPARC IIIi processor, a *fp_exception_other* exception occurs with `FSR.ftt` = *unfinished_FPop* (even though the processor is operating in nonstandard floating-point mode), then system software emulates the instruction, and the correct numerical result is calculated (instead of replacing the result with zero).

Therefore, the processor may produce a different (albeit more accurate) result than in previous processors in the following situation:

> `FADD` or `FSUB` produces a subnormal result
> `FSR.NS` = 1

Notes –

1) The processor does not implement (in hardware) the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

2) For `FADDs`, `FADDd`, `FSUBs`, `FSUBd`, an *fp_exception_other* with `ftt` = *unfinished_FPop* can occur if either operand is NaN.

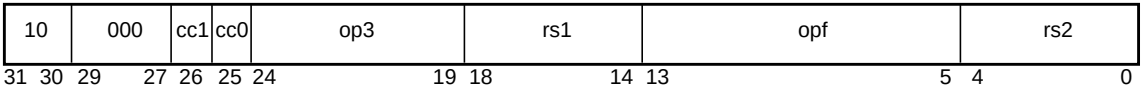
Exceptions

fp_disabled
fp_exception_ieee_754 (OF, UF, NX, NV)
fp_exception_other (`ftt` = *unimplemented_FPop* (`FADDq` and `FSUBq` only))
fp_exception_other (`ftt` = *unifinished_FPop* (`FADDs`, `FADDd`, `FSUBs`, `FSUBd` only))

A.14 Floating-Point Compare

Opcode	op3	opf	Operation
`FCMPs`	11 0101	0 0101 0001	Compare Single
`FCMPd`	11 0101	0 0101 0010	Compare Double
`FCMPq`	11 0101	0 0101 0011	Compare Quad
`FCMPEs`	11 0101	0 0101 0101	Compare Single and Exception if Unordered
`FCMPEd`	11 0101	0 0101 0110	Compare Double and Exception if Unordered
`FCMPEq`	11 0101	0 0101 0111	Compare Quad and Exception if Unordered

Format (3)



Assembly Language Syntax	
`fcmps`	`%fcc`*n*, $freg_{rs1}$, $freg_{rs2}$
`fcmpd`	`%fcc`*n*, $freg_{rs1}$, $freg_{rs2}$
`fcmpq`	`%fcc`*n*, $freg_{rs1}$, $freg_{rs2}$
`fcmpes`	`%fcc`*n*, $freg_{rs1}$, $freg_{rs2}$
`fcmped`	`%fcc`*n*, $freg_{rs1}$, $freg_{rs2}$
`fcmpeq`	`%fcc`*n*, $freg_{rs1}$, $freg_{rs2}$

cc1 cc0	Condition Code
00	`fcc0`
01	`fcc1`
10	`fcc2`
11	`fcc3`

Description

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

fcc value	Relation
0	$freg_{rs1} = freg_{rs2}$
1	$freg_{rs1} < freg_{rs2}$
2	$freg_{rs1} > freg_{rs2}$
3	$freg_{rs1}$? $freg_{rs2}$ (unordered)

The “?” in the preceding table means that the comparison is unordered. The unordered condition occurs when one or both of the operands to the compare is a signalling or quiet NaN.

The “compare and cause exception if unordered” (`FCMPEs`, `FCMPEd`, and `FCMPEq`) instructions cause an invalid (NV) exception if either operand is a NaN.

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

Compatibility Note – Unlike SPARC-V8, SPARC-V9 does 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 a zero in the `r[rd]` field. In SPARC-V9, bits 26 and 25 of the `r[rd]` field specify the floating-point condition code to be set. Legal SPARC-V8 code will work on SPARC-V9 because the zeroes in the `r[rd]` field are interpreted as `fcc0` and the `FBfcc` instruction branches according to `fcc0`.

Note – The processor does not implement (in hardware) the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

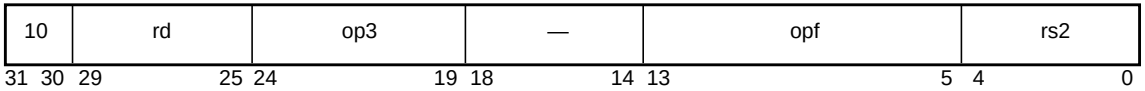
Exceptions

fp_disabled
fp_exception_ieee_754 (NV)
fp_exception_other (`ftt =` *unimplemented_FPop* (`FCMPq`, `FCMPEq` only))

A.15 Convert Floating-Point to Integer

Opcode	op3	opf	Operation
`FsTOx`	11 0100	0 1000 0001	Convert Single to 64-bit Integer
`FdTOx`	11 0100	0 1000 0010	Convert Double to 64-bit Integer
`FqTOx`	11 0100	0 1000 0011	Convert Quad to 64-bit Integer
`FsTOi`	11 0100	0 1101 0001	Convert Single to 32-bit Integer
`FdTOi`	11 0100	0 1101 0010	Convert Double to 32-bit Integer
`FqTOi`	11 0100	0 1101 0011	Convert Quad to 32-bit Integer

Format (3)



Assembly Language Syntax	
`fstox`	*freg*$_{rs2}$, *freg*$_{rd}$
`fdtox`	*freg*$_{rs2}$, *freg*$_{rd}$
`fqtox`	*freg*$_{rs2}$, *freg*$_{rd}$
`fstoi`	*freg*$_{rs2}$, *freg*$_{rd}$
`fdtoi`	*freg*$_{rs2}$, *freg*$_{rd}$
`fqtoi`	*freg*$_{rs2}$, *freg*$_{rd}$

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(s) specified by `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 specified by `rd`.

The result is always rounded toward zero; that is, the rounding direction (`RD`) field of the `FSR` register is ignored.

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 a *fp_exception_ieee_754* "invalid" exception occurs.

Note – The processor does not implement (in hardware) the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

The following floating-point-to-integer conversion instructions generate an *unfinished_FPop* exception for certain ranges of floating-point operands, as shown in TABLE A-6.

TABLE A-6 Floating-Point to Integer *unfinished_FPop* Exception Conditions

Instruction	Unfinished Trap Ranges
FsTOi	result $< -2^{31}$, result $\geq 2^{31}$, Inf, NaN
FsTOx	$\lvert result \rvert \geq 2^{52}$, Inf, NaN
FdTOi	result $< -2^{31}$, result $\geq 2^{31}$, Inf, NaN
FdTOx	$\lvert result \rvert \geq 2^{52}$, Inf, NaN

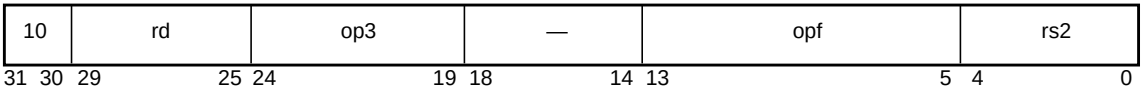
Exceptions

fp_disabled
fp_exception_ieee_754 (NV, NX)
unfinished_FPop
fp_exception_other (`ftt` = *unimplemented_FPop* (`FqTOi`, `FqTOx` only))

A.16 Convert Between Floating-Point Formats

Opcode	op3	opf	Operation
FsTOd	11 0100	0 1100 1001	Convert Single to Double
FsTOq	11 0100	0 1100 1101	Convert Single to Quad
FdTOs	11 0100	0 1100 0110	Convert Double to Single
FdTOq	11 0100	0 1100 1110	Convert Double to Quad
FqTOs	11 0100	0 1100 0111	Convert Quad to Single
FqTOd	11 0100	0 1100 1011	Convert Quad to Double

Format (3)



Assembly Language Syntax	
`fstod`	$freg_{rs2}$, $freg_{rd}$
`fstoq`	$freg_{rs2}$, $freg_{rd}$
`fdtos`	$freg_{rs2}$, $freg_{rd}$
`fdtoq`	$freg_{rs2}$, $freg_{rd}$
`fqtos`	$freg_{rs2}$, $freg_{rd}$
`fqtod`	$freg_{rs2}$, $freg_{rd}$

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`.

Rounding is performed as specified by the `FSR.RD` field.

`FqTOd`, `FqTOs`, and `FdTOs` (the "narrowing" conversion instructions) can raise `OF`, `UF`, and `NX` exceptions. `FdTOq`, `FsTOq`, and `FsTOd` (the "widening" conversion instructions) cannot.

Any of these six instructions can trigger an NV exception if the source operand is a signalling NaN.

Notes –

1) The UltraSPARC IIIi processor does not implement (in hardware) the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

2) For `FdTOs` and `FsTOd`, a *fp_exception_other* with `ftt` = *unfinished_FPop* can occur if the source operand is NaN or subnormal, or out of range of the destination format.

The following floating-point to floating-point conversion instructions generate an *unfinished_FPop* exception for certain ranges of floating-point operands, as shown in TABLE A-7.

TABLE A-7 Floating-Point/Floating-Point *unfinished_FPop* Exception Conditions

Instruction	Unfinished Trap Ranges
`FdTOs`	$\|result\| \geq 2^{52}$, $\|result\| < 2^{-31}$, operand $< -2^{22}$, operand $\geq 2^{22}$, NaN

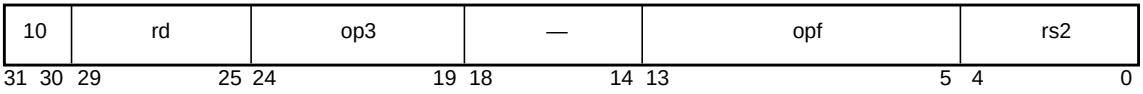
Exceptions

fp_disabled
fp_exception_ieee_754 (OF, UF, NV, NX)
fp_exception_other (`ftt` = *unimplemented_FPop* (`FsTOq`, `FdTOq`, `FqTOs`, `FqTOd` only))
unfinished_FPop
fp_exception_other (`ftt` = *unfinished_FPop* (`FdTOs` and `FsTOd` only))

A.17 Convert Integer to Floating-Point

Opcode	op3	opf	Operation
`FxTOs`	11 0100	0 1000 0100	Convert 64-bit Integer to Single
`FxTOd`	11 0100	0 1000 1000	Convert 64-bit Integer to Double
`FxTOq`	11 0100	0 1000 1100	Convert 64-bit Integer to Quad
`FiTOs`	11 0100	0 1100 0100	Convert 32-bit Integer to Single
`FiTOd`	11 0100	0 1100 1000	Convert 32-bit Integer to Double
`FiTOq`	11 0100	0 1100 1100	Convert 32-bit Integer to Quad

Format (3)



Assembly Language Syntax	
`fxtos`	$freg_{rs2}$, $freg_{rd}$
`fxtod`	$freg_{rs2}$, $freg_{rd}$
`fxtoq`	$freg_{rs2}$, $freg_{rd}$
`fitos`	$freg_{rs2}$, $freg_{rd}$
`fitod`	$freg_{rs2}$, $freg_{rd}$
`fitoq`	$freg_{rs2}$, $freg_{rd}$

Description

FxTOs, FxTOd, and FxTOq convert the 64-bit signed integer operand in the floating-point registers specified by rs2 into a floating-point number in the destination format.

FiTOs, FiTOd, and FiTOq convert the 32-bit signed integer operand in floating-point register(s) specified by rs2 into a floating-point number in the destination format. All write their result into the floating-point register(s) specified by rd.

FiTOs, FxTOs, and FxTOd round as specified by the FSR.RD field.

Note – The UltraSPARC IIIi processor does not implement (in hardware) the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with ftt = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

The following integer-to-floating-point conversion instructions generate an *unfinished_FPop* exception for certain ranges of integer operands, as shown in TABLE A-8.

TABLE A-8 Integer/Floating-Point *unfinished_FPop* Exception Conditions

Instruction	Unfinished Trap Ranges
FiTOs	operand $< -2^{22}$, operand $\geq 2^{22}$
FxTOs	operand $< -2^{22}$, operand $\geq 2^{22}$
FxTOd	operand $< -2^{51}$, operand $\geq 2^{51}$

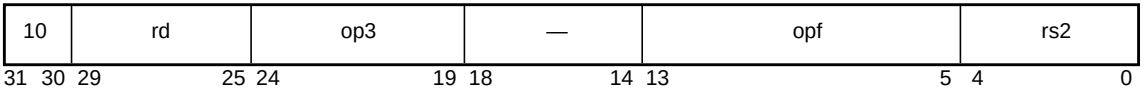
Exceptions

fp_disabled
fp_exception_ieee_754 (NX (FiTOs, FxTOs, FxTOd only))
unfinished_FPop
fp_exception_other (ftt = *unimplemented_FPop* (FiTOq, FxTOq only))

A.18 Floating-Point Move

Opcode	op3	opf	Operation
FMOVs	11 0100	0 0000 0001	Move Single
FMOVd	11 0100	0 0000 0010	Move Double
FMOVq	11 0100	0 0000 0011	Move Quad
FNEGs	11 0100	0 0000 0101	Negate Single
FNEGd	11 0100	0 0000 0110	Negate Double
FNEGq	11 0100	0 0000 0111	Negate Quad
FABSs	11 0100	0 0000 1001	Absolute Value Single
FABSd	11 0100	0 0000 1010	Absolute Value Double
FABSq	11 0100	0 0000 1011	Absolute Value Quad

Format (3)



Assembly Language Syntax	
fmovs	$freg_{rs2}$, $freg_{rd}$
fmovd	$freg_{rs2}$, $freg_{rd}$
fmovq	$freg_{rs2}$, $freg_{rd}$
fnegs	$freg_{rs2}$, $freg_{rd}$
fnegd	$freg_{rs2}$, $freg_{rd}$
fnegq	$freg_{rs2}$, $freg_{rd}$
fabss	$freg_{rs2}$, $freg_{rd}$
fabsd	$freg_{rs2}$, $freg_{rd}$
fabsq	$freg_{rs2}$, $freg_{rd}$

Description

The single-precision versions of these instructions copy the contents of a single-precision floating-point register to the destination. The double-precision versions copy the contents of a double-precision floating-point register to the destination. The quad-precision versions copy a quad-precision value in floating-point registers to the destination.

`FMOV` copies the source to the destination unaltered.

`FNEG` copies the source to the destination with the sign bit complemented.

`FABS` copies the source to the destination with the sign bit cleared.

These instructions do not round.

Note – The processor does not implement (in hardware) the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

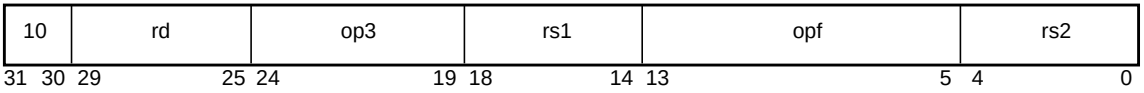
Exceptions

fp_disabled
fp_exception_other (`ftt` = *unimplemented_FPop* (`FMOVq`, `FNEGq`, `FABSq` only))

A.19 Floating-Point Multiply and Divide

Opcode	op3	opf	Operation
FMULs	11 0100	0 0100 1001	Multiply Single
FMULd	11 0100	0 0100 1010	Multiply Double
FMULq	11 0100	0 0100 1011	Multiply Quad
FsMULd	11 0100	0 0110 1001	Multiply Single to Double
FdMULq	11 0100	0 0110 1110	Multiply Double to Quad
FDIVs	11 0100	0 0100 1101	Divide Single
FDIVd	11 0100	0 0100 1110	Divide Double
FDIVq	11 0100	0 0100 1111	Divide Quad

Format (3)



Assembly Language Syntax	
fmuls	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmuld	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmulq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fsmuld	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fdmulq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fdivs	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fdivd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fdivq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

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.

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 the FSR.RD field.

Notes –

1) The processor does not implement (in hardware) the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with ftt = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

2) For FDIVs and FDIVd, a *fp_exception_other* with ftt = *unfinished_FPop* can occur if the divide unit detects certain unusual conditions.

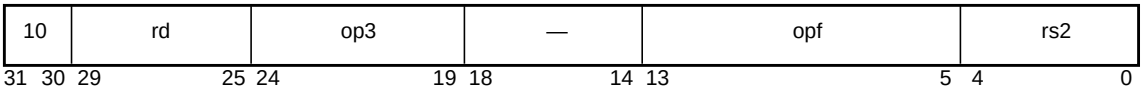
Exceptions

fp_disabled
fp_exception_ieee_754 (OF, UF, DZ (FDIV only), NV, NX)
fp_exception_other (ftt = *unimplemented_FPop* (FMULq, FdMULq, FDIVq)
fp_exception_other (ftt = *unifinished_FPop* (FMULs, FMULd, FSMULd, FDIVs, FDIV))

A.20 Floating-Point Square Root

Opcode	op3	opf	Operation
FSQRTs	11 0100	0 0010 1001	Square Root Single
FSQRTd	11 0100	0 0010 1010	Square Root Double
FSQRTq	11 0100	0 0010 1011	Square Root Quad

Format (3)



Assembly Language Syntax	
fsqrts	$freg_{rs2}$, $freg_{rd}$
fsqrtd	$freg_{rs2}$, $freg_{rd}$
fsqrtq	$freg_{rs2}$, $freg_{rd}$

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 the `FSR.RD` field.

Note – The processor does not implement (in hardware) the instructions that refer to a quad floating-point register. Execution of such an instruction generates *fp_exception_other* (with `ftt` = *unimplemented_FPop*), which causes a trap. Supervisor software then emulates these instructions.

For `FSQRTs` and `FSQRTd` a *fp_exception_other* (with `ftt` = *unfinished_FPop*) can occur if the operand to the square root is positive denormalized.

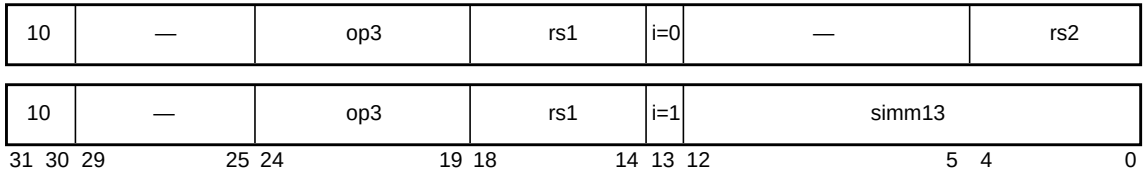
Exceptions

fp_disabled
fp_exception_ieee_754 (*IEEE_754_exception* (NV, NX))
fp_exception_other (*unimplemented_FPop*) (Quad forms)
fp_exception_other (*unfinished_FPop*) (FSQRTs, FSQRTd)

A.21 Flush Instruction Memory

Opcode	op3	Operation
FLUSH	11 1011	Flush Instruction Memory

Format (3)



Assembly Language Syntax	
flush	*address*

Description

FLUSH ensures that the doubleword specified as the effective address is consistent across any local caches, and in a multiprocessor system, will eventually become consistent everywhere.

In the following discussion P_{FLUSH} refers to the processor that executed the FLUSH instruction.

FLUSH 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 processors. FLUSH behaves as if it were a store with respect to MEMBAR-induced orderings. See Section A.34, “Memory Barrier.”

The effective address operand for the FLUSH instruction is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1. The least significant two address bits of the effective address are unused and should be supplied as zeroes by software. Bit 2 of the address is ignored because FLUSH operates on at least a doubleword.

Programming Note –

1. Typically, FLUSH is used in self-modifying code. The use of self-modifying code is discouraged.
2. 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.
3. 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.
4. FLUSH may be time consuming.
5. In a multiprocessor system, the time it takes for a FLUSH to take effect is dependent on the system. No mechanism is provided to ensure or test completion.
6. Because FLUSH is designed to act on a doubleword and on some implementations FLUSH may trap to system software, system software should provide a user-callable service routine for flushing arbitrarily sized regions of memory. On some processor 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 IIIi processor:

- A FLUSH instruction flushes the processor pipeline and synchronizes the processor.
- The instruction cache is kept coherent; therefore, there is no need to perform any action on it.
- The address provided with the FLUSH instruction is ignored. However, for portability across all SPARC-V9 implementations, software must supply the target effective address in FLUSH instructions.

FLUSH synchronizes code and data spaces after code space is modified during program execution. The FLUSH effective address is ignored. FLUSH does not access the data MMU and cannot generate a data MMU miss or exception.

SPARC-V9 specifies that the FLUSH instruction has no latency on the issuing processor. In other words, a store to instruction space prior to the FLUSH instruction is visible immediately after the completion of FLUSH. When a FLUSH operation is performed, the processor guarantees that earlier code modifications will be visible across the whole system.

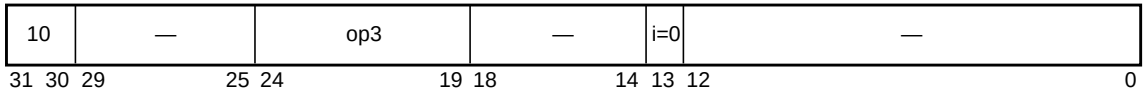
Exceptions

None

A.22 Flush Register Windows

Opcode	op3	Operation
FLUSHW	10 1011	Flush Register Windows

Format (3)



Assembly Language Syntax	
flushw	

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 switch memory stacks or to examine register contents for previous stack frames.

FLUSHW acts as a NOP if CANSAVE = NWINDOWS – 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** NWINDOWS).

Programming Note – Typically, the spill handler saves a window on a memory stack and returns to re-execute the `FLUSHW` instruction. Thus, `FLUSHW` traps and re-executes until all active windows other than the current window have been spilled.

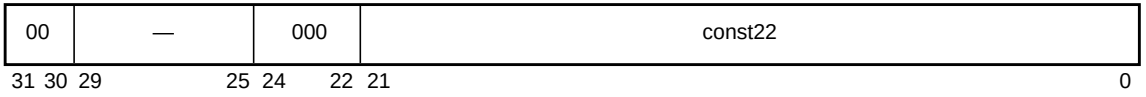
Exceptions

spill_n_normal
spill_n_other

A.23 Illegal Instruction Trap

Opcode	op	op2	Operation
`ILLTRAP`	00	000	*illegal_instruction* trap

Format (2)



Assembly Language Syntax	
`illtrap`	*const22*

Description

The `ILLTRAP` instruction causes an *illegal_instruction* exception. The `const22` value is ignored by the hardware; specifically, this field is *not* reserved by the architecture for any future use.

Compatibility Note – Except for its name, this instruction is identical to the SPARC-V8 `UNIMP` instruction.

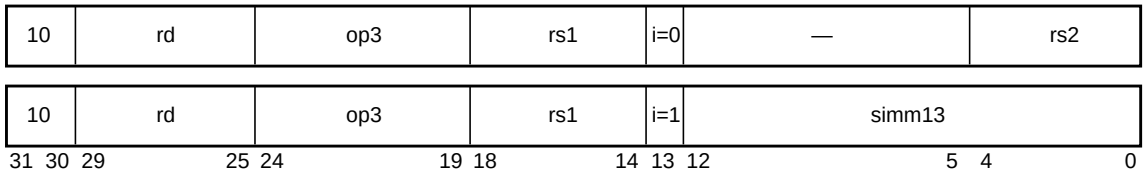
Exceptions

illegal_instruction

A.24 Jump and Link

Opcode	op3	Operation
JMPL	11 1000	Jump and Link

Format (3)



Assembly Language Syntax	
jmpl	*address*, reg_{rd}

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].

If either of the low-order two bits of the jump address is nonzero, a *mem_address_not_aligned* exception occurs.

Programming Note – 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.

Exceptions

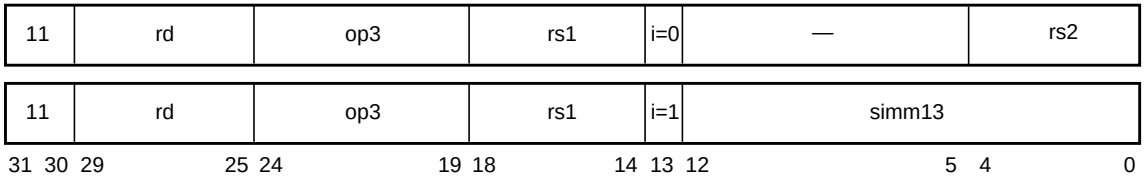
mem_address_not_aligned

A.25 Load Floating-Point

Opcode	op3	rd	Operation
LDF	10 0000	0–31	Load Floating-Point Register
LDDF	10 0011	†	Load Double Floating-Point Register
LDQF	10 0010	†	Load Quad Floating-Point Register
LDXFSR	10 0001	1	Load Floating-Point State Register
—	10 0001	2–31	*Reserved*

† Encoded floating-point register value.

Format (3)



Assembly Language Syntax	
`ld`	[address], $freg_{rd}$
`ldd`	[address], $freg_{rd}$
`ldq`	[address], $freg_{rd}$
`ldx`	[address], `%fsr`

Description

The load single floating-point instruction (`LDF`) copies a word from memory into `f[rd]`.

The load doubleword floating-point instruction (`LDDF`) copies a word-aligned doubleword from memory into a double-precision floating-point register.

The load quad floating-point instruction (`LDQF`) traps to software.

The load floating-point state register instruction (`LDXFSR`) waits for all FPop instructions that have not finished execution to complete and then loads a doubleword from memory into the `FSR`.

Load floating-point instructions access the primary address space (ASI = 80_{16}). The effective address for these instructions is "`r[rs1]` + `r[rs2]`" if `i` = 0, or "`r[rs1]` + `sign_ext(simm13)`" if `i` = 1.

`LDF` causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned. `LDXFSR` causes a *mem_address_not_aligned* exception if the address is not doubleword aligned. If the floating-point unit is not enabled (per `FPRS.FEF` and `PSTATE.PEF`) or if no FPU is present, then a load floating-point instruction causes an *fp_disabled* exception.

`LDDF` requires doubleword aligned. If word alignment is used, then the `LDDF` causes an *LDDF_mem_address_not_aligned* exception. The trap handler software shall emulate the `LDDF` instruction and return.

Programming Note – In SPARC-V8, some compilers issued sequences of single-precision loads when they could not determine that doubleword or quadword operands were properly aligned. For SPARC-V9, since emulation of misaligned loads is expected to be fast, compilers are recommended to issue sets of single-precision loads only when they can determine that doubleword or quadword operands are *not* properly aligned.

If a load floating-point instruction traps with any type of access error, the contents of the destination floating-point register(s) is undefined.

In the UltraSPARC IIIi processor, an LDDF instruction causes an *LDDF_mem_address_not_aligned* trap if the effective address is 32-bit aligned but not 64-bit (doubleword) aligned.

Exceptions

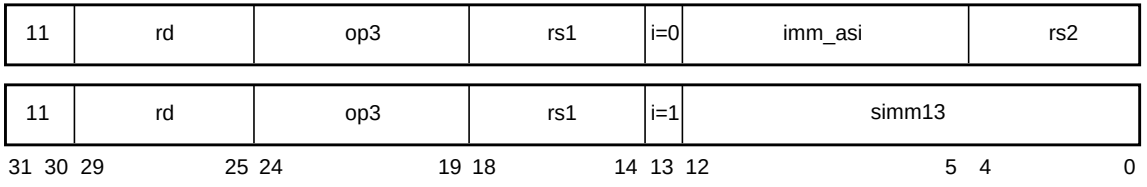
illegal_instruction (op3 = 21_{16} and rd = 2–31)
fp_disabled
LDDF_mem_address_not_aligned (LDDF only)
mem_address_not_aligned
data_access_exception
PA_watchpoint
VA_watchpoint
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.26 Load Floating-Point from Alternate Space

Opcode	op3	rd	Operation
LDFA^PASI	11 0000	0–31	Load Floating-Point Register from Alternate Space
LDDFA^PASI	11 0011	†	Load Double Floating-Point Register from Alternate Space
LDQFA^PASI	11 0010	†	Load Quad Floating-Point Register from Alternate Space

† Encoded floating-point register value.

Format (3)



Assembly Language Syntax	
lda	[*regaddr*] *imm_asi, freg$_{rd}$*
lda	[*reg_plus_imm*] %asi, *freg$_{rd}$*
ldda	[*regaddr*] *imm_asi, freg$_{rd}$*
ldda	[*reg_plus_imm*] %asi, *freg$_{rd}$*
ldqa	[*regaddr*] *imm_asi, freg$_{rd}$*
ldqa	[*reg_plus_imm*] %asi, *freg$_{rd}$*

Description

The load single floating-point from alternate space instruction (LDFA) copies a word from memory into f[rd].

The load double floating-point from alternate space instruction (LDDFA) copies a word-aligned doubleword from memory into a double-precision floating-point register.

The load quad floating-point from alternate space instruction (LDQFA) traps to software.

Load floating-point 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 zero; 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.

LDFA causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned. If the floating-point unit is not enabled (per FPRS.FEF and PSTATE.PEF) or if no FPU is present, then load floating-point from alternate space instructions cause an *fp_disabled* exception.

LDDFA with certain target ASIs is defined to be a 64-byte block-load instruction. See Section A.4, "Block Load and Block Store (VIS I)" for details.

Implementation Note – LDFA and LDDFA cause a *privileged_action* exception if PSTATE.PRIV = 0 and bit 7 of the ASI is zero.

LDDF requires doubleword alignment. If word alignment is used, then the LDDF causes an *LDDF_mem_address_not_aligned* exception. The trap handler software shall emulate the LDDF instruction and return.

Programming Note – In SPARC-V8, some compilers issued sequences of single-precision loads when they could not determine that doubleword or quadword operands were properly aligned. For SPARC-V9, 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.

If a load floating-point instruction traps with any type of access error, the contents of the destination floating-point register(s) is undefined.

In the UltraSPARC IIIi processor, an LDDFA instruction causes an *LDDF_mem_address_not_aligned* trap if the effective address is 32-bit aligned but not 64-bit (doubleword) aligned.

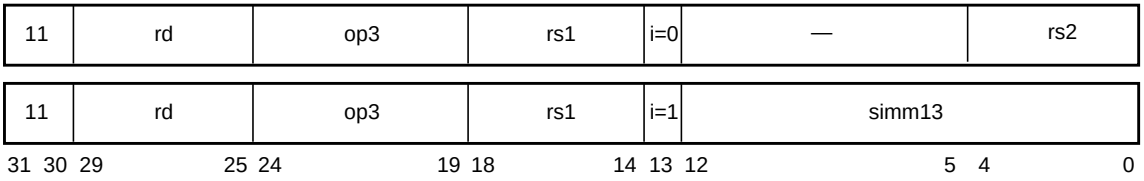
Exceptions

illegal_instruction (LDQFA only)
fp_disabled
LDDF_mem_address_not_aligned (LDDFA only)
mem_address_not_aligned
privileged_action
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
VA_watchpoint
PA_watchpoint

A.27 Load Integer

Opcode	op3	Operation
LDSB	00 1001	Load Signed Byte
LDSH	00 1010	Load Signed Halfword
LDSW	00 1000	Load Signed Word
LDUB	00 0001	Load Unsigned Byte
LDUH	00 0010	Load Unsigned Halfword
LDUW	00 0000	Load Unsigned Word
LDX	00 1011	Load Extended Word

Format (3)



Assembly Language Syntax		
`ldsb`	[address], reg_{rd}	
`ldsh`	[address], reg_{rd}	
`ldsw`	[address], reg_{rd}	
`ldub`	[address], reg_{rd}	
`lduh`	[address], reg_{rd}	
`lduw`	[address], reg_{rd}	(*synonym*: `ld`)
`ldx`	[address], reg_{rd}	

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 the primary address space (ASI = 80_{16}). 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.

`LDUH` and `LDSH` cause a *mem_address_not_aligned* exception if the address is not halfword aligned. `LDUW` and `LDSW` cause a *mem_address_not_aligned* exception if the effective address is not word aligned. `LDX` causes a *mem_address_not_aligned* exception if the address is not doubleword aligned.

Compatibility Note – The SPARC-V8 `LD` instruction has been renamed `LDUW` in SPARC-V9. The `LDSW` instruction is new in SPARC-V9.

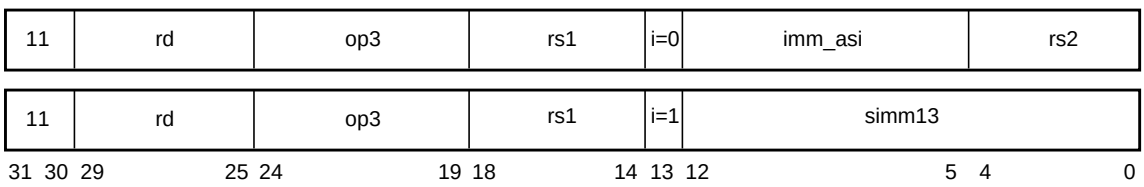
Exceptions

mem_address_not_aligned (all except LDSB, LDUB)
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
VA_watchpoint
PA_watchpoint

A.28 Load Integer from Alternate Space

Opcode	op3	Operation
LDSBA$^{P_{ASI}}$	01 1001	Load Signed Byte from Alternate Space
LDSHA$^{P_{ASI}}$	01 1010	Load Signed Halfword from Alternate Space
LDSWA$^{P_{ASI}}$	01 1000	Load Signed Word from Alternate Space
LDUBA$^{P_{ASI}}$	01 0001	Load Unsigned Byte from Alternate Space
LDUHA$^{P_{ASI}}$	01 0010	Load Unsigned Halfword from Alternate Space
LDUWA$^{P_{ASI}}$	01 0000	Load Unsigned Word from Alternate Space
LDXA$^{P_{ASI}}$	01 1011	Load Extended Word from Alternate Space

Format (3)



Assembly Language Syntax		
`ldsba`	[*regaddr*] *imm_asi*, reg_{rd}	
`ldsha`	[*regaddr*] *imm_asi*, reg_{rd}	
`ldswa`	[*regaddr*] *imm_asi*, reg_{rd}	
`lduba`	[*regaddr*] *imm_asi*, reg_{rd}	
`lduha`	[*regaddr*] *imm_asi*, reg_{rd}	
`lduwa`	[regaddr] imm_asi, reg_{rd}	(*synonym*: `lda`)
`ldxa`	[*regaddr*] *imm_asi*, reg_{rd}	
`ldsba`	[*reg_plus_imm*] `%asi`, reg_{rd}	
`ldsha`	[*reg_plus_imm*] `%asi`, reg_{rd}	
`ldswa`	[*reg_plus_imm*] `%asi`, reg_{rd}	
`lduba`	[*reg_plus_imm*] `%asi`, reg_{rd}	
`lduha`	[*reg_plus_imm*] `%asi`, reg_{rd}	
`lduwa`	[*reg_plus_imm*] `%asi`, reg_{rd}	(*synonym:* `lda`)
`ldxa`	[*reg_plus_imm*] `%asi`, reg_{rd}	

Description

The load integer from alternate space instructions copy a byte, halfword, 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 zero; 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.

`LDUHA` and `LDSHA` cause a *mem_address_not_aligned* exception if the address is not halfword aligned. `LDUWA` and `LDSWA` cause a *mem_address_not_aligned* exception if the effective address is not word aligned; `LDXA` causes a *mem_address_not_aligned* exception if the address is not doubleword aligned.

These instructions cause a *privileged_action* exception if `PSTATE.PRIV` = 0 and bit 7 of the ASI is zero.

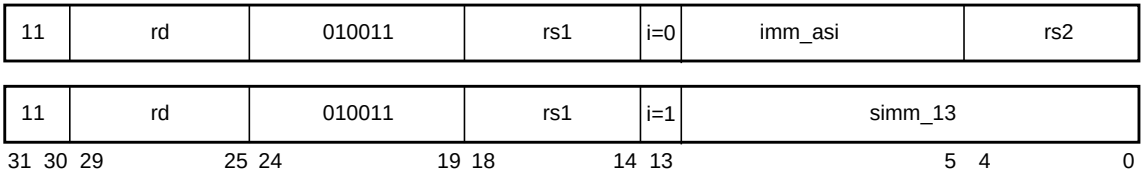
Exceptions

privileged_action
mem_address_not_aligned (all except LDSBA and LDUBA)
data_access_exception
PA_watchpoint
VA_watchpoint
fast_data_access_MMU_miss
fast_data_access_protection
data_access_error

A.29 Load Quadword, Atomic (VIS I)

Opcode	imm_asi	ASI Value	Operation
LDDA	ASI_NUCLEUS_QUAD_LDD	24_{16}	128-bit atomic load
LDDA	ASI_NUCLEUS_QUAD_LDD_L	$2C_{16}$	128-bit atomic load, little-endian
LDDA	ASI_QUAD_LDD_PHYS	34_{16}	128-bit atomic load
LDDA	ASI_QUAD_LDD_PHYS_L	$3C_{16}$	128-bit atomic load, little-endian

Format (3) LDDA



Assembly Language Syntax	
`ldda`	[*reg_addr*] *imm_asi*, reg_{rd}
`ldda`	[*reg_plus_imm*] `%asi`, reg_{rd}

Description

ASIs 24_{16} and $2C_{16}$ are used with the LDDA instruction to atomically read a 128-bit, virtually addressed data item. They are intended to be used by a TLB miss handler to access TSB entries without requiring locks. The data is placed in an even/odd pair of 64-bit registers. The lowest-address 64 bits are placed in the even register; the highest-address 64 bits are placed in the odd-numbered register. The reference is made from the nucleus context. ASIs 24_{16} and $2C_{16}$ are translated by the MMU into physical addresses according to normal translation rules for the nucleus context.

To reduce the number of locked pages in D-TLB a new ASI load instruction, atomic quad load physical (ldda ASI_QUAD_LDD_PHYS) was added. It allows a full TTE entry (128 bits, tag and data) in TSB to be read directly with PA, bypassing the VA-to-PA translation. In the D-TLB miss handler, a TTE entry is read using two ldx instructions. ASIs 34_{16} and $3C_{16}$ are not translated by the MMU and addresses provided are interpreted directly as physical addresses.

Since quad load with these ASIs bypasses the D-MMU, the physical address is set equal to the truncated virtual address, that is, PA[42:0] = VA[42:0]. Internally in hardware, the physical page attribute bits of these ASIs are hardcoded (not coming from DCU Control Register) as follows:

```
CP = 1, CV = 0, IE = 0, E = 0, P = 0, W = 0, NFO = 0, Size = 8K
```

Note that (CP, CV) = 10 means it is cacheable in L2-cache, W-cache, and P-cache, but not D-cache (since D-cache is VA-indexed). Therefore, this atomic quad load physical instruction can only be used with cacheable PA.

Semantically, ASI_QUAD_LDD_PHYS is like a combination of ASI_NUCLEUS_QUAD_LDD and ASI_PHYS_USE_EC.

An *illegal_instruction* occurs if an odd "rd" register number is used. If non-privileged software tries to use this ASI, a *privileged_action* exception occurs. If the physical address of the data referenced matches the watchpoint register (ASI_DMMU_PA_WATCHPOINT_REG), the *PA_watchpoint* exception occurs.

In addition to the usual traps for LDDA using a privileged ASI, a *data_access_exception* trap occurs for a non-cacheable access or if a quadword-load ASI is used with any instruction other than LDDA. A *mem_address_not_aligned* trap is taken if the access is not aligned on a 128-byte boundary.

Exceptions

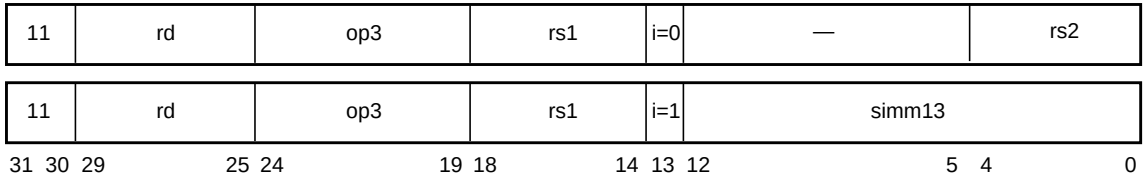
privileged_action
PA_watchpoint (recognized on only the first 8 bytes of an access)
VA_watchpoint (recognized on only the first 8 bytes of an access)
illegal_instruction (misaligned rd)
mem_address_not_aligned

data_access_exception (an attempt to access a page marked as non-cacheable)
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.30 Load-Store Unsigned Byte

Opcode	op3	Operation
LDSTUB	00 1101	Load-Store Unsigned Byte

Format (3)



Assembly Language Syntax	
ldstub	**[***address***],** reg_{rd}

Description

The load-store unsigned byte instruction copies a byte from memory into `r[rd]`, then rewrites the addressed byte in memory to all ones. 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 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.

The effective address for these instructions 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 processors and I/O DMA memory accesses is maintained for cacheable memory space.

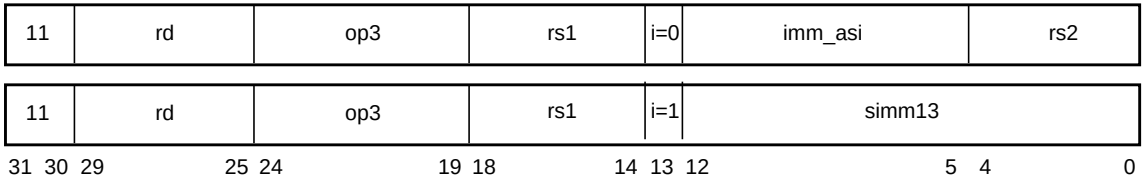
Exceptions

data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
VA_watchpoint
PA_watchpoint

A.31 Load-Store Unsigned Byte to Alternate Space

Opcode	op3	Operation
LDSTUBA^{PASI}	01 1101	Load-Store Unsigned Byte into Alternate Space

Format (3)



Assembly Language Syntax	
`ldstuba`	[*regaddr*] *imm_asi*, reg_{rd}
`ldstuba`	[*reg_plus_imm*] `%asi`, reg_{rd}

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 ones. 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 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.

LDSTUBA 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 zero; otherwise, it is not privileged. The effective address is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

LDSTUBA causes a *privileged_action* exception if PSTATE.PRIV = 0 and bit 7 of the ASI is zero.

The coherence and atomicity of memory operations between processors and I/O DMA memory accesses is maintained for cacheable memory space.

Exceptions

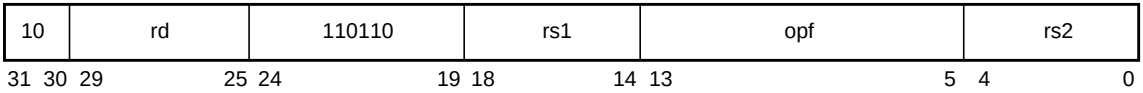
privileged_action
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
VA_watchpoint
PA_watchpoint

A.32 Logical Operate Instructions (VIS I)

Opcode	opf	Operation
FZERO	0 0110 0000	Zero fill
FZEROS	0 0110 0001	Zero fill, single precision
FONE	0 0111 1110	One fill
FONES	0 0111 1111	One fill, single precision
FSRC1	0 0111 0100	Copy src1
FSRC1S	0 0111 0101	Copy src1, single precision
FSRC2	0 0111 1000	Copy src2
FSRC2S	0 0111 1001	Copy src2, single precision
FNOT1	0 0110 1010	Negate (ones-complement) src1
FNOT1S	0 0110 1011	Negate (ones-complement) src1, single precision
FNOT2	0 0110 0110	Negate (ones-complement) src2
FNOT2S	0 0110 0111	Negate (ones-complement) src2, single precision
FOR	0 0111 1100	Logical OR
FORS	0 0111 1101	Logical OR, single precision
FNOR	0 0110 0010	Logical NOR
FNORS	0 0110 0011	Logical NOR, single precision
FAND	0 0111 0000	Logical AND
FANDS	0 0111 0001	Logical AND, single precision
FNAND	0 0110 1110	Logical NAND
FNANDS	0 0110 1111	Logical NAND, single precision
FXOR	0 0110 1100	Logical XOR
FXORS	0 0110 1101	Logical XOR, single precision
FXNOR	0 0111 0010	Logical XNOR
FXNORS	0 0111 0011	Logical XNOR, single precision
FORNOT1	0 0111 1010	Negated src1 OR src2
FORNOT1S	0 0111 1011	Negated src1 OR src2, single precision
FORNOT2	0 0111 0110	src1 OR negated src2
FORNOT2S	0 0111 0111	src1 OR negated src2, single precision
FANDNOT1	0 0110 1000	Negated src1 AND src2
FANDNOT1S	0 0110 1001	Negated src1 AND src2, single precision

Opcode	opf	Operation
FANDNOT2	0 0110 0100	src1 AND negated src2
FANDNOT2S	0 0110 0101	src1 AND negated src2, single precision

Format (3)



Assembly Language Syntax	
fzero	$freg_{rd}$
fzeros	$freg_{rd}$
fone	$freg_{rd}$
fones	$freg_{rd}$
fsrc1	$freg_{rs1}$, $freg_{rd}$
fsrc1s	$freg_{rs1}$, $freg_{rd}$
fsrc2	$freg_{rs2}$, $freg_{rd}$
fsrc2s	$freg_{rs2}$, $freg_{rd}$
fnot1	$freg_{rs1}$, $freg_{rd}$
fnot1s	$freg_{rs1}$, $freg_{rd}$
fnot2	$freg_{rs2}$, $freg_{rd}$
fnot2s	$freg_{rs2}$, $freg_{rd}$
for	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fnor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fnors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fand	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fand	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fnands	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fnands	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fxor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fxors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fxnor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fxnors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Assembly Language Syntax	
`fornot1`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fornot1s`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fornot2`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fornot2s`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fandnot1`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fandnot1s`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fandnot2`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fandnot2s`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

The standard 64-bit versions of these instructions perform 1 of 16 64-bit logical operations between the 64-bit floating-point registers specified by `rs1` and `rs2`. The result is stored in the 64-bit floating-point destination register specified by `rd`. The 32-bit (single-precision) version of these instructions perform 32-bit logical operations.

Note – For good performance, the result of a single logical instruction should not be used as part of a 64-bit graphics instruction source operand in the next three instruction groups. Similarly, the result of a standard logical should not be used as a 32-bit graphics instruction source operand in the next three instruction groups.

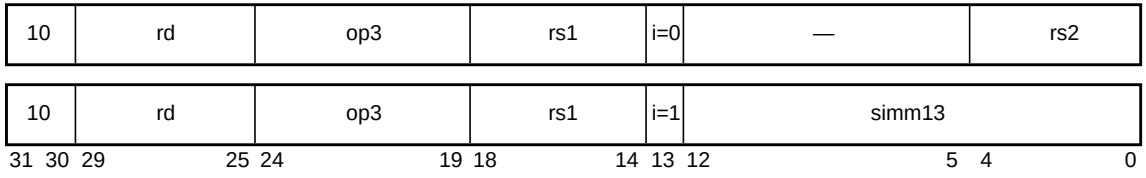
Exceptions

fp_disabled

A.33 Logical Operations

Opcode	op3	Operation
AND	00 0001	AND
ANDcc	01 0001	AND and modify condition codes
ANDN	00 0101	AND Not
ANDNcc	01 0101	AND Not and modify condition codes
OR	00 0010	Inclusive OR
ORcc	01 0010	Inclusive OR and modify condition codes
ORN	00 0110	Inclusive OR Not
ORNcc	01 0110	Inclusive OR Not and modify condition codes
XOR	00 0011	Exclusive OR
XORcc	01 0011	Exclusive OR and modify condition codes
XNOR	00 0111	Exclusive NOR
XNORcc	01 0111	Exclusive NOR and modify condition codes

Format (3)



Assembly Language Syntax	
`and`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`andcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`andn`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`andncc`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`or`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`orcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`orn`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`orncc`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`xor`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`xorcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`xnor`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`xnorcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}

Description

These instructions implement bitwise logical 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`, `ANDNcc`, `ORcc`, `ORNcc`, `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` to zero
- `icc.n` to bit 31 of the result
- `xcc.n` to bit 63 of the result
- `icc.z` to one if bits 31:0 of the result are zero (otherwise to zero)
- `xcc.z` to one if all 64 bits of the result are zero (otherwise to zero)

`ANDN`, `ANDNcc`, `ORN`, and `ORNcc` logically negate their second operand before applying the main (AND or OR) operation.

Programming Note – XNOR and XNORcc are identical to the XOR-Not and XOR-Not-cc logical operations, respectively.

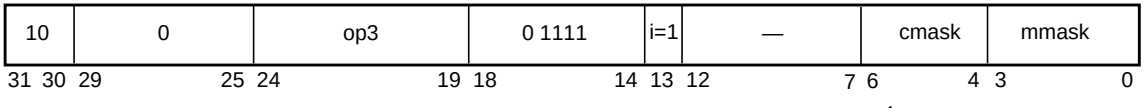
Exceptions

None

A.34 Memory Barrier

Opcode	op3	Operation
MEMBAR	10 1000	Memory Barrier

Format (3)



Assembly Language Syntax	
membar	*membar_mask*

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 CASXA 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. MEMBAR applies to all memory operations in all address spaces referenced by the issuing processor, but it has no effect on memory references by other 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 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 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 A-9 specifies the order constraint that each bit of mmask (selected when set to one) imposes on memory references appearing before and after the MEMBAR. From zero to four, mask bits may be selected in the mmask field.

TABLE A-9 MEMBAR mmask Encodings

Mask Bit	Name	Description
mmask<3>	#StoreStore	The effects of all stores appearing prior to the MEMBAR instruction must be visible to all processors before the effect of any stores following the MEMBAR; it is equivalent to the deprecated STBAR instruction.
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 processor.
mmask<1>	#StoreLoad	The effects of all stores appearing prior to the MEMBAR instruction must be visible to all 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 A-10, 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 A-10 MEMBAR cmask Encodings

Mask Bit	Function	Name	Description
cmask[2]	Synchronization barrier	#Sync	All operations (including non-memory 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.

TABLE A-10 MEMBAR cmask Encodings *(Continued)*

Mask Bit	Function	Name	Description
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.

The encoding of MEMBAR is identical to that of the RDASR instruction, except that rs1 = 15, rd = 0, and i = 1.

The coherence and atomicity of memory operations between processors and I/O DMA memory accesses is maintained for cacheable memory space.

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.

The information included in this section should not be used for the decision as to when MEMBARs should be added to software that needs to be compliant across all UltraSPARC-based platforms. The operations of block load/block store (BLD/BST) on the UltraSPARC IIIi processor are generally more ordered with respect to other operations, compared to the UltraSPARC I processor and the UltraSPARC II processor. Code written and found to "work" on the UltraSPARC IIIi processor may not work on the UltraSPARC I processor and the UltraSPARC II processor if it does not follow the rules for BLD/BST specified for those processors. Code that happens to work on the UltraSPARC I processor and the UltraSPARC II processor may not work on the UltraSPARC IIIi processor if it did not meet the coding guidelines specified for those processors. In no case is the coding requirement for the UltraSPARC IIIi processor more restrictive than that for the UltraSPARC I and the UltraSPARC II processors.

Software developers should not use the information in this section for determining the need for MEMBARs but instead should rely on the SPARC-V9 MEMBAR rules. These UltraSPARC IIIi processor rules are less restrictive than SPARC-V9, UltraSPARC I processor, and the UltraSPARC II processor rules and are never more restrictive.

MEMBAR Rules

The UltraSPARC IIIi hardware uses the following rules to guide the interlock implementation.

1. Non-cacheable load or store with side-effect bit on will always be blocked.
2. Cacheable or non-cacheable BLD will not be blocked.

3. VA<12:5> of a load (cacheable or non-cacheable) will be compared with the VA<12:5> of all entries in Store Queue. When a matching is detected, this load (cacheable or non-cacheable) will be blocked.

4. An insertion of MEMBAR is required if Strong Ordering is desired while not fitting rules 1 to 3.

TABLE A-11 and TABLE A-12 reflect the hardware interlocking mechanism implemented in the UltraSPARC IIIi processor. The tables are read from Row to Column, the first memory operation in program order being in Row followed by the memory operation found in Column. The following two symbols are used as table entries:

- # — No intervening operation required because Fireplane-compliant systems automatically order R before C.
- M — MEMBAR #Sync *or* MEMBAR #MemIssue *or* MEMBAR #StoreLoad required.

For VA<12:5> of a column operation not matching with VA<2:5> of a row operation while a strong ordering is desired, the MEMBAR rules summarized in TABLE A-11 reflect the UltraSPARC IIIi processor's hardware implementation.

TABLE A-11 MEMBAR Rules for Column VA <12:5> ≠ Row VA <12:5> While Desiring Strong Ordering

	To Column Operation C:													
From Row Operation R:	**load**	**load from internal ASI**	**store**	**store to internal ASI**	**atomic**	**load_nc_e**	**store_nc_e**	**load_nc_ne**	**store_nc_ne**	**bload**	**bstore**	**bstore_commit**	**bload_nc**	**bstore_nc**
load	#	#	#	#	#	#	#	#	#	M	M	#	M	M
load from internal ASI	#	#	#	#	#	#	#	#	#	#	#	#	#	#
store	M	#	#	#	#	M	#	M	#	M	M	#	M	M
store to internal ASI	#	M	#	#	#	#	#	#	#	M	#	#	M	M
atomic	#	#	#	#	#	#	#	#	#	M	M	#	M	M
load_nc_e	#	#	#	#	#	#	#	#	#	M	M	#	M	M
store_nc_e	M	#	#	#	#	#	#	M	#	M	M	#	M	M
load_nc_ne	#	#	#	#	#	#	#	#	#	M	M	#	M	M
store_nc_ne	M	#	#	#	#	M	#	M	#	M	M	#	M	M
bload	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bstore	M	#	M	#	M	M	M	M	M	M	M	#	M	M

TABLE A-11 MEMBAR Rules for Column VA <12:5> ≠ Row VA <12:5> While Desiring Strong Ordering *(Continued)*

From Row Operation R:	To Column Operation C: load	load from internal ASI	store	store to internal ASI	atomic	load_nc_e	store_nc_e	load_nc_ne	store_nc_ne	bload	bstore	bstore_commit	bload_nc	bstore_nc
bstore_commit	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bload_nc	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bstore_nc	M	#	M	#	M	M	M	M	M	M	M	#	M	M

When VA<12:5> of a column operation matches VA<12:5> of a row operation, the MEMBAR rules summarized in TABLE A-12 reflect the UltraSPARC IIIi's hardware implementation.

TABLE A-12 MEMBAR Rules for Column VA<12:5> = Row VA<12:5> While Desiring Strong Ordering

From Row Operation R:	To Column Operation C: load	load from internal ASI	store	store to internal ASI	atomic	load_nc_e	store_nc_e	load_nc_ne	store_nc_ne	bload	bstore	bstore_commit	bload_nc	bstore_nc
load	#	#	#	#	#	#	#	#	#	#	#	#	#	#
load from internal ASI	#	#	#	#	#	#	#	#	#	#	#	#	#	#
store	#	#	#	#	#	#	#	#	#	M	#	#	#	#
store to internal ASI	#	M	#	#	#	#	#	#	#	M	#	#	M	M
atomic	#	#	#	#	#	#	#	#	#	#	#	#	#	#
load_nc_e	#	#	#	#	#	#	#	#	#	#	#	#	#	#
store_nc_e	#	#	#	#	#	#	#	#	#	M	#	#	M	#
load_nc_ne	#	#	#	#	#	#	#	#	#	#	#	#	#	#
store_nc_ne	#	#	#	#	#	#	#	#	#	M	#	#	M	#
bload	#	#	#	#	#	#	#	#	#	#	#	#	#	#
bstore	#	#	#	#	#	#	#	#	#	M	#	#	#	#

TABLE A-12 MEMBAR Rules for Column VA<12:5> = Row VA<12:5> While Desiring Strong Ordering *(Continued)*

		To Column Operation C:												
From Row Operation R:	**load**	**load from internal ASI**	**store**	**store to internal ASI**	**atomic**	**load_nc_e**	**store_nc_e**	**load_nc_ne**	**store_nc_ne**	**bload**	**bstore**	**bstore_commit**	**bload_nc**	**bstore_nc**
bstore_commit	M	#	M	#	M	M	M	M	M	M	M	#	M	M
bload_nc	#	#	#	#	#	#	#	#	#	#	#	#	#	#
bstore_nc	#	#	#	#	#	#	#	#	#	#	#	#	M	#

Special Rules for Quad LDD (ASI 24_{16} and ASI $2C_{16}$)

MEMBAR is only required before quad LDD if VA<12:5> of a preceding store to the same address space matches VA<12:5> of the quad LDD.

Exceptions

None

A.35 Move Floating-Point Register on Condition (FMOVcc)

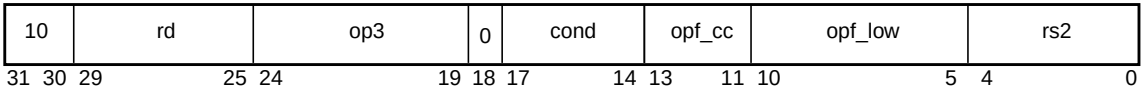
For Integer Condition Codes

Opcode	op3	cond	Operation	*icc*/*xcc* Test
FMOVA	11 0101	1000	Move Always	1
FMOVN	11 0101	0000	Move Never	0
FMOVNE	11 0101	1001	Move if Not Equal	**not** Z
FMOVE	11 0101	0001	Move if Equal	Z
FMOVG	11 0101	1010	Move if Greater	**not** (Z **or** (N **xor** V))
FMOVLE	11 0101	0010	Move if Less or Equal	Z **or** (N **xor** V)
FMOVGE	11 0101	1011	Move if Greater or Equal	**not** (N **xor** V)
FMOVL	11 0101	0011	Move if Less	N **xor** V
FMOVGU	11 0101	1100	Move if Greater Unsigned	**not** (C **or** Z)
FMOVLEU	11 0101	0100	Move if Less or Equal Unsigned	(C **or** Z)
FMOVCC	11 0101	1101	Move if Carry Clear (Greater or Equal, Unsigned)	**not** C
FMOVCS	11 0101	0101	Move if Carry Set (Less than, Unsigned)	C
FMOVPOS	11 0101	1110	Move if Positive	**not** N
FMOVNEG	11 0101	0110	Move if Negative	N
FMOVVC	11 0101	1111	Move if Overflow Clear	**not** V
FMOVVS	11 0101	0111	Move if Overflow Set	V

For Floating-Point Condition Codes

Opcode	op3	cond	Operation	*fcc* Test
FMOVFA	11 0101	1000	Move Always	1
FMOVFN	11 0101	0000	Move Never	0
FMOVFU	11 0101	0111	Move if Unordered	U
FMOVFG	11 0101	0110	Move if Greater	G
FMOVFUG	11 0101	0101	Move if Unordered or Greater	G **or** U
FMOVFL	11 0101	0100	Move if Less	L
FMOVFUL	11 0101	0011	Move if Unordered or Less	L **or** U
FMOVFLG	11 0101	0010	Move if Less or Greater	L **or** G
FMOVFNE	11 0101	0001	Move if Not Equal	L **or** G **or** U
FMOVFE	11 0101	1001	Move if Equal	E
FMOVFUE	11 0101	1010	Move if Unordered or Equal	E **or** U
FMOVFGE	11 0101	1011	Move if Greater or Equal	E **or** G
FMOVFUGE	11 0101	1100	Move if Unordered or Greater or Equal	E **or** G **or** U
FMOVFLE	11 0101	1101	Move if Less or Equal	E **or** L
FMOVFULE	11 0101	1110	Move if Unordered or Less or Equal	E **or** L **or** U
FMOVFO	11 0101	1111	Move if Ordered	E **or** L **or** G

Format (4)



Encoding of the opf_cc Field

opf_cc	Condition Code
000	fcc0
001	fcc1
010	fcc2
011	fcc3
100	icc
101	—
110	xcc
111	—

Encoding of opf Field (opf_cc opf_low)

Instruction Variation		opf_cc	opf_low	opf
FMOVScc	%fcc*n*,*rs2*,*rd*	0*nn*	00 0001	0 *nn*00 0001
FMOVDcc	%fcc*n*,*rs2*,*rd*	0*nn*	00 0010	0 *nn*00 0010
FMOVQcc	%fcc*n*,*rs2*,*rd*	0*nn*	00 0011	0 *nn*00 0011
FMOVScc	%icc, *rs2*,*rd*	100	00 0001	1 0000 0001
FMOVDcc	%icc, *rs2*,*rd*	100	00 0010	1 0000 0010
FMOVQcc	%icc, *rs2*,*rd*	100	00 0011	1 0000 0011
FMOVScc	%xcc, *rs2*,*rd*	110	00 0001	1 1000 0001
FMOVDcc	%xcc, *rs2*,*rd*	110	00 0010	1 1000 0010
FMOVQcc	%xcc, *rs2*,*rd*	110	00 0011	1 1000 0011

For Integer Condition Codes

Assembly Language Syntax		
`fmov{s,d,q}a`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}n`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}ne`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	(*synonyms:* `fmov{s,d,q}nz`)
`fmov{s,d,q}e`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	(*synonyms:* `fmov{s,d,q}z`)
`fmov{s,d,q}g`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}le`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}ge`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}l`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}gu`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}leu`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}cc`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	(*synonyms*: `fmov{s,d,q}geu`)
`fmov{s,d,q}cs`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	(*synonyms*: `fmov{s,d,q}lu`)
`fmov{s,d,q}pos`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}neg`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}vc`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}vs`	*i_or_x_cc*, $freg_{rs2}$, $freg_{rd}$	

Programming Note – To select the appropriate condition code, include `%icc` or `%xcc` before the registers.

For Floating-Point Condition Codes

Assembly Language Syntax		
`fmov{s,d,q}a`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}n`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}u`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}g`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}ug`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}l`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}ul`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}lg`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}ne`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	(*synonyms:* `fmov{s,d,q}nz`)
`fmov{s,d,q}e`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	(*synonyms:* `fmov{s,d,q}z`)
`fmov{s,d,q}ue`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}ge`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}uge`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}le`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}ule`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	
`fmov{s,d,q}o`	`%fcc`*n*, $freg_{rs2}$, $freg_{rd}$	

Description

These 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 condition code. 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 do not modify any condition codes.

Programming Note – In general, branches cause the processor's performance to degrade. 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 only executed if the branch is 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.

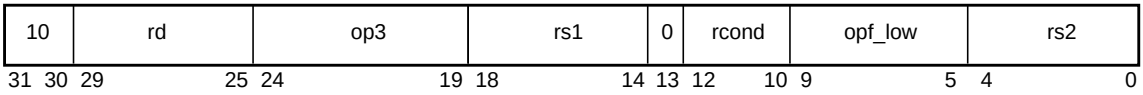
Exceptions

fp_disabled
fp_exception_other (ftt = *unimplemented_FPop* (opf_cc = 101_2 or 111_2 and quad forms))

A.36 Move Floating-Point Register on Integer Register Condition (FMOVr)

Opcode	op3	rcond	Operation	Test
—	11 0101	000	*Reserved*	—
FMOVRZ	11 0101	001	Move if Register Zero	r[rs1] = 0
FMOVRLEZ	11 0101	010	Move if Register Less Than or Equal to Zero	r[rs1] ≤ 0
FMOVRLZ	11 0101	011	Move if Register Less Than Zero	r[rs1] < 0
—	11 0101	100	*Reserved*	—
FMOVRNZ	11 0101	101	Move if Register Not Zero	r[rs1] ≠ 0
FMOVRGZ	11 0101	110	Move if Register Greater Than Zero	r[rs1] > 0
FMOVRGEZ	11 0101	111	Move if Register Greater Than or Equal to Zero	r[rs1] ≥ 0

Format (4)



Encoding of opf_low Field

Instruction variation	opf_low
FMOVS*rcond* *rs1*, *rs2*, *rd*	0 0101
FMOVD*rcond* *rs1*, *rs2*, *rd*	0 0110
FMOVQ*rcond* *rs1*, *rs2*, *rd*	0 0111

Assembly Language Syntax		
`fmovr{s,d,q}e`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	(*synonym:* `fmovr{s,d,q}z`)
`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}ne`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	(*synonym:* `fmovr{s,d,q}nz`)
`fmovr{s,d,q}gz`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	
`fmovr{s,d,q}gez`	*reg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	

Description

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

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

Implementation Note – The UltraSPARC IIIi processor does not implement this instruction by tagging each register value. The UltraSPARC IIIi processor looks at the full 64-bit register to determine a negative or zero.

Exceptions

fp_disabled
fp_exception_other (*unimplemented_FPop* (`rcond` = 000_2 or 100_2 and quad forms))

A.37 Move Integer Register on Condition (MOVcc)

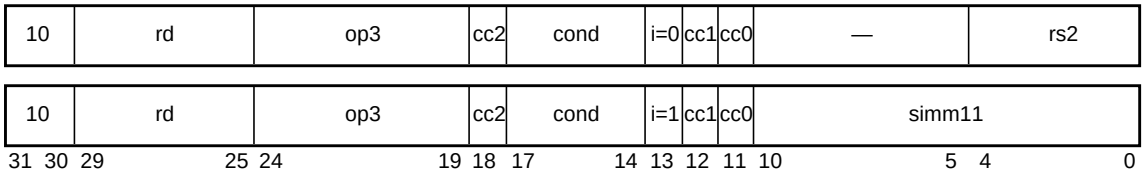
For Integer Condition Codes

Opcode	op3	cond	Operation	icc/xcc Test
MOVA	10 1100	1000	Move Always	1
MOVN	10 1100	0000	Move Never	0
MOVNE	10 1100	1001	Move if Not Equal	**not** Z
MOVE	10 1100	0001	Move if Equal	Z
MOVG	10 1100	1010	Move if Greater	**not** (Z **or** (N **xor** V))
MOVLE	10 1100	0010	Move if Less or Equal	Z **or** (N **xor** V)
MOVGE	10 1100	1011	Move if Greater or Equal	**not** (N **xor** V)
MOVL	10 1100	0011	Move if Less	N **xor** V
MOVGU	10 1100	1100	Move if Greater Unsigned	**not** (C **or** Z)
MOVLEU	10 1100	0100	Move if Less or Equal Unsigned	(C **or** Z)
MOVCC	10 1100	1101	Move if Carry Clear (Greater or Equal, Unsigned)	**not** C
MOVCS	10 1100	0101	Move if Carry Set (Less than, Unsigned)	C
MOVPOS	10 1100	1110	Move if Positive	**not** N
MOVNEG	10 1100	0110	Move if Negative	N
MOVVC	10 1100	1111	Move if Overflow Clear	**not** V
MOVVS	10 1100	0111	Move if Overflow Set	V

For Floating-Point Condition Codes

Opcode	op3	cond	Operation	fcc Test
MOVFA	10 1100	1000	Move Always	1
MOVFN	10 1100	0000	Move Never	0
MOVFU	10 1100	0111	Move if Unordered	U
MOVFG	10 1100	0110	Move if Greater	G
MOVFUG	10 1100	0101	Move if Unordered or Greater	G **or** U
MOVFL	10 1100	0100	Move if Less	L
MOVFUL	10 1100	0011	Move if Unordered or Less	L **or** U
MOVFLG	10 1100	0010	Move if Less or Greater	L **or** G
MOVFNE	10 1100	0001	Move if Not Equal	L **or** G **or** U
MOVFE	10 1100	1001	Move if Equal	E
MOVFUE	10 1100	1010	Move if Unordered or Equal	E **or** U
MOVFGE	10 1100	1011	Move if Greater or Equal	E **or** G
MOVFUGE	10 1100	1100	Move if Unordered or Greater or Equal	E **or** G **or** U
MOVFLE	10 1100	1101	Move if Less or Equal	E **or** L
MOVFULE	10 1100	1110	Move if Unordered or Less or Equal	E **or** L **or** U
MOVFO	10 1100	1111	Move if Ordered	E **or** L **or** G

Format (4)



cc2	cc1	cc0	Condition Code
000			fcc0
001			fcc1
010			fcc2
011			fcc3
100			icc
101			*Reserved*
110			xcc
111			*Reserved*

For Integer Condition Codes

Assembly Language Syntax		
`mova`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movn`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movne`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	(*synonym:* `movnz`)
`move`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	(*synonym:* `movz`)
`movg`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movle`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movge`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movl`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movgu`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movleu`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movcc`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	(*synonym:* `movgeu`)
`movcs`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	(*synonym:* `movlu`)
`movpos`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movneg`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movvc`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	
`movvs`	*i_or_x_cc, reg_or_imm11,* reg_{rd}	

Programming Note – To select the appropriate condition code, include `%icc` or `%xcc` before the register or immediate field.

For Floating-Point Condition Codes

Assembly Language Syntax		
`mova`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movn`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movu`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movg`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movug`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movl`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movul`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movlg`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movne`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	(*synonym:* `movnz`)
`move`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	(*synonym*: `movz`)
`movue`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movge`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movuge`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movle`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movule`	`%fcc`*n*, *reg_or_imm11*, reg_{rd}	
`movo`	`%fccn`, *reg_or_imm11*, reg_{rd}	

Programming Note – To select the appropriate condition code, include `%fcc0`, `%fcc1`, `%fcc2`, or `%fcc3` before the register or immediate field.

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 the `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 – In general, branches cause the processor performance to degrade. Frequently, the MOVcc and FMOVcc instructions can be used to avoid branches. For example, consider 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:...
```

This takes 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.

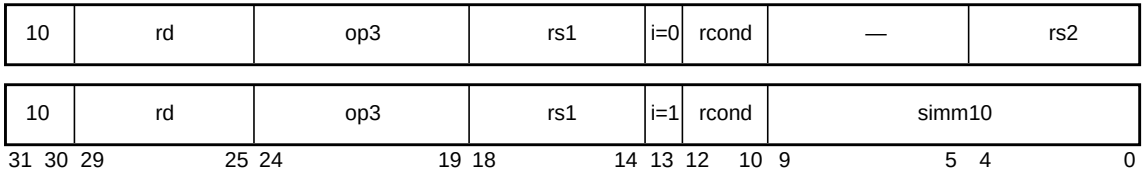
Exceptions

illegal_instruction (cc2 || cc1 || cc0 = 101_2 or 111_2)
fp_disabled (cc2 || cc1 || cc0 = 000_2, 001_2, 010_2, or 011_2 and the FPU is disabled)

A.38 Move Integer Register on Register Condition (MOVr)

Opcode	op3	rcond	Operation	Test
—	10 1111	000	*Reserved*	—
MOVRZ	10 1111	001	Move if Register Zero	r[rs1] = 0
MOVRLEZ	10 1111	010	Move if Register Less Than or Equal to Zero	r[rs1] ≤ 0
MOVRLZ	10 1111	011	Move if Register Less Than Zero	r[rs1] < 0
—	10 1111	100	*Reserved*	—
MOVRNZ	10 1111	101	Move if Register Not Zero	r[rs1] ≠ 0
MOVRGZ	10 1111	110	Move if Register Greater Than Zero	r[rs1] > 0
MOVRGEZ	10 1111	111	Move if Register Greater Than or Equal to Zero	r[rs1] ≥ 0

Format (3)



Assembly Language Syntax		
movrz	reg_{rs1}, *reg_or_imm10*, reg_{rd}	(*synonym:* movre)
movrlez	reg_{rs1}, *reg_or_imm10*, reg_{rd}	
movrlz	reg_{rs1}, *reg_or_imm10*, reg_{rd}	
movrnz	reg_{rs1}, *reg_or_imm10*, reg_{rd}	(*synonym:* movrne)
movrgz	reg_{rs1}, *reg_or_imm10*, reg_{rd}	
movrgez	reg_{rs1}, *reg_or_imm10*, reg_{rd}	

Description

If the contents of integer register `r[rs1]` satisfy the condition specified in the `rcond` field, these instructions copy `r[rs2]` (if `i` = 0) or `sign_ext(simm10)` (if `i` = 1) 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.

Implementation Note – The UltraSPARC IIIi processor does not implement this instruction by tagging each register value. The UltraSPARC IIIi processor looks at the full 64-bit register to determine a negative or zero.

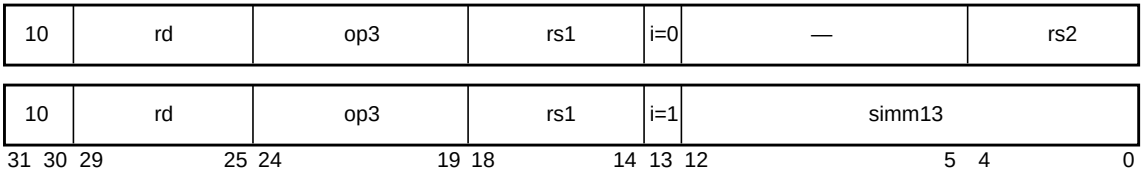
Exceptions

illegal_instruction (`rcond` = 000_2 or 100_2)

A.39 Multiply and Divide (64-bit)

Opcode	op3	Operation
`MULX`	00 1001	Multiply (signed or unsigned)
`SDIVX`	10 1101	Signed Divide
`UDIVX`	00 1101	Unsigned Divide

Format (3)



Assembly Language Syntax	
`mulx`	*reg_{rs1}, reg_or_imm, reg_{rd}*
`sdivx`	*reg_{rs1}, reg_or_imm, reg_{rd}*
`udivx`	*reg_{rs1}, reg_or_imm, reg_{rd}*

Description

`MULX` computes "`r[rs1]` × `r[rs2]`" if `i` = 0 or "`r[rs1]` × `sign_ext(simm13)`" if `i` = 1, and writes the 64-bit product into `r[rd]`. `MULX` can be used to calculate the 64-bit product for signed or unsigned operands (the product is the same).

`SDIVX` and `UDIVX` compute "`r[rs1]` ÷ `r[rs2]`" if `i` = 0 or "`r[rs1]` ÷ `sign_ext(simm13)`" if `i` = 1, and write the 64-bit result into `r[rd]`. `SDIVX` operates on the operands as signed integers and produces a corresponding signed result. `UDIVX` operates on the operands as unsigned integers and produces a corresponding unsigned result.

For `SDIVX`, if the largest negative number is divided by –1, the result should be the largest negative number. That is:

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

These instructions do not modify any condition codes.

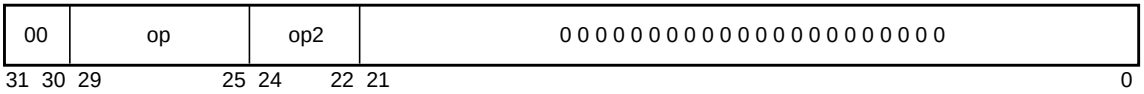
Exceptions

division_by_zero

A.40 No Operation

Opcode	op	op2	Operation
`NOP`	0 0000	100	No Operation

Format (2)



Assembly Language Syntax	
nop	

Description

The NOP instruction changes no program-visible state (except that of the PC and nPC).

NOP is a special case of the SETHI instruction, with imm22 = 0 and rd = 0.

Exceptions

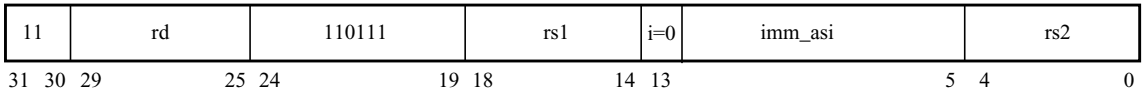
None

A.41 Partial Store (VIS I)

Opcode	imm_asi	ASI Value	Operation
STDFA	ASI_PST8_P	$C0_{16}$	Eight 8-bit conditional stores to primary address space
STDFA	ASI_PST8_S	$C1_{16}$	Eight 8-bit conditional stores to secondary address space
STDFA	ASI_PST8_PL	$C8_{16}$	Eight 8-bit conditional stores to primary address space, little-endian
STDFA	ASI_PST8_SL	$C9_{16}$	Eight 8-bit conditional stores to secondary address space, little-endian
STDFA	ASI_PST16_P	$C2_{16}$	Four 16-bit conditional stores to primary address space
STDFA	ASI_PST16_S	$C3_{16}$	Four 16-bit conditional stores to secondary address space
STDFA	ASI_PST16_PL	CA_{16}	Four 16-bit conditional stores to primary address space, little-endian
STDFA	ASI_PST16_SL	CB_{16}	Four 16-bit conditional stores to secondary address space, little-endian
STDFA	ASI_PST32_P	$C4_{16}$	Two 32-bit conditional stores to primary address space
STDFA	ASI_PST32_S	$C5_{16}$	Two 32-bit conditional stores to secondary address space

Opcode	imm_asi	ASI Value	Operation
STDFA	ASI_PST32_PL	CC_{16}	Two 32-bit conditional stores to primary address space, little-endian
STDFA	ASI_PST32_SL	CD_{16}	Two 32-bit conditional stores to secondary address space, little-endian

Format (3)



Assembly Language Syntax[1]	
stda	*freg*$_{rd}$, *reg*$_{rs2}$, [*reg*$_{rs1}$] *imm_asi*

1. 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 assemblers may recognize only the original syntax.

Description

The partial store instructions are selected by one of the partial store ASIs with the STDFA instruction.

Two 32-bit, four 16-bit, or eight 8-bit values from the 64-bit floating-point register specified by rd are conditionally stored at the address specified by r[rs1], using the mask specified in r[rs2]. The value in r[rs2] has the same format as the result specified by the pixel compare instructions (see Section A.44, "Pixel Compare (VIS I)"). The most significant bit of the mask (not the entire register) corresponds to the most significant part of the floating-point register specified by rd. The data is stored in little-endian form in memory if the ASI name has an "L" suffix; otherwise, it is stored in big-endian format.

A partial store instruction can cause a virtual (or physical) watchpoint exception when the following conditions are met:

- The virtual (physical) address in r[rs1] matches the address in the VA (PA) Data Watchpoint Register.
- The byte store mask in r[rs2] indicates that a byte is to be stored.

- The Virtual (Physical) Data Watchpoint Mask in DCUCR indicates that one or more of the bytes to be stored at the watched address is being watched.

Watchpoint exceptions on partial store instructions behaves as if every partial store always stores all 8 bytes. The DCUCR Data Watchpoint masks are only checked for nonzero value (watchpoint enabled). 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).

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.

Note – If the byte ordering is little-endian, the byte enables generated by this instruction are swapped with respect to big-endian.

Exceptions

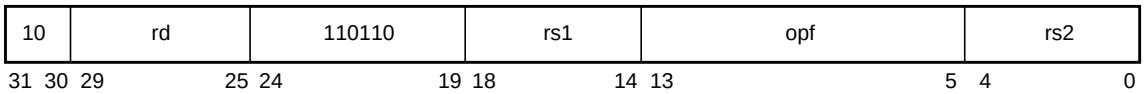
fp_disabled
illegal_instruction (When i = 1, no immediate mode is supported.)
PA_watchpoint
VA_watchpoint
mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.42 Partitioned Add/Subtract Instructions (VIS I)

Opcode	opf	Operation
FPADD16	0 0101 0000	Four 16-bit Add
FPADD16S	0 0101 0001	Two 16-bit Add
FPADD32	0 0101 0010	Two 32-bit Add
FPADD32S	0 0101 0011	One 32-bit Add
FPSUB16	0 0101 0100	Four 16-bit Subtract
FPSUB16S	0 0101 0101	Two 16-bit Subtract
FPSUB32	0 0101 0110	Two 32-bit Subtract

Opcode	opf	Operation
FPSUB32S	0 0101 0111	One 32-bit Subtract

Format (3)



Assembly Language Syntax	
fpadd16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpadd16s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpadd32	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpadd32s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpsub16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpsub16s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpsub32	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpsub32s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

The standard versions of these instructions perform four 16-bit or two 32-bit partitioned adds or subtracts between the corresponding fixed-point values contained in the source operands (the 64-bit floating-point registers specified by `rs1` and `rs2`). For subtraction, the second operand (`rs2`) is subtracted from the first operand (`rs1`). The result is placed in the 64-bit destination register specified by `rd`.

The single-precision versions of these instructions (FPADD16S, FPSUB16S, FPADD32S, FPSUB32S) perform two 16-bit or one 32-bit partitioned add(s) or subtract(s); only the low 32 bits of the destination register are affected.

Note – For good performance, the result of a single FPADD should not be used as part of a source operand of a 64-bit graphics instruction in the next instruction group. Similarly, the result of a standard FPADD should not be used as a 32-bit graphics instruction source operand in the next three instruction groups.

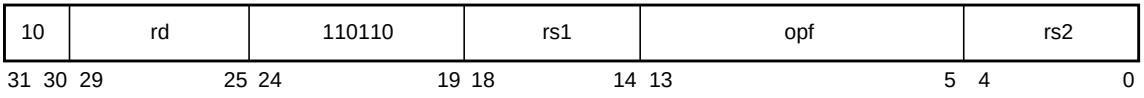
Exceptions

fp_disabled

A.43 Partitioned Multiply Instructions (VIS I)

Opcode	opf	Operation
FMUL8x16	0 0011 0001	8-bit x 16-bit Partitioned Product
FMUL8x16AU	0 0011 0011	8-bit x 16-bit Upper α Partitioned Product
FMUL8x16AL	0 0011 0101	8-bit x 16-bit Upper α Partitioned Product
FMUL8SUx16	0 0011 0110	Upper 8-bit x 16-bit Partitioned Product
FMUL8ULx16	0 0011 0111	Lower Unsigned 8-bit x 16-bit Partitioned Product
FMULD8SUx16	0 0011 1000	Upper 8-bit x 16-bit Partitioned Product
FMULD8ULx16	0 0011 1001	Lower Unsigned 8-bit x 16-bit Partitioned Product

Format (3)



Assembly Language Syntax	
fmul8x16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmul8x16au	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmul8x16al	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmul8sux16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmul8ulx16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmuld8sux16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmuld8ulx16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

Note – For good performance, the result of a partitioned multiply should not be used as a 32-bit graphics instruction source operand in the next three instruction groups.

Programming Note – When software emulates an 8-bit unsigned by16-bit signed multiply, the unsigned value must be zero-extended and the 16-bit value sign-extended before the multiplication.

Note – For good performance, the result of a partitioned multiply should not be used as a source operand of a 32-bit graphics instruction in the next three instruction groups.

The following sections describe the versions of partitioned multiplies.

Exceptions

fp_disabled

A.43.1 FMUL8x16 Instruction

`FMUL8x16` multiplies each unsigned 8-bit value (that is, a pixel) in `f[rs1]` by the corresponding (signed) 16-bit fixed-point integer in the 64-bit floating-point register specified by `rs2`; it rounds the 24-bit product (assuming binary point between bits 7 and 8) and stores the upper 16 bits of the result into the corresponding 16-bit field in the 64-bit floating-point destination register specified by `rd`. FIGURE A-5 illustrates the operation.

Note – This instruction treats the pixel 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.

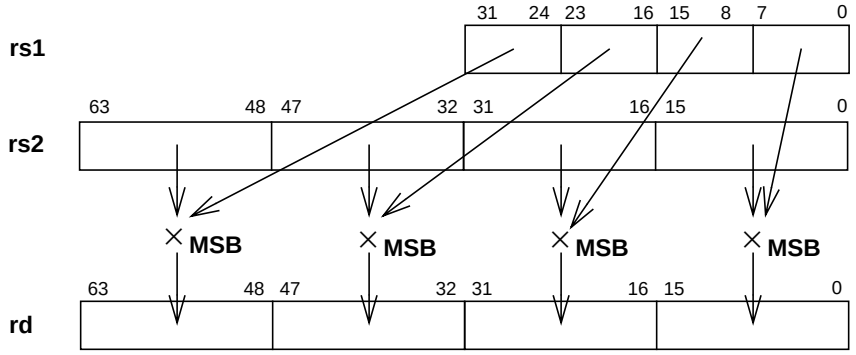


FIGURE A-5 FMUL8x16 Operation

A.43.2 FMUL8x16AU Instruction

`FMUL8x16AU` is the same as `FMUL8x16`, except that one 16-bit fixed-point value is used for all four multiplies. This value is the most significant 16 bits of the 32-bit register `f[rs2]`, which is typically a proportional value. FIGURE A-6 illustrates the operation.

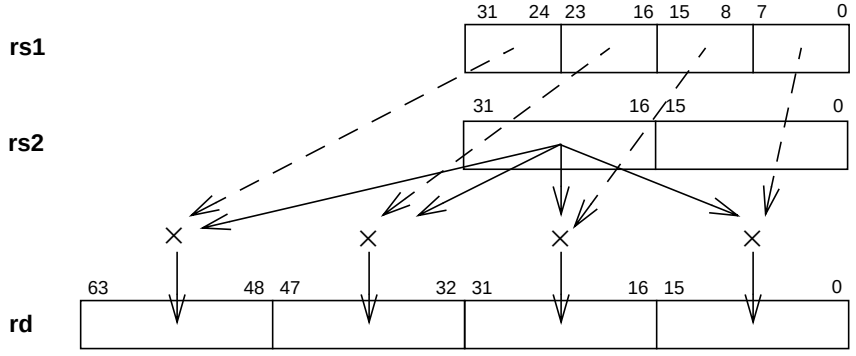


FIGURE A-6 FMUL8x16AU Operation

A.43.3 FMUL8x16AL Instruction

`FMUL8x16AL` is the same as `FMUL8x16AU`, except that the least significant 16 bits of the 32-bit register `f[rs2]` register are used as a proportional value. FIGURE A-7 illustrates the operation.

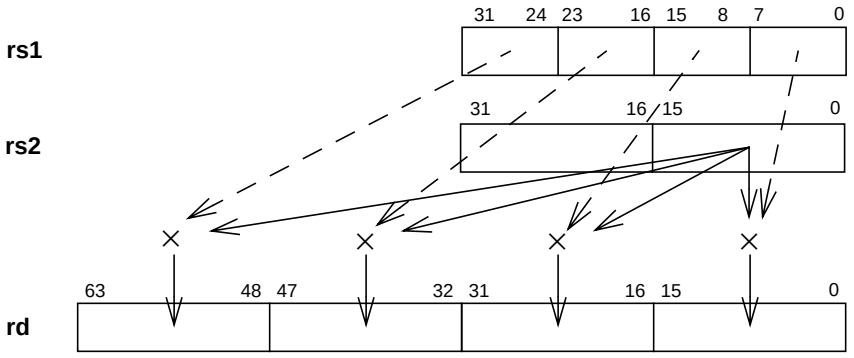


FIGURE A-7 FMUL8x16AL Operation

A.43.4 FMUL8SUx16 Instruction

FMUL8SUx16 multiplies the upper 8 bits of each 16-bit signed value in the 64-bit floating-point register specified by rs1 by the corresponding signed, 16-bit, fixed-point, signed integer in the 64-bit floating-point register specified by rs2. It rounds the 24-bit product toward the nearest representable value and then stores the upper 16 bits of the result into the corresponding 16-bit field of the 64-bit floating-point destination register specified by rd. If the product is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE A-8 illustrates the operation.

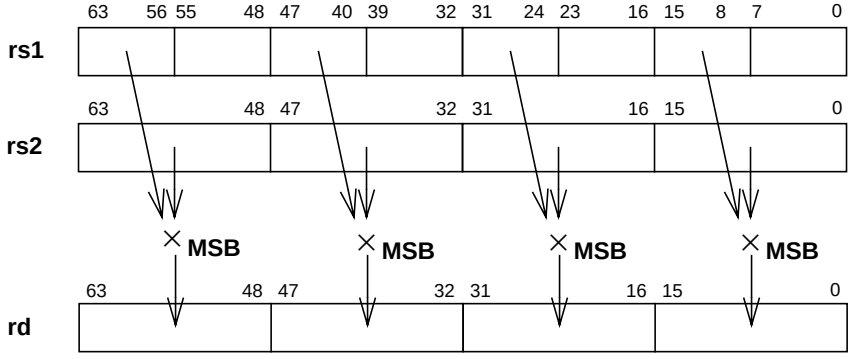


FIGURE A-8 FMUL8SUx16 Operation

A.43.5 FMUL8ULx16 Instruction

FMUL8ULx16 multiplies the unsigned lower 8 bits of each 16-bit value in the 64-bit floating-point register specified by rs1 by the corresponding fixed-point signed integer in the 64-bit floating-point register specified by rs2. Each 24-bit product is sign-extended to 32 bits. The upper 16 bits of the sign-extended value are rounded to nearest representable value and then stored in the corresponding 16-bit field of the 64-bit floating-point destination register specified by rd. If the result is exactly halfway between two integers, the result is rounded toward positive infinity. FIGURE A-9 illustrates the operation. CODE EXAMPLE A-5 shows an example.

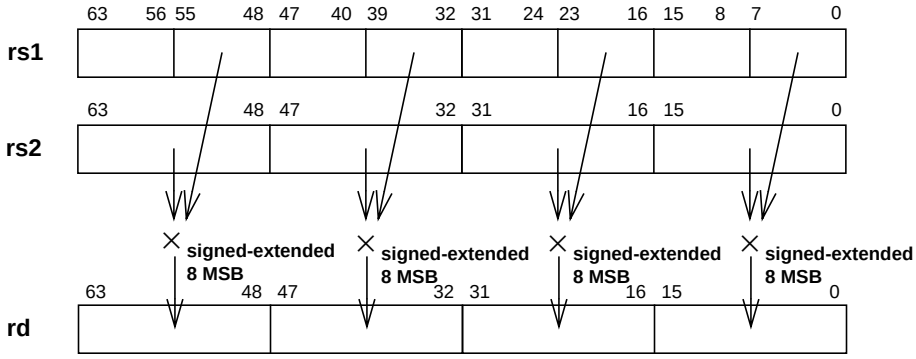


FIGURE A-9 FMUL8LUx16 Operation

CODE EXAMPLE A-5 FMUL8LUx16 Operation

```
    fmul8sux16   %f0, %f1, %f2

    fmul8ulx16   %f0, %f1, %f3

    fpadd16      %f2, %f3, %f4
```

A.43.6 FMULD8SUx16 Instruction

FMULD8SUx16 multiplies the upper 8 bits of each 16-bit signed value in f[rs1] by the corresponding signed 16-bit fixed-point signed integer in f[rs2]. Each 24-bit product is shifted left by 8 bits to make up a 32-bit result, which is then stored in the 64-bit floating-point register specified by rd. FIGURE A-10 illustrates the operation.

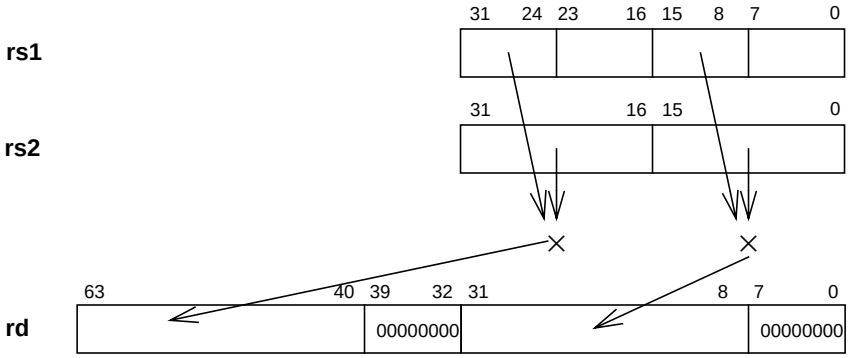


FIGURE A-10 FMULD8SUx16 Operation

A.43.7 FMULD8ULx16 Instruction

FMULD8ULx16 multiplies the unsigned lower 8 bits of each 16-bit value in f[rs1] by the corresponding fixed-point signed integer in f[rs2]. Each 24-bit product is sign-extended to 32 bits and stored in the 64-bit floating-point register specified by rd. FIGURE A-11 illustrates the operation; CODE EXAMPLE A-6 exemplifies the operation.

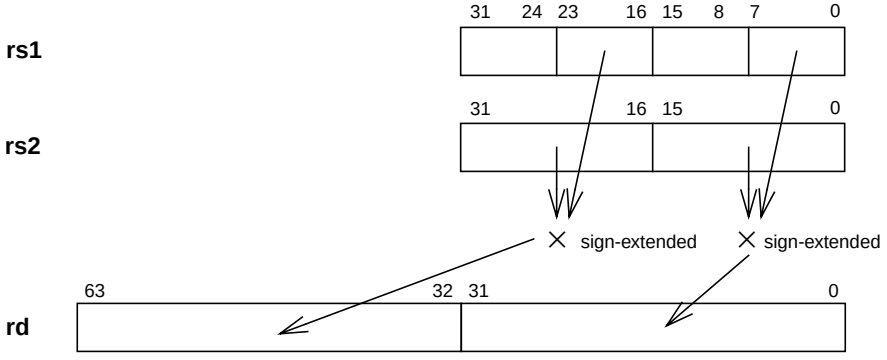


FIGURE A-11 FMULD8ULx16 Operation

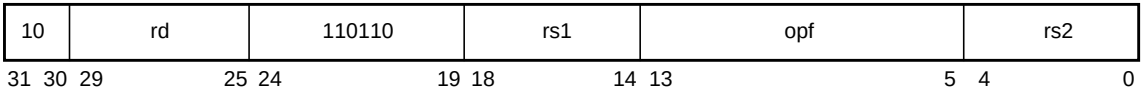
CODE EXAMPLE A-6 FMULD8ULx16 Operation

```
fmuld8sux16  %f0, %f1, %f2

fmuld8ulx16  %f0, %f1, %f3

fpadd32      %f2, %f3, %f4
```

A.44 Pixel Compare (VIS I)

Opcode	opf	Operation
FCMPGT16	0 0010 1000	Four 16-bit Compares; set rd if src1 > src2
FCMPGT32	0 0010 1100	Two 32-bit Compares; set rd if src1 > src2
FCMPLE16	0 0010 0000	Four 16-bit Compares; set rd if src1 ≤ src2
FCMPLE32	0 0010 0100	Two 32-bit Compares; set rd if src1 ≤ src2
FCMPNE16	0 0010 0010	Four 16-bit Compares; set rd if src1 ≠ src2
FCMPNE32	0 0010 0110	Two 32-bit Compares; set rd if src1 ≠ src2
FCMPEQ16	0 0010 1010	Four 16-bit Compares; set rd if src1 = src2
FCMPEQ32	0 0010 1110	Two 32-bit Compares; set rd if src1 = src2

Format (3)



Assembly Language Syntax	
`fcmpgt16`	*freg$_{rs1}$, freg$_{rs2}$, reg$_{rd}$*
`fcmpgt32`	*freg$_{rs1}$, freg$_{rs2}$, reg$_{rd}$*
`fcmple16`	*freg$_{rs1}$, freg$_{rs2}$, reg$_{rd}$*
`fcmple32`	*freg$_{rs1}$, freg$_{rs2}$, reg$_{rd}$*
`fcmpne16`	*freg$_{rs1}$, freg$_{rs2}$, reg$_{rd}$*
`fcmpne32`	*freg$_{rs1}$, freg$_{rs2}$, reg$_{rd}$*
`fcmpeq16`	*freg$_{rs1}$, freg$_{rs2}$, reg$_{rd}$*
`fcmpeq32`	*freg$_{rs1}$, freg$_{rs2}$, reg$_{rd}$*

Description

Either four 16-bit or two 32-bit fixed-point values in the 64-bit floating-point source registers specified by `rs1` and `rs2` are compared. The 4-bit or 2-bit results are stored in the least significant bits in the integer destination register `r[rd]`. Signed comparisons are used. Bit 0 of `r[rd]` corresponds to the least significant 16-bit or 32-bit comparison.

For `FCMPGT`, each bit in the result is set if the corresponding value in the first source operand is greater than the value in the second source operand. Less-than comparisons are made by swapping the operands.

For `FCMPLE`, each bit in the result is set if the corresponding value in the first source operand is less than or equal to the value in the second source operand. Greater-than-or-equal comparisons are made by swapping the operands.

For `FCMPEQ`, each bit in the result is set if the corresponding value in the first source operand is equal to the value in the second source operand.

For `FCMPNE`, each bit in the result is set if the corresponding value in the first source operand is not equal to the value in the second source operand.

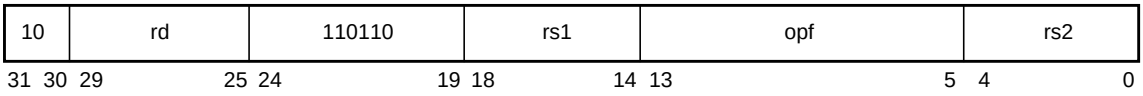
Exceptions

fp_disabled

A.45 Pixel Component Distance (PDIST) (VIS I)

Opcode	opf	Operation
PDIST	0 0011 1110	Distance between eight 8-bit components

Format (3)



Assembly Language Syntax	
pdist	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

Eight unsigned 8-bit values are contained in the 64-bit floating-point source registers specified by `rs1` and `rs2`. The corresponding 8-bit values in the source registers are subtracted (that is, the second source operand from the first source operand). The sum of the absolute value of each difference is added to the integer in the 64-bit floating-point destination register specified by `rd`. The result is stored in the destination register. Typically, this instruction is used for motion estimation in video compression algorithms.

Note – For good performance, the `rd` operand of `PDIST` should not reference the result of a non-`PDIST` instruction in the five previously executed instruction groups.

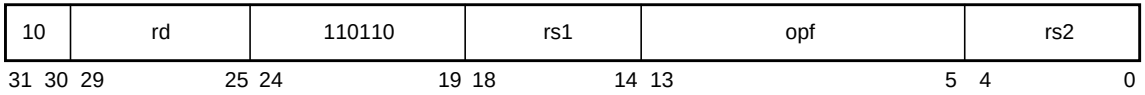
Exceptions

fp_disabled

A.46 Pixel Formatting (VIS I)

Opcode	opf	Operation
FPACK16	0 0011 1011	Four 16-bit packs into 8 unsigned bits
FPACK32	0 0011 1010	Two 32-bit packs into 8 unsigned bit
FPACKFIX	0 0011 1101	Four 16-bit packs into 16 signed bits
FEXPAND	0 0100 1101	Four 16-bit expands
FPMERGE	0 0100 1011	Two 32-bit merges

Format (3)



Assembly Language Syntax	
fpack16	*freg*$_{rs2}$, *freg*$_{rd}$
fpack32	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
fpackfix	*freg*$_{rs2}$, *freg*$_{rd}$
fexpand	*freg*$_{rs2}$, *freg*$_{rd}$
fpmerge	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

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.

Programming Note – For good performance, the result of an FPACK (including FPACK32) should not be used as part of a 64-bit graphics instruction source operand in the next three instruction groups.

FEXPAND performs the inverse of the FPACK16 operation.

FPMERGE interleaves four 8-bit values from each of two 32-bit registers into a single 64-bit destination register.

Programming Note – The result of FEXPAND or FPMERGE should not be used as a 32-bit graphics instruction source operand in the next three instruction groups.

Exceptions

fp_disabled

A.46.1 FPACK16

FPACK16 takes four 16-bit fixed values from the 64-bit floating-point register specified by rs2, scales, truncates, and clips them into four 8-bit unsigned integers, and stores the results in the 32-bit destination register, f[rd]. FIGURE A-12 illustrates the FPACK16 operation.

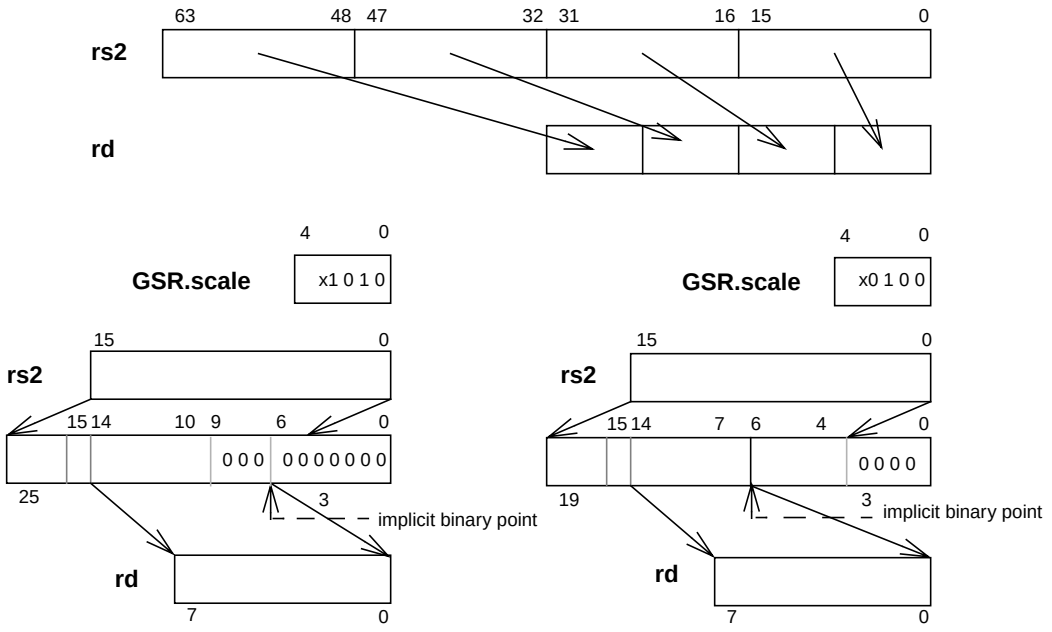


FIGURE A-12 FPACK16 Operation

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 the 64-bit floating-point register specified by 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), zero 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[rd].

A.46.2 FPACK32

FPACK32 takes two 32-bit fixed values from the second source operand (the 64-bit floating-point register specified by 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 with each 32-bit word in the 64-bit floating-point register specified by rs1, left-shifted by 8 bits. The 64-bit result is stored in the 64-bit floating-point register specified by rd. Thus, successive FPACK32 instructions can assemble two pixels by using three or four pairs of 32-bit fixed values. FIGURE A-13 illustrates the FPACK32 operation.

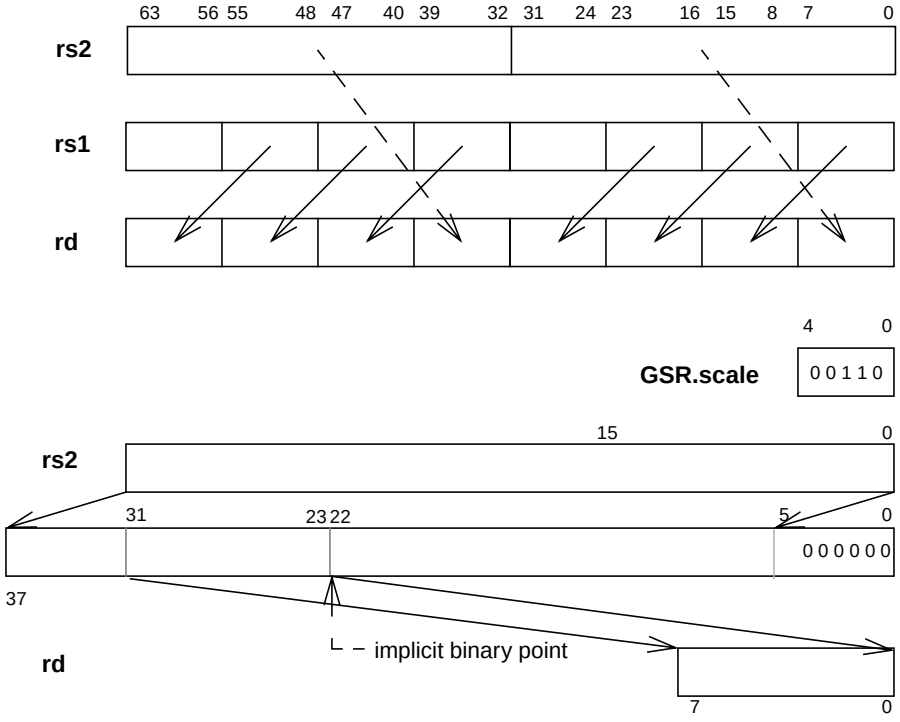


FIGURE A-13 FPACK32 Operation

This operation is carried out as follows:

1. Left-shift each 32-bit value from the second source operand 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 converts the scaled value into a signed integer (that is,

round toward negative infinity). If the resulting value is negative (that is, MSB is set), then zero 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 the first source operand (the 64-bit floating-point register specified by `rs1`) by 8 bits.
4. Merge the two clipped 8-bit unsigned values into the corresponding least significant byte positions in the left-shifted value from the second source operand.
5. Store the result in the `rd` register.

A.46.3 FPACKFIX

`FPACKFIX` takes two 32-bit fixed values from the 64-bit floating-point register specified by `rs2`, scales, truncates, and clips them into two 16-bit unsigned integers, and then stores the result in the 32-bit destination register `f[rd]`. FIGURE A-14 illustrates the `FPACKFIX` operation.

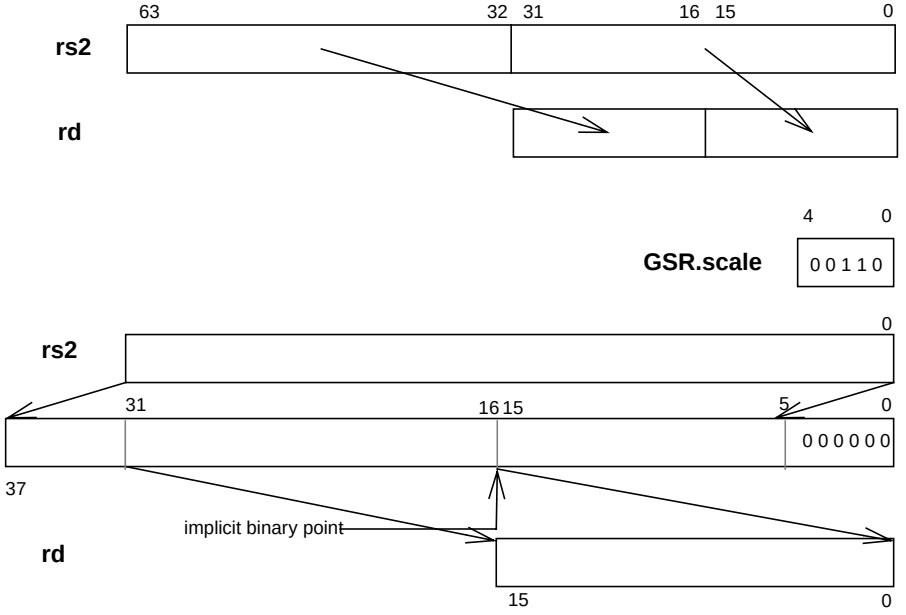


FIGURE A-14 `FPACKFIX` Operation

This operation is carried out as follows:

1. Left-shift each 32-bit value from the source operand (the 64-bit floating-point register specified by `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 converts 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[rd]`.

A.46.4 FEXPAND

FEXPAND takes four 8-bit unsigned integers from `f[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 specified by `rd`. FIGURE A-15 illustrates the operation.

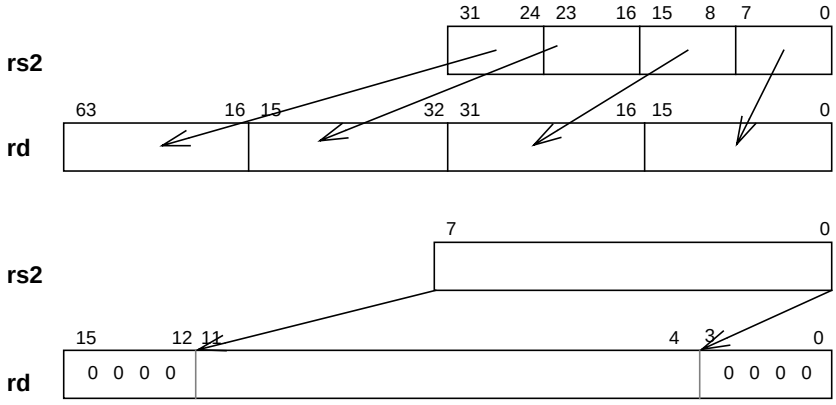


FIGURE A-15 FEXPAND Operation

This operation is carried out as follows:

1. Left-shift each 8-bit value by four and zero-extend the results to a 16-bit fixed value.
2. Store the result in the destination register.

A.46.5 FPMERGE

FPMERGE interleaves four corresponding 8-bit unsigned values in f[rs1] and f[rs2] to produce a 64-bit value in the 64-bit floating-point destination register specified by 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 A-16 illustrates the operation.

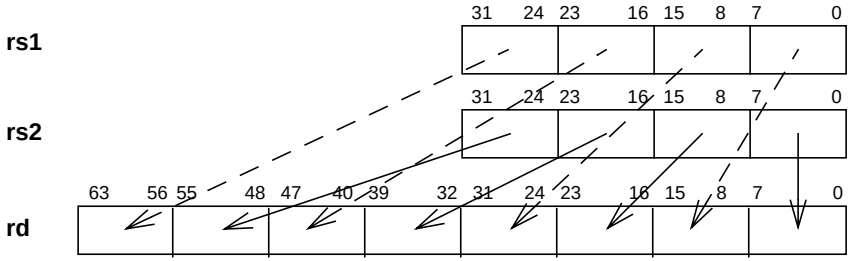


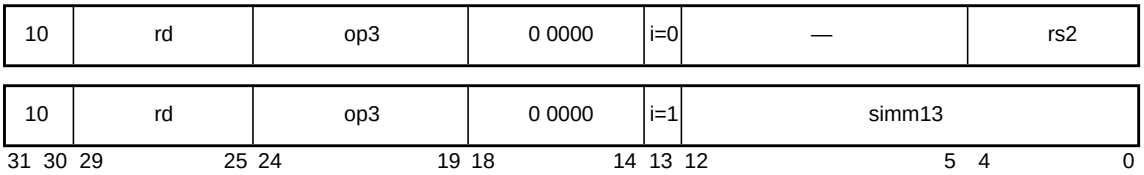
FIGURE A-16 FPMERGE Operation

Back-to-back FPMERMGEs cannot be done on adjacent cycles.

A.47 Population Count

Opcode	op3	Operation
POPC	10 1110	Population Count

Format (3)



Assembly Language Syntax	
`popc`	*reg_or_imm, reg*$_{rd}$

Description

`POPC` counts the number of one bits in `r[rs2]` if `i` = 0, or the number of one bits in `sign_ext(simm13)` if `i` = 1, and stores the count in `r[rd]`. This instruction does not modify the condition codes.

Note – The UltraSPARC IIIi processor does not implement this instruction in hardware; instead, it traps to software. The instruction is emulated in supervisor software.

Exceptions

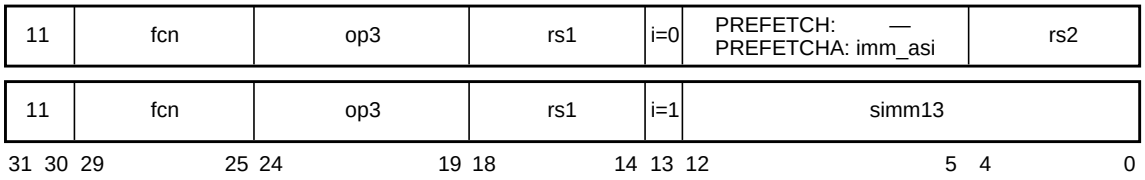
illegal_instruction

A.48 Prefetch Data

Opcode	op3	Operation
`PREFETCH`	10 1101	Prefetch Data
`PREFETCHA`$^{P_{ASI}}$	11 1101	Prefetch Data from Alternate Space

Implementation Note – The PREFETCH{A} instructions are supported in the UltraSPARC IIIi processor.

Format (3) PREFETCH{A}



Assembly Language Syntax	
`prefetch`	[*address*], *prefetch_fcn*
`prefetcha`	[*regaddr*] *imm_asi, prefetch_fcn*
`prefetcha`	[*reg_plus_imm*] `%asi`, *prefetch_fcn*

Description

Prefetching is used to help manage data memory cache(s). A prefetch to a non-prefetchable location has no effect. Non-cacheable and non-prefetchable locations are not the same.

Variants of the prefetch instruction are used to prepare the memory system for different types of memory accesses.

In non-privileged code, a prefetch instruction has no observable effect. Its execution is nonblocking and cannot cause an observable trap. In particular, a prefetch instruction shall not trap if it is applied to an illegal or nonexistent memory address.

Programming Note – When software needs to prefetch 64 bytes beginning at an arbitrary *address*, issue two prefetch instructions to canvas all bytes:
`prefetch[`*address*`],` *prefetch_fcn*
`prefetch[`*address* `+ 63],` *prefetch_fcn*

PREFETCH A

Prefetch instructions that do *not* load from an alternate address space access the primary address space (`ASI_PRIMARY{_LITTLE}`). Prefetch instructions that *do* load from an alternate address space 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 zero; 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.

Exceptions

illegal_instruction

A.48.1 Prefetch Instruction Variants

PREFETCH(A) instructions with `fcn` = 0–3 are implemented.

Each prefetch variant reflects an intent on the part of the compiler or programmer. This is different from other instructions in SPARC-V9 (except `BPN`), all of which specify specific actions.

The prefetch instruction variants are intended to provide scalability for future improvements in both hardware and compilers.

The prefetch variant is selected by the `fcn` field of the instruction. In accordance with SPARC-V9, `fcn` values 4–15 cause an *illegal_instruction* exception.

A prefetch with `fcn` = 16 invalidates the P-cache line corresponding to the effective address of the prefetch. Use this characteristic to prefetch non-cacheable data after data are loaded into registers from the P-cache. A prefetch invalidate is issued to remove the data from the P-cache so it will not be found by a later reference. Prefetch with `fcn` = 20, 21, 22, 23 are new.

TABLE A-13 lists the types of software prefetch instructions. Note that the table contains hexadecimal values for `fcn` unlike the decimal values in the explanation above.

TABLE A-13 Types of Software Prefetch Instructions

`fcn` Value (hex)	Instruction Type	Prefetch into:	Instruction Strength UltraSPARC IIIi	Request Exclusive Ownership
00	Prefetch read many	P-cache and L2-cache	weak	No
01	Prefetch read once	P-cache only	weak	No
02	Prefetch write many	L2-cache only	weak	Yes
03	Prefetch write once[1]	L2-cache only	weak	No
04	*reserved*	Undefined		
05 - 0F	*reserved*	Undefined		
10	Prefetch invalidate	Invalidates a P-cache line, no data is prefetched.		N/A
11 - 13	*reserved*	Undefined		

TABLE A-13 Types of Software Prefetch Instructions *(Continued)*

`fcn` Value (hex)	Instruction Type	Prefetch into:	Instruction Strength UltraSPARC IIIi	Request Exclusive Ownership
14	Same as `fcn` = 00		weak[2]	No
15	Same as `fcn` = 01		weak[2]	No
16	Same as `fcn` = 02		weak[2]	Yes
17	Same as `fcn` = 03		weak[2]	No
18 - 1F	*reserved*	Undefined		

1. Although the name is "prefetch write once," the actual use is prefetch to L2-cache for a future read.
2. These weak instructions may be implemented as strong in future implementations.

A.48.2 New Error Handling of PREFETCH,2 and Other Prefetches

Since PREFETCH,2 request for cache line ownership (RTO/R_RTO), an error occurs while processing it will be handled differently compared to other prefetch requests with RTS/R_RTS, as described in TABLE A-14.

TABLE A-14 Error Handling of Prefetch Requests

Prefetch Type	L2-cache Hit/Miss	Error Type	L2-cache Action	P-cache Action	Error Logging	Trap
PREFETCH,2 (RTO/R_RTO)	Hit	Tag, Hardware-corrected	No state change	None	THCE	Disrupting
	Miss	Tag, Hardware-corrected	Install data, state change to M	None	THCE	Disrupting
	"Hit" (tag error)	Tag, uncorrectable	No data install, no state change	None	TUE	Fatal Error
	Hit	Data, Hardware-corrected	No state change	None	EDC	Disrupting
	Hit	Data, uncorrectable	No state change	None	EDU	Disrupting
	Miss	Data, Hardware-corrected	Install data, state change to M	None	CE	Disrupting
	Miss	Data, uncorrectable	Install uncorrected data, state change to M	None	DUE	Disrupting
	Miss	Mtag, Hardware-corrected	Install data, state change to M	None	EMC	Disrupting
	Miss	Mtag, uncorrectable	Install data if L2-cache state is M or Os	None	EMU	Fatal Error

TABLE A-14 Error Handling of Prefetch Requests *(Continued)*

Prefetch Type	L2-cache Hit/Miss	Error Type	L2-cache Action	P-cache Action	Error Logging	Trap
PREFETCH,0 PREFETCH,1 PREFETCH,3 Hardware prefetch (RTS/R_RTS)	Hit	Tag, Hardware-corrected	No state change	Install data (except PREFETCH, 3)	THCE	Disrupting
	Miss	Tag, Hardware-corrected	Install data, state change to S or E	Install data (except PREFETCH, 3)	THCE	Disrupting
	"Hit" (tag error)	Tag, uncorrectable	No data install, no state change	Cancel install	TUE	Fatal Error
	Hit	Data, Hardware-corrected	No state change	Install data (except PREFETCH, 3)	EDC	Disrupting
	Hit	Data, uncorrectable	No state change	Cancel install	EDU	Disrupting
	Miss	Data, Hardware-corrected	Install data, state change to S or E	Install data (except PREFETCH, 3)	CE	Disrupting
	Miss	Data, uncorrectable	-If RTS, cancel install, no state change. -If R_RTS, install uncorrected data, state change to Os.	Cancel install	DUE	Disrupting
	Miss	Mtag, Hardware-corrected	Install data, state change to S or E	None	EMC	Disrupting
	Miss	Mtag, uncorrectable	Install data if L2-cache state is M or Os	None	EMU	Fatal Error

A.48.2.1 New Column in Coherence Table

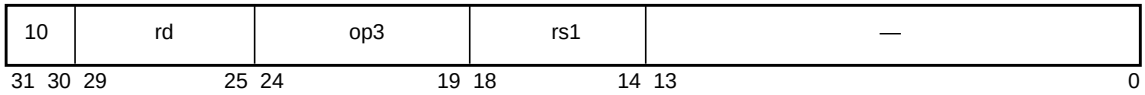
A new column has been added to the UltraSPARC IIIi Coherence Table to describe the processor action on write prefetch RTO. Basically, the behavior of coherence state change is the following:

- On L2-cache hit: same as Load request (no state change)
- On L2-cache miss: same as Store request (send RTO/R_RTO to get M state)

A.49 Read Privileged Register

Opcode	op3	Operation
RDPR[P]	10 1010	Read Privileged Register

Format (3)



rs1	Privileged Register
0	TPC
1	TNPC
2	TSTATE
3	TT
4	TICK
5	TBA
6	PSTATE
7	TL
8	PIL
9	CWP
10	CANSAVE
11	CANRESTORE
12	CLEANWIN
13	OTHERWIN
14	WSTATE
15	FQ
16–30	—
31	VER

Assembly Language Syntax	
rdpr	%tpc, reg_{rd}
rdpr	%tnpc, reg_{rd}
rdpr	%tstate, reg_{rd}
rdpr	%tt, reg_{rd}
rdpr	%tick, reg_{rd}
rdpr	%tba, reg_{rd}
rdpr	%pstate, reg_{rd}
rdpr	%tl, reg_{rd}
rdpr	%pil, reg_{rd}
rdpr	%cwp, reg_{rd}
rdpr	%cansave, reg_{rd}
rdpr	%canrestore, reg_{rd}
rdpr	%cleanwin, reg_{rd}
rdpr	%otherwin, reg_{rd}
rdpr	%wstate, reg_{rd}
rdpr	%fq, reg_{rd}
rdpr	%ver, reg_{rd}

Description

The rs1 field in the instruction determines the privileged register that is read. There are MAXTL copies of the TPC, TNPC, TT, and TSTATE registers. A read from one of these registers returns the value in the register indexed by the current value in the trap level register (TL). A read of TPC, TNPC, TT, or TSTATE when the trap level is zero (TL = 0) causes an *illegal_instruction* exception.

RDPR instructions with rs1 in the range 16 –30 are reserved; executing an RDPR instruction with rs1 in that range causes an *illegal_instruction* exception.

Programming Note – On this implementation with precise floating-point traps, the address of a trapping instruction will be in the TPC[TL] register when the trap code begins execution.

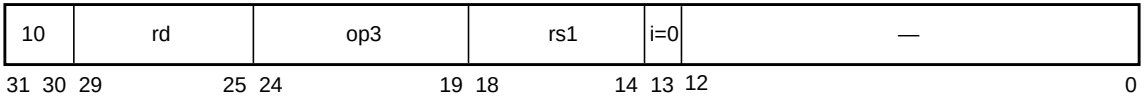
Exceptions

privileged_opcode
illegal_instruction ((rs1 = 16–30) or ((rs1 ≤ 3) and (TL = 0)))

A.50 Read State Register

Opcode	op3	rs1	Operation
RDY[D]	10 1000	0	Read Y Register; deprecated (see Section A.70.9, “Read Y Register”)
—	10 1000	1	*Reserved, do not access; attempt to access causes in illegal_instruction exception.*
RDCCR	10 1000	2	Read Condition Codes Register
RDASI	10 1000	3	Read ASI Register
RDTICK[PNPT]	10 1000	4	Read Tick Register
RDPC	10 1000	5	Read Program Counter
RDFPRS	10 1000	6	Read Floating-Point Registers Status Register
—	10 1000	7–14	*Reserved, do not access; attempt to access causes in illegal_instruction exception.*
See section description	10 1000	15	STBAR, MEMBAR, or *Reserved*; see section description.
RDASR	10 1000	16–31	Read non-SPARC-V9 ASRs
RDPCR[PPCR]		16	Read Performance Control Registers (PCR)
RDPIC[PPIC]		17	Read Performance Instrumentation Counters (PIC)
RDDCR[P]		18	Read Dispatch Control Register (DCR)
RDGSR		19	Read Graphic Status Register (GSR)
—		20–21	*Reserved, do not access; attempt to access causes in illegal_instruction exception.*
RDSOFTINT[P]		22	Read per-processor Soft Interrupt Register
RDTICK_CMPR[P]		23	Read Tick Compare Register
RDSTICK[PNPT]		24	Read System TICK Register
RDSTICK_CMPR[P]		25	Read System TICK Compare Register
—		26–31	*Reserved, do not access; attempt to access causes in illegal_instruction exception.*

Format (3)



Assembly Language Syntax	
rd	%ccr, reg_{rd}
rd	%asi, reg_{rd}
rd	%tick, reg_{rd}
rd	%pc, reg_{rd}
rd	%fprs, reg_{rd}
rd	%pcr, reg_{rd}
rd	%pic, reg_{rd}
rd	%dcr, reg_{rd}
rd	%gsr, reg_{rd}
rd	%softint, reg_{rd}
rd	%tick_cmpr, reg_{rd}
rd	%sys_tick, reg_{rd}
rd	%sys_tick_cmpr, reg_{rd}

Description

These instructions read the state register specified by rs1 into r[rd].

Values 7–14 of rs1 are reserved for future versions of the architecture. A Read State Register instruction with rs1 = 15, rd = 0, and i = 0 is defined to be a (deprecated) STBAR instruction (see Section A.70.10, “Store Barrier”). An RDASR instruction with rs1 = 15, rd = 0, and i = 1 is defined to be a MEMBAR instruction. RDASR with rs1 = 15 and rd ≠ 0 is reserved for future versions of the architecture; it causes an *illegal_instruction* exception.

For RDPC, the processor writes the full 64-bit program counter value to the destination register of a CALL, JMPL, or RDPC instruction. When PSTATE.AM = 1 and a trap occurs, the processor writes the full 64-bit program counter value to TPC[TL].

RDFPRS waits for all pending FPops and loads of floating-point registers to complete before reading the FPRS register.

RDGSR causes a *fp_disabled* exception if PSTATE.PEF = 0 or FPRS.FEF = 0.

RDTICK causes a *privileged_action* exception if PSTATE.PRIV = 0 and TICK.NPT = 1. RDSTICK causes a *privileged_action* exception if PSTATE.PRIV = 0 and STICK.NPT = 1.

RDPIC causes a *privileged_action* exception if PSTATE.PRIV = 0 and PCR.PRIV = 1.

RDPCR causes a *privileged_opcode* exception due to access privilege violation.

Implementation Note – Ancillary state registers include, for example, timer, counter, diagnostic, self-test, and trap-control registers.

Compatibility Note – The SPARC-V8 RDPSR, RDWIM, and RDTBR instructions do not exist in SPARC-V9 since the PSR, WIM, and TBR registers do not exist in SPARC-V9.

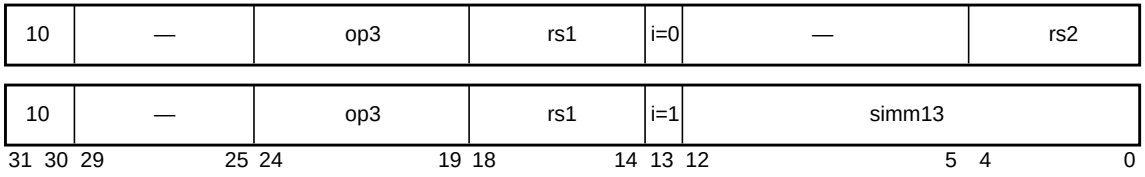
Exceptions

privileged_opcode (RDDCR, RDSOFTINT, RDTICK_CMPR, RDSTICK, RDSTICK_CMPR, and RDPCR)
illegal_instruction (RDASR with rs1 = 1 or 7–14;
RDASR with rs1 = 15 and rd ≠ 0;
RDASR with rs1 = 20–21, 26–31)
privileged_action (RDTICK with PSTATE.PRIV = 0 and TICK.NPT = 1;
RDPIC with PSTATE.PRIV = 0 and PCR.PRIV = 1;
RDSTICK with PSTATE.PRIV = 0 and STICK.NPT = 1)
fp_disabled (RDGSR with PSTATE.PEF = 0 or FPRS.FEF = 0)

A.51 RETURN

Opcode	op3	Operation
RETURN	11 1001	Return

Format (3)



Assembly Language Syntax	
`return`	*address*

Description

The `RETURN` instruction causes a delayed transfer of control to the target address and has the window semantics of a `RESTORE` instruction; that is, it restores the register window prior to the last `SAVE` instruction. The target address is "`r[rs1]` + `r[rs2]`" if `i` = 0, or "`r[rs1]` + `sign_ext(simm13)`" if `i` = 1. Registers `r[rs1]` and `r[rs2]` come from the *old* window.

The `RETURN` instruction may cause an exception. It may cause a *window_fill* exception as part of its `RESTORE` semantics, or it may cause a *mem_address_not_aligned* exception if either of the two low-order bits of the target address is nonzero.

Programming Note – To re-execute the trapped instruction when returning from a user trap handler, use the `RETURN` instruction in the delay slot of a `JMPL` instruction, for example:

```
jmpl     %l6,%g0   | Trapped PC supplied to user trap handler
return   %l7       | Trapped nPC supplied to user trap handler
```

Programming Note – A routine that uses a register window may be structured either as:

```
save     %sp,-framesize, %sp
. . .
ret                | Same as jmpl %i7+8, %g0
restore            | Something useful like "restore
                   | %o2,%l2,%o0"
```

or,

```
save     %sp,-framesize, %sp
. . .
return   %i7+8
nop                | Could do some useful work in the caller's
                   | window, for example, "or %o1, %o2,%o0"
```

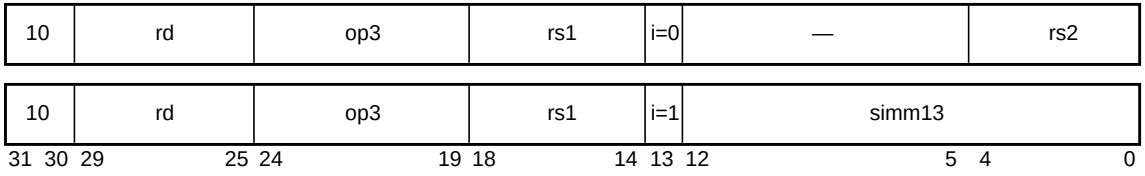
Exceptions

mem_address_not_aligned
fill_n_normal (*n* = 0–7)
fill_n_other (*n* = 0–7)

A.52 SAVE and RESTORE

Opcode	op3	Operation
SAVE	11 1100	Save Caller's Window
RESTORE	11 1101	Restore Caller's Window

Format (3)



Assembly Language Syntax	
`save`	reg_{rs1}, *reg_or_imm*, reg_{rd}
`restore`	reg_{rs1}, *reg_or_imm*, reg_{rd}

Description (Effect on Non-Privileged State)

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.

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 spill or fill trap is not generated, SAVE and RESTORE behave like normal ADD instructions, 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 windows, NWINDOWS.

Programming Note – Typically, if a SAVE (RESTORE) instruction traps, the spill (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.

The SAVE instruction can be used to atomically allocate a new window in the register file and a new software stack frame in memory.

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 the SAVE instruction does not trap, it increments the CWP (**mod** NWINDOWS) 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).

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).

If the RESTORE instruction does not trap, it decrements the CWP (**mod** NWINDOWS) 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. 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 spill and fill traps can be controlled by setting the value of the OTHERWIN and WSTATE registers appropriately.

The spill (fill) handler normally will end with a SAVED (RESTORED) instruction followed by a RETRY instruction.

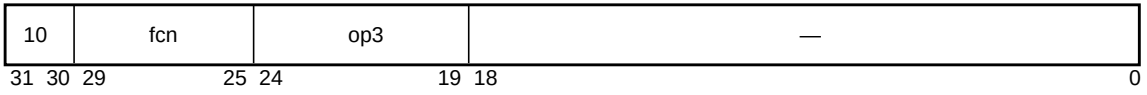
Exceptions

clean_window (SAVE only)
fill_n_normal (RESTORE only, $n = 0–7$)
fill_n_other (RESTORE only, $n = 0–7$)
spill_n_normal (SAVE only, $n = 0–7$)
spill_n_other (SAVE only, $n = 0–7$)

A.53 SAVED and RESTORED

Opcode	op3	fcn	Operation
SAVED[P]	11 0001	0	Window has been saved
RESTORED[P]	11 0001	1	Window has been restored
—	11 0001	2–31	*Reserved*

Format (3)



Assembly Language Syntax	
saved	
restored	

Description

SAVED and RESTORED adjust the state of the register-windows control registers.

SAVED increments CANSAVE. If OTHERWIN = 0, SAVED decrements CANRESTORE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

RESTORED increments CANRESTORE. If CLEANWIN < (NWINDOWS–1), then RESTORED increments CLEANWIN. If OTHERWIN = 0, it decrements CANSAVE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

Programming Note – The spill (fill) handlers use the SAVED (RESTORED) instruction to indicate that a window has been spilled (filled) successfully.

Normal privileged software would probably not do a SAVED or RESTORED 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 (RESTORED) instruction outside of a window spill (fill) trap handler is likely to create an inconsistent window state. Hardware will not signal an exception, however, since maintaining a consistent window state is the responsibility of privileged software.

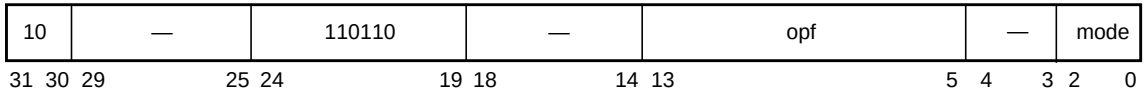
Exceptions

privileged_opcode
illegal_instruction (fcn = 2–31)

A.54 Set Interval Arithmetic Mode (VIS II)

Opcode	opf	Operation
SIAM	0 1000 0001	Set the interval arithmetic mode fields in the GSR

Format (3)



Assembly Language Syntax	
`siam`	*mode*

Description

The `SIAM` instruction sets the `GSR.IM` and `GSR.IRND` fields as follows:

`GSR.IM` = mode<2>

`GSR.IRND` = mode<1:0>

Note – `SIAM` is a groupable, break-after instruction. It enables the interval rounding mode to be changed every cycle without flushing the pipeline. FPops in the same instruction group as an `SIAM` instruction use the previous rounding mode.

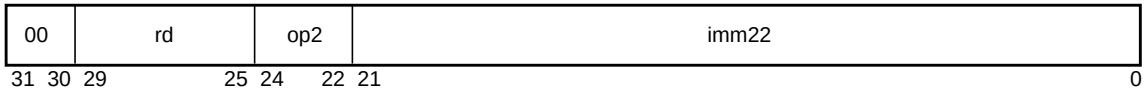
Exceptions

fp_disabled

A.55 SETHI

Opcode	op2	Operation
SETHI	100	Set High 22 Bits of Low Word

Format (2)



Assembly Language Syntax	
sethi	*const22*, *reg*$_{rd}$
sethi	%hi (*value*), *reg*$_{rd}$

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 has a special use:

- rd = 0 and imm22 = 0: has no architectural effect and is defined to be a NOP instruction
- rd = 0 and imm22 ≠ 0 is used to trigger hardware performance counters. See Chapter 11 "Performance Instrumentation" for details.

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 get ones 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
```

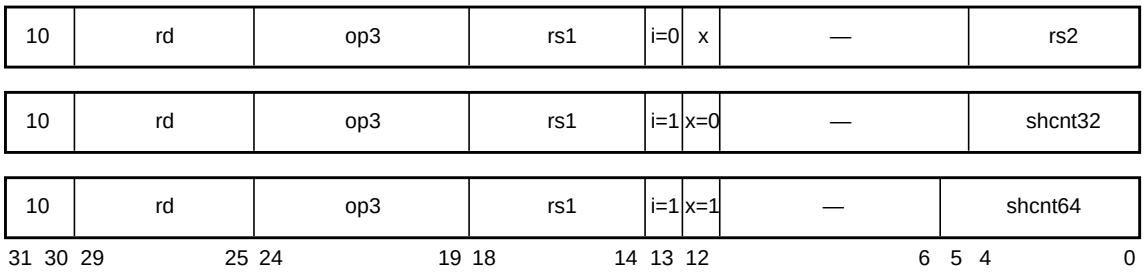
Exceptions

None

A.56 Shift

Opcode	op3	x	Operation
SLL	10 0101	0	Shift Left Logical – 32 bits
SRL	10 0110	0	Shift Right Logical – 32 bits
SRA	10 0111	0	Shift Right Arithmetic – 32 bits
SLLX	10 0101	1	Shift Left Logical – 64 bits
SRLX	10 0110	1	Shift Right Logical – 64 bits
SRAX	10 0111	1	Shift Right Arithmetic – 64 bits

Format (3)



Assembly Language Syntax	
`sll`	*reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$
`srl`	*reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$
`sra`	*reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$
`sllx`	*reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$
`srlx`	*reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$
`srax`	*reg*$_{rs1}$, *reg_or_shcnt*, *reg*$_{rd}$

Description

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 A-15 shows the shift count encodings for all values of `i` and `x`.

TABLE A-15 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 zero, but the high-order bits are affected by the 32-bit shifts as noted above.

These instructions do not modify the condition codes.

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

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

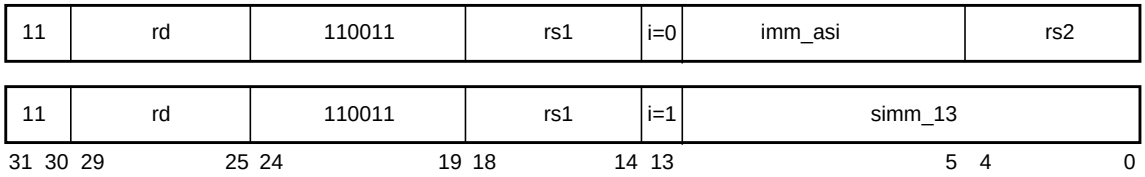
Exceptions

None

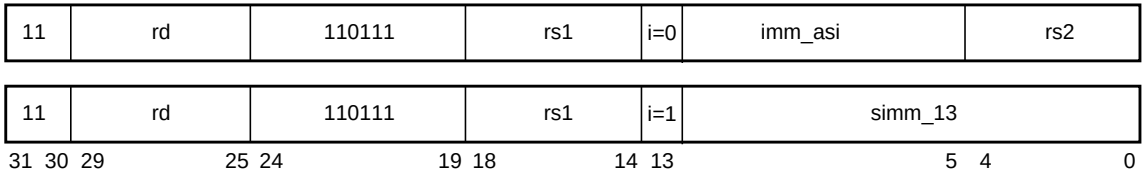
A.57 Short Floating-Point Load and Store (VIS I)

Opcode	imm_asi	ASI Value	Operation
LDDFA STDFA	ASI_FL8_P	$D0_{16}$	8-bit load/store from/to primary address space
LDDFA STDFA	ASI_FL8_S	$D1_{16}$	8-bit load/store from/to secondary address space
LDDFA STDFA	ASI_FL8_PL	$D8_{16}$	8-bit load/store from/to primary address space, little-endian
LDDFA STDFA	ASI_FL8_SL	$D9_{16}$	8-bit load/store from/to secondary address space, little-endian
LDDFA STDFA	ASI_FL16_P	$D2_{16}$	16-bit load/store from/to primary address space
LDDFA STDFA	ASI_FL16_S	$D3_{16}$	16-bit load/store from/to secondary address space
LDDFA STDFA	ASI_FL16_PL	DA_{16}	16-bit load/store from/to primary address space, little-endian
LDDFA STDFA	ASI_FL16_SL	DB_{16}	16-bit load/store from/to secondary address space, little-endian

Format (3) LDDFA



Format (3) STDFA



Assembly Language Syntax	
`ldda`	[*reg_addr*] *imm_asi*, *freg*$_{rd}$
`ldda`	[*reg_plus_imm*] `%asi`, *freg*$_{rd}$
`stda`	*freg*$_{rd}$, [*reg_addr*] *imm_asi*
`stda`	*freg*$_{rd}$, [*reg_plus_imm*] `%asi`

Description

Short floating-point load and store instructions are selected by means of one of the short ASIs with the `LDDFA` and `STDFA` instructions.

These ASIs allow 8- and 16-bit loads or stores to be performed to/from the floating-point registers. Eight-bit loads can be performed to arbitrary byte addresses. For 16-bit loads, the least significant bit of the address must be zero or a *mem_address_not_aligned* trap is taken. Short loads are zero-extended to the full floating-point register. Short stores access the low-order 8 or 16 bits of the register.

Little-endian ASIs transfer data in little-endian format in memory; otherwise, memory is assumed to be big-endian. Short loads and stores are typically used with the `FALIGNDATA` instruction (see Section A.2, “Alignment Instructions (VIS I)”) to assemble or store 64 bits on noncontiguous components.

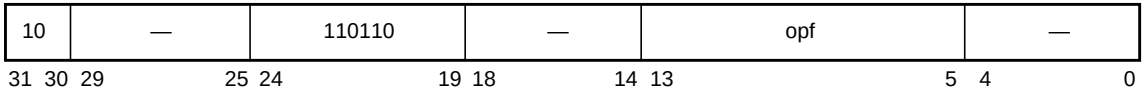
Exceptions

fp_disabled
PA_watchpoint
VA_watchpoint
mem_address_not_aligned (odd memory address for a 16-bit load or store)
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.58 SHUTDOWN (VIS I)

Opcode	opf	Operation
SHUTDOWN^{P}	0 1000 0000	Shut down to enter power-down mode

Format (3)



Assembly Language Syntax
`shutdown`

Description

SHUTDOWN is a privileged instruction.

The SHUTDOWN instruction executes as a NOP. An external system signal is used to enter and leave Low Power mode.

Because SHUTDOWN is a privileged instruction, an attempt to execute it while in non-privileged mode causes a *privileged_opcode* trap.

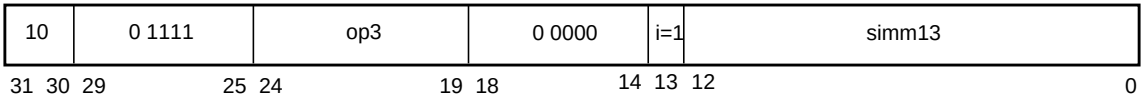
Exceptions

privileged_opcode

A.59 Software-Initiated Reset

Opcode	op3	rd	Operation
SIR	11 0000	15	Software-Initiated Reset

Format (3)



Assembly Language Syntax	
sir	*simm13*

Description

SIR is used to generate a software-initiated reset (SIR). As with other traps, a software-initiated reset performs different actions when TL = MAXTL than it does when TL < MAXTL.

When executed in non-privileged mode, SIR acts like a NOP with no visible effect.

Exceptions

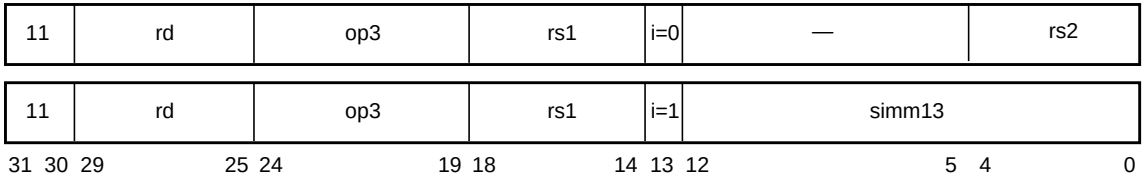
software_initiated_reset

A.60 Store Floating-Point

Opcode	op3	rd	Operation
STF	10 0100	0–31	Store Floating-Point Register
STDF	10 0111	†	Store Double Floating-Point Register
STQF	10 0110	†	Store Quad Floating-Point Register
STXFSR	10 0101	1	Store Floating-Point State Register
—	10 0101	2–31	*Reserved*

† Encoded floating-point register value.

Format (3)



Assembly Language Syntax	
st	*freg*$_{rd}$, [*address*]
std	*freg*$_{rd}$, [*address*]
stq	*freg*$_{rd}$, [*address*]
stx	%fsr, [*address*]

Description

The store single floating-point instruction (STF) copies f[rd] into memory.

The store double floating-point instruction (STDF) copies a doubleword from a double floating-point register into a word-aligned doubleword in memory.

The store quad floating-point instruction (STQF) traps to software.

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 until it is known that the store will not cause a precise trap.

The effective address for these instructions is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

STF requires word alignment otherwise a *mem_address_not_aligned* exception occurs.

STDF instruction causes a *STDF_mem_address_not_aligned* trap if the effective address is 32-bit aligned but not 64-bit (doubleword) aligned. In this case, the trap handler software shall emulate the STDF instruction and return.

STXFSR requires doubleword alignment; otherwise, it causes a *mem_address_not_aligned* exception. In this case, the trap handler software shall emulate the STXFSR instruction and return.

If the floating-point unit is not enabled for the source register rd (per FPRS.FEF and PSTATE.PEF) or if the FPU is not present, then a store floating-point instruction causes a *fp_disabled* exception.

Programming Note – In SPARC-V8, some compilers issued sets of single-precision stores when they could not determine that doubleword or quadword operands were properly aligned. For SPARC-V9, since emulation of misaligned stores is expected to be fast, it is recommended that compilers issue sets of single-precision stores only when they can determine that doubleword or quadword operands are *not* properly aligned.

Exceptions

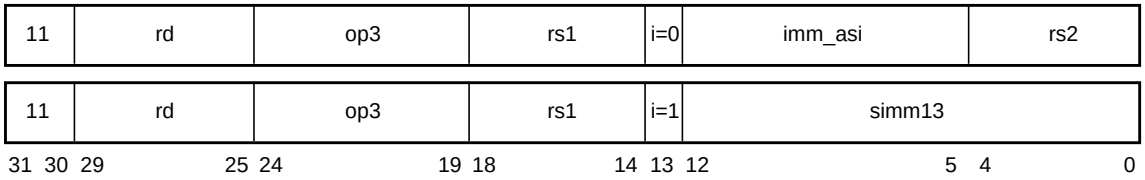
illegal_instruction (op3 = 25_{16} and rd = 2–31)
fp_disabled
mem_address_not_aligned
STDF_mem_address_not_aligned (STDF only)
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.61 Store Floating-Point into Alternate Space

Opcode	op3	rd	Operation
STFA$^{P_{ASI}}$	11 0100	0–31	Store Floating-Point Register to Alternate Space
STDFA$^{P_{ASI}}$	11 0111	†	Store Double Floating-Point Register to Alternate Space
STQFA$^{P_{ASI}}$	11 0110	†	Store Quad Floating-Point Register to Alternate Space

† Encoded floating-point register value.

Format (3)



Assembly Language Syntax	
`sta`	*freg$_{rd}$, [regaddr] imm_asi*
`sta`	*freg$_{rd}$, [reg_plus_imm]* `%asi`
`stda`	*freg$_{rd}$, [regaddr] imm_asi*
`stda`	*freg$_{rd}$, [reg_plus_imm]* `%asi`
`stqa`	*freg$_{rd}$, [regaddr] imm_asi*
`stqa`	*freg$_{rd}$, [reg_plus_imm]* %asi

Description

The store single floating-point into alternate space instruction (STFA) copies `f[rd]` into memory.

The store double floating-point into alternate space instruction (STDFA) copies a doubleword from a double floating-point register into a word-aligned doubleword in memory.

The store quad floating-point into alternate space instruction (STQFA) traps to software.

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 zero; 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.

STFA requires word alignment; otherwise, a *mem_address_not_aligned* exception occurs.

STDFA instruction causes a *STDF_mem_address_not_aligned* trap if the effective address is 32-bit aligned but not 64-bit (doubleword) aligned. In this case, the trap handler software shall emulate the STDF instruction and return.

STDFA with certain target ASI is defined to be a 64-byte block-store instruction. See Section A.4, "Block Load and Block Store (VIS I)" for details.

If the floating-point unit is not enabled for the source register `rd` (per `FPRS.FEF` and `PSTATE.PEF`) or if the FPU is not present, store floating-point into alternate space instructions cause a *fp_disabled* exception.

Implementation Note – This check is not made for STQFA. STFA and STDFA cause a *privileged_action* exception if `PSTATE.PRIV` = 0 and bit 7 of the ASI is zero.

Programming Note – In SPARC-V8, some compilers issued sets of single-precision stores when they could not determine that doubleword or quadword operands were properly aligned. For SPARC-V9, since emulation of misaligned stores is expected to be fast, compilers are recommended to issue sets of single-precision stores only when they can determine that doubleword or quadword operands are *not* properly aligned.

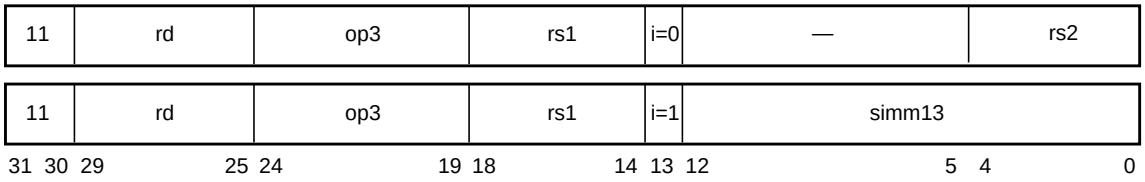
Exceptions

illegal_instruction
fp_disabled
mem_address_not_aligned
STDF_mem_address_not_aligned (STDFA only)
privileged_action
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.62 Store Integer

Opcode	op3	Operation
STB	00 0101	Store Byte
STH	00 0110	Store Halfword
STW	00 0100	Store Word
STX	00 1110	Store Extended Word

Format (3)



Assembly Language Syntax		
stb	*reg*$_{rd}$, [*address*]	(*synonyms*: stub, stsb)
sth	*reg*$_{rd}$, [*address*]	(*synonyms*: stuh, stsh)
stw	*reg*$_{rd}$, [*address*]	(*synonyms*: st, stuw, stsw)
stx	*reg*$_{rd}$, [*address*]	

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.

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, store extended) instruction operates atomically.

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.

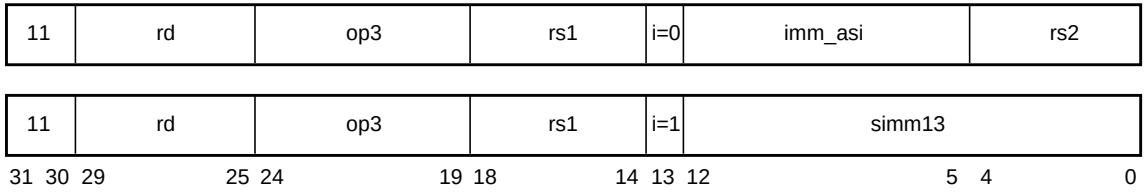
Exceptions

mem_address_not_aligned (all except STB)
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.63 Store Integer into Alternate Space

Opcode	op3	Operation
STBA$^{P_{ASI}}$	01 0101	Store Byte into Alternate Space
STHA$^{P_{ASI}}$	01 0110	Store Halfword into Alternate Space
STWA$^{P_{ASI}}$	01 0100	Store Word into Alternate Space
STXA$^{P_{ASI}}$	01 1110	Store Extended Word into Alternate Space

Format (3)



Assembly Language Syntax		
stba	reg_{rd}, [*regaddr*] *imm_asi*	(*synonyms*: stuba, stsba)
stha	reg_{rd}, [*regaddr*] *imm_asi*	(*synonyms*: stuha, stsha)
stwa	reg_{rd}, [*regaddr*] *imm_asi*	(*synonyms*: sta, stuwa, stswa)
stxa	reg_{rd}, [*regaddr*] *imm_asi*	
stba	reg_{rd}, [*reg_plus_imm*] %asi	(*synonyms*: stuba, stsba)
stha	reg_{rd}, [*reg_plus_imm*] %asi	(*synonyms*: stuha, stsha)
stwa	reg_{rd}, [*reg_plus_imm*] %asi	(*synonyms*: sta, stuwa, stswa)
stxa	reg_{rd}, [*reg_plus_imm*] %asi	

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 zero; 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, store extended) instruction operates atomically.

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.

A store integer into alternate space instruction causes a *privileged_action* exception if PSTATE.PRIV = 0 and bit 7 of the ASI is zero.

Compatibility Note – The SPARC-V8 STA instruction is renamed STWA in SPARC-V9.

Exceptions

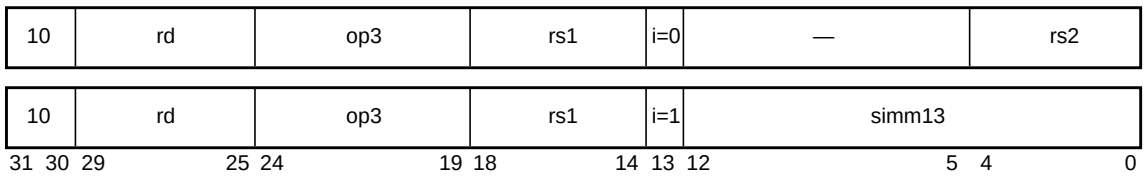
privileged_action
mem_address_not_aligned (all except STBA)
data_access_exception
data_access_error

fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.64 Subtract

Opcode	op3	Operation
SUB	00 0100	Subtract
SUBcc	01 0100	Subtract and modify condition codes
SUBC	00 1100	Subtract with Carry
SUBCcc	01 1100	Subtract with Carry and modify condition codes

Format (3)



Assembly Language Syntax	
sub	reg_{rs1}, *reg_or_imm*, reg_{rd}
subcc	reg_{rs1}, *reg_or_imm*, reg_{rd}
subc	reg_{rs1}, *reg_or_imm*, reg_{rd}
subccc	reg_{rs1}, *reg_or_imm*, reg_{rd}

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 Note – A SUBcc with rd = 0 can be used to effect a signed or unsigned integer comparison.

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).

Exceptions

None

A.65 Tagged Add

Opcode	op3	Operation
TADDcc	10 0000	Tagged Add and modify condition codes

Format (3)

Assembly Language Syntax	
taddcc	reg_{rs1}, *reg_or_imm*, reg_{rd}

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 the sum of bit 31 is different).

If a `TADDcc` causes a tag overflow, the 32-bit overflow bit (`CCR.icc.v`) is set to one; if `TADDcc` does not cause a tag overflow, `CCR.icc.v` is set to zero.

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, based on the normal 64-bit arithmetic overflow condition, like a normal 64-bit add.

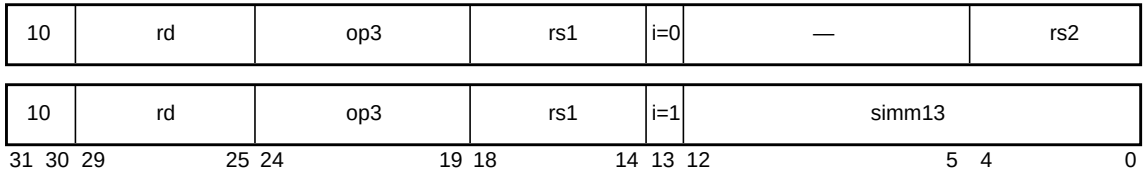
Exceptions

None

A.66 Tagged Subtract

Opcode	op3	Operation
TSUBcc	10 0001	Tagged Subtract and modify condition codes

Format (3)



Assembly Language Syntax	
`tsubcc`	*reg_{rs1}, reg_or_imm, reg_{rd}*

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 one; if `TSUBcc` does not cause a tag overflow, `CCR.icc.v` is set to zero.

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). The `CCR.xcc.v` setting is based only on the normal 64-bit arithmetic overflow condition, like a normal 64-bit subtract.

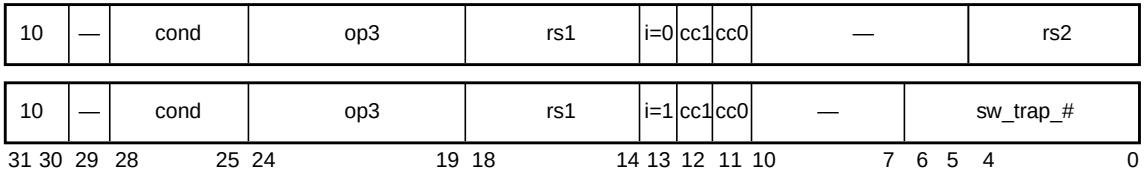
Exceptions

None

A.67 Trap on Integer Condition Codes (Tcc)

Opcode	op3	cond	Operation	*icc* Test
TA	11 1010	1000	Trap Always	1
TN	11 1010	0000	Trap Never	0
TNE	11 1010	1001	Trap on Not Equal	**not** Z
TE	11 1010	0001	Trap on Equal	Z
TG	11 1010	1010	Trap on Greater	**not** (Z **or** (N **xor** V))
TLE	11 1010	0010	Trap on Less or Equal	Z **or** (N **xor** V)
TGE	11 1010	1011	Trap on Greater or Equal	**not** (N **xor** V)
TL	11 1010	0011	Trap on Less	N **xor** V
TGU	11 1010	1100	Trap on Greater Unsigned	**not** (C **or** Z)
TLEU	11 1010	0100	Trap on Less or Equal Unsigned	(C **or** Z)
TCC	11 1010	1101	Trap on Carry Clear (Greater than or Equal, Unsigned)	**not** C
TCS	11 1010	0101	Trap on Carry Set (Less Than, Unsigned)	C
TPOS	11 1010	1110	Trap on Positive or zero	**not** N
TNEG	11 1010	0110	Trap on Negative	N
TVC	11 1010	1111	Trap on Overflow Clear	**not** V
TVS	11 1010	0111	Trap on Overflow Set	V

Format (4)



cc1 cc0	Condition Codes
00	icc
01	—
10	xcc
11	—

Assembly Language Syntax		
ta	i_or_x_cc, software_trap_number	
tn	i_or_x_cc, software_trap_number	
tne	i_or_x_cc, software_trap_number	(*synonym*: tnz)
te	i_or_x_cc, software_trap_number	(*synonym*: tz)
tg	i_or_x_cc, software_trap_number	
tle	i_or_x_cc, software_trap_number	
tge	i_or_x_cc, software_trap_number	
tl	i_or_x_cc, software_trap_number	
tgu	i_or_x_cc, software_trap_number	
tleu	i_or_x_cc, software_trap_number	
tcc	i_or_x_cc, software_trap_number	(*synonym*: tgeu)
tcs	i_or_x_cc, software_trap_number	(*synonym*: tlu)
tpos	i_or_x_cc, software_trap_number	
tneg	i_or_x_cc, software_trap_number	
tvc	i_or_x_cc, software_trap_number	
tvs	i_or_x_cc, software_trap_number	

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* exception is generated. If FALSE, a *trap_instruction* exception does not occur and the instruction behaves like a NOP.

The software trap number is specified by the least significant seven bits of "r[rs1] + r[rs2]" if i = 0, or the least significant seven bits of "r[rs1] + sw_trap_#" if i = 1.

When i = 1, bits 7 through 10 are reserved and should be supplied as zeroes by software. When i = 0, bits 5 through 10 are reserved, the most significant 57 bits of "r[rs1] + r[rs2]" are unused, and both should be supplied as zeroes by software.

Description (Effect on Privileged State)

If a *trap_instruction* traps, 256 plus the software trap number is written into TT[TL]. Then the trap is taken, and the processor performs the normal trap entry procedure.

Programming Note – `Tcc` can be used to implement breakpointing, tracing, and calls to supervisor software. It can also be used for runtime checks, such as out-of-range array indexes, integer overflow, and so on.

Compatibility Note – `Tcc` is upward compatible with the SPARC-V8 `Ticc` instruction, with one qualification: a `Ticc` with `i` = 1 and `simm13` < 0 may execute differently on a SPARC-V9 processor. Use of the `i` = 1 form of `Ticc` is believed to be rare in SPARC-V8 software, and `simm13` < 0 is probably not used at all; therefore, it is believed in practice, that full software compatibility will be achieved.

Exceptions

trap_instruction
illegal_instruction (`cc1` ▯ `cc0` = 01_2 or 11_2, or reserved fields nonzero)

A.68 Write Privileged Register

Opcode	op3	Operation
`WRPR`[P]	11 0010	Write Privileged Register

Format (3)

rd	Privileged Register
0	TPC
1	TNPC
2	TSTATE
3	TT
4	TICK
5	TBA
6	PSTATE
7	TL
8	PIL
9	CWP
10	CANSAVE
11	CANRESTORE
12	CLEANWIN
13	OTHERWIN
14	WSTATE
15–31	*Reserved*

Assembly Language Syntax	
wrpr	reg_{rs1}, *reg_or_imm*, %tpc
wrpr	reg_{rs1}, *reg_or_imm*, %tnpc
wrpr	reg_{rs1}, *reg_or_imm*, %tstate
wrpr	reg_{rs1}, *reg_or_imm*, %tt
wrpr	reg_{rs1}, *reg_or_imm*, %tick
wrpr	reg_{rs1}, *reg_or_imm*, %tba
wrpr	reg_{rs1}, *reg_or_imm*, %pstate
wrpr	reg_{rs1}, *reg_or_imm*, %tl
wrpr	reg_{rs1}, *reg_or_imm*, %pil
wrpr	reg_{rs1}, *reg_or_imm*, %cwp
wrpr	reg_{rs1}, *reg_or_imm*, %cansave
wrpr	reg_{rs1}, *reg_or_imm*, %canrestore
wrpr	reg_{rs1}, *reg_or_imm*, %cleanwin
wrpr	reg_{rs1}, *reg_or_imm*, %otherwin
wrpr	reg_{rs1}, *reg_or_imm*, %wstate

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 an exclusive OR.

The rd field in the instruction determines the privileged register that is written. There are at least four 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 write to TPC, TNPC, TT, or TSTATE when the trap level is zero (TL = 0) causes an *illegal_instruction* exception.

A WRPR of TL does not cause a trap or return from trap; it does not alter any other machine state.

Programming Note – A WRPR of TL can be used to read the values of TPC, TNPC, and TSTATE for any trap level; however, make sure 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 processor state made by the WRPR.

WRPR instructions with rd in the range 15–31 are reserved for future versions of the architecture; executing a WRPR instruction with rd in that range causes an *illegal_instruction* exception.

Implementation Note – Some WRPR instructions could serialize the processor in some implementations.

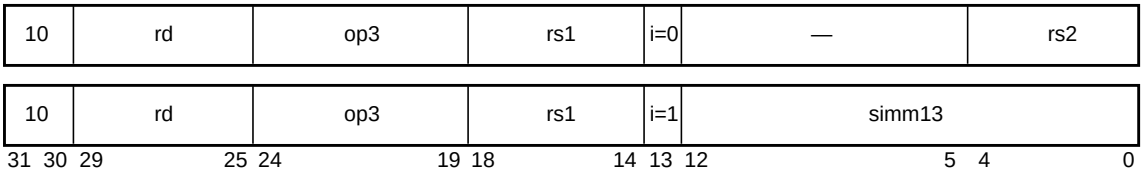
Exceptions

privileged_opcode
illegal_instruction ((rd = 15–31) or ((rd ≤ 3) and (TL = 0)))

A.69 Write State Register

Opcode	op3	rd	Operation
WRY[D]	11 0000	0	Write Y register; deprecated (see Section A.70.18, "Write Y Register").
—	11 0000	1	*Reserved, do not access; attempt to access causes an illegal_instruction exception.*
WRCCR	11 0000	2	Write Condition Codes Register
WRASI	11 0000	3	Write Graphics Status Register
—	11 0000	4, 5	*Reserved, do not access; attempt to access causes an illegal_instruction exception.*
WRFPRS	11 0000	6	Write Floating-Point Registers Status Register
—	11 0000	7–14	*Reserved, do not access; attempt to access causes an illegal_instruction exception.*
—	11 0000	15	Software-initiated reset (see Section A.59, "Software-Initiated Reset").
WRASR	11 0000	16–31	Write non-SPARC-V9 ASRs
WRPCR[P$_{PCR}$]		16	Write Performance Control Registers (PCR)
WRPIC[P$_{PIC}$]		17	Write Performance Instrumentation Counters (PIC)
WRDCR[P]		18	Write Dispatch Control Register (DCR)
WRGSR		19	Write Graphic Status Register (GSR)
WRSOFTINT_SET[P]		20	Set bits of per-processor Soft Interrupt Register
WRSOFTINT_CLR[P]		21	Clear bits of per-processor Soft Interrupt Register
WRSOFTINT[P]		22	Write per-processor Soft Interrupt Register
WRTICK_CMPR[P]		23	Write Tick Compare Register
WRSTICK[P]		24	Write System TICK Register
WRSTICK_CMPR[P]		25	Write System TICK Compare Register
—		26–31	*Reserved, do not access; attempt to access causes an illegal_instruction exception.*

Format (3)



Assembly Language Syntax		
wr	*reg*$_{rs1}$, *reg_or_imm*,	%ccr
wr	*reg*$_{rs1}$, *reg_or_imm*,	%asi
wr	*reg*$_{rs1}$, *reg_or_imm*,	%fprs
wr	*reg*$_{rs1}$, *reg_or_imm*,	%pcr
wr	*reg*$_{rs1}$, *reg_or_imm*,	%pic
wr	*reg*$_{rs1}$, *reg_or_imm*,	%dcr
wr	*reg*$_{rs1}$, *reg_or_imm*,	%gsr
wr	*reg*$_{rs1}$, *reg_or_imm*,	%set_softint
wr	*reg*$_{rs1}$, *reg_or_imm*,	%clear_softint
wr	*reg*$_{rs1}$, *reg_or_imm*,	%softint
wr	*reg*$_{rs1}$, *reg_or_imm*,	%tick_cmpr
wr	*reg*$_{rs1}$, *reg_or_imm*,	%sys_tick
wr	*reg*$_{rs1}$, *reg_or_imm*,	%sys_tick_cmpr

Description

These instructions store 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 state register.

Note – The operation is an exclusive OR.

WRASR writes a value to the ancillary state register (ASR) indicated by rd. The operation performed to generate the value written may be rd dependent or implementation dependent (see below). A WRASR instruction is indicated by op = 2, rd = ≥ 16, and op3 = 30_{16}.

The WRASR opcode for rd = 15, rs1 = 0, and i = 1 is used for the software-initiated reset (SIR) instruction (see Section A.59, "Software-Initiated Reset").

The WRCCR, WRFPRS, and WRASI instructions are *not* delayed-write instructions. The instruction immediately following a WRCCR, WRFPRS, or WRASIR 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.

WRGSR causes a *fp_disabled* trap if PSTATE.PEF = 0 or FPRS.FEF = 0.

WRPIC causes a *privileged_action* exception if PSTATE.PRIV = 0 and PCR.PRIV = 1.

WRPCR causes a *privileged_opcode* exception due to access privilege violation.

Implementation Note – Ancillary state registers may include, for example, timer, counter, diagnostic, self-test, and trap-control registers.

Compatibility Note – The SPARC-V8 WRIER, WRPSR, WRWIM, and WRTBR instructions do not exist in SPARC-V9 because the IER, PSR, TBR, and WIM registers do not exist in SPARC-V9.

Implementation Note – Some WRASR instructions could serialize the processor in some implementations.

Exceptions

software_initiated_reset (rd = 15, rs1 = 0, and i = 1 only)
privileged_opcode (WRDCR, WRSOFTINT_SET, WRSOFTINT_CLR, WRSOFTINT, WRTICK_CMPR, WRSTICK, WRSTICK_CMPR, and WRPCR)
illegal_instruction (WRASR with rd = 1, 4, 5, 7–14, 26–31; WRASR with rd = 15 and rs1 ≠ 0 or i ≠ 1)
privileged_action (WRPIC with PSTATE.PRIV = 0 and PCR.PRIV = 1)
fp_disabled (WRGSR with PSTATE.PEF = 0 or FPRS.FEF = 0)

A.70 Deprecated Instructions

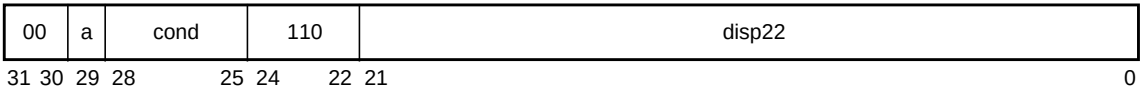
The following instructions are deprecated; they are provided only for compatibility with previous versions of the architecture. They should not be used in new SPARC-V9 software. For each deprecated instruction, another instruction is recommended to be used instead. Please see TABLE A-2 for the page number at which you can find a description of the preferred instruction.

A.70.1 Branch on Floating-Point Condition Codes (FBfcc)

The `FBfcc` instructions are deprecated. Use the `FBPfcc` instructions instead.

Opcode	cond	Operation	*fcc* Test
FBA^{D}	1000	Branch Always	1
FBN^{D}	0000	Branch Never	0
FBU^{D}	0111	Branch on Unordered	U
FBG^{D}	0110	Branch on Greater	G
FBUG^{D}	0101	Branch on Unordered or Greater	G **or** U
FBL^{D}	0100	Branch on Less	L
FBUL^{D}	0011	Branch on Unordered or Less	L **or** U
FBLG^{D}	0010	Branch on Less or Greater	L **or** G
FBNE^{D}	0001	Branch on Not Equal	L **or** G **or** U
FBE^{D}	1001	Branch on Equal	E
FBUE^{D}	1010	Branch on Unordered or Equal	E **or** U
FBGE^{D}	1011	Branch on Greater or Equal	E **or** G
FBUGE^{D}	1100	Branch on Unordered or Greater or Equal	E **or** G **or** U
FBLE^{D}	1101	Branch on Less or Equal	E **or** L
FBULE^{D}	1110	Branch on Unordered or Less or Equal	E **or** L **or** U
FBO^{D}	1111	Branch on Ordered	E **or** L **or** G

Format (2)



Assembly Language Syntax		
`fba{,a}`	*label*	
`fbn{,a}`	*label*	
`fbu{,a}`	*label*	
`fbg{,a}`	*label*	
`fbug{,a}`	*label*	
`fbl{,a}`	*label*	
`fbul{,a}`	*label*	
`fblg{,a}`	*label*	
`fbne{,a}`	*label*	(*synonym*: `fbnz`)
`fbe{,a}`	*label*	(*synonym*: `fbz`)
`fbue{,a}`	*label*	
`fbge{,a}`	*label*	
`fbuge{,a}`	*label*	
`fble{,a}`	*label*	
`fbule{,a}`	*label*	
`fbo{,a}`	*label*	

Programming Note – To set the annul 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 zero, an `FBN` (Branch Never) instruction acts like a NOP. If its annul field is one, 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 one, the delay instruction is annulled (not executed). If the annul field is zero, 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 field. If a conditional branch is not taken and the annul (a) field is one, the delay instruction is annulled (not executed).

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

Compatibility Note – Unlike SPARC-V8, SPARC-V9 does not require an instruction between a floating-point compare operation and a floating-point branch (FBfcc, FBPfcc).

If FPRS.FEF = 0 or PSTATE.PEF = 0, or if an FPU is not present, the FBfcc instruction is not executed and instead generates a *fp_disabled* exception.

Exceptions

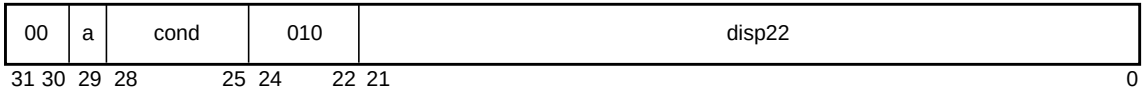
fp_disabled

A.70.2 Branch on Integer Condition Codes (Bicc)

Use the BPcc instructions in place of Bicc instructions.

Opcode	cond	Operation	*icc* Test
BA^{D}	1000	Branch Always	1
BN^{D}	0000	Branch Never	0
BNE^{D}	1001	Branch on Not Equal	**not** Z
BE^{D}	0001	Branch on Equal	Z
BG^{D}	1010	Branch on Greater	**not** (Z **or** (N **xor** V))
BLE^{D}	0010	Branch on Less or Equal	Z **or** (N **xor** V)
BGE^{D}	1011	Branch on Greater or Equal	**not** (N **xor** V)
BL^{D}	0011	Branch on Less	N **xor** V
BGU^{D}	1100	Branch on Greater Unsigned	**not** (C **or** Z)
BLEU^{D}	0100	Branch on Less or Equal Unsigned	C **or** Z
BCC^{D}	1101	Branch on Carry Clear (Greater Than or Equal, Unsigned)	**not** C
BCS^{D}	0101	Branch on Carry Set (Less Than, Unsigned)	C
BPOS^{D}	1110	Branch on Positive	**not** N
BNEG^{D}	0110	Branch on Negative	N
BVC^{D}	1111	Branch on Overflow Clear	**not** V
BVS^{D}	0111	Branch on Overflow Set	V

Format (2)



Assembly Language Syntax		
`ba{,a}`	*label*	
`bn{,a}`	*label*	
`bne{,a}`	*label*	(*synonym*: `bnz`)
`be{,a}`	*label*	(*synonym*: `bz`)
`bg{,a}`	*label*	
`ble{,a}`	*label*	
`bge{,a}`	*label*	
`bl{,a}`	*label*	
`bgu{,a}`	*label*	
`bleu{,a}`	*label*	
`bcc{,a}`	*label*	(*synonym*: `bgeu`)
`bcs{,a}`	*label*	(*synonym*: `blu`)
`bpos{,a}`	*label*	
`bneg{,a}`	*label*	
`bvc{,a}`	*label*	
`bvs{,a}`	*label*	

Programming Note – To set the annul 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.

Description

Unconditional branches and icc-conditional branches are described below:

- **Unconditional branches (BA, BN)** — If its annul field is zero, a `BN` (Branch Never) instruction is treated as a NOP. If its annul field is one, 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 field of the branch instruction is one, the delay instruction is annulled (not executed). If the annul field is zero, 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 (a) field is one, the delay instruction is annulled (not executed).

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

Exceptions

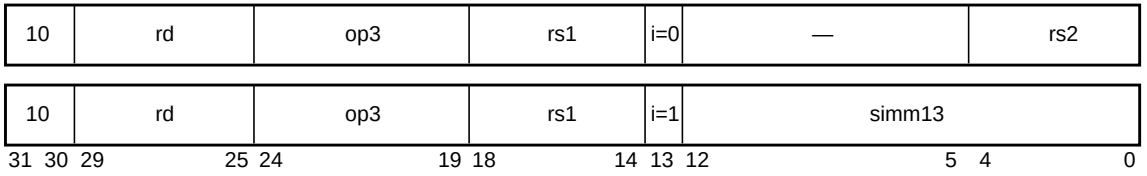
None

A.70.3 Divide (64-bit / 32-bit)

The UDIV, UDIVcc, SDIV, and SDIVcc instructions are deprecated. Use the UDIVX and SDIVX instructions instead.

Opcode	op3	Operation
UDIV^{D}	00 1110	Unsigned Integer Divide
SDIV^{D}	00 1111	Signed Integer Divide
UDIVcc^{D}	01 1110	Unsigned Integer Divide and modify condition codes
SDIVcc^{D}	01 1111	Signed Integer Divide and modify condition codes

Format (3)



Assembly Language Syntax	
`udiv`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*
`sdiv`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*
`udivcc`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*
`sdivcc`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description

The divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If `i` = 0, they compute "(`Y` ▯ `r[rs1]`<31:0>) ÷ `r[rs2]`<31:0>." Otherwise (that is, if `i` = 1), the divide instructions compute "(`Y` ▯ `r[rs1]`<31:0>) ÷ (`sign_ext(simm13)`<31:0>)." In either case, if overflow does not occur, the less significant 32 bits of the integer quotient are sign-extended 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.

In the UltraSPARC IIIi processor, `LDD` is implemented in hardware.

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 A-16.

TABLE A-16 `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	UDIVcc
`icc.N`	Set if `r[rd]`<31> = 1
`icc.Z`	Set if `r[rd]`<31:0> = 0
`icc.V`	Set if overflow (per TABLE A-16)
`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

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 A-17.

TABLE A-17 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	SDIVcc
icc.N	Set if r[rd]<31> = 1
icc.Z	Set if r[rd]<31:0> = 0
icc.V	Set if overflow (per TABLE A-17)
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

Exceptions

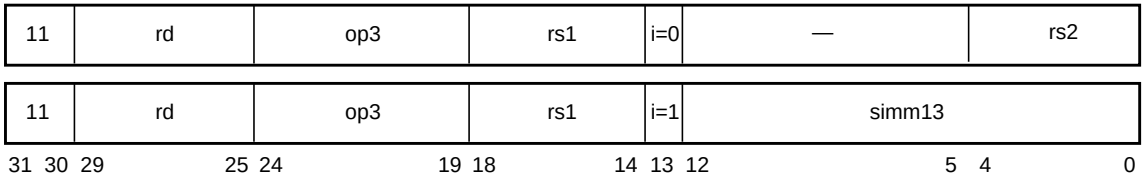
division_by_zero

A.70.4 Load Floating-Point Status Register

The LDFSR instruction is deprecated. Use the LDXFSR instruction instead.

Opcode	op3	rd	Operation
LDFSR^{D}	10 0001	0	Load Floating-Point State Register Lower

Format (3)



Assembly Language Syntax	
`ld`	[*address*], `%fsr`

Description

The load floating-point state register lower instruction (LDFSR) waits for all FPop instructions that have not finished execution to complete and then loads a word from memory into the less significant 32 bits of the FSR. The upper 32 bits of FSR are unaffected by LDFSR.

LDFSR causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned.

Compatibility Note – SPARC-V9 supports two different instructions to load the FSR: the SPARC-V8 LDFSR instruction is defined to load only the less significant 32 bits of the FSR, whereas LDXFSR allows SPARC-V9 programs to load all 64 bits of the FSR.

Exceptions

mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

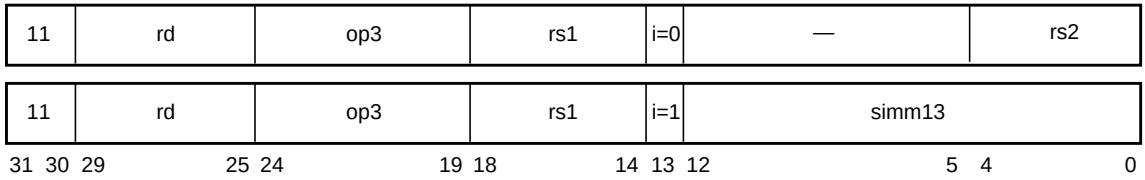
A.70.5 Load Integer Doubleword

The LDD instruction is deprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC-V9 software. Use the LDX instruction instead.

Please refer to Section A.27, “Load Integer” for the current load integer instructions.

Opcode	op3	Operation
LDD[D]	00 0011	Load doubleword

Format (3)



Assembly Language Syntax	
ldd	*[address], reg$_{rd}$*

Description

The load doubleword integer instruction (LDD) copies a doubleword from memory into an r register pair. The word at the effective memory address is copied into the even r register. The word at the effective memory address + 4 is copied into the following odd-numbered r register. The upper 32 bits of both the even-numbered and odd-numbered r registers are zero-filled.

Notes – A load doubleword with `rd` = 0 modifies only `r[1]`. The least significant bit of the `rd` field in an `LDD` instruction is unused and should be set to zero by software. An attempt to execute a load doubleword instruction that refers to a misaligned (odd-numbered) destination register causes an *illegal_instruction* exception.

With respect to little-endian memory, an `LDD` instruction behaves as if it is composed of two 32-bit loads, each of which is byte swapped independently before being written into each destination register.

Load integer doubleword instructions access the primary address space (ASI = 80_{16}). The effective address is "`r[rs1]` + `r[rs2]`" if `i` = 0, or "`r[rs1]` + `sign_ext(simm13)`" if `i` = 1.

A successful load doubleword instruction operates atomically.

Programming Note – `LDD` is provided for compatibility with SPARC-V8. It may execute slowly on SPARC-V9 machines because of data path and register-access difficulties.

Exceptions

illegal_instruction (`LDD` with odd `rd`)
mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

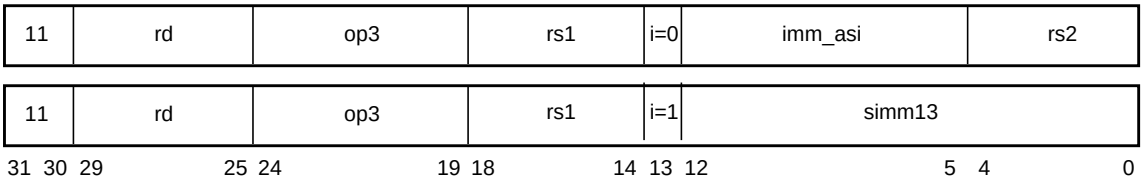
A.70.6 Load Integer Doubleword from Alternate Space

The `LDDA` instruction is deprecated. Use the `LDXA` instruction in its place.

Please refer to Section A.28, "Load Integer from Alternate Space" for current load integer from alternate space instructions.

Opcode	op3	Operation
$\texttt{LDDA}^{D,\ P_{ASI}}$	01 0011	Load Doubleword from Alternate Space

Format (3)



Assembly Language Syntax	
`ldda`	[*regaddr*] *imm_asi*, reg_{rd}
`ldda`	[*reg_plus_imm*] `%asi`, reg_{rd}

Description

The load doubleword integer from alternate space instruction (`LDDA`) copies a doubleword from memory into an `r` register pair. The word at the effective memory address is copied into the even `r` register. The word at the effective memory address + 4 is copied into the following odd-numbered `r` register. The upper 32 bits of both the even-numbered and odd-numbered `r` registers are zero-filled.

Notes – A load doubleword with `rd` = 0 modifies only `r[1]`. The least significant bit of the `rd` field in an `LDDA` instruction is unused and should be set to zero by software. An attempt to execute a load doubleword instruction that refers to a misaligned (odd-numbered) destination register causes an *illegal_instruction* exception.

With respect to little-endian memory, an `LDDA` instruction behaves as if it is composed of two 32-bit loads, each of which is byte-swapped independently before being written into each destination register.

The load integer doubleword 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 zero; 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 doubleword instruction operates atomically.

`LDDA` causes a *mem_address_not_aligned* exception if the address is not doubleword aligned.

These instructions cause a *privileged_action* exception if `PSTATE.PRIV` = 0 and bit 7 of the ASI is zero.

In the UltraSPARC IIIi processor, LDDA is implemented in hardware.

LDDA with ASI=24_{16} or $2C_{16}$ is defined to be a Load Quadword Atomic instruction. See Section A.29, “Load Quadword, Atomic (VIS I)” for details.

Programming Note – LDDA is provided for compatibility with SPARC-V8. It may execute slowly on SPARC-V9 machines because of data path and register-access difficulties.

If LDDA is emulated in software, an LDXA instruction should be used for the memory access in order to preserve atomicity.

Exceptions

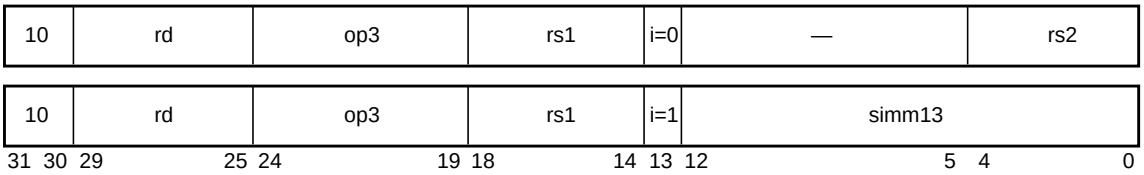
privileged_action
illegal_instruction (LDDA with odd rd)
mem_address_not_aligned
data_access_exception
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.70.7 Multiply (32-bit)

The UMUL, UMULcc, SMUL, and SMULcc instructions are deprecated. Use the MULX instruction instead.

Opcode	op3	Operation
UMUL^{D}	00 1010	Unsigned Integer Multiply
SMUL^{D}	00 1011	Signed Integer Multiply
UMULcc^{D}	01 1010	Unsigned Integer Multiply and modify condition codes
SMULcc^{D}	01 1011	Signed Integer Multiply and modify condition codes

Format (3)



Assembly Language Syntax	
`umul`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
`smul`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
`umulcc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
`smulcc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description

The multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute "`r[rs1]`<31:0> × `r[rs2]`<31:0>" if `i` = 0, or "`r[rs1]`<31:0> × `sign_ext(simm13)`<31:0>" if `i` = 1. They write the 32 most significant bits of the product into the `Y` register and all 64 bits of the product into `r[rd]`.

Unsigned multiply instructions (`UMUL`, `UMULcc`) operate on unsigned integer word operands and compute an unsigned integer doubleword product. Signed multiply instructions (`SMUL`, `SMULcc`) operate on signed integer word operands and compute a signed integer doubleword product.

`UMUL` and `SMUL` do not affect the condition code bits. `UMULcc` and `SMULcc` write the integer condition code bits, `icc` and `xcc`, as shown in TABLE A-18.

Note – Zero (`icc.Z`) and 32-bit negative (`icc.N`) condition codes are set according to the *less* significant word of the product, not according to the full 64-bit result.

TABLE A-18 UMULcc / SMULcc Condition Code Settings

Bit	UMULcc / SMULcc
`icc.N`	Set if product<31> = 1
`icc.Z`	Set if product<31:0>= 0
`icc.V`	0
`icc.C`	0
`xcc.N`	Set if product<63> = 1
`xcc.Z`	Set if product<63:0> = 0
`xcc.V`	0
`xcc.C`	0

Programming Notes – 32-bit overflow after `UMUL`/`UMULcc` is indicated by `Y` ≠ 0.

`Y` ≠ (`r[rd]` >> 31) is indicates 32-bit overflow after `SMUL`/`SMULcc`, where ">>" indicates 32-bit arithmetic right-shift.

Exceptions

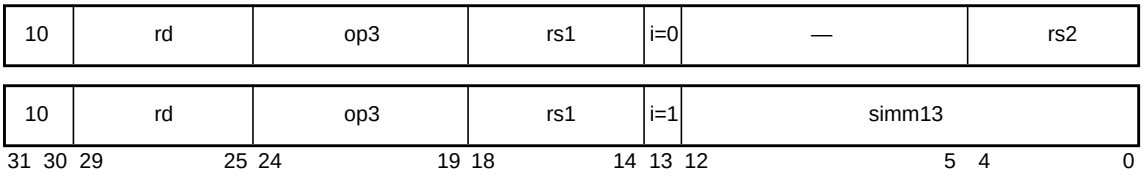
None

A.70.8 Multiply Step

The `MULScc` instruction is deprecated. Use the `MULX` instruction instead.

Opcode	op3	Operation
`MULScc`^{D}	10 0100	Multiply Step and modify condition codes

Format (3)



Assembly Language Syntax	
`mulscc`	*reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description

`MULScc` treats the less significant 32 bits of both `r[rs1]` and 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 adds, 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 – A standard `MULScc` instruction has `rs1` = `rd`.

`MULScc` operates as follows:

1. The multiplicand is `r[rs2]` if `i` = 0, or `sign_ext(simm13)` if `i` = 1.
2. A 32-bit value is computed by shifting `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 `Y` = 0, then zero is added to the shifted value from step (2).
4. The sum from step (3) is written into `r[rd]`. The upper 32 bits of `r[rd]` are undefined. The integer condition codes are updated according to the addition performed in step (3). The values of the extended condition codes are 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.

Exceptions

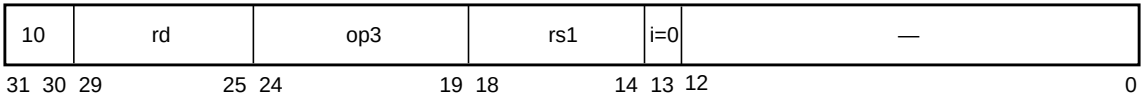
None

A.70.9 Read Y Register

The RDY instruction from the Read State Register instructions (Section A.50, “Read State Register”) is deprecated. It is recommended that all instructions that reference the Y register be avoided.

Opcode	op3	rs1	Operation
RDY[D]	10 1000	0	Read Y Register

Format (3)



Assembly Language Syntax	
rd	%y, reg_{rd}

Description

This instruction reads the Y register into r[rd].

Exceptions

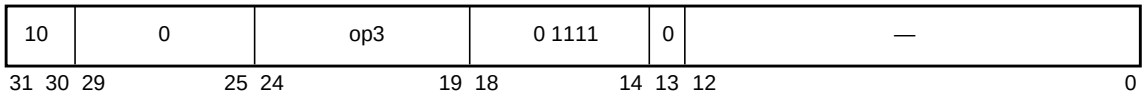
None

A.70.10 Store Barrier

The STBAR instruction is deprecated. Use the MEMBAR instruction instead.

Opcode	op3	Operation
STBAR[D]	10 1000	Store Barrier

Format (3)



Assembly Language Syntax
`stbar`

Description

The store barrier instruction (STBAR) forces *all* store and atomic load-store operations issued by a processor prior to the STBAR to complete their effects on memory before *any* store or atomic load-store operations issued by that processor subsequent to the STBAR are executed by memory.

Note – The encoding of STBAR is identical to that of the RDASR instruction except that `rs1` = 15 and `rd` = 0, and it is identical to that of the MEMBAR instruction except that bit 13 (`i`) = 0.

Compatibility Note – STBAR is identical in function to a MEMBAR instruction with `mmask` = 8_{16}. STBAR is retained for compatibility with SPARC-V8.

Implementation Note – For correctness, it is sufficient for a processor to stop issuing new store and atomic load-store operations when an STBAR is encountered and to resume after all stores have completed and are observed in memory by all processors. More efficient implementations may take advantage of the fact that the processor is allowed to issue store and load-store operations after the STBAR, as long as those operations are guaranteed not to become visible before all the earlier stores and atomic load-stores have become visible to all processors.

Exceptions

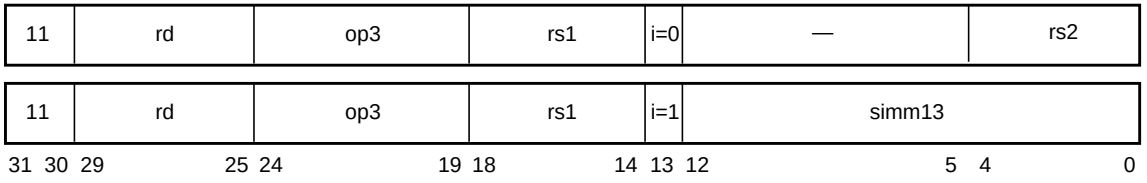
None

A.70.11 Store Floating-Point Status Register Lower

The STFSR instruction is deprecated. Use the STXFSR instruction instead.

Opcode	op3	rd	Operation
STFSR^{D}	10 0101	0	Store Floating-Point State Register Lower

Format (3)



Assembly Language Syntax	
st	%fsr, [*address*]

Description

The store floating-point state register lower instruction (STFSR) waits for any currently executing FPop instructions to complete, and then it writes the less significant 32 bits of the FSR into memory.

Compatibility Note – SPARC-V9 needs two store-FSR instructions, since the SPARC-V8 STFSR instruction is defined to store only 32 bits of the FSR into memory. STXFSR allows SPARC-V9 programs to store all 64 bits of the FSR.

STFSR zeroes FSR.ftt after writing the FSR to memory.

Implementation Note – FSR.ftt should not be zeroed until it is known that the store will not cause a precise trap.

The effective address for this instruction is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

STFSR causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned.

Exceptions

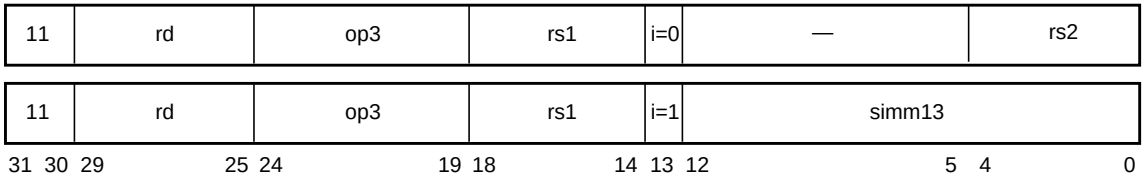
illegal_instruction (op3 = 25_{16} and rd = 2–31)
fp_disabled
mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.70.12 Store Integer Doubleword

The STD instruction is deprecated. Use the STX instruction instead.

Opcode	op3	Operation
STD[D]	00 0111	Store Doubleword

Format (3)



Assembly Language Syntax	
`std`	*reg*$_{rd}$, [*address*]

Description

The store doubleword integer instruction (`STD`) 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 doubleword instruction is unused and should always be set to zero by software. An attempt to execute a store doubleword instruction that refers to a misaligned (odd-numbered) `rd` causes an *illegal_instruction* exception.

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 doubleword instruction operates atomically.

`STD` causes a *mem_address_not_aligned* exception if the effective address is not doubleword aligned.

In the UltraSPARC IIIi processor, `STD` is implemented in hardware.

Programming Notes – `STD` is provided for compatibility with SPARC-V8. It may execute slowly on SPARC-V9 machines because of data path and register-access difficulties. Therefore, programmers should avoid using `STD`.

If `STD` is emulated in software, `STX` should be used to preserve atomicity.

With respect to little-endian memory, a `STD` instruction behaves as if it is composed of two 32-bit stores, each of which is byte-swapped independently before being written into each destination memory word.

Exceptions

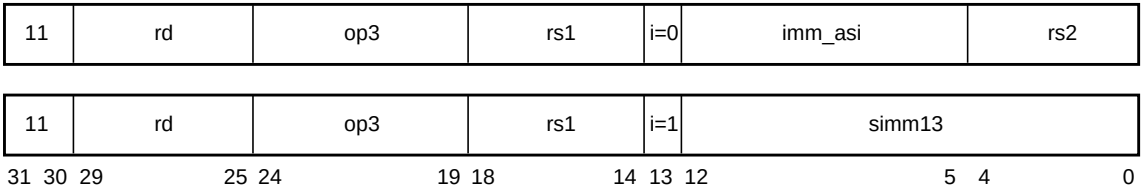
illegal_instruction (STD with odd rd)
mem_address_not_aligned (all except STB)
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.70.13 Store Integer Doubleword into Alternate Space

The STDA instruction is deprecated. Instead, use the STXA instruction.

Opcode	op3	Operation
STDA$^{D, PASI}$	01 0111	Store Doubleword into Alternate Space

Format (3)



Assembly Language Syntax	
stda	*reg$_{rd}$*, [*reg_plus_imm*] %asi

Description

The store doubleword integer instruction (STDA) 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 doubleword instruction is unused and should always be set to zero by software. An attempt to execute a store doubleword instruction that refers to a misaligned (odd-numbered) rd causes an *illegal_instruction* exception.

Store integer doubleword 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 zero; 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 doubleword instruction operates atomically.

STDA causes a *mem_address_not_aligned* exception if the effective address is not doubleword aligned.

A store integer into alternate space instruction causes a *privileged_action* exception if PSTATE.PRIV = 0 and bit 7 of the ASI is zero.

In the UltraSPARC IIIi processor, STDA is implemented in hardware.

Programming Note – STDA is provided for compatibility with SPARC-V8. It may execute slowly on SPARC-V9 machines because of data path and register-access difficulties. Therefore, programmers should avoid using STDA.

Exceptions

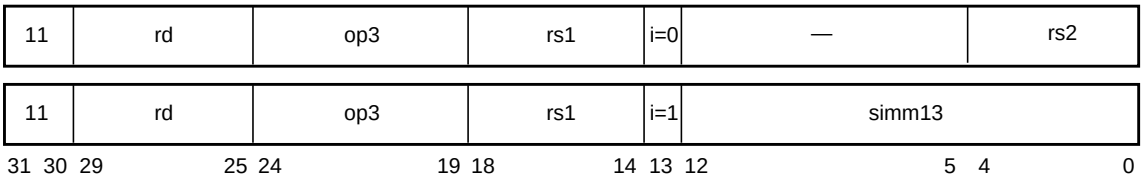
illegal_instruction (STDA with odd rd)
privileged_action
mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.70.14 Swap Register with Memory

The SWAP instruction is deprecated. Use the CASA or CASXA instruction in its place.

Opcode	op3	Operation
SWAP[D]	00 1111	Swap Register with Memory

Format (3)



Assembly Language Syntax	
`swap`	[*address*], *reg*$_{rd}$

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 zero. The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more 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 effective address for these instructions 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.

The coherence and atomicity of memory operations between processors and I/O DMA memory accesses are implementation dependent.

Implementation Note – See *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, a document available from SPARC International, for information on the presence of hardware support for these instructions in the various SPARC-V9 implementations.

Exceptions

mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss

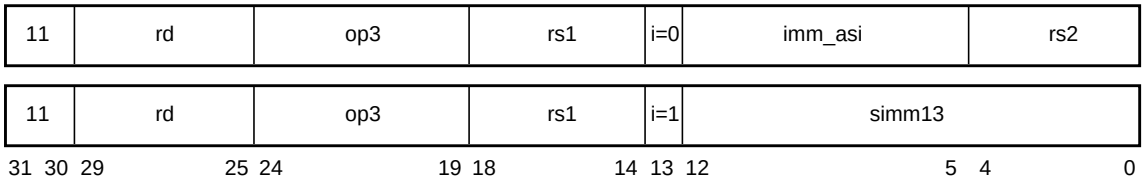
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.70.15 Swap Register with Alternate Space Memory

The SWAPA instruction is deprecated. Use the CASXA instruction instead.

Opcode	op3	Operation
SWAPA$^{D, P_{ASI}}$	01 1111	Swap register with Alternate Space Memory

Format (3)



Assembly Language Syntax	
`swapa`	[*regaddr*] *imm_asi*, *reg$_{rd}$*
`swapa`	[*reg_plus_imm*] `%asi`, *reg$_{rd}$*

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 zero. The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more 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 zero; 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 zero.

The coherence and atomicity of memory operations between processors and I/O DMA memory accesses are implementation dependent.

Implementation Note – See *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, a document available from SPARC International, for information on the presence of hardware support for this instruction in the various SPARC-V9 implementations.

Exceptions

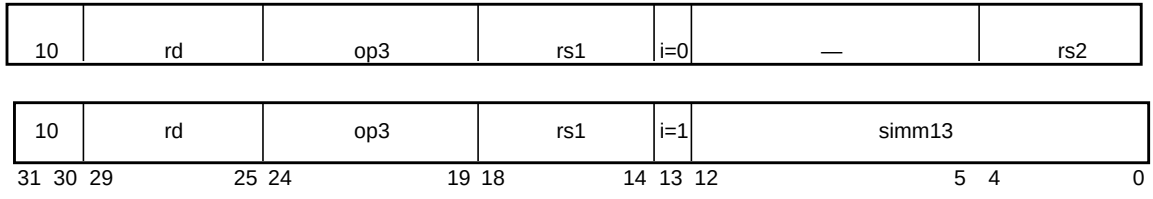
mem_address_not_aligned
privileged_action
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

A.70.16 Tagged Add and Trap on Overflow

The TADDccTV instruction is deprecated. Use the TADDcc followed by BPVS instruction (with instructions to save the pre-TADDcc integer condition codes if necessary).

Opcode	op3	Operation
TADDccTV^{D}	10 0010	Tagged Add and modify condition codes, or Trap on Overflow

Format (3)



Assembly Language Syntax	
`taddcctv`	*reg_{rs1}, reg_or_imm, reg_{rd}*

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.

A *tag_overflow* exception 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 the sum of bit 31 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 zero 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, based only on the normal 64-bit arithmetic overflow condition, like a normal 64-bit add.

Compatibility Note – `TADDccTV` traps based on the 32-bit overflow condition, just as in SPARC-V8. Although the tagged add instructions set the 64-bit condition codes `CCR.xcc`, there is no form of the instruction that traps the 64-bit overflow condition.

Exceptions

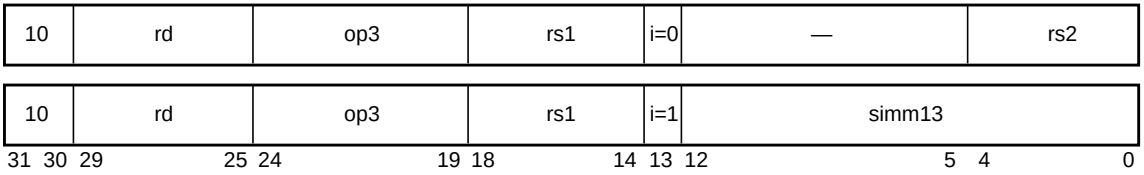
tag_overflow

A.70.17 Tagged Subtract and Trap on Overflow

The `TSUBccTV` instruction is deprecated. Use the `TSUBcc` instruction followed by `BPVS` (with instructions to save the pre-`TSUBcc` integer condition codes if necessary).

Opcode	op3	Operation
TSUBccTV[D]	10 0011	Tagged Subtract and modify condition codes, or Trap on Overflow

Format (3)



Assembly Language Syntax
`tsubcctv` *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

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 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 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 zero 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, based only on the normal 64-bit arithmetic overflow condition, like a normal 64-bit subtract.

Compatibility Note – TSUBccTV traps are based on the 32-bit overflow condition, just as in SPARC-V8. Although the tagged-subtract instructions set the 64-bit condition codes CCR.xcc, there is no form of the instruction that traps on 64-bit overflow.

Exceptions

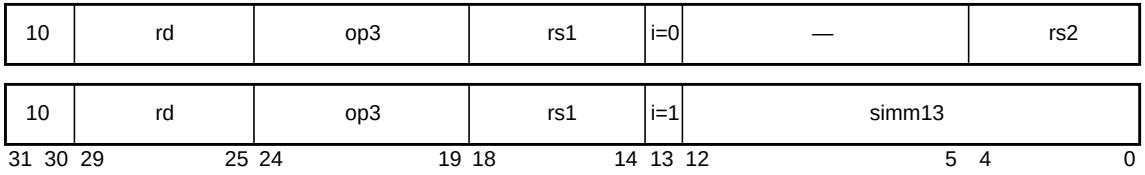
tag_overflow

A.70.18 Write Y Register

The WRY instruction is deprecated. It is recommended that all instructions that reference the Y register be avoided.

Opcode	op3	rd	Operation
WRY[D]	11 0000	0	Write Y register
—	11 0000	1–31	See Section A.69, “Write State Register”

Format (3)



Assembly Language Syntax	
wr	*reg*$_{rs1}$, *reg_or_imm*,%y

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 Y register.

Note – The operation is an exclusive OR.

The WRY instruction is *not* a delayed-write instruction. The instruction immediately following a WRY observes the new value of the Y register.

Exceptions

None

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z