UltraSPARC® III Cu

User's Manual

Version 2.2.1

January 2004

Table of Contents

List of Figures

List of Tables

Preface

Welcome to the UltraSPARC® III Cu User's Manual. This book contains information about the architecture and programming of the UltraSPARC III Cu 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 III Cu 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 III Cu 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-endian and little-endian byte orders

The UltraSPARC processor 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 III Cu processor. However, sometimes it 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 III Cu processor.

This manual is divided into multiple sections. The following sections are described:

Processor Introduction

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

Architecture and Functions

This section discusses the details of the UltraSPARC III Cu architecture and the functions of various CPU 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.

Supervisor Programming

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

Performance Programming

This section explores the opportunities to exploit the high-performance architecture of the processor. Chapters are devoted to performance instrumentation and prefetch, two special features of the UltraSPARC III Cu processor.

Instruction Definitions Appendix

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

SPARC V9 Architecture

The SPARC Architecture Manual, Version 9 was used to implement the CPU in 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* font is used for emphasis, book titles, the first instance of a word that is defined, and assembly language terms.
- *Italic sans serif* font is used for exception and trap names. "The *privileged_action* exception..." is an example of how this font is used.
- `Typewriter` font (Courier) 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 □ 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) ← 0 □ 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 seven 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 III Cu processor unless otherwise restricted to a particular processor in the family.

Implementation Note – Implementation notes contain information that contains implementation specific information the UltraSPARC III Cu 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.
BBC	Bootbus controller (UltraSPARC III Cu processors).
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.
CDS	Crossbar Data Switch. Data bus crossbars for the Sun™ Fireplane interconnect of the UltraSPARC III Cu Processor. Also known as Dual CPU Data Switch (**DCDS**).
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.
D-TLB	Data Translation Lookaside Buffer.
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, data paths, 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**.
I-TLB	Instruction Translation Lookaside Buffer.
I-TSB	Instruction Translation Storage Buffer.
IU	Integer Unit.
L2-cache	Second level 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*.
LPA	Local Physical (or Processor) Address. Used in the context of Scalable Shared Memory (SSM) system architectures.
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**.
Microtag	A partial virtual address tag used for early way select of a virtually indexed, physically tagged set associative cache. Microtag is often referred to as utag or Utag in this documentation.

MMU	Memory Management Unit. The address translation hardware in the UltraSPARC III Cu implementation that translates 64-bit virtual address into physical addresses. The MMU is composed of the translation lookaside buffers (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	Non-coherent pending queue.
next program counter	*See* **nPC**.
NFO	Non-fault 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 an unrecoverable 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^{TM} 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.
RAW	Read-After-Write.
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.
SIU	System Interface Unit (Sun Fireplane interconnect).
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.
SSM	Scalable shared memory. A directory based data coherency mechanism.
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**

Processor Introduction

1.1 Overview

The UltraSPARC III Cu processor is a high-performance, highly integrated superscalar processor that implements the 64-bit SPARC V9 RISC architecture. It can sustain the execution of up to four instructions per cycle, even in the presence of conditional branches and cache misses, mainly because the units asynchronously feed instructions and data to the rest of the pipeline. Instructions that are predicted to be executed are issued in program order to multiple functional units, executed in parallel, and for added parallelism can be completed out-of-order. To further increase the number of instructions executed per cycle, instructions from two basic blocks can be issued in the same group.

The chip supports a 64-bit virtual address space and a 43-bit physical address space. The core instruction set has been extended to include graphics instructions that provide the most common operations related to two-dimensional image processing, two- and three-dimensional graphics and image compression algorithms, and parallel operations on pixel data with 8- and 16-bit components.

The processor is designed to offer very high clock speeds as well as wide superscalar issue to exploit instruction-level parallelism. The processor offers large Level-1 instruction and data caches, large flexible memory management units (MMUs), and support for large L2-cache. The processor was designed to work in systems ranging from single processor workstations through cache coherent servers with over a hundred processors. For building large systems, the processor has built-in support for both snooping-based cache coherency and directory-based cache coherency.

The architecture and implementation coupled with new compiler techniques make it possible to reduce each component while not degrading the other two.

1.2 CPU Features

The UltraSPARC III Cu processor is a richly featured processor. Features include:

- SPARC V9 Architecture with the VIS™ II Instruction Set, SPARC Binary code hardware compatible
- 4-way superscalar processor with nine execution units and six execution pipelines
- 14 stage, non-stalling pipeline
- Improved memory latency
- 64-bit data paths, 64-bit ALUs, 64-bit address arithmetic
- 64-bit virtual address and 43-bit physical address space
- Data prefetching mechanism
- L2-cache unit that supports a 2-way set associative cache
- Comprehensive error detection and recovery
- Data Memory Management Unit with 1040 Translation Lookaside Buffer (TLB) entries that can support up to 4 MB pages

1.3 Cache Features

The UltraSPARC III Cu processor cache features include:

- 32 KB, 4-way set associative primary instruction cache memory with parity
- 64 KB, 4-way set associative primary data cache memory with parity
- 2 KB, 4-way set associative Prefetch cache for software prefetch
- 2 KB, 4-way set associative Write cache reduces store bandwidth to Level 2 cache
- 8 MB, 2-way set associative external unified L2-cache with ECC protection (single bit correction, double bit detection)

1.4 Technology

The UltraSPARC III Cu processor technology features include:

- 0.18 µ, 7-layer Cu metal, CMOS process
- 1.6 V core and 1.5 V I/O power supplies

- 232 mm^2 die size
- 1368 pin ceramic LGA package
- 900 MHz and higher frequency
- 80 W power dissipation at 900 MHz

1.5 UltraSPARC III Cu Differences

The UltraSPARC III Cu processor differs from previous UltraSPARC processors in several key areas, including:

- Bootbus limitations
- Instruction set extensions
- Instruction differences
- Memory subsystem
- Interrupts
- Address space size
- Error correction
- Registers
- Non-cacheable store compression
- RAS Architecture

This section describes the UltraSPARC III Cu chip differences and includes a summary table of those differences. The section concludes with a discussion of the UltraSPARC III Cu processor performance enhancements and RAS architecture.

1.5.1 Bootbus Limitations

All bootbus addresses must be mapped as side-effect pages with the `TTE.E` bit set. In addition, programmers must not issue the following memory operations to any bootbus address:

- Prefetch instructions
- Block load and block store instructions
- Any memory operations with `ASI_PHYS_USE_EC` or `ASI_PHYS_USE_EC_LITTLE`
- Partial store instructions

1.5.2 Instruction Set Extensions

The UltraSPARC III Cu processor has added Sun proprietary extensions to the SPARC V9 Instruction Set Architecture (ISA), in addition to those implemented in UltraSPARC I. The extensions are in the areas of VIS extensions, prefetch enhancement, and interval arithmetic support.

1.5.2.1 Visual Instruction Set (VIS) Extensions

Three new VIS instructions were added:

- **Byte Mask** — Sets the Graphics Status Register (GSR) for a following byte shuffle operation. One byte mask can be issued per instruction group as the last instruction of the group.

  Byte Mask is a break-after instruction.
- **Byte Shuffle** — Allows any set of 8 bytes to be extracted from a pair of double-precision, floating-point registers and written to a destination double-precision, floating-point register. The 32-bit byte mask field of the GSR specifies the pattern of source bytes for the byte shuffle instruction.
- **Edge(ncc)** — Two variants: the original instruction sets the integer condition codes, and the new instruction does not set condition codes. Differences between the variants are as follows:

Edge	Edge(ncc)
Sets integer condition codes	Does not set integer condition codes
Single instruction group	Groupable

  Because of implementation restrictions in the pipeline, all instructions that set condition codes and execute in the MS pipeline stage must be in a single instruction group.

1.5.2.2 Prefetch Enhancement

The processor supports an instruction to invalidate a prefetched line. It invalidates a prefetch cache line after prefetched data has been loaded into registers and on error conditions.

1.5.2.3 Interval Arithmetic Support

One new instruction was added to improve the efficiency of interval arithmetic computations. The Set Interval Arithmetic Mode (SIAM) instruction enables the rounding mode bits in the Floating-Point Status Register (FSR) to be overridden without the overhead of modifying the RD field of the FSR. Updates directly to FSR are expensive because they flush the pipeline.

1.5.3 Instruction Differences

Several instructions have changed relative to the previous UltraSPARC processors.

- **`SHUTDOWN`** — Low power mode compliance is achieved through a different mechanism than that used by the UltraSPARC I processor. For compatibility, the `SHUTDOWN` instruction in the UltraSPARC III Cu processor executes as a NOP.
- **`FLUSH`** — Since the processor maintains consistency between the instruction cache and all store and atomic instructions, the `FLUSH` instruction is used only to clear the pipeline. Unlike the case with the UltraSPARC I processor, the `FLUSH` address is ignored. It is not used for instruction cache flushing and is not propagated to the system.
- **Floating-point conversion instructions** — Because of implementation restrictions, the following integer to floating-point conversion instructions generate an *unfinished_FPop* exception for certain ranges of integer operands, as shown in TABLE 1-1.

TABLE 1-1 Integer/Floating-Point *unfinished_FPop* Exception Conditions

Instruction	Unfinished Trap Ranges
`FsTOi`	result < -2^{31}, result ≥ 2^{31}, Inf, NaN
`FsTOx`	$\lvert$result$\rvert \geq 2^{52}$, Inf, NaN
`FdTOi`	result < -2^{31}, result ≥ 2^{31}, Inf, NaN
`FdTOx`	$\lvert$result$\rvert \geq 2^{52}$, Inf, NaN
`FdTOs`	$\lvert$result$\rvert \geq 2^{52}$, $\lvert$result$\rvert < 2^{-31}$, operand < -2^{22}, operand ≥ 2^{22}, NaN
`FiTOs`	operand < -2^{22}, operand ≥ 2^{22}
`FxTOs`	operand < -2^{22}, operand ≥ 2^{22}
`FxTOd`	operand < -2^{51}, operand ≥ 2^{51}

When the above instructions take an *unfinished_FPop* trap, system software must properly emulate the instruction and resume execution.

- **NaN handling** — Because of implementation restrictions, the processor generates an *unfinished_FPop* exception for operations that use the floating-point adder when one or more of the operands is NaN. Previous UltraSPARC processors would propagate the NaN in hardware.
- **Floating-point subnormal** — Because of implementation restrictions, the processor generates an *unfinished_FPop* exception in nonstandard mode for floating-point addition and floating-point subtraction operations when the result is a subnormal value. Previous UltraSPARC processors handled these in hardware. When an *unfinished_FPop* trap is generated, it is expected that system software will properly emulate the instruction and resume execution.
- **`Ticc` reserved field checking** — The processor checks the reserved field of the `Ticc` instruction for zero and generates an *illegal_instruction* trap if the field is nonzero. Neither UltraSPARC I nor UltraSPARC II processors checked the `Ticc` reserved field for zero.

1.5.4 Memory Subsystem

The memory subsystem design is new. Differences include changes in the caches, cache flushing, and TLBs.

1.5.4.1 Caches

The UltraSPARC III Cu memory system comprises five caches: four on-chip and one external to the chip.

- **Data cache (D-cache)** — A 64 KB, 4-way associative, virtually indexed, physically tagged (VIPT) cache. The D-cache is write-through, no write-allocate, not included in the L2-cache. The line size is 32 bytes with no sub-blocking. The D-cache needs to be flushed only if an alias is created with virtual address (`VA`) bit 13. `VA`<13> is the only virtual bit used to index the D-cache.
- **Instruction cache (I-cache)** — A 32 KB, 4-way associative, VIPT cache. The I-cache is not included in the L2-cache.

  The line size is 32 bytes - no sub-blocking. The I-cache is kept consistent with the store stream of the processor as well as with external stores from other processors.

  You never need to flush the I-cache, not even for address aliases.
- **Prefetch cache (P-cache)** — A 2 KB, 4-way associative cache. It is virtually indexed, virtually tagged (VIVT) cache for lookup and install to the cache. It is physically indexed and physically tagged for snoop and invalidate operations. The P-cache is not included in the L2-cache. The line size is 64 bytes with 32-byte sub-blocks.

  The P-cache is globally invalidated if any of the following conditions occur:

  - If the context registers are written
  - If there is a demap operation in the DMU
  - When the DMU 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 for autonomous hardware prefetches from the L2-cache.

  Software never needs to flush the P-cache, not even for address aliases.
- **Write cache (W-cache)** — A 2 KB, 4-way associative, PIPT cache. The line size is 64 bytes with 32-byte sub-blocks. The W-cache reduces bandwidth to the L2-cache by coalescing and bursting stores to the L2-cache.

  The W-cache is included in the L2-cache; all lines in the W-cache have a corresponding line allocated in the L2-cache. The data state of the W-cache line always supersedes the state of the data in the corresponding L2-cache line.

It is necessary to flush the W-cache for stable storage. Flushing the L2-cache implicitly forces the flush of the W-cache.

- **L2-cache** — A 1 MB to 8 MB, direct mapped or 2-way set associative, PIPT cache. The L2-cache is write-allocate, write-back.

  It is necessary to flush the L2-cache for stable storage.

1.5.4.2 Cache Flushing

The following are flushing requirements for specific caches:

- **Data cache** — The UltraSPARC III Cu D-cache differs in size and organization from the UltraSPARC I D-cache and so requires changes to the algorithms used to flush the cache.

  The virtually indexed caches need to be flushed when a virtual address alias is created. Caches that contain modified data need to be flushed for stable storage.

  The UltraSPARC III Cu D-cache is the only cache that needs to be flushed when a virtual address alias is created. Like the UltraSPARC I D-cache, the UltraSPARC III Cu D-cache uses one virtual address bit for indexing the cache and thus creates an alias boundary of 16 KB for the D-cache.
- **Instruction cache** — The processor maintains consistency of the on-chip I-cache with the stores from all processors so that a `FLUSH` instruction is needed only to ensure the pipeline is consistent. This means a single flush is sufficient at the end of a sequence of stores that updates the instruction stream to ensure correct operation.

  Unlike the case with the UltraSPARC I processor, the `FLUSH` instruction does not propagate externally since all I-caches in an UltraSPARC III Cu multiprocessor system are maintained consistent. Since the I-cache is a PIPT cache, it does not have to be flushed for virtual address aliases. The I-cache never contains modified data; therefore, it does not need to be flushed for stable storage.
- **Prefetch cache** — The P-cache is physically indexed and tagged. It cannot contain modified data, so it never needs to be flushed.
- **L2-caches and write caches** — Since the L2-cache and W-cache can contain modified data, they must be flushed for stable storage. The W-cache is included in the L2-cache, so it is sufficient to flush a block from the L2-cache; if there is a corresponding block in the W-cache, it will also be flushed. The recommended procedure to flush modified data from the L2-cache back to memory is as follows:
  - Load the block (64 bytes) into the floating-point registers by using FP loads or Block Load.
  - Write the floating-point registers to memory with a Block Store Commit.
  - Issue `MEMBAR #Sync` to ensure completion.

  The Block Store with Commit instruction will invalidate the block from both the L2-cache and the W-cache. Both of these caches are physically indexed, so they do not need to be flushed for address aliases.

1.5.4.3 Translation Lookaside Buffers (TLBs)

The implementation of instruction and data TLBs for the UltraSPARC III Cu processor is described in this section.

The following are two instruction TLBs that are accessed in parallel:

- A 16-entry, fully associative TLB to hold entries for 8 KB, 64 KB, 512 KB, and 4 MB page sizes. This TLB contains locked and unlocked pages of any size.
- A 128 entry, 2-way associative TLB used exclusively for 8 KB page entries. This TLB contains only unlocked pages.

The following are three data TLBs that are accessed in parallel.

- A 16-entry, fully associative TLB to hold entries for 8 KB, 64 KB, 512 KB, and 4 MB page sizes. This TLB contains locked and unlocked pages of any size.
- Two 512 entry, 2-way set associative that can each be programmed to support lookup of any one page size at a given time. However, multiple page sizes can be resident. This TLB contains only unlocked pages.

Other TLB differences are described below:

- **TLB flushing** — Both the instruction and data TLBs now have a demap-all operation that removes all unlocked Translation Table Entries (TTEs).
- **TTE format** — The UltraSPARC III Cu processor now has the additional elements in the TTE format:
  - Physical Address field: Expanded from 28 bits (PA<40:13>, TTE<40:13>) to 30 bits (PA<42:13>, TTE<42:13>).
- **Synchronous Fault Status Registers (SFSR) extensions** — A new fault type was added to the FT field of the SFSR to indicate an I/D-TLB miss, and one status bit was added to the D-TLB SFSR:
  - NF: Set to signify that the faulting operation was a speculative load instruction.
- **Instruction/Data Translation Storage Buffer (i/dTSB) Register** — Three new register extensions of the i/dTSB register were added to the UltraSPARC III Cu 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, the processor selects which TSB Extension Register to use to form the TSB base address, based on the virtual space accessed by the faulting instruction.
- **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.
- **TLB Diagnostic Register** — A new register replaces the function of the diagnostic bits in the TTE.

1.5.5 Interrupts

The UltraSPARC III Cu processor extends the interrupt architecture previously implemented in the UltraSPARC I processor in these areas:

- **Module ID fields** — Extended from 5 bits to 10 bits.
- **Interrupt Transmit** BUSY/NACK **bits** — Extended from 1 pair to 32 pairs, enabling pipelining of outgoing interrupts.
- **Data Dispatch and Receive Registers** — Expanded from 3 to 8, enabling up to 64 bytes to be transmitted in an interrupt.
- **System Tick Interrupt bit** — Added to the soft interrupt register.

1.5.6 Address Space Size

The UltraSPARC III Cu processor extends both the virtual and physical address space previously implemented. It implements the full 64-bit virtual address range defined in the SPARC V9 architecture. There are no VA holes, compared to the UltraSPARC I. The physical address range has also been extended from 41 bits to 43 bits.

Address space with PA<42> = 1 is considered as the non-cacheable address space. Physical address $400\,0000\,0000_{16}$ to $7FF\,FFFF\,FFFF_{16}$ is in the non-cacheable area.

1.5.7 Registers

Differences in registers include enhancements to ASI registers and ASR registers.

Address Space Identifier (ASI) Registers

Changes to the ASI registers include those to the following registers:

- **SRAM diagnostic registers** — Several new diagnostic ASI registers were added for the following on-chip SRAMs:
  - Prefetch cache
  - Write cache
  - Branch predict array
  - I/D-TLB CAM

  Changes were made to fields of existing UltraSPARC II diagnostic ASI registers for the following on-chip SRAMs:
  - Data cache
  - L2-cache
  - Instruction cache

The following ASI registers were removed:

- UDB Error Register
- UDB Control Register

- **Asyncronous Fault Status Register (AFSR)** — Several changes were made to add new fault types (L2-cache ECC errors) and remove old fault types (SDB errors). AFSR accumulates errors.
- **Asynchronous Fault Address Register (AFAR)** — The AFAR was extended to handle a 43-bit physical address. It is now updated on several errors that previously did not capture the address. AFAR overwrites for higher priority errors if a more severe error occurs.
- **Secondary Asyncronous Fault Status Register (AFSR)** — Secondary AFSR captures the first error.
- **Asynchronous Fault Address Register (AFAR)** — Secondary AFAR captures the address associated with the first error and it locks until it is explicitly cleared by software.
- **Software Interrupt Register (SOFTINT)** — The SOFTINT register has an additional bit added to signal SYSTEM TICK COMPARE interrupts.
- **System Interface Registers** — The UPA interface ASI has been reused for two new Fireplane Interconnect registers: a configuration register and an address register.

Ancillary State Registers (ASRs)

Changes to the ASRs include changes to the following registers:

- **System Tick and Compare** — Two new ASRs were added to support a system clock: ASR 18_{16}, a System Tick Register (analogous to the per-processor tick register ASR 4_{16}), and ASR 19_{16}, a System Tick Compare Register (analogous to the per-processor tick compare register 17_{16}).
- **Graphics Status Register (GSR)** — New fields were added to the GSR:
  - 32-bit MASK field used by the BSHUFFLE instruction
  - 1-bit IM field to enable interval arithmetic round mode
  - 2-bit IRD field to specify round mode for interval arithmetic
- **Performance Control Register (PCR)** — The PCR has been extended to enable a larger number of performance events to be measured.
- **Dispatch Control Register (DCR)** — Many control fields were added to the DCR to aid in debugging first silicon.

1.5.8 Non-Cacheable Store Compression

Like previous implementations, the UltraSPARC III Cu processor uses a 16-byte buffer to merge adjacent non-cacheable stores into a single external data transaction. This merging greatly increases store bandwidth to the graphics frame buffer. A change in the algorithm for determining when to break merging improves store bandwidth to graphics devices.

1.5.9 Error Correction

Error correction differs from the UltraSPARC I processor and the UltraSPARC II processor handling, as follows:

- **L2-cache** — The processor uses ECC protection on the L2-cache instead of parity protection. It requires software correction and recovery for single bit L2-cache ECC read errors, which are signaled as a precise error.
- **System interface** — A new ECC code has been defined for ECC protection across 132 data bits (nine ECC bits) and three MTag bits (four ECC bits) on the system bus and on the data switch. The syndromes for these codes differ from the syndromes used previously. The processor requires software correction and recovery for single-bit system ECC errors, which are signaled as disrupting errors.

1.5.10 SRAM Protection and RAS Features

TABLE 1-2 lists all UltraSPARC III Cu on-chip SRAM protection and other RAS features.

TABLE 1-2 RAS Features

No.	Feature	Feature Description
1.	D-cache Data array parity protection	2 parity bits per 64-bit data
2.	D-cache Physical Tag array parity protection	1-bit parity per tag entry
3.	D-cache Snoop Tag array parity protection	1-bit parity per tag entry
4.	I-cache Data array parity protection	1-bit parity per partially decoded instruction
5.	I-cache Physical Tag array parity protection	1-bit parity per tag entry
6.	I-cache Snoop Tag array parity protection	1-bit parity per tag entry
7.	L2-cache data array ECC protection	ECC Protection with 1-bit correction and 2-bit detection
8.	L2-cache Tag array ECC protection	ECC Protection with 1-bit correction and 2-bit detection
9.	Dual AFSR/AFAR	Secondary AFSR/AFAR captures on the first error event while primary AFSR/AFAR accumulates in case of multiple events

CHAPTER 2

System Introduction

This chapter discusses how the UltraSPARC III Cu processor is used in systems.

2.1 System Configurations

The UltraSPARC III Cu processor can be used in a variety of configurations from two-processor systems to very large, high-performance Symmetric Multiprocessor (SMP) Systems.

2.1.1 Two-Processor Configuration with UltraSPARC III Cu

FIGURE 2-1 is an example of a two-processor UltraSPARC III Cu configuration. This configuration is the basic building block that can be used to build SMP systems.

In this configuration, both UltraSPARC III Cu processors are connected to a Dual CPU Data Switch (DCDS) via a 16-byte interface at 150 MHz. Each processor has an address and control interface to the SDRAMs. However, the data from the SDRAM is connected directly to the DCDS via a 64-byte interface running at 75 MHz. Note that the ranges for the SDRAM on different UltraSPARC III Cu CPUs should not overlap. Each UltraSPARC III Cu CPU connects to the Fireplane Interconnect Address bus. The UltraSPARC III Cu CPU also connects with an L2-cache via a 32-byte interface and with the Boot PROM via the Boot Bus interface. Both UltraSPARC III Cu CPUs provide a JTAG interface.

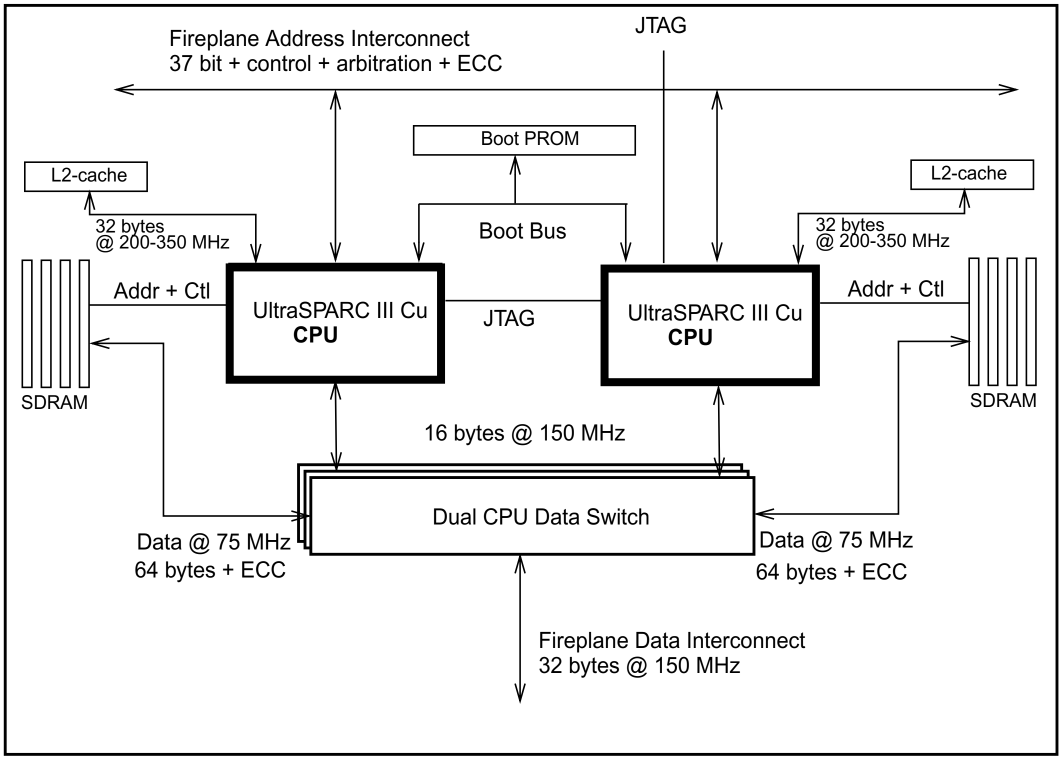


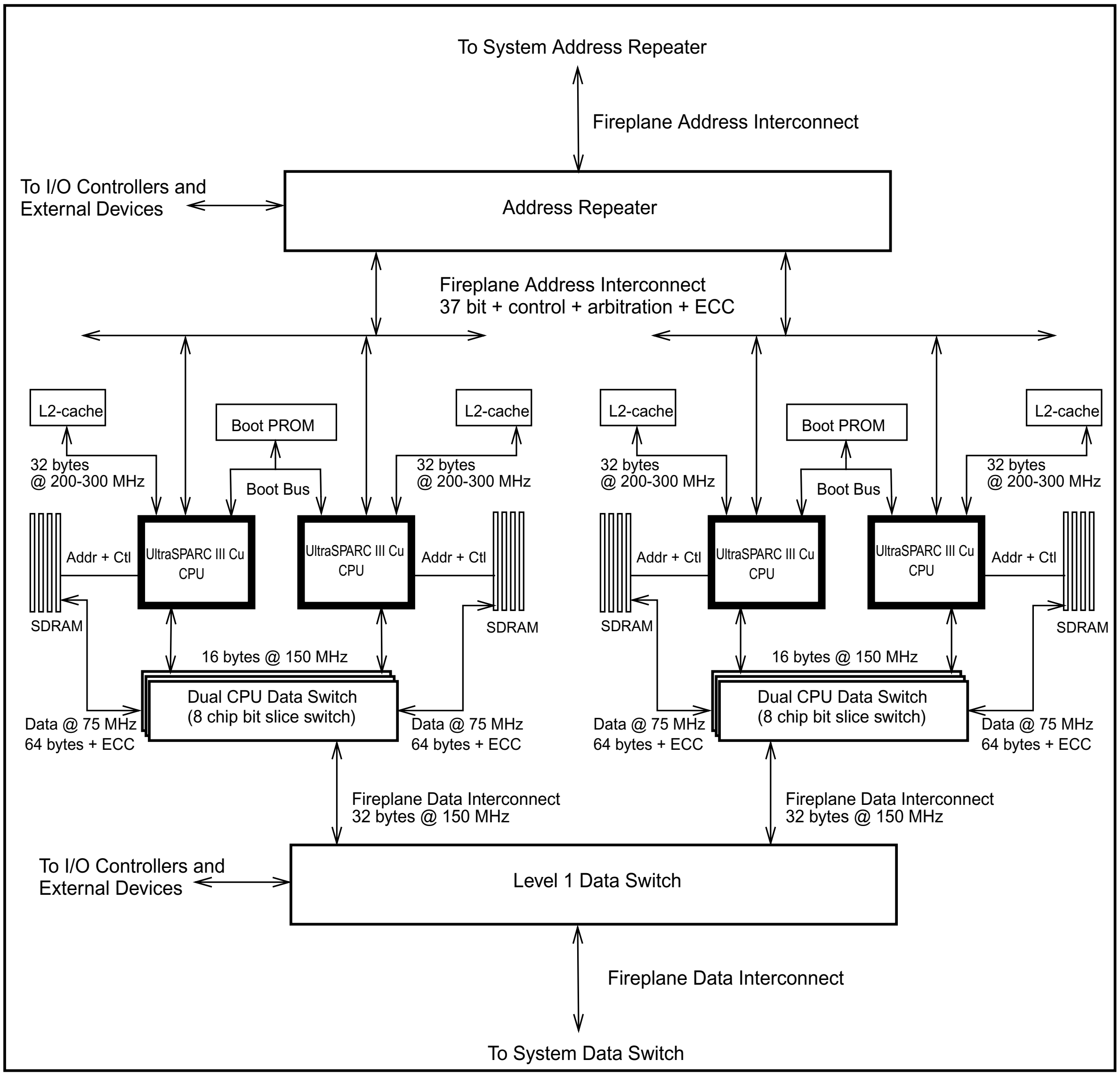
FIGURE 2-1 Two-Processor Configuration with the UltraSPARC III Cu

---

Note – Some processors or memories may not be present in the actual system.

2.1.2 Four-Processor Configuration with UltraSPARC III Cu

A four-processor configuration, shown in , is built from a two-processor configuration. This example also demonstrates how larger systems can be created by using an address repeater and a Level 1 data switch. I/O controllers, bridge chips, or external devices are connected to the system via the address repeater and the data switch. This configuration can be repeated to create a high performance SMP system.


To System Address Repeater
Fireplane Address Interconnect
To I/O Controllers and External Devices
Address Repeater
Fireplane Address Interconnect
37 bit + control + arbitration + ECC
L2-cache
Boot PROM
L2-cache
L2-cache
Boot PROM
L2-cache
32 bytes @ 200-300 MHz
Boot Bus
32 bytes @ 200-300 MHz
32 bytes @ 200-300 MHz
Boot Bus
32 bytes @ 200-300 MHz
Addr + Ctl
UltraSPARC III Cu CPU
UltraSPARC III Cu CPU
Addr + Ctl
Addr + Ctl
UltraSPARC III Cu CPU
UltraSPARC III Cu CPU
Addr + Ctl
SDRAM
SDRAM
SDRAM
SDRAM
16 bytes @ 150 MHz
16 bytes @ 150 MHz
Dual CPU Data Switch (8 chip bit slice switch)
Dual CPU Data Switch (8 chip bit slice switch)
Data @ 75 MHz 64 bytes + ECC
Data @ 75 MHz 64 bytes + ECC
Data @ 75 MHz 64 bytes + ECC
Data @ 75 MHz 64 bytes + ECC
Fireplane Data Interconnect 32 bytes @ 150 MHz
Fireplane Data Interconnect 32 bytes @ 150 MHz
To I/O Controllers and External Devices
Level 1 Data Switch
Fireplane Data Interconnect
To System Data Switch

Four-Processor Configuration with the UltraSPARC III Cu

2.1.3 Multiprocessor System with the UltraSPARC III Cu

A multiprocessor system, as shown in FIGURE 2-2, can be built using the two-processor configuration shown in FIGURE 2-1. A system address and data repeater are used to build such systems. Various devices can be connected to the system via PCI interfaces. Level 1 and Level 2 address repeaters and data switches are used to build systems that can accommodate up to six processor/memory boards and up to four I/O subsystems.

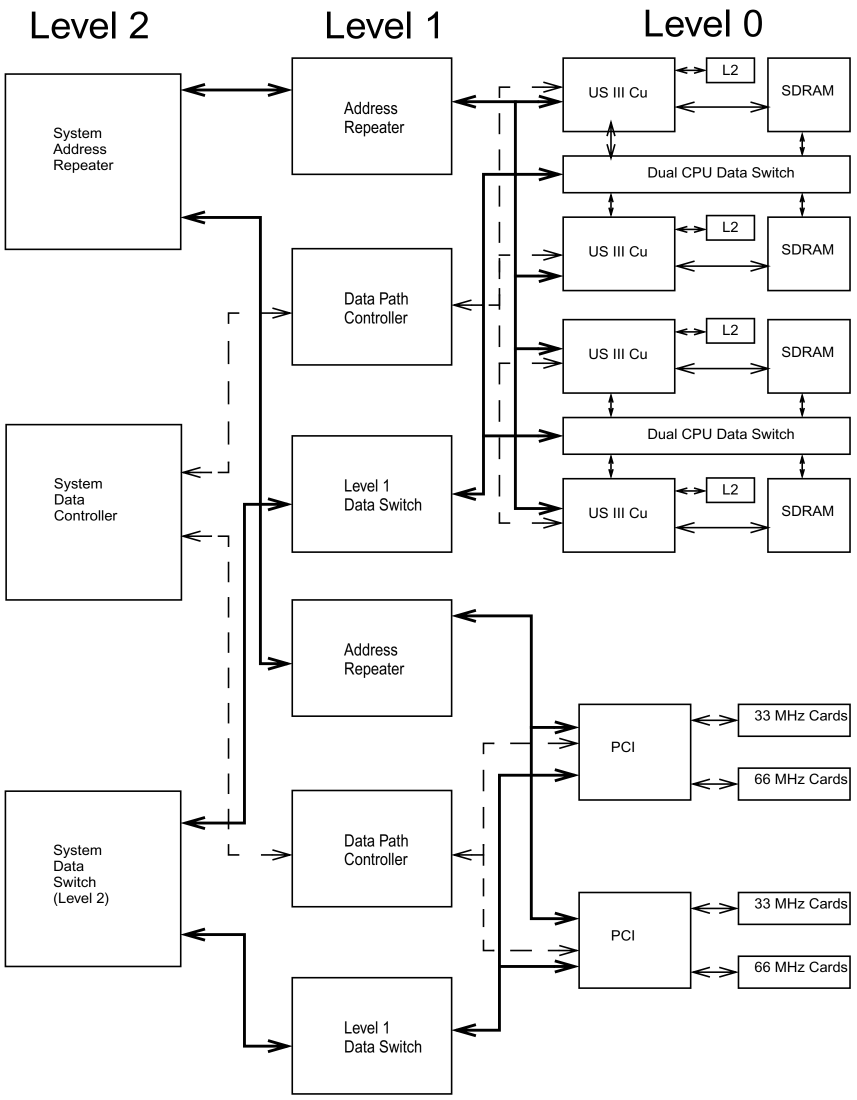


FIGURE 2-2 Multiprocessor System with the UltraSPARC III Cu

2.1.4 Very Large Multiprocessor System with the UltraSPARC III Cu

A very large multiprocessor system, as shown in FIGURE 2-3, can be built by interconnecting multiple configurations, shown in FIGURE 2-2, using Level 3, 18-port crossbar switches.

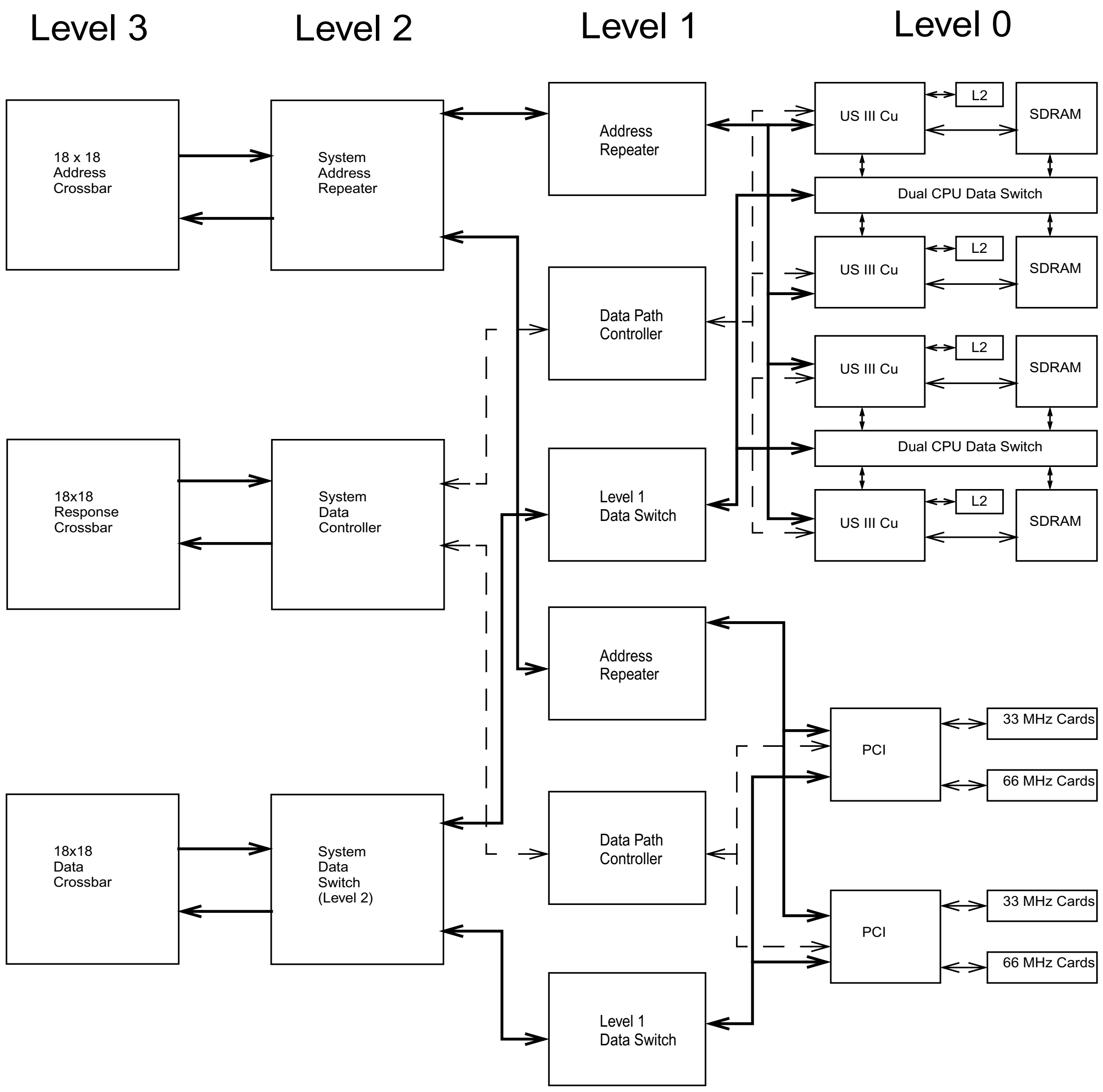


FIGURE 2-3 Very Large Multiprocessor System with the UltraSPARC III Cu

2.2 Cache Coherence

UltraSPARC III Cu-based systems support a “snooping” based cache coherence protocol and a directory based cache coherence protocol (also known as Scalable Shared Memory (SSM)).

For small to medium systems, the UltraSPARC III Cu processor uses a MOESI snooping protocol. In this protocol, when a processor wants a line, it broadcasts the request to all other processors, which check their caches to see if they have the line.

For larger systems, the UltraSPARC III Cu processor has built-in support for a directory based coherence protocol (SSM).

In practice, there are small clusters of processors that are connected together with a snooping based coherence protocol. A directory based cache coherence protocol is used between clusters.

2.3 System Interfaces

There are multiple interfaces on the UltraSPARC III Cu processor. This section summarizes the various interfaces.

2.3.1 Fireplane Interconnect

The Fireplane Interconnect has two parts. Both typically run in the range of 150 MHz.

- A hierarchical bus for address and control.
- A point-to-point data interconnect.

The Fireplane Interconnect Address Bus has 37 bits of address plus control, arbitration and ECC bits. All UltraSPARC III Cu processors connect to this bus directly. To build larger multiprocessor systems, a repeater chip is used to create a hierarchical bus.

The Fireplane Interconnect Data Bus has 32 bytes of data along with ECC and routing information. All UltraSPARC III Cu processors connect to the Fireplane Interconnect Data Bus through a DCDS switch. To build larger multiprocessor systems, a point-to-point data network built with Level 1 data switches can be used.

2.3.2 SDRAM Interface

The UltraSPARC III Cu processor provides address and control bits to the SDRAM. The data from the SDRAM is directly driven to the DCDS via a 64-byte interface running at one-half of the Fireplane frequency.

2.3.3 DCDS Interface

The UltraSPARC III Cu processor connects to the Dual CPU Data Switch (DCDS) via a 16-byte interface running at 150 MHz and capable of delivering a peak bandwidth of 2.4 GB/s. DCDS is an eight chip, bit slice switch that converts it into a 32-byte Fireplane Interconnect Data Port running at 150 MHz and capable of delivering a peak bandwidth of 4.8 GB/s.

2.3.4 L2-Cache Interface

The UltraSPARC III Cu processor connects to an up to 8 MB, 2-way set associative external unified Level 2 (L2) cache with ECC protection via a 32-byte interface, running at 200 MHz or higher.

2.3.5 Boot Bus Interface

The UltraSPARC III Cu processor has a Boot Bus interface that connects to a Boot PROM, other booting mechanisms, and diagnostic and recovery mechanisms.

2.3.6 JTAG Interface

The UltraSPARC III Cu processor provides a standard 1149.1 compliant JTAG interface. This interface can be used to scan out the internal state of the processor for fault diagnosis. The full scan is a destructive operation that requires the CPU to go through a Power-on reset (POR) before being used again. In addition, the UltraSPARC III Cu processor also provides a shadow JTAG interface that allows a subset of the state to be scanned while the processor is running.

SECTION II

Architecture and Functions

CHAPTER **3**

CPU Architecture Basics

The UltraSPARC III Cu processor is a high-performance, highly integrated, superscalar processor. The UltraSPARC III Cu fully implements the 64-bit SPARC V9 architecture, supporting a 64-bit virtual address space and a 43-bit physical address space. The core instruction set is extended to include new SIMD operations. The processor was designed to offer very high clock speeds as well as wide superscalar issue to exploit instruction-level parallelism. The processor offers large Level-1 instruction and data caches, large flexible memory management units (MMUs), and support for a large L2-cache. The processor was designed to work in systems ranging from single processor workstations through cache-coherent servers with more than a thousand processors. For building a wide range of system configurations, the processors has built-in support for both snooping-based cache coherency and directory-based cache coherency.

The UltraSPARC III Cu processor also offers a number of performance enhancements over previous UltraSPARC processors. The processor incorporates a number of data prefetching mechanisms to exploit memory-level parallelism. The processor offers an enhanced data memory management unit (D-MMU) that has 1040 TLB entries and more support for flexibly using large pages, up to 4 MB pages, to more effectively map gigabytes of data. The processor supports a 2-way set associative L2-cache instead of a direct-mapped cache.

3.1 Component Overview

The processor consists of a high-performance, instruction fetch engine, called the instruction issue unit, that is decoupled from the rest of the pipeline by an instruction buffer. Instructions are steered to either floating-point execution units, integer execution units, or a load/store unit for execution. Integrated on the processor are controls for the L2-cache, interface to the Fireplane bus, and a memory controller. A simple block diagram of the UltraSPARC III Cu processor is shown in FIGURE 3-1.

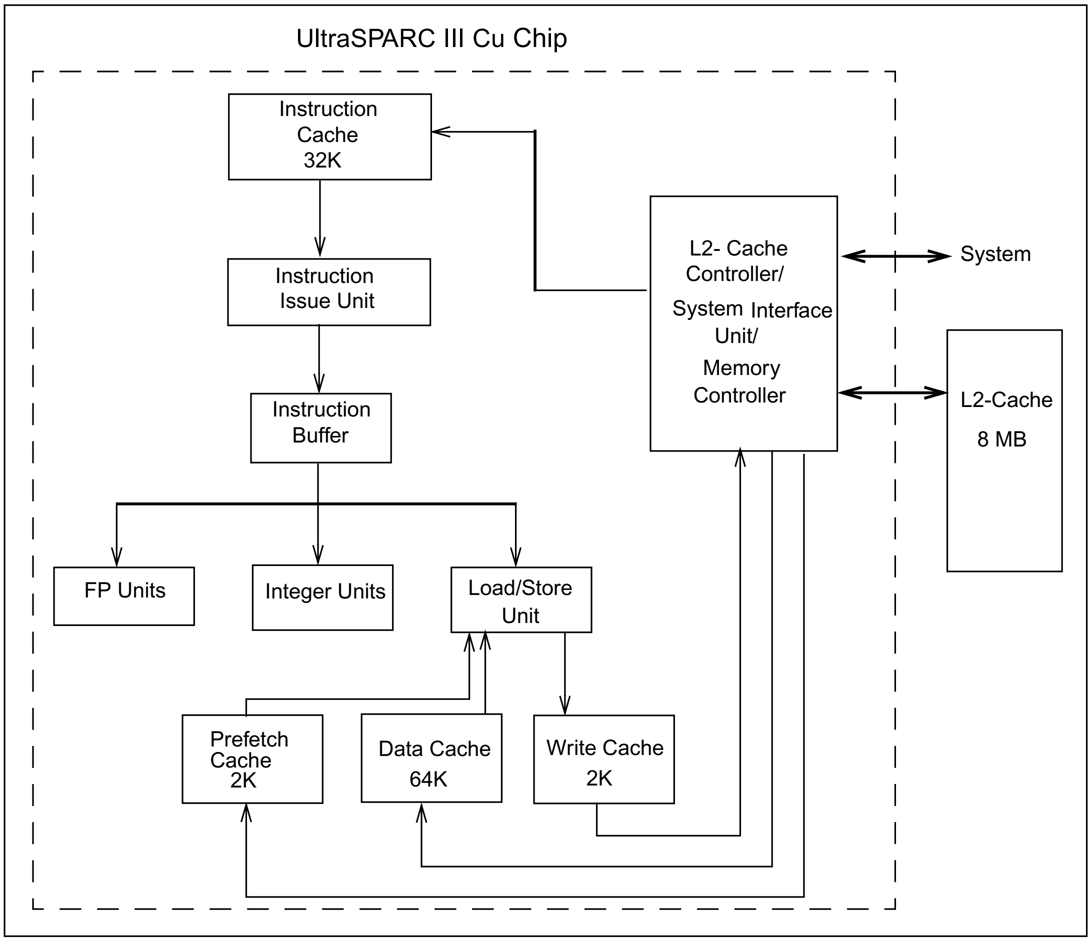


FIGURE 3-1 UltraSPARC III Cu Architecture

3.1.1 Instruction Fetch and Buffering

The instruction issue unit in the UltraSPARC III Cu processor is responsible for fetching, caching, and supplying groups of instructions to its execution pipelines. Instructions are fetched and decoded in groups of up to four instructions. Instruction fetching can be done in every cycle if the request hits instruction cache and other conditions are met. If the request misses instruction cache, a fill request is sent to the lower memory hierarchy and the requested 32-byte line is installed in instruction cache. If the requested line is the first half of a 64-byte boundary, the second half of the 64-byte boundary is prefetched and the line is filled in the Instruction Prefetch Buffer (IPB) for a potential reference in the future.

The UltraSPARC III Cu instruction cache is a 32 KB size, 32-byte line size (eight instructions), physically tagged, 4-way set associative cache. Its throughput is one cycle with two-cycle latency. In addition to data and tag array, it also has a microtag, predecode, Load Prediction Bit (LPB), and snoop tag array. The microtag uses eight bits of virtual address to enable way-select to be performed before the physical address translation is completed. The predecode bits include information about which pipeline each instruction will issue 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 III Cu 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 III Cu processor uses a 16K entry branch predictor to predict fetch direction of conditional branches. The target must be determined for branches that are either predicted or known to redirect instruction fetches. For PC relative branches, the target of the branch is computed; this adds a one-cycle branch taken penalty, but avoids 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.

Between the instruction fetch pipeline and the execution pipeline is an instruction buffer that can hold up to 16 instructions. The instruction buffer decouples the fetch and execute pipelines and buffers burstiness in each pipeline from each other. The buffer can effectively hide low latency issues like the taken branch penalty and even hides some of the penalty of instruction cache misses.

3.1.2 Execution Pipelines

The UltraSPARC III Cu processor has six execution pipelines and can issue up to four instructions per cycle. The six execution pipelines consist of the following:

- Two integer arithmetic and logic (ALU) pipelines
- Branch pipeline
- Load/store pipeline which also handles special instructions
- Floating-point multiply pipeline which also handles SIMD instructions
- Floating-point addition pipeline which also handles SIMD instructions

The integer ALU pipelines can issue integer addition, subtraction, logic operations, and shifts. These pipelines have single-cycle latency and throughput. The branch pipeline handles all branch instructions and can resolve one branch each cycle. The load/store pipeline can handle one load or store instruction each cycle and is discussed in more detail in

Section 3.1.3. Integer multiplication and division is performed by the load/store pipeline. 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 pipelines are each four-cycle latency pipelines but are fully pipelined (one instruction per cycle per pipeline). These pipelines handle single and double precision floating-point operations and a set of data parallel operations 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 or 20 cycles for single and double precision, respectively. Floating-point square root requires 23 or 29 cycles for single and double precision, respectively.

3.1.3 Load/Store Unit

As stated earlier, 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.

Loads have either a two- or three-cycle latency. Integer loads that are for unsigned words or double words have a two-cycle latency. All other load instructions have a three-cycle latency. Data can be forwarded from earlier stores still in the store queue to subsequent loads if a RAW hazard is detected. Data forwarding requires a three-cycle latency. For those instructions that can have a two-cycle latency, there is a prediction bit in the instruction cache used to identify those loads that often require store forwarding, which will be issued as three-cycle loads. If a two-cycle load is not correctly predicted to have a RAW hazard, the load must be reissued.

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 can coalesce stores to the same cache line. The store queue allows non-catchable stores (for example, stores to a graphics frame buffer) to be coalesced together such that the required bandwidth to the device is greatly reduced.

The data cache is a 64 KB, 4-way associative, two-cycle latency, one-cycle throughput, virtually indexed, physically tagged (VIPT) cache. The data cache, like the instruction cache, uses 8-bit microtags to do way-selection based on virtual addresses. The data cache is write-through, no write-allocate, and not included in the L2-cache. The line size is 32 bytes with no sub-blocking. The data cache needs to be flushed only 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 SPARC V9 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 per byte dirty bits.

3.1.3.1 Data Prefetching Support

The UltraSPARC III Cu processor makes use of an advanced data prefetching mechanism. This mechanism is used to both overlap load misses to increase memory-level parallelism and to hide load-miss latency. This mechanism allows software to explicitly expose the memory-level parallelism and to schedule memory operations. This mechanism is extremely important because the UltraSPARC III Cu processor has blocking loads; when the processor reaches a load instruction that misses in the cache, the processor waits for the load to complete before executing any other instructions. The processor supports software prefetching where the compiler (or Java^{TM} JIT) can schedule prefetching of data to exploit memory-level parallelism. Some versions of the processor will also support hardware prefetching, where the processor observes common data sequences and attempts to prefetch the data automatically.

There are a number of variations of software prefetches. Software prefetches can specify if the data should be brought into the processor either for reading or 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.

One of the main mechanisms for implementing prefetches is 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. 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 sub-blocks 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. The MMUs consist of a set of translation lookaside buffers (TLBs) that are tables of translations from virtual to physical addresses. As long as a virtual address can be translated using one of the entries in a TLB, the operation proceeds without interruption. If there is no translation available for a virtual address, the processor traps to software to update the TLBs with a valid translation.

For the instruction address stream translation, there are two TLBs accessed in parallel. The first TLB is a 16-entry, fully associative TLB. This TLB can translate page sizes of 8K, 64K, 512K, and 4M, and locked page always reside in this TLB. The second TLB is a 64 set, 2-way set associative (128 entries) TLB. This large TLB is used to translate a page size of 8K.

The D-MMU of the UltraSPARC III Cu processor is enhanced to provide more translation entries and more support for using large pages for translation. For the data reference address stream translation there are three TLBs accessed in parallel. The first TLB is a 16-entry fully-associative TLB. This TLB can translate page sizes of 8K, 64K, 512K, and 4M. The second TLB is a 256-set, 2-way set-associative (512 entries) TLB. This TLB can translate at 8K, 64K, 512K, and 4M page sizes, but at any one time it is configured to only handle one of the page sizes. The third TLB is identical to the second. This TLB, like the second, can handle one of four page sizes and can be configured to the same or a different page size than the second TLB.

The two large TLBs is very important for general use of large pages for translation. 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 L2-cache Unit (Level-2 Unified Cache)

The UltraSPARC III Cu processor can support an L2-cache of 1 MB, 4 MB or 8 MB. The L2-cache is 2-way set associative, PIPT cache. The line size of the L2-cache depends on the cache size (64 bytes for 1 MB, up to 512 bytes for 8 MB). Regardless of the line size, the cache uses 64-byte sub-blocks that are the unit of fill and the unit of coherency. The L2-cache is write-allocate, write-back. The tags for the L2-cache are on the CPU chip.

For data, the L2-cache uses standard SRAM parts running at either one-third, one-fourth, or one-fifth of the processor speed. The interface between the processor and the SRAM is 32 bytes wide. The L2-cache for the UltraSPARC III Cu processor is fully protected with error correcting code (ECC). Single bit errors in the L2-cache are corrected and double bit errors are detected. These result in UltraSPARC III Cu state-of-the-art reliability.

3.1.6 System Interface Unit

The system interface unit (SIU) is the UltraSPARC III Cu processor's port to the external world. All data is transferred between the UltraSPARC III Cu processor and local DRAM, main memory associated with another CPU, or the system bus passes through the SIU. The SIU is the engine for the cache coherency protocol for multiprocessor systems.

The SIU supports clock divisors of 4, 5, and 6 between the system clock and the internal CPU clock. When the system reset becomes inactive, both the internal CPU clock and the system interface clock are synchronized at the rising edge.

The system interface allows for a low-cost interconnect of up to six agents. An agent may be another UltraSPARC III Cu processor, an I/O controller, bus repeater or an SSM controller. The bandwidth of the external interface buses allows the system interface to be implemented using a snooping coherence protocol. A snooping interface allows each agent to maintain coherency without the need for an external coherency controller. An UltraSPARC III Cu processor snoops coherent transaction requests issued on the system interface buses and follows a write-invalidate MOESI cache coherence policy. Snooping-based systems can be built with tens of processors.

SSM is a directory-based protocol used for creating very large multiprocessor systems. Smaller groups of processors using a snooping interface can utilize SSM controllers to create systems with over a hundred processors. The SIU of the UltraSPARC III Cu processor has built-in support for working with an SSM controller to facilitate the creation of large systems.

3.1.7 Memory Controller Unit

The UltraSPARC III Cu processor has an on-chip Memory Controller Unit (MCU). The UltraSPARC III Cu memory system supports a minimum of 128 MB and a maximum of 16 GB of main memory. The MCU supports 75 MHz SDRAM and interfaces to various densities of DRAM and single or multiple banks of memory. The MCU only sends out control signals to the DRAM. The UltraSPARC III Cu SIU is responsible for delivering data to the data switch for write operations and retrieving data from the data switch for read operations.

3.2 CPU Operating Modes

The UltraSPARC III Cu 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 ASIs. There are certain features that can be accessed only in privileged mode. Non-privileged software is not allowed to access these features.

Privileged mode execution is typically used by the kernel and operating system.

3.2.2 Non-Privileged Mode

This mode is a "**non-supervisor**" mode. In this mode, the software is 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 III Cu processor can be reset using various mechanisms. This section deals with the reset and RED_state for the UltraSPARC III Cu processor.

3.2.3.1 RED_state Characteristics

A processor enters RED_state by one of the two ways.

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

When the processor enters the 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 POK pin is activated and stays asserted until the processor is within its specified operating range. When the POK pin is active, all other resets and traps are ignored. POR has a trap type of 1 at physical address offset 20_{16}. 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.

The MCU refresh control register as well as the Fireplane configuration register must be initialized after a POR.

In SSM systems, the MTags contained in memory must be initialized before any Fireplane transactions are generated.

Caution – Executing a `DONE` or `RETRY` instruction when `TSTATE` is not initialized 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 Reset pin is activated. When the Reset pin is active, all other resets and traps are ignored. System reset has a trap type of 1 at physical address offset 20_{16}. Any pending external transactions are canceled.

Note – Memory refresh continues uninterrupted during a system reset. 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 an external hardware pin. It causes a SPARC V9 XIR, which has a trap type 3_{16} at physical address offset 60_{16}. 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 80_{16}. SIR affects only one processor rather than the entire system.

RED_state Trap Vector

When the UltraSPARC III Cu 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 III Cu processor provides extensive support for detecting and correcting errors. Note that some errors may still be uncorrectable.

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 shutdown 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 signal to its interconnect, expecting to be halted/reset by the system. System actions induced by the ERROR signal are system implementation dependent.

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 signalled either through a disrupting trap to the processor or through an ERROR signal to system hardware to induce 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 with an ERROR are meant either to be loss of system consistency or a protocol error on system interconnect.

On the other hand, 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.

Corrective Actions

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

- **Reset-inducing ERROR signal.** The most severe fatal error generates an ERROR signal to induce a system reset. Both, an error detected in the course of instruction execution and an error asynchronous to instruction execution may generate an ERROR signal.
- **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.

3.2.5 Debug and Diagnostics Mode

The UltraSPARC III Cu 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 a couple of 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.

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 to access the register (all internal registers are 8 bytes and must be accessed as 8-byte units with 8-byte aligned 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, large memories on chip, such as caches, can have each element of the memory array be directly read and written. These 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 (all internal state must be accessed in 8-byte units with 8-byte aligned addresses). Most structures can be directly read and many structures can also be directly written or quickly cleared.

The UltraSPARC III Cu 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 can also 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 a full scan 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 to access the register (all internal registers are 8 bytes and must be accessed as 8-byte units with 8-byte aligned 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. See Chapter 12, “Traps and Trap Handling,” for a detailed description of exception and trap processing.

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 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

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

Pipeline Stage	Definition
B	**B**ranch target computation
I	**I**nstruction group formation
J	**J**: grouping
R	**R**egister access (dispatch/dependency checking stage)
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, "CPU 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 illustrates each pipeline stage in detail and the relationship between high level, large architectural structures.

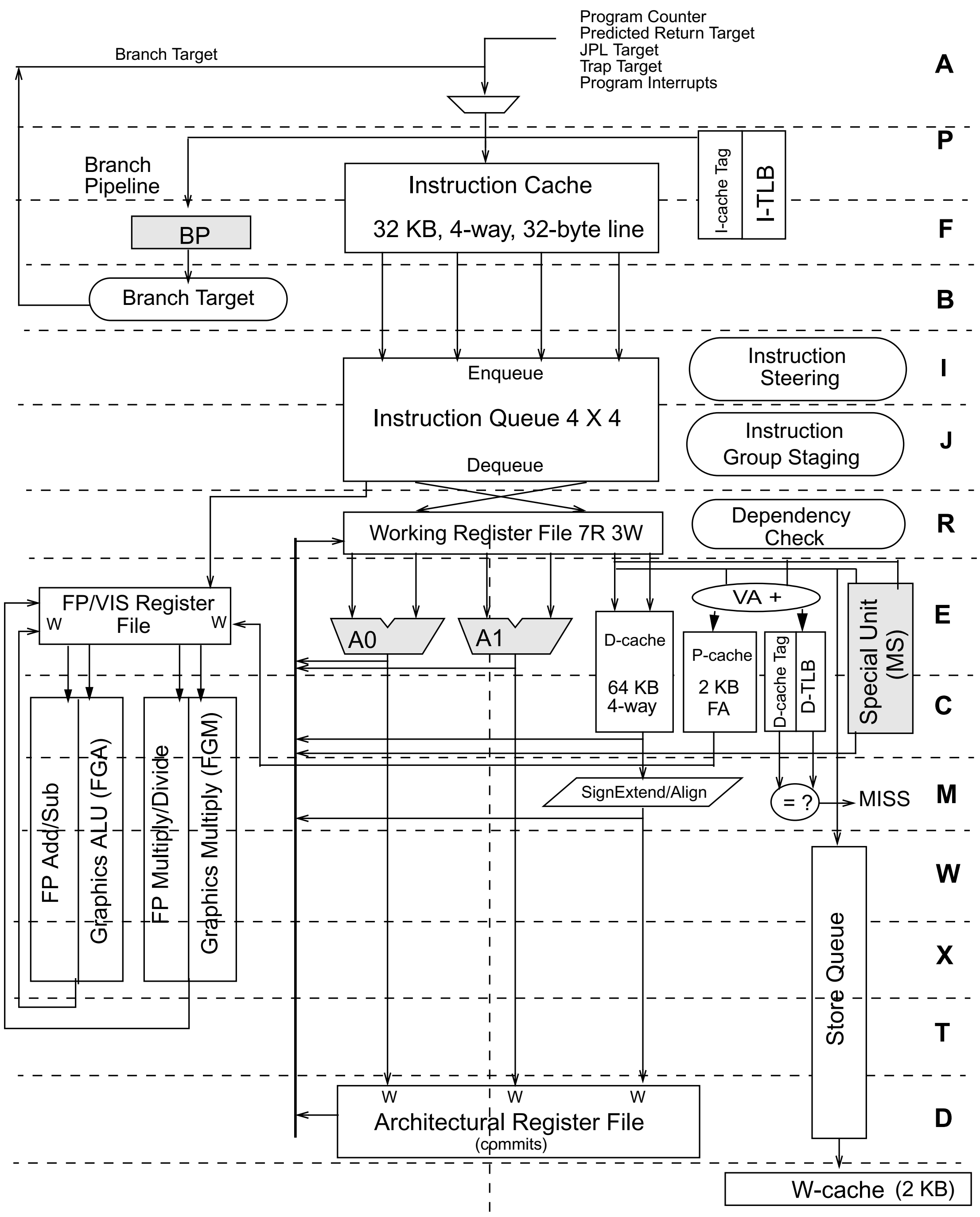


FIGURE 4-1 Instruction Pipeline Diagram Instruction Dependencies

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

4.2.1 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 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” for details.

4.2.2 Dispatch Dependencies

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

- Working register operand not available
- Functional Unit not available
- Store-load sequence 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.

4.2.3 Execution Dependencies

The pipeline assumes all load instructions will hit in a primary cache, allowing the pipeline to operate at full speed. A cache miss 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.

4.2.4 Instruction-Fetch Stages

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

4.2.4.1 A-stage (Address Generation)

The address stage generates and selects the fetch address to be used by the instruction 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

4.2.4.2 P-stage (Preliminary Fetch)

The preliminary fetch stage starts fetching four instructions from the instruction 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.

4.2.4.3 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).

4.2.4.4 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 instruction 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.

4.2.5 Instruction Issue and Queue Stages

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

4.2.5.1 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

4.2.5.2 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.

4.2.5.3 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.

4.2.6 Execution Pipeline

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

4.2.6.1 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. The 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 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 lookup 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.

4.2.6.2 C-stage (Cache)

The data 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.

4.2.6.3 M-stage (Miss)

Data 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 are available to the floating-point pipelines in the M-stage.

4.2.6.4 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.

4.2.6.5 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.

4.2.7 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.

4.2.7.1 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.

4.2.7.2 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 functional unit 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 re-fetched 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 regroup 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:

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

Before continuing, we define a few terms that apply to instructions.

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. We call those extra flows after the initial instruction flow *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, we simplify bypass logic and 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 can be in any order relative to the branch are allowed, 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/not taken 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 operand 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 FGM pipeline.

In the stage in which a divide/square root is moved from the first part to the last part, we cannot issue any instructions 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 depend 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 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 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 III Cu processor does not allow the RDASR and RDPR instructions to be dispatched. Instead, we wait for all pipeline states 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 we force bubbles after the trap exit instructions to give us the cycles to do it all. We will not accept a new instruction 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.

Pipelines 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 4-4) dealing with dispatching properties and latencies of operations. The static or nominal properties are modelled in the following terms (columns in TABLE 4-4), 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, 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 holding 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 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 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 4-4 UltraSPARC III Cu 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		
`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

TABLE 4-4 UltraSPARC III Cu Instruction Latencies and Dispatching Properties *(2 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
`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		
`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		

TABLE 4-4 UltraSPARC III Cu Instruction Latencies and Dispatching Properties *(3 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
`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		
`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		

TABLE 4-4 UltraSPARC III Cu Instruction Latencies and Dispatching Properties *(4 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
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	N/A		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		
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]	

TABLE 4-4 UltraSPARC III Cu Instruction Latencies and Dispatching Properties *(5 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
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		
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	

TABLE 4-4 UltraSPARC III Cu Instruction Latencies and Dispatching Properties *(6 of 6)*

Instruction	Latency	Dispatch Blocking After	Pipeline	Break	SIG
`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 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 stall 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. The floating-point formats conform to the IEEE Standard for Binary Floating-point Arithmetic, IEEE Std 754-1985. In tagged words, the least significant two bits are treated as a tag; the remaining 30 bits are treated as a signed integer.

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

		Range	
Data Type	**Width (bits)**	**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	$2^{64} - 1$
(Unsigned) tagged integer word	32	0	$2^{30} - 1$

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 (extendedword)*	`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.

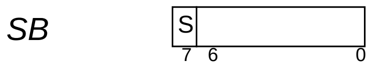


FIGURE 5-1 Signed Integer Byte Data Format

5.1.3.2 Signed Integer Halfword

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

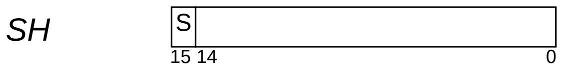


FIGURE 5-2 Signed Integer Halfword Data Format

5.1.3.3 Signed Integer Word

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

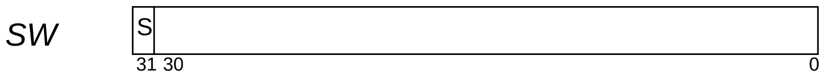


FIGURE 5-3 Signed Integer Word Data Format

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.

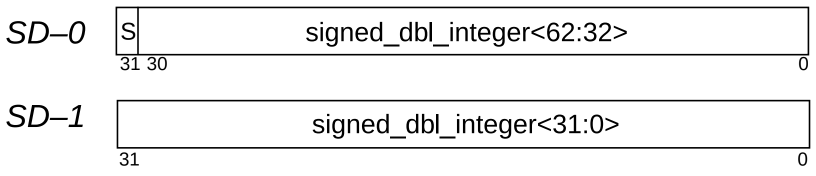


FIGURE 5-4 Signed Integer Double Data Format

5.1.3.5 Signed Extended Integer

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

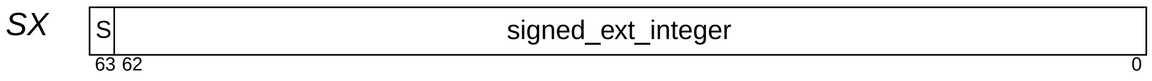


FIGURE 5-5 Signed Extended Integer Data Format

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.

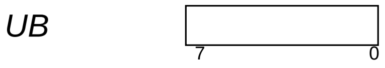


FIGURE 5-6 Unsigned Integer Byte Data Format

5.1.4.2 Unsigned Integer Halfword

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

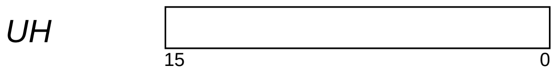


FIGURE 5-7 Unsigned Integer Halfword Data Format

5.1.4.3 Unsigned Integer Word

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

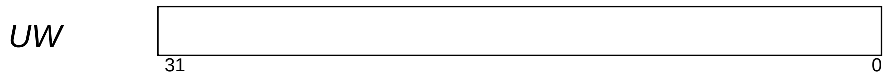


FIGURE 5-8 Unsigned Integer Word Data Format

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.

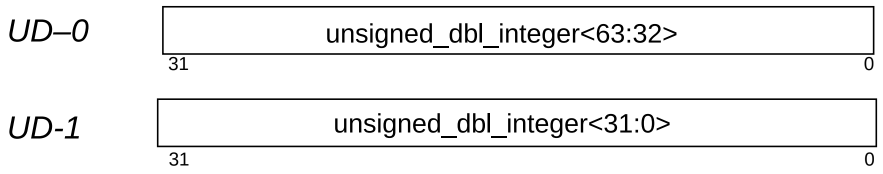


FIGURE 5-9 Unsigned Integer Double Data Format

5.1.4.5 Unsigned Extended Integer

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

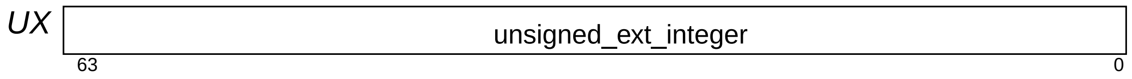


FIGURE 5-10 Unsigned Extended Integer Data Format

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 least significant bit (LSB) positions. Bit 31 is the overflow bit.

FIGURE 5-11 illustrates the tagged word data format.

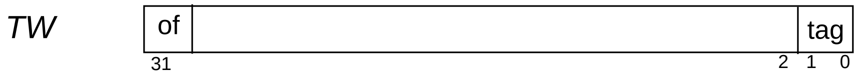


FIGURE 5-11 Tagged Word Data Format

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*
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.

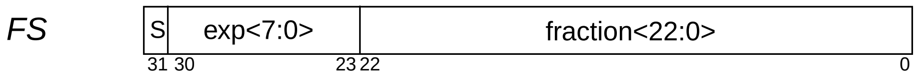


FIGURE 5-12 Floating-Point Single-Precision Data Format

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

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

Normalized value (0 < e < 255)	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal value (e = 0)	$(-1)^s \times 2^{-126} \times 0.f$
Zero (e = 0)	$(-1)^s \times 0$
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*
$-\infty$ (negative infinity)	s = 1; e = 255 (max); f = .000--00
$+\infty$ (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.

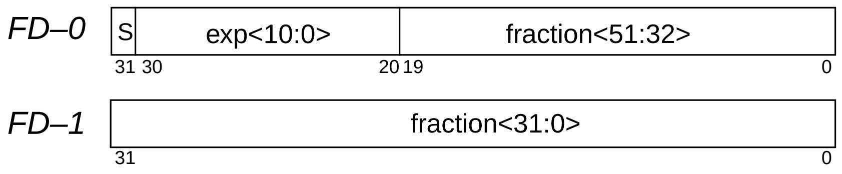


FIGURE 5-13 Floating-Point Double-Precision Double Word Data Format

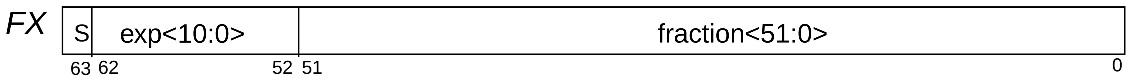


FIGURE 5-14 Floating-Point Double-Precision Extended Word Data Format

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*
$-\infty$ (negative infinity)	s = 1; e = 2047 (max); f = .000--00
$+\infty$ (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.

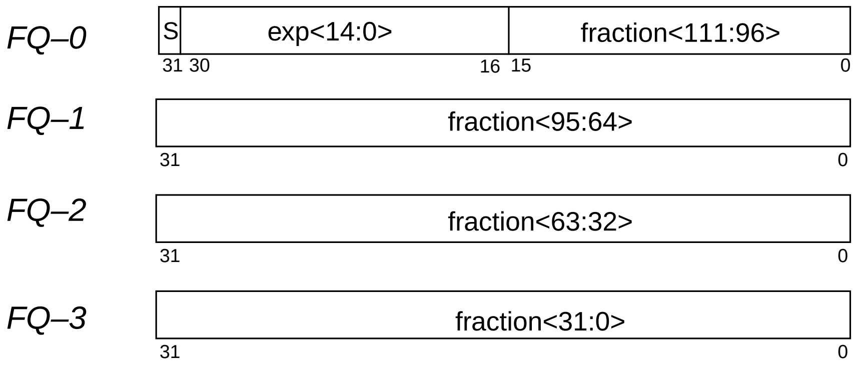


FIGURE 5-15 Floating-Point Quad-Precision Data Format

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

s = sign (1-bit)
e = biased exponent (15 bits)
f = fraction (112 bits)
u = undefined	

Normalized value (0 < e < 32767)	$(-1)^s \times 2^{e-16383} \times 1.f$
Subnormal value (e = 0)	$(-1)^s \times 2^{-16382} \times 0.f$
Zero (e = 0)	$(-1)^s \times 0$
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*
$-\infty$ (negative infinity)	s = 1; e = 32767 (max); f = .000--00
$+\infty$ (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.

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 in FIGURE 5-16.

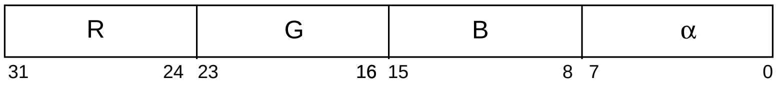


FIGURE 5-16 Pixel Data Format with Band Sequential Ordering Shown

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.

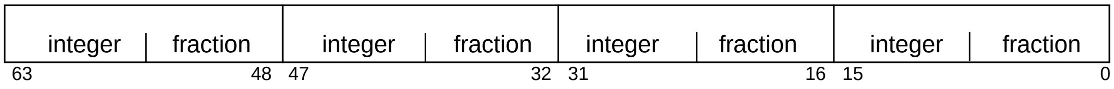


FIGURE 5-17 Fixed16 VIS Data Format

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.

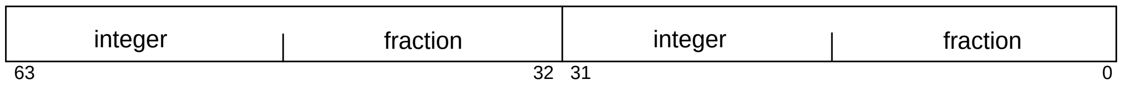


FIGURE 5-18 Fixed32 VIS Data Format

CHAPTER 6

Registers

This chapter discusses the following topics:

6.1 Introduction

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

There are separate working registers for the integer and floating-point units (FPUs). Both of 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 new diagrams and tables appear for the first time. Contents of this chapter applies to non-privileged mode unless stated otherwise.

6.2 Integer Unit General-Purpose `r` Registers

An UltraSPARC III Cu 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 *locals* and eight *ins*) and a half register set (eight *outs*) 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`, `.IG`, and `.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	Comments
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]	

TABLE 6-1 Integer Unit General-Purpose Registers *(Continued)*

Windowed Register Name	r Register Address	Source	Comments
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 zero

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

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 "Processor State (PSTATE) Privileged Register 6" on page 112 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.

An illustration showing the current IU registers is shown in FIGURE 6-1.

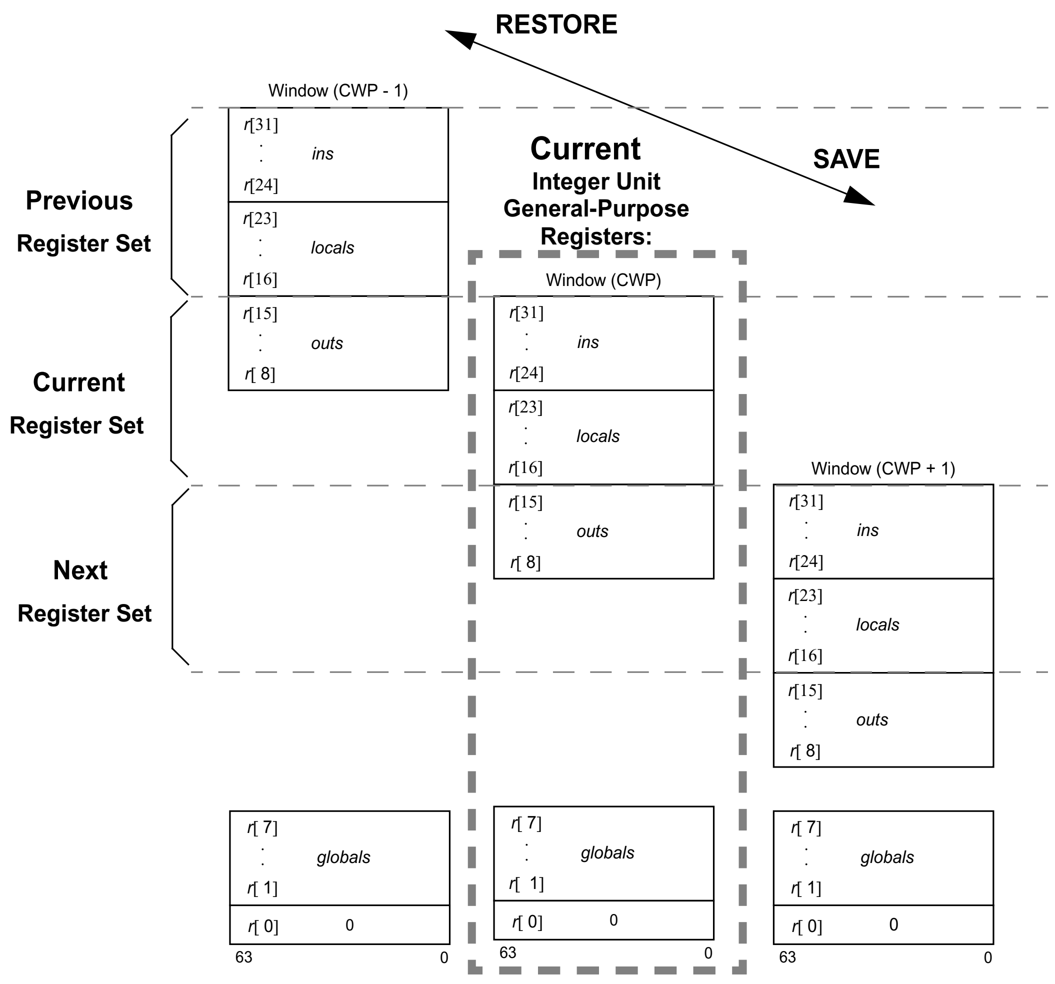


FIGURE 6-1 Three Overlapping Windows and the Eight Global Registers

---

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: 8 *normal* global registers, 8 *alternate* global registers, 8 *interrupt* global registers, 8 *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 $8 \leq o \leq 15$, 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 $24 \leq i \leq 31$, 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.

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.

6.2.3 128-bit Operand Considerations

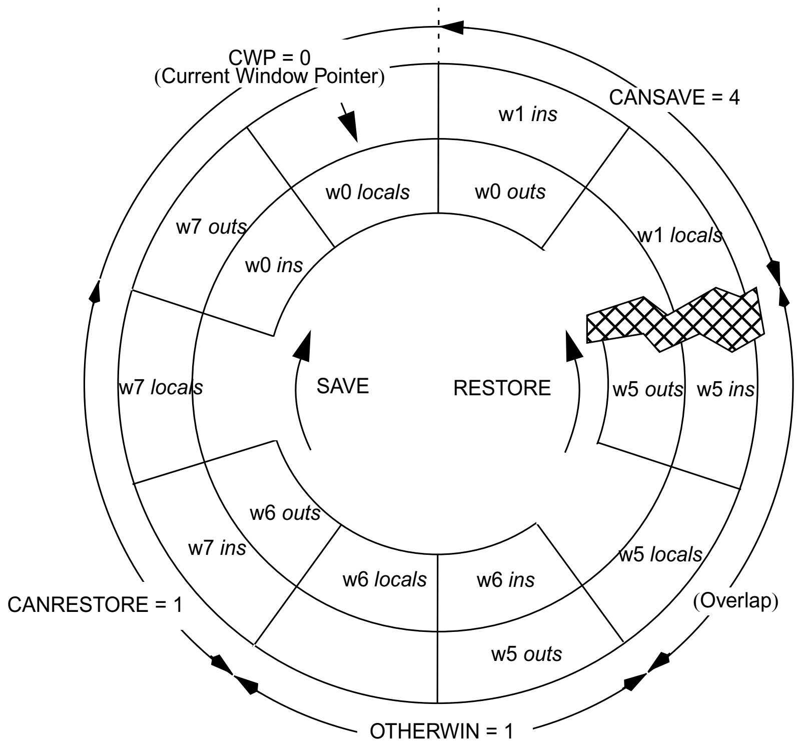
LDD, LDDA, STD, and STDA instructions access 128-bit data associated with adjacent r registers and require even-odd register alignment. An attempt to execute an LDD, LDDA, STD, or STDA instruction that refers to a misaligned (odd) destination register number causes an *illegal_instruction* trap.

6.3 Register Window Management

Note – Register window management is the responsibility of the operating system (supervisor code). The user code sees an unlimited stack of register windows and does not have to worry about register window management. The operating system provides support for underflow and overflow of the stack. This mechanism is transparent to the user code.

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 are 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 the SPARC V8 architecture, 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 a UltraSPARC III Cu 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).

Programming Note – 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*).

Programming Note – 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.

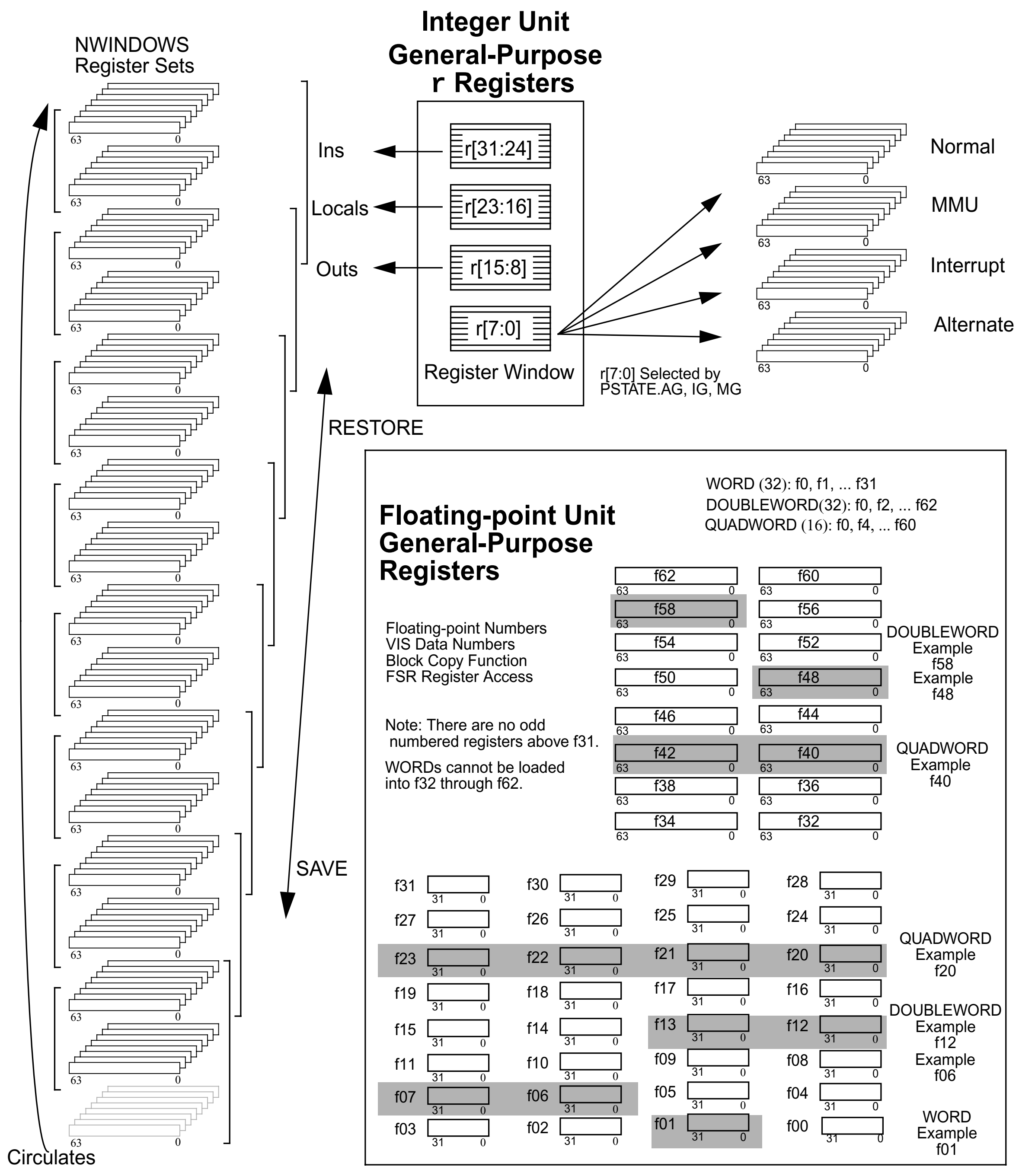


FIGURE 6-3 Integer Unit r Registers and Floating-Point Unit Working Registers

6.5 Control and Status Register Summary

This section presents a summary of control and status registers.

6.5.1 State and Ancillary State Register Summary

See FIGURE 6-4 and TABLE 6-6 for more information on state and ancillary state registers (ASRs).

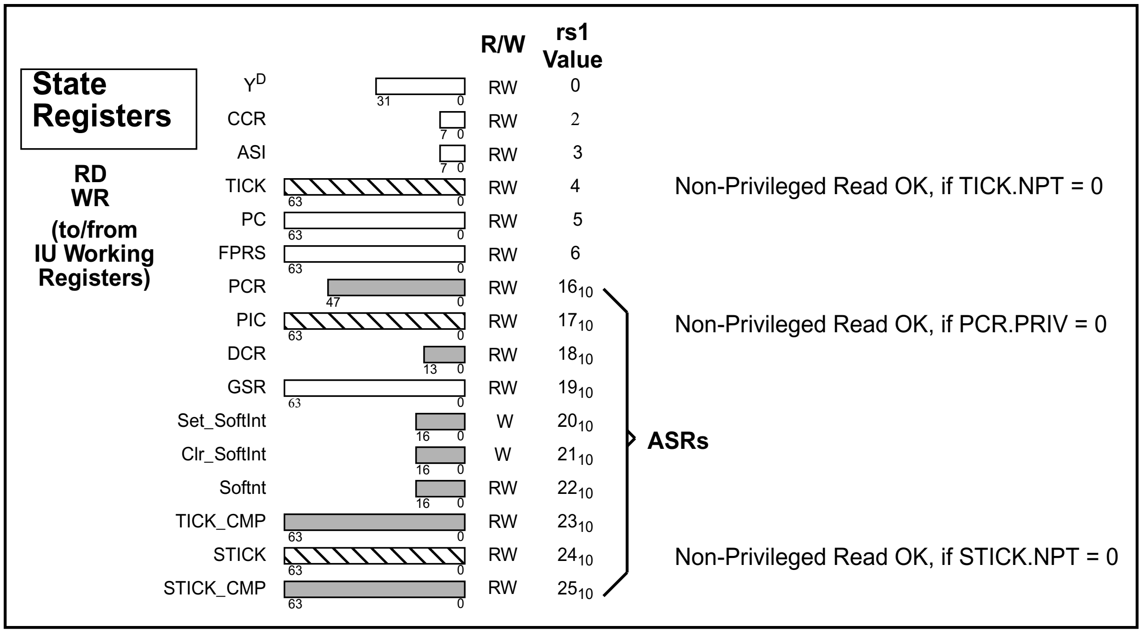


FIGURE 6-4 State and Ancillary State Registers

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 CPU 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.		
ASR 16	Privileged	RW	PCR	Performance Instrumentation	Chapter 14, “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	CPU 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

See FIGURE 6-5 and TABLE 6-7 for more information on privileged registers.

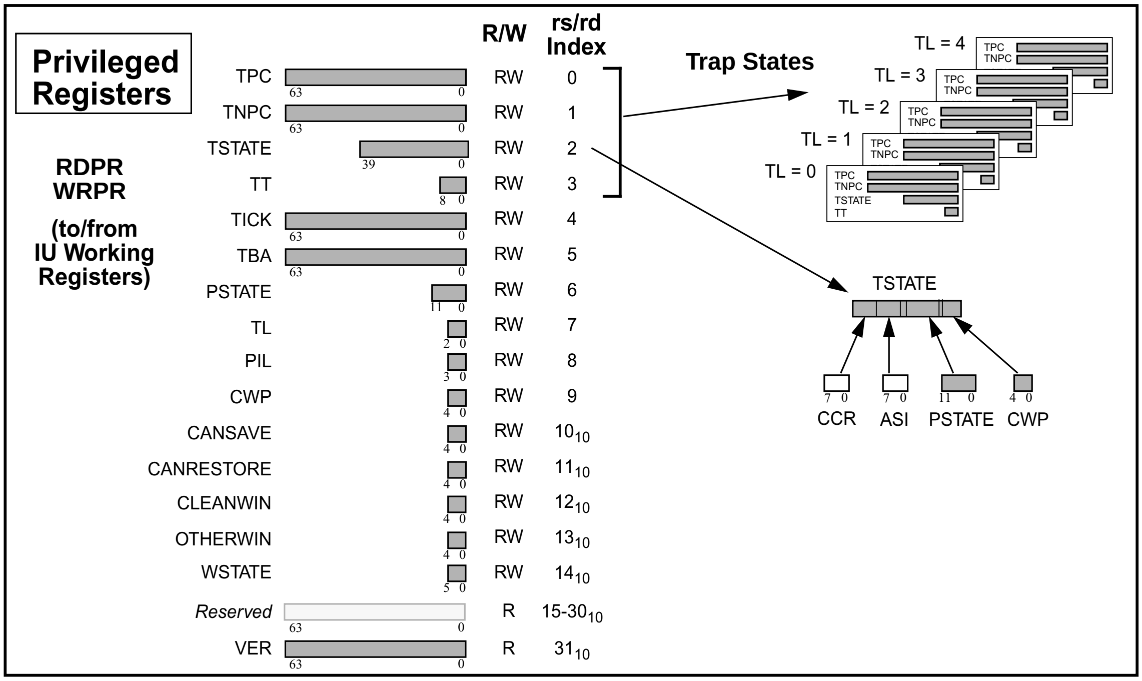


FIGURE 6-5 Privileged Registers

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	CPU 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	Saveable 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

See FIGURE 6-6 and TABLE 6-8 for more information on ASI and specially accessed registers.

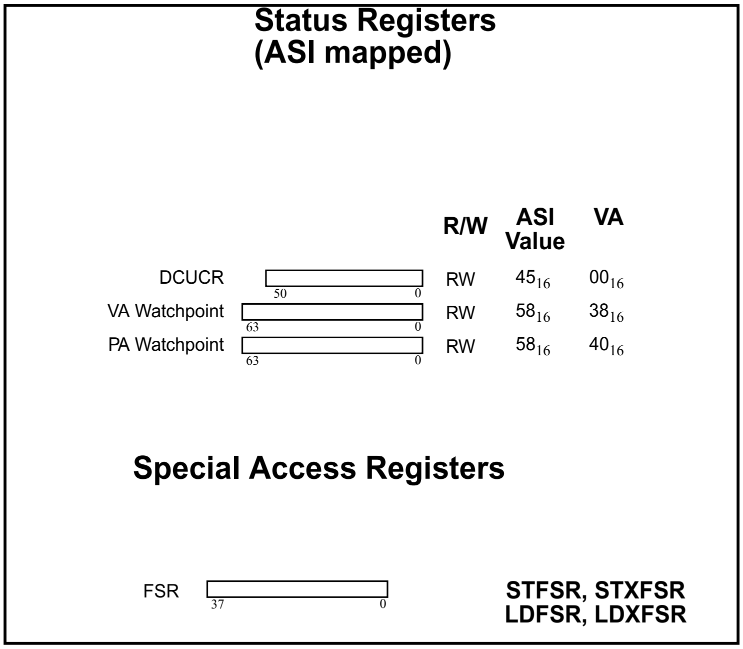


FIGURE 6-6 ASI and Specially Accessed Registers

TABLE 6-8 ASI and Specially Accessed Registers

Type	Abbreviation	Description	Access Restriction	R/W	Reference Section	Notes
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 ASRs, numbered from 7 through 31. The eight State Registers, 0 through 7, are defined by the SPARC V9 architecture.

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.

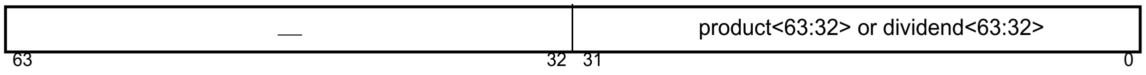


FIGURE 6-7 Y Register

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.

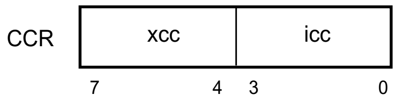


FIGURE 6-8 Condition Codes Register

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.

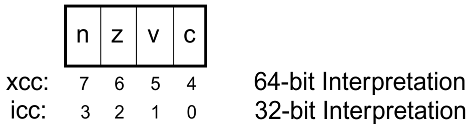


FIGURE 6-9 Integer Condition Codes (`CCR_icc` and `CCR_xcc`)

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 = non-negative.

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

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

The `c` bits indicate whether a 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 (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 equal to zero select restricted ASIs. When a non-privileged instruction makes an access that uses an ASI with bit 7 equal to zero, a *privileged_action* exception is generated.

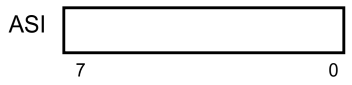


FIGURE 6-10 Address Space Identifier Register

6.6.4 TICK Register (TICK) ASR4

See Section 6.7.4, "Timer State Registers: ASRs 4, 23, 24, 25" 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 (FPU) is presented in Section 6.9.1.

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.

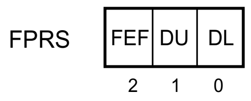


FIGURE 6-11 Floating-Point Registers State Register

6.6.6.1 FPRS_enable_fp (FEF)

Bit 2, FEF, determines whether the FPU is enabled. If it is disabled, executing a floating-point instruction causes a *fp_disabled* trap. 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.

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 it pessimistically; it may be set 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 it pessimistically; it may be set 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.

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 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.

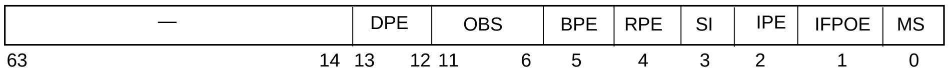


FIGURE 6-12 Dispatch Control Register (ASR 18)

TABLE 6-9 `DCR` Bit Description

Bit	**Field**	**Type**	**Description**
63:14	—		*Reserved*.
13:12	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 automatically and correctly by HW.
11:6	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`).
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`).

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

Bit	Field	Type	Description
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 automatically and correctly by HW.
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`).

Note – Both IPE and DPE will default to 0 (disable) after power-on or system reset.

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 III Cu processor forces a *fp_disabled* trap when an interrupt occurs on FP-only code. The trap handler is then responsible for checking whether the floating-point 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 a 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.

Accesses to the GSR cause a *fp_disabled* trap if `PSTATE.PEF` or `FPRS.FEF` is zero.

The `GSR` is illustrated in FIGURE 6-13 and described in TABLE 6-10.

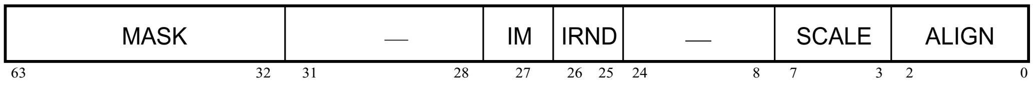


FIGURE 6-13 Graphic Status Register (ASR 19)

TABLE 6-10 `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`.
26:25	IRND<1:0>	IEEE Std 754-1985 rounding direction to use in Interval Mode (`GSR.IM` = 1), as follows: **IRND** / **Round toward** 0 / Nearest (even if tie) 1 / 0 2 / + ∞ 3 / − ∞ When `GSR.IM` = 1, the value in `GSR.IRND` overrides the value in `FSR.RD`.

TABLE 6-10 GSR Bit Description *(Continued)*

Bit	Field	Description
24:8	*Reserved.*	
7:3	SCALE<4:0>	Shift count in the range 0–31, used by the `PACK` instructions for formatting.
2:0	ALIGN<2:0>	Least three significant bits of the address computed by the last executed `ALIGNADDRESS` or `ALIGNADDRESS_LITTLE` instruction.

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 6-11 and FIGURE 6-14 for more details.

TABLE 6-11 Register-Window State Registers

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

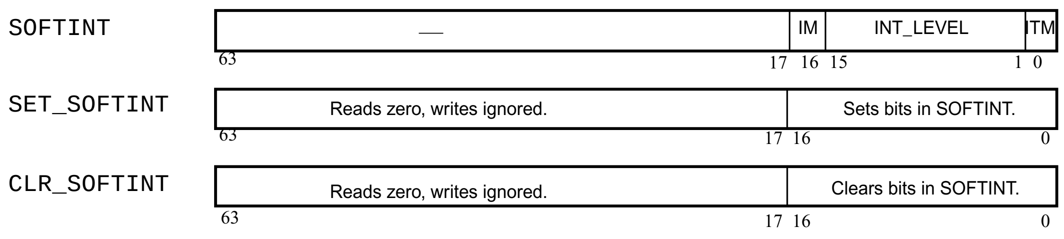


FIGURE 6-14 `SOFTINT`, `SET_SOFTINT`, and `CLR_SOFTINT` Register Formats

SOFTINT Register

The operating system uses the `SOFTINT` to schedule interrupts. The field definitions are described in TABLE 6-12.

TABLE 6-12 `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" 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 "TICK_COMPARE Register" 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 CPU clock. The STICK timer is driven by the system clock. Four registers are used to implement the timer and support the timer interrupts.

TABLE 6-13 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

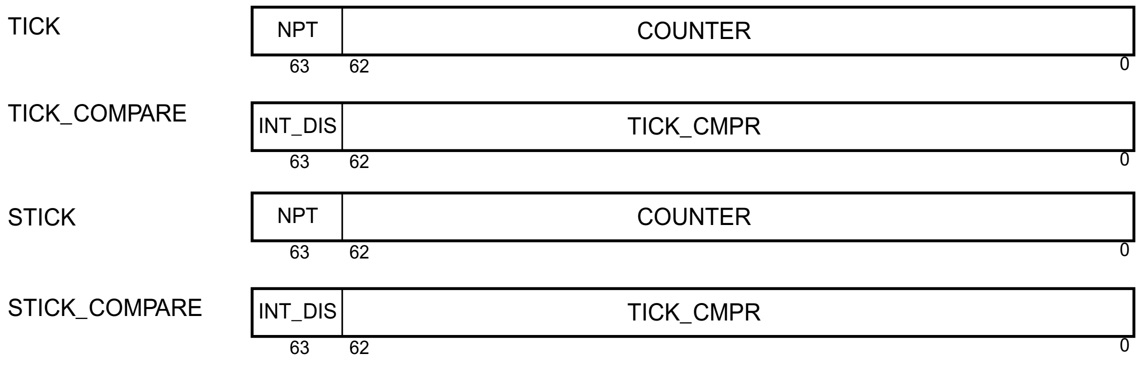


FIGURE 6-15 Timer State Registers

TICK Register

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

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 writable 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.

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.

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 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 6-16 shows the format for this register group. This figure is followed by a description of each register. FIGURE 6-17 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.

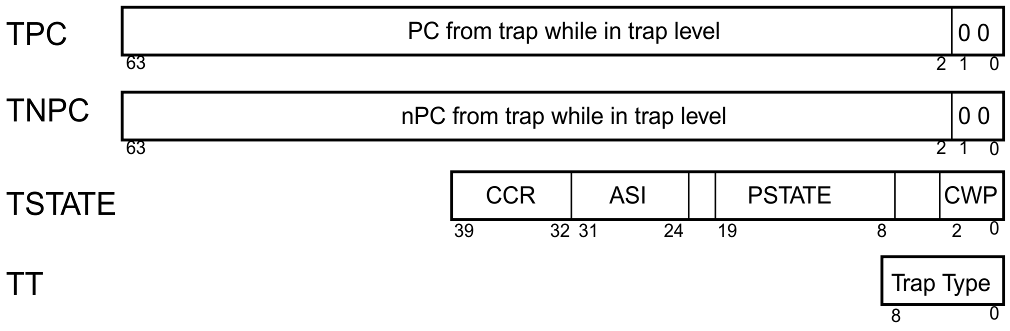


FIGURE 6-16 Trap State Register Format

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 shown in FIGURE 6-17.

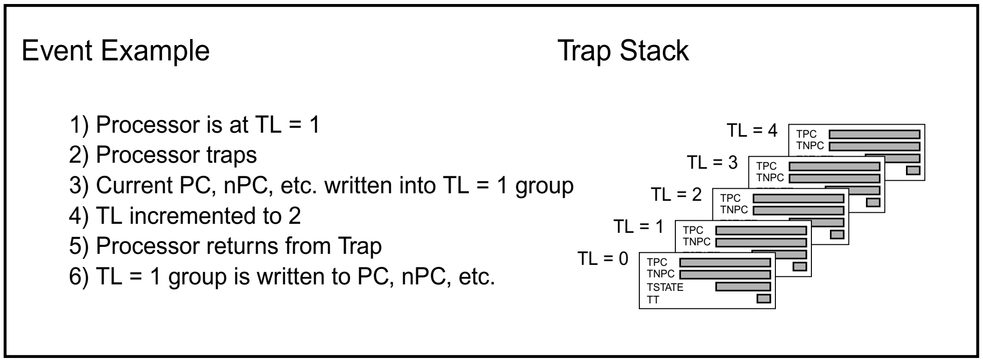

Event Example
Trap Stack
1) Processor is at TL = 1
2) Processor traps
3) Current PC, nPC, etc. written into TL = 1 group
4) TL incremented to 2
5) Processor returns from Trap
6) TL = 1 group is written to PC, nPC, etc.
TL = 4
TL = 3
TL = 2
TL = 1
TL = 0
TPC
TNPC
TSTATE
TT

FIGURE 6-17 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 6-14. During normal operation, the trap stack register values defined for the trap levels above the current one are undefined.

TABLE 6-14 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 6-18, 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.

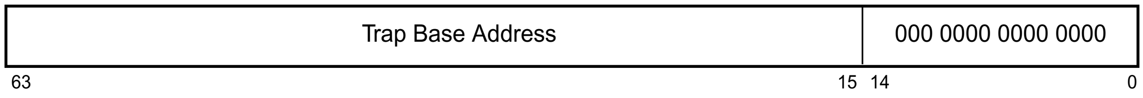


FIGURE 6-18 Trap Base Address Register

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 6-19.

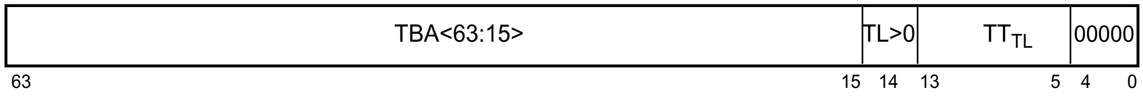


FIGURE 6-19 Trap Vector Address Format

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. See Chapter 12, "Traps and Trap Handling" for more details on trap vectors.

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 (FIGURE 6-20) 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. See Chapter 12, "Traps and Trap Handling" for more details.

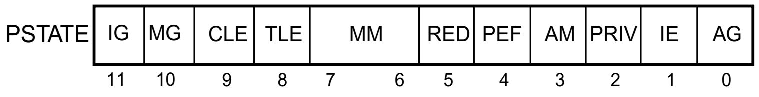


FIGURE 6-20 PSTATE Fields

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 113 through page 114 describe the fields contained in the PSTATE register.

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

The UltraSPARC III Cu 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 6-15.

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

TABLE 6-15 PSTATE Global Register Selection Events

		PSTATE Settings		
Event	**Globals Selected for Use**	**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
Any trap other than those listed above	Alternate Global registers	1	0	0
	Reserved.	1	0	1
	Reserved.	1	1	0
	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 – UltraSPARC III Cu processors 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 114. When an *interrupt_vector* trap (trap type = 60_{16}) is taken, 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 MG and clears IG and 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 `AG` and clearing `IG` and `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 6-16 for MM Encodings.

TABLE 6-16 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 III Cu 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 the following method:

- Execute 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 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 (D-SFAR).

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

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 6-21, 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. See Chapter 12, “Traps and Trap Handling” for more details about the TL register.

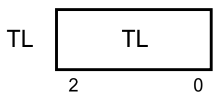


FIGURE 6-21 Trap Level Register

Programming Note – Writing the TL register with a value greater than MAXTL (five for UltraSPARC III Cu) 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 6-22, 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.

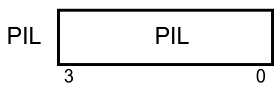


FIGURE 6-22 Processor Interrupt Level Register

On SPARC V8 processors, the Level-15 interrupt is considered to be nonmaskable; therefore, 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.

Register-window management is described in a separate chapter.

TABLE 6-17 Register-Window State Privileged Registers

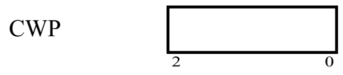
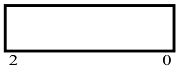
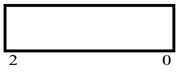
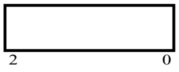
Register-Window State Registers	Value Range	Description
Current Window Pointer 	0 to 7	**State Register 9:** The `CWP` register is a counter that identifies the current window into the set of integer registers. See Chapter 12, “Traps and Trap Handling” for information on how hardware manipulates the `CWP` register.
Saveable Window Sets	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.

TABLE 6-17 Register-Window State Privileged Registers *(Continued)*

Register-Window State Registers	Value Range	Description
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 6-17. 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 the SPARC V8 and SPARC V9 architectures are visible only to privileged software; they are invisible to non-privileged software.

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

2. PSR.CWP in the SPARC V8 architecture is changed on each trap. In the SPARC V9 architecture, 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 6-23, 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].

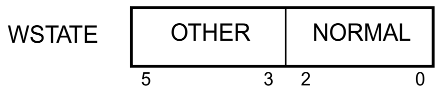


FIGURE 6-23 WSTATE Register

6.8.8 Version (VER) Privileged Register 31

The version register, shown in FIGURE 6-24, 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.

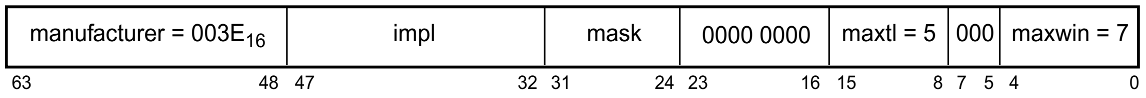


FIGURE 6-24 Version Register

VER.manuf Field

The `VER.manuf` field contains our 16-bit manufacturer code, $003E_{16}$, which is our 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 6-18 shows the processor implementation codes.

TABLE 6-18 Processor Implementation Codes

Processor	**VER.impl**
UltraSPARC I	0010_{16}
UltraSPARC II	0011_{16}
UltraSPARC IIi	0012_{16}
UltraSPARC IIe	0013_{16}
UltraSPARC III Cu	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 6-19 lists the UltraSPARC III Cu Processor Mask Version.

TABLE 6-19 UltraSPARC III Cu 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 6-24, contain FPU mode and status information. Section 6.6.6, "Floating-Point Registers State (FPRS) Register 6" presents state information about the FPU.

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 6-25 illustrates the FSR fields.

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

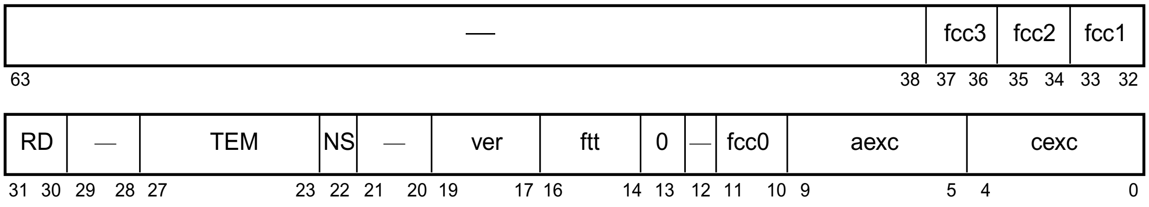


FIGURE 6-25 FSR Fields Reserved Bits

Reserved Bits

Bits 63–38, 29–28, 21–20, and 12 are reserved. When read by an STXFSR instruction, these bits shall 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 123 through page 131 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 – SPARC V9 architecture's fcc0 is the same as SPARC V8 architecture's fcc.

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 fcc*n* 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 6-20, 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 (SNaN) or a quiet NaN (QNaN). If FCMP or FCMPE generates a *fp_exception_ieee_754* exception, then fcc*n* is unchanged. TABLE 6-20 shows the floating-point condition codes Fields of FSR.

TABLE 6-20 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 6-21 shows the rounding direction fields.

TABLE 6-21 Rounding Direction (RD) Field of FSR

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

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, a *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*)

For the UltraSPARC III family of processors, the value in `FSR.ver` is zero.

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 an 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 an 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 6-22.

Note – The value "7" is reserved for future expansion.

TABLE 6-22 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

TABLE 6-22 Floating-Point Trap Type (`ftt`) Field of FSR *(Continued)*

ftt	Trap Type	Trap Vector
5	*hardware_error*	Reserved, unimplemented
6	*invalid_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 standard (recommended) set of conditions is listed in TABLE 6-23. An implementation may cause *fp_exception_other* with *unfinished_FPop* under a different (but specified) set of conditions.

TABLE 6-23 Standard Conditions in 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 – 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
`fitos`	N/A	N/A	$- 2^{22} <= operand < 2^{22}$
`fxtos`	N/A	N/A	$- 2^{22} <= operand < 2^{22}$
`fitod`	N/A	N/A	None
`fxtod`	N/A	N/A	$- 2^{51} <= operand < 2^{51}$

TABLE 6-23 Standard Conditions in 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
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 FPops 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 6-26 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.

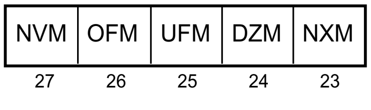


FIGURE 6-26 Trap Enable Mask (TEM) Fields of FSR

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 6-27 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.

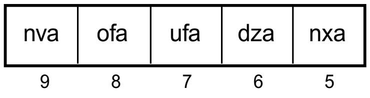


FIGURE 6-27 Accrued Exception Bits (`aexc`) Fields of `FSR`

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 6-28 for an illustration.

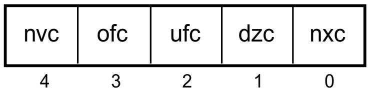


FIGURE 6-28 Current Exception Bits (`cexc`) Fields of `FSR`

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 754-1985 exception.

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

- 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 6-24 (where “x” indicates “don’t care”):

TABLE 6-24 Setting of FSR.cexc Bits

Exception(s) Detected in FP Operation			**Trap Enable Mask bits (in FSR.TEM)**			*fp_exception_ ieee_754* **Trap Occurs?**	**Current Exception bits (in FSR.cexc)**			**Notes**
of	**uf**	**nx**	**OFM**	**UFM**	**NXM**		**ofc**	**ufc**	**nxc**	
-	-	-	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)
-	-	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 ÷ 0.0 and ∞ – ∞ 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 III Cu 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 an FSR that is fully compliant with IEEE Std 754-1985.

6.10 ASI Mapped Registers

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

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 6-29 and described in TABLE 6-25. In the table, the field definitions and bits are grouped by function rather than by strict bit sequence.

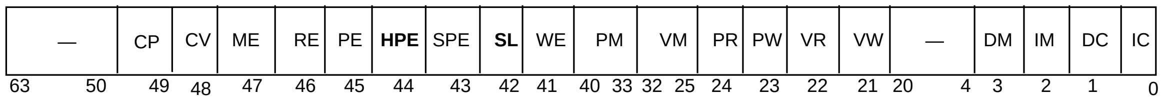


FIGURE 6-29 DCU Control Register Access Data Format (ASI 45_{16})

TABLE 6-25 DCUCR Bit Field Descriptions *(1 of 3)*

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
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.	

TABLE 6-25 DCUCR Bit Field Descriptions *(2 of 3)*

Bits	Field	Type	Description	Note
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 an 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.	
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 data cache is disabled, its contents are not updated. When the D-cache is re-enabled, 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 instruction cache is disabled, its contents are not updated. When the I-cache is re-enabled, 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 6-25 DCUCR Bit Field Descriptions *(3 of 3)*

Bits	Field	Type	Description	Note
32:25	VM<7:0>		*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.	4

Watchpoint Mask (`PM` **and** `VM`)	**Least Significant 3 Bits of Address of Bytes Watched** **`7654 3210`**
00_{16}	Watchpoint disabled
01_{16}	`0000 0001`
32_{16}	`0011 0010`
FF_{16}	`1111 1111`

Bits	Field	Type	Description	Note
24, 23	PR, PW		*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.	
22, 21	VR, VW		*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.	

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 III Cu 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

UltraSPARC III Cu processors implement "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 6-26 lists ASIs that are affected by the two watchpoint traps.

TABLE 6-26 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 6-30 illustrates the Virtual Address Watchpoint Register,

where: **DB_VA** is the most significant 61 bits of the 64-bit virtual data watchpoint address.

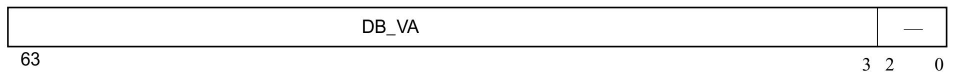


FIGURE 6-30 VA Data Watchpoint Register Format

Physical Address Data Watchpoint Register

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

Name: PA Data Watchpoint Register

FIGURE 6-31 illustrates the PA Data Watchpoint Register,

where: **DB_PA** is the most significant 61 bits of the physical data watchpoint address. The width of an UltraSPARC III Cu physical address is 43 bits.

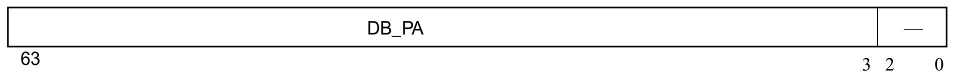


FIGURE 6-31 PA Data Watchpoint Register Format

Compatibility Note – The UltraSPARC III Cu 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

Reference Section

- Data Prefetch Instructions
- Instruction Formats and Fields
- Reserved Opcodes and Instruction Fields
- 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 III Cu processor further extends this instruction execution environment.

The UltraSPARC III Cu 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 improved 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

The SPARC V9 architecture uses big-endian byte order by default; the address of a quadword, doubleword, word, or halfword is the address of its most significant byte. Increasing the address means decreasing the significance of the unit being accessed. All instruction accesses are performed using big-endian byte order. The SPARC V9 architecture also 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 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:

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

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 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.

Chapter 8, "Address Space Identifiers" describes the ASIs in more detail. The chapter summary table associates ASI values to specific instructions.

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 redefined) 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 9, "Memory Models" using the FLUSH and Section 7.6, "Data Coherency Instructions" describes MEMBAR 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. Section 7.4.2, "FPU/VIS Data Access Instructions" discusses floating-point register load and store instructions.

The CPU 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 ORing 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 the next section).

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 can 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 optionally be 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.

Appendix A “Instruction Definitions” describes the tagged instructions.

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 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 uses 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 an 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 Std 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-endian 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 FP Arithmetic Instructions

Single-precision and double-precision FP is executed in hardware. Quad precision (128-bit) instructions are recognized by the CPU 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 instruction 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 FP 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 FP Compare Instructions

The same precision operands are compared and the `fcc` condition codes are set.

7.4.6 FP 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 uses 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 is 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 9, “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 CPU 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 instruction 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`s induce a partial ordering between sets of loads and stores and future loads and stores. *Sequencing* `MEMBAR`s 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 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 (±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 nPC is changed, the effect of the transfer of control is delayed by one instruction. Most control transfers are of delayed variety. The instruction following a delayed CTI is said to be in the *delay slot* of the CTI. Some CTI (branches) can optionally annul, that is, not execute, the instruction in the delay slot, depending upon whether the transfer is taken or not taken. Annulled instructions have no effect upon the program-visible state, nor can they cause a trap.

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

The SPARC V8 architecture 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. See Chapter 12, “Traps and Trap Handling” for more information.

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.

- NOP Instruction

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. Refer to Appendix A, "Prefetch Data" on page 560 for further details.

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

Chapter 4, “Instruction Execution” explains instruction grouping rules in detail.

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. Chapter 4, “Instruction Execution” explains execution latencies in detail.

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 $\rightarrow$ 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 $\rightarrow$ IU register & Compare logic; IU register $\rightarrow$ 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 $\rightarrow$ IU register		

TABLE 7-3 Instruction Summary for the Integer Execution Environment *(2 of 3)*

Instruction	**Description**		**Notes**
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		
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		

TABLE 7-3 Instruction Summary for the Integer Execution Environment *(3 of 3)*

Instruction	Description		Notes
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)		
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 $\rightarrow$ FPU register (lower 32-bit)			
LDXFSR	Load FSR into FP reg file: FSR $\rightarrow$ FPU register (64-bit)			
STFSR^{D}	Store FSR register: FPU (lower 32-bit) $\rightarrow$ FSR register			
STXFSR	Store FSR register: FPU $\rightarrow$ 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 source 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 instruction 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			
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 mode, 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	Tells processor to fetch data			
PREFETCHA^{PASI}	Tells 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.

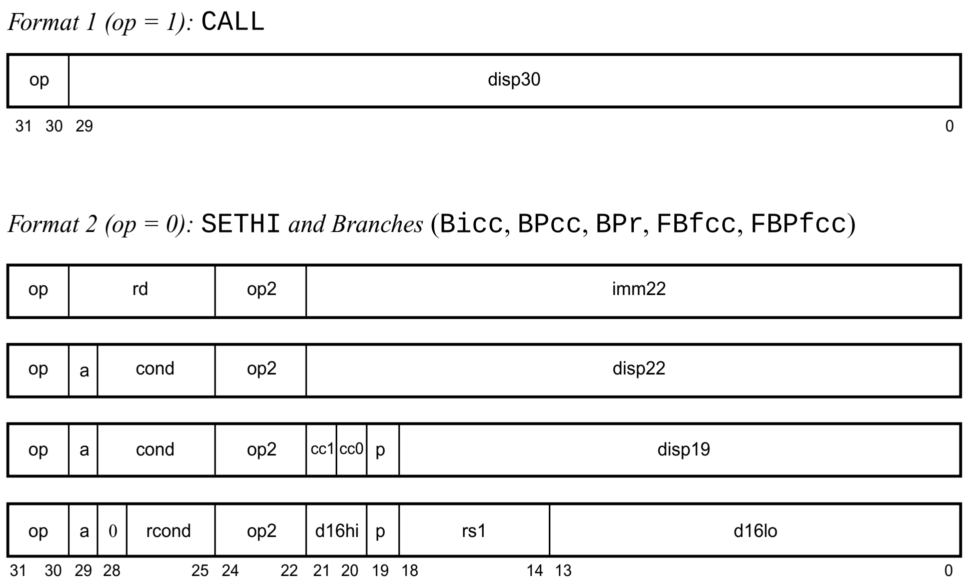


FIGURE 7-1 Summary of Instruction Formats: Formats 1 and 2

Format 3 (op = 2 or 3): Arithmetic, Logical, `MOVr`, `MEMBAR`, Prefetch, Load, *and* Store

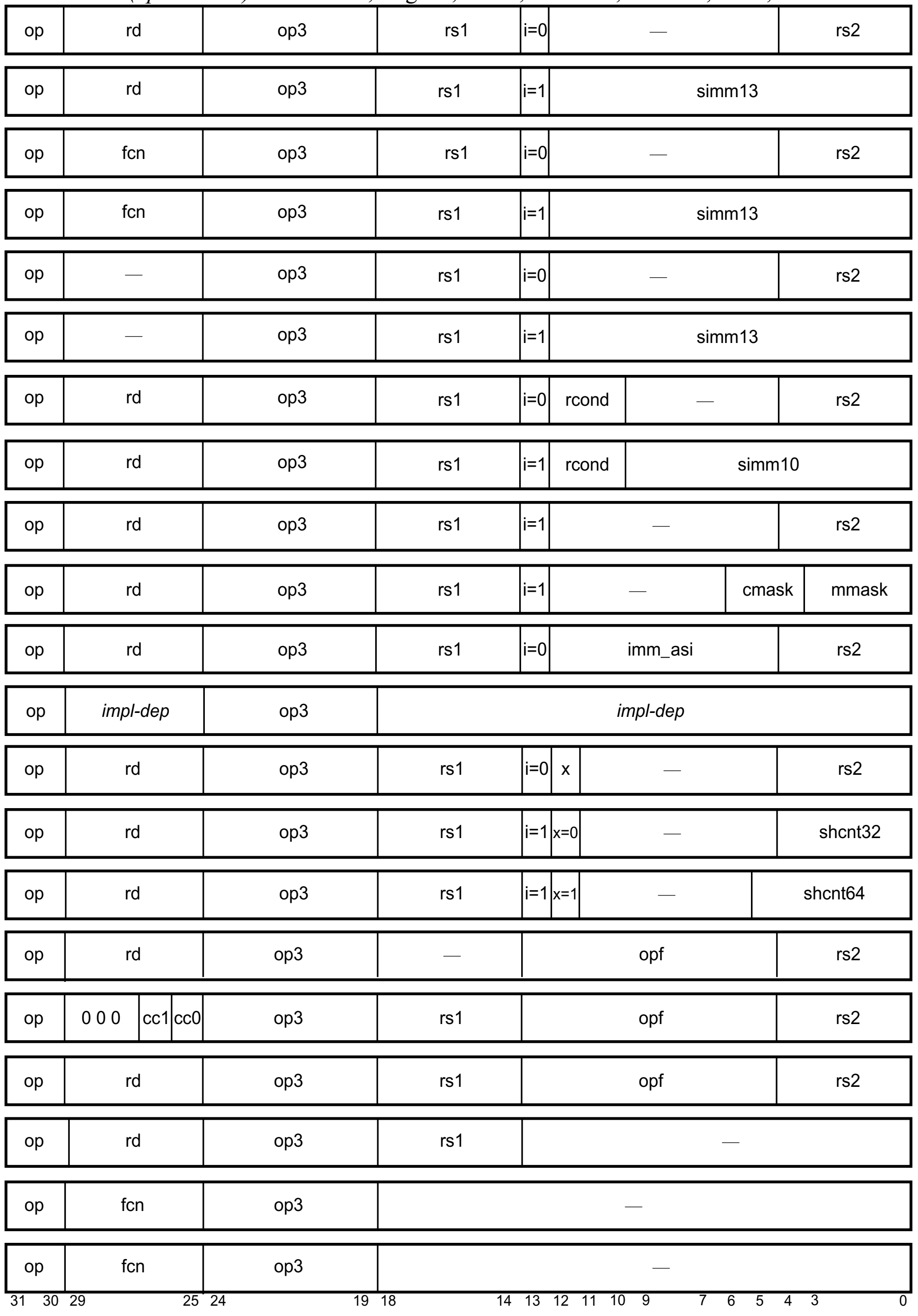

op | rd | op3 | rs1 | i=0 | — | rs2
op | rd | op3 | rs1 | i=1 | simm13
op | fcn | op3 | rs1 | i=0 | — | rs2
op | fcn | op3 | rs1 | i=1 | simm13
op | — | op3 | rs1 | i=0 | — | rs2
op | — | op3 | rs1 | i=1 | simm13
op | rd | op3 | rs1 | i=0 | rcond | — | rs2
op | rd | op3 | rs1 | i=1 | rcond | simm10
op | rd | op3 | rs1 | i=1 | — | rs2
op | rd | op3 | rs1 | i=1 | — | cmask | mmask
op | rd | op3 | rs1 | i=0 | imm_asi | rs2
op | impl-dep | op3 | impl-dep
op | rd | op3 | rs1 | i=0 | x | — | rs2
op | rd | op3 | rs1 | i=1 | x=0 | — | shcnt32
op | rd | op3 | rs1 | i=1 | x=1 | — | shcnt64
op | rd | op3 | — | opf | rs2
op | 0 0 0 | cc1 | cc0 | op3 | rs1 | opf | rs2
op | rd | op3 | rs1 | opf | rs2
op | rd | op3 | rs1 | —
op | fcn | op3 | —
op | fcn | op3 | —
31 30 29 25 24 19 18 14 13 12 11 10 9 7 6 5 4 3 0

FIGURE 7-2 Summary of Instruction Formats: Format 3

Format 4 (op = 2): `MOVcc`, `FMOVr`, `FMOVcc`*, and* `Tcc`

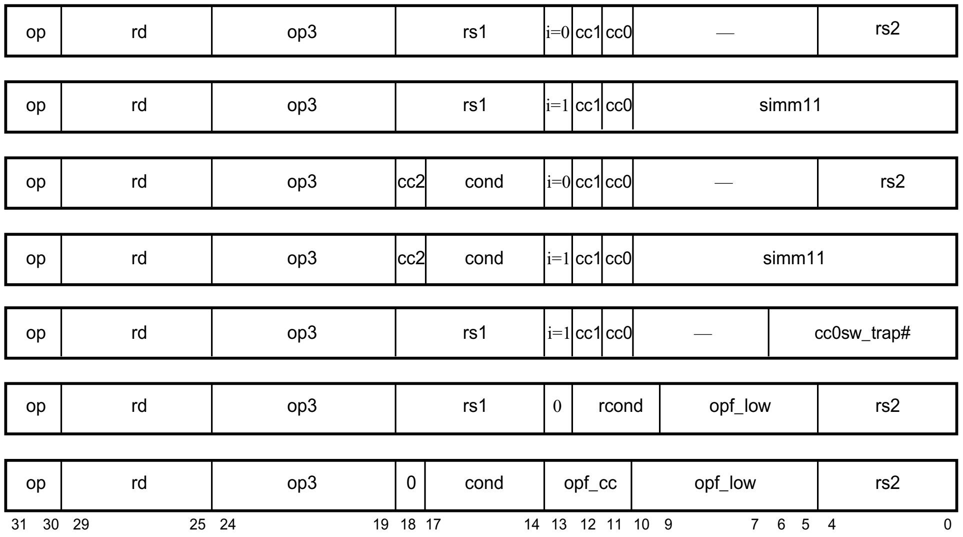


FIGURE 7-3 Summary of Instruction Formats: Format 4

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.

TABLE 7-10 Instruction Field Interpretation *(2 of 3)*

Field	Description
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.
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. **p** | **Branch Prediction** 0 | Predict that branch will not be taken 1 | Predict that branch will be taken
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.

TABLE 7-10 Instruction Field Interpretation *(3 of 3)*

Field	Description
`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.
`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, an UltraSPARC III Cu User's Manual 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") 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. See Chapter 8, "Address Space Identifiers" for details.

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.

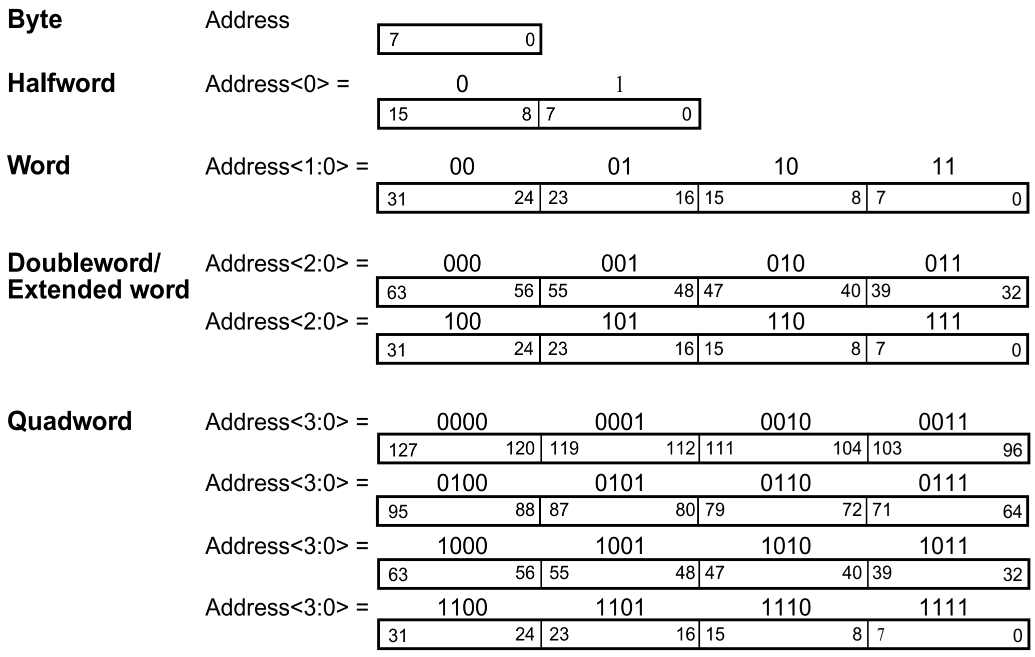


FIGURE 7-4 Big-endian Addressing Conventions

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.

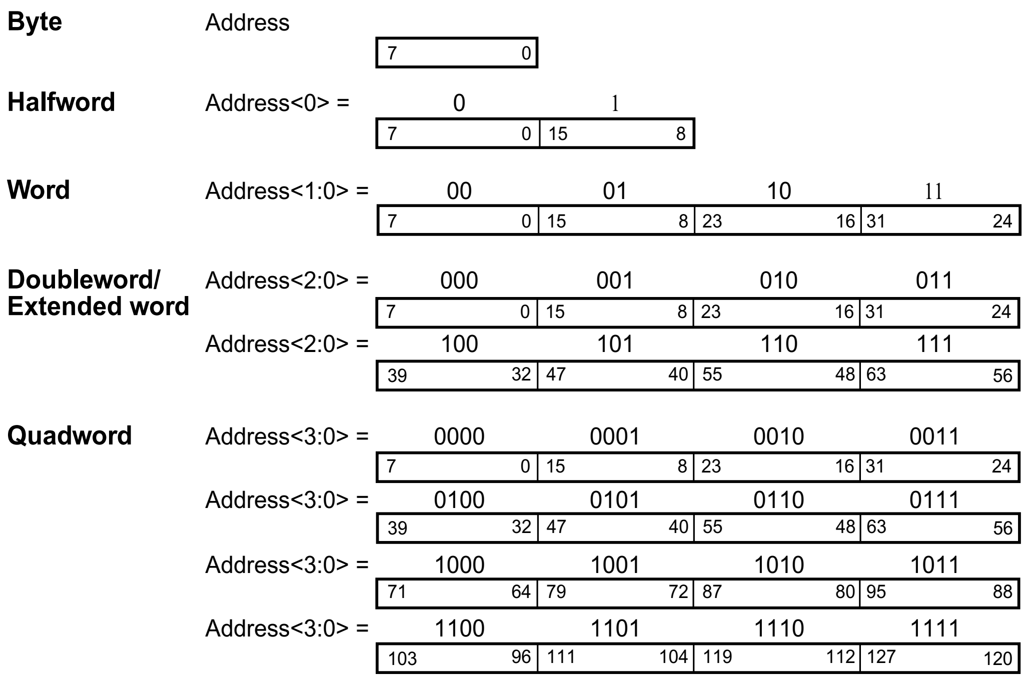


FIGURE 7-5 Little-endian Addressing Conventions

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.

CHAPTER 8

Address Space Identifiers

The address space identifiers (ASIs) are described in the following sections:

- ASI Introduction
- ASI Heredity
- ASI Groups
- Instructions Associated with the ASIs
- Using ASIs
- List of ASI Definitions
- Special Memory Access ASIs

8.1 ASI Introduction

Every instruction fetch, data load, or data store operation is specified by a 64-bit virtual address. In the SPARC architecture, there is always an ASI along with the virtual address. In most cases the ASI is implicit, but can be explicitly specified when appropriate. The ASI can provide rules for how the Memory Management Unit (MMU) should translate a virtual address to a physical address. The ASI can provide attributes for how an operation should be performed. The ASI can also be used to address internal state of the processor.

SPARC Compatibility Note – The SPARC V9 architecture has also extended the limit of virtual addresses from 32-bit (SPARC V8) to 64-bit for each address space. The SPARC V9 architecture supports 32-bit addressing through masking of the upper 32 bits to zero when the address mask (AM) bit in the PSTATE register is set.

Every instruction fetch, load or store address in the processor has an 8-bit ASI appended to the virtual address (VA). The VA plus the ASI fully specify the address. These address spaces map to main memory, the processor subsystems, and internal control, status, and diagnostics registers (CSRs) within a processor. These ASIs are internal to the processor and are not visible outside. ASIs can create special transactions on internal processor busses and assert special internal control signals.

For instruction fetches and data loads or stores that do not use the load or store alternate instructions, the ASI is an implicit ASI generated by the hardware. If a load alternate or store alternate instruction is used, the value of the ASI can be specified in the `%asi` register or as an immediate value in the instruction.

8.1.1 Load/Store Instructions

Not all load/store instructions have explicit ASIs.

8.1.2 Processor State

Privileged versus non-privileged mode affects the way some ASIs are interpreted.

8.1.3 Default ASIs

Default ASIs are precoded. They are selected based on privileged/non-privileged mode.

8.1.4 Non-Translating and Bypassing ASIs

Non-translating and bypassing ASIs use the VA for the PA. The bypassing ASIs select the processor control, status, and diagnostic registers and RAM arrays for diagnostic purposes. Non-translating ASIs essentially use the VA to address main memory, the system, and memory mapped control, status, and diagnostic registers.

Sometimes, bypassing ASIs are included when referring to non-translating ASIs.

8.1.5 Datapath

The datapath supports byte swapping for endianess and variable length data sizes from single bytes (partial store) to 64-byte blocks (block load/store). FIGURE 8-1 shows the datapath and logic for ASIs.

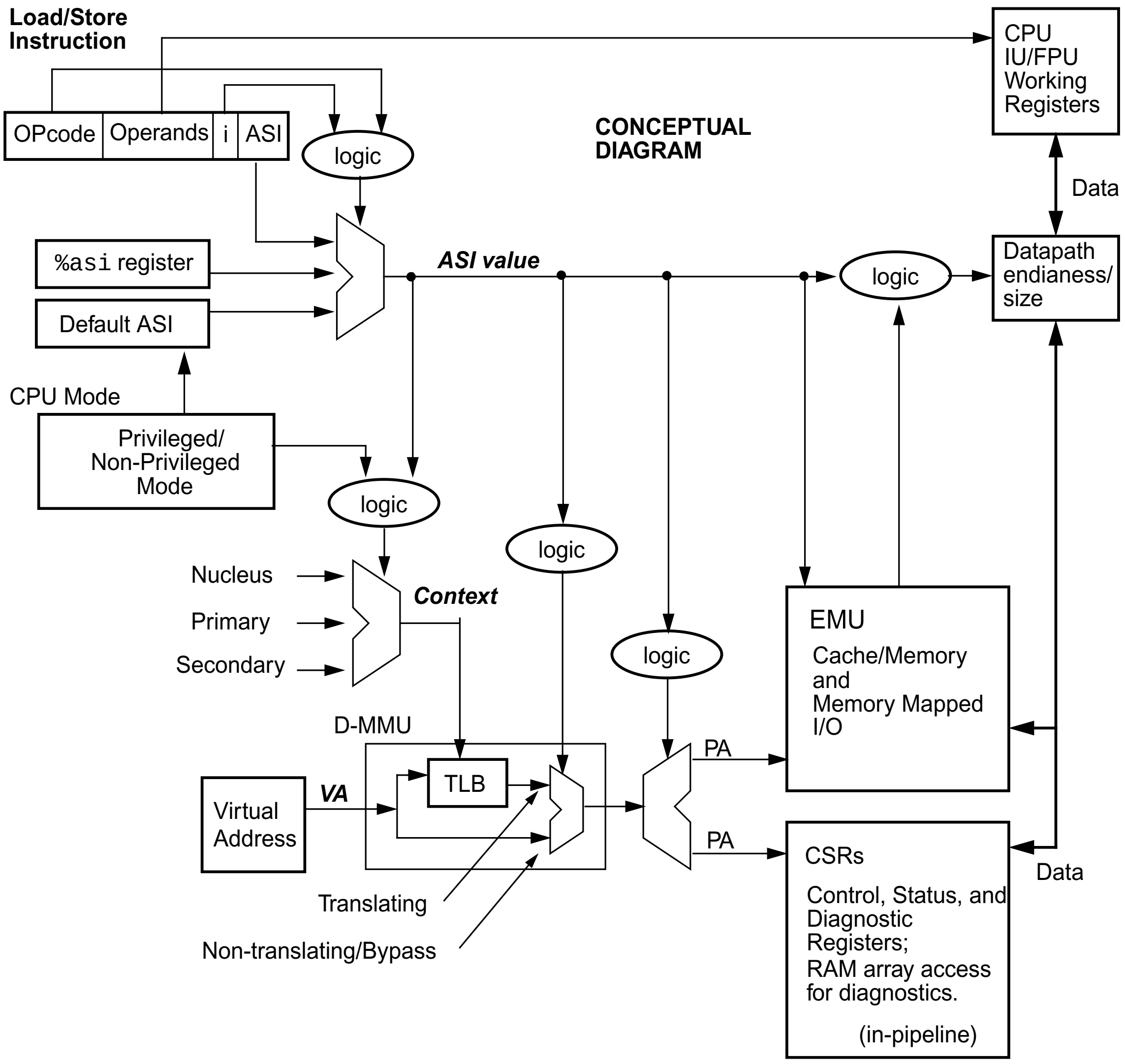


FIGURE 8-1 ASI Source and Function Conceptual Diagram

8.2 ASI Heredity

FIGURE 8-2, illustrates how the ASIs are defined at different processor architectural levels. The most universal ASIs are defined by the SPARC V9 architecture. The next level is defined for all processors based on the UltraSPARC III Family of processors. The last level defines ASIs belonging to a particular processor.

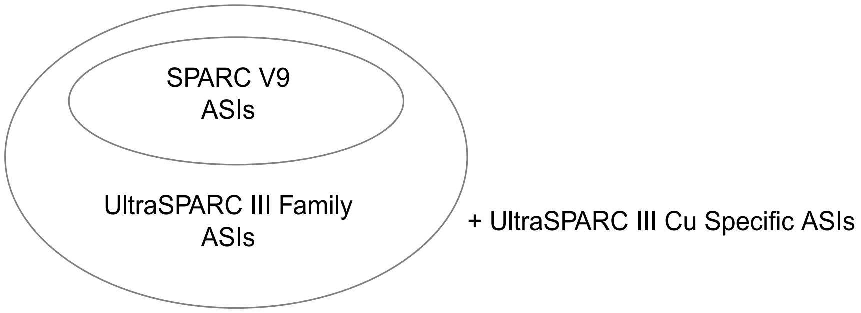


FIGURE 8-2 ASI Groups

ASIs defined by common architecture definitions should work across the processors within that family. All SPARC V9 architecture defined ASIs will work on all processors defined by the SPARC V9 architecture including all UltraSPARC processors, but an ASI defined for a specific processor will not necessarily work on other SPARC V9 processors.

8.2.1 SPARC V9 ASIs

The SPARC V9 architecture defines a set of required ASIs for SPARC V9 processors. These ASIs are supported in all UltraSPARC processors. Some of the ASIs have been deprecated in favor of newer ASIs that take advantage of the 64-bit architecture.

8.2.2 UltraSPARC UltraSPARC III Family ASIs

The UltraSPARC UltraSPARC III Family currently contains processors similar to the UltraSPARC III Cu processor. All SPARC V9 ASIs will work for the UltraSPARC UltraSPARC III Family of processors.

8.2.3 UltraSPARC III Cu Specific ASIs

The UltraSPARC III Cu processor specific ASIs are also defined. ASI $4A_{16}$ is an example of how one ASI is defined differently for the Sun Fireplane interconnect in the UltraSPARC III Cu processor. All UltraSPARC UltraSPARC III Family ASIs will work with the UltraSPARC III Cu processor. TABLE 8-1 lists processor specific ASIs.

TABLE 8-1 Processor Specific ASIs

ASI Value	UltraSPARC III Cu
4A	Fireplane Interconnect CSRs
72	Memory Control Unit CSRs

8.3 ASI Groups

The ASI is evenly divided into restricted and unrestricted halves, defined by the SPARC V9. ASIs in the range 00_{16}–$7F_{16}$ are restricted. ASIs in the range 80_{16}–FF_{16} are unrestricted. An attempt to access a restricted ASI in non-privileged mode causes a *privileged_action* trap.

Normal or *translating* ASIs cause the CPU's VA to be translated to a physical one by the MMU. *Non-translating*, or *bypassing* ASIs, cause the CPU to not translate the VA; instead, the MMU passes the lower 43 bits of the CPU's virtual addresses as 43-bit physical addresses.

Access restrictions and translating abilities are summarized in TABLE 8-2.

ASIs can be grouped together to help understand the nature of the ASIs. The ASIs that map to a PA are targeted toward physical memory, memory mapped CSRs, and the processor's subsystem, depending on the VA and the ASI value.

The UltraSPARC III Cu processor implements the standard SPARC V9 ASIs and many processor specific ASIs for endian support and address CSR registers.

Processor Compatibility Note – In TABLE 8-2, text in **bold** means the ASI was not implemented in UltraSPARC I or UltraSPARC II. Text with ~~strike through~~ means the ASI was implemented in UltraSPARC I or UltraSPARC II but not in the UltraSPARC III Family.

TABLE 8-2 ASI Summary Table

ASI Values	Destination	Translating VA to PA	Special Operations	Architecture Definition
Restricted, Accessible in Privileged Mode Only				
04h, 0Ch	Physical Address	Translating		SPARC V9 and UltraSPARC III Family
10h, 11h				SPARC V9
14h, 15h		Bypassing		UltraSPARC III Family
18h, 19h		Translating		SPARC V9
1Ch, 1Dh		Bypassing		UltraSPARC III Family
24h, 2Ch		Translating		
30-34h, 38-3Ch, 40-44h, 45-49h	CSR	Non-translating		
4Ah	CSR (Bus i/f)			UltraSPARC III Cu
4B-4Eh, 50-5Fh, **60h**, 66h, 67h, 68h, ~~6Eh~~, 6Fh	CSR			UltraSPARC III Family
70h, 71h	Physical Address	Translating	Block Load/Store	
72h	CSR (MCU)	Non-translating		UltraSPARC III Cu
74-75h, 76h, 77h	CSR	Non-translating		
78h, 79h	Physical Address	Translating	Block Load/Store	
7Eh, 7Fh	CSR	Non-translating		
Non-Restricted, Accessible in Privileged or Non-privileged Mode				
80h, 81h, 82h, 83h, 88h, 89h, 8Ah, 8Bh	Physical Address	Translating		SPARC V9
C0-C5h, C8-CDh			Partial Store	UltraSPARC III Family
D0-D3h, D8-DBh			Short FP Load/Store	
E0h, E1h, F0h, F1h, F8h, F9h			Block Load/Store	

8.4 Instructions Associated with the ASIs

ASIs are used with load and store instructions. Their usage and restrictions are described in TABLE 8-4. Additional information is described in Section 8.5, “Using ASIs.”

8.4.1 Block Load and Block Store ASIs

Block load (BLD) and block store (BST) operations are generated by using the `LDDFA` and `STDFA` instructions with the 70_{16}, 71_{16}, 78_{16}, 79_{16}, $E0_{16}$, $E1_{16}$, $F0_{16}$, $F1_{16}$, $F8_{16}$, and $F9_{16}$ ASIs.

If the operand address is not 64-byte aligned, then a *mem_address_not_aligned* exception is generated.

If these ASIs are used with any other instruction, then a *data_access_exception* is generated and *mem_address_not_aligned* is not generated.

8.4.2 Partial Store ASIs

Partial store operations are generated by using the `STDFA` instruction with the $C0_{16}$–$C5_{16}$ and $C8_{16}$–CD_{16} ASIs.

If the operand address is not 8-byte aligned, then a *mem_address_not_aligned* exception is generated and if `i` = 1 in the instruction, then an *illegal_instruction* exception is generated instead.

If these ASIs are used with any other instruction, then a *data_access_exception* exception is generated and neither a *mem_address_not_aligned* nor a *illegal_instruction* (for `i` = 1) is generated.

8.4.3 Short Floating-Point Load and Store ASIs

Short floating-point load and store operations are generated by using the `LDDFA` and `STDFA` instructions with the $D0_{16}$–$D3_{16}$ and $D8_{16}$–DB_{16} ASIs to load and store byte and halfword values in the floating-point registers.

If the data is not aligned, then a *mem_address_not_aligned is generated.*

If these ASIs are used with any other instruction, a *data_access_exception* is generated and *mem_address_not_aligned* will not be generated.

8.4.3.1 Halfword Alignment

If the operand address for a $D2_{16}$, $D3_{16}$, DA_{16}, or DB_{16} ASI (halfword) is not halfword aligned, then a *mem_address_not_aligned* exception is generated.

8.5 Using ASIs

8.5.1 Data Widths

The ASIs for the UltraSPARC III Cu processor and the entire UltraSPARC UltraSPARC III Family of processors are accessible using 64-bit `LDXA`, `STXA`, `LDDFA`, and `STDFA` instructions, except where noted. SPARC V9 ASIs are accessible using aligned 8-, 16-, 32- and 64-bit load and store (read and write) instructions, except where noted. The UltraSPARC UltraSPARC III Family of processors and specific implementations require 64-bit aligned accesses, except where noted.

8.5.2 Operand Alignment

Addresses must align to the boundary of the data width. TABLE 8-3 shows the data size and address offset for operand alignment.

TABLE 8-3 Operand Alignment

Data Size	**Address Offset (binary)**
Halfword	xxxx.xxx0
Word	xxxx.xx00
Double word (8 bytes)	xxxx.x000
Quad word (16 bytes)	xxxx.0000
Block (64 bytes)	xx00.0000

8.5.3 Common Exceptions

Using ASIs improperly will generate one of the following exceptions in the CPU.

8.5.3.1 data_access_exception

When the wrong instruction is used with an ASI, a *data_access_exception* is generated. This is sometimes referred to as the invalid_ASI_exception.

8.5.3.2 mem_address_not_aligned

When the address operand does not align to the boundary of the data size, a *mem_address_not_aligned* exception is generated.

8.5.3.3 privileged_action

If a restricted, privileged mode only ASI is accessed in non-privileged mode, then a *privileged_mode_exception* is generated.

8.6 List of ASI Definitions

TABLE 8-4 lists all the ASIs in the UltraSPARC III Cu processor.

TABLE 8-4 ASI Definitions *(1 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
00-7F	**Restricted, Accessible in Privileged Mode only**					
00–03	—	—			Implementation-dependent, Unassigned	1
04	SPARC V9 and UltraSPARC III Family	`ASI_NUCLEUS (ASI_N)`	RW		Implicit address space, nucleus privilege, TL>0	
05–0B	—	—			Implementation-dependent, Unassigned	1
0C	SPARC V9 and UltraSPARC III Family	`ASI_NUCLEUS_LITTLE (ASI_NL)`	RW		Implicit address space, nucleus privilege, TL>0, little-endian	
0D–0F	—	—			Implementation-dependent, Unassigned	1
10	SPARC V9	`ASI_AS_IF_USER_PRIMARY (ASI_AIUP)`	RW		Primary address space, user privilege	2, 3

TABLE 8-4 ASI Definitions *(2 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
11	SPARC V9	ASI_AS_IF_USER_SECONDARY (ASI_AIUS)	RW		Secondary address space, user privilege	2, 3
12–13	—	—			Implementation-dependent, Unassigned	1
14	UltraSPARC III Family	ASI_PHYS_USE_EC	RW		Physical address external cacheable only	4, 5
15	UltraSPARC III Family	ASI_PHYS_BYPASS_EC_WITH_EBIT	RW		Physical address, non-cacheable, with side-effect	4
16–17	—	—			Implementation-dependent, Unassigned	1
18	SPARC V9	ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)	RW		Primary address space, user privilege, little-endian	3
19	SPARC V9	ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)	RW		Secondary address space, user privilege, little-endian	3
1A–1B	—	—			Implementation-dependent, Unassigned	1
1C	UltraSPARC III Family	ASI_PHYS_USE_EC_LITTLE (ASI_PHYS_USE_EC_L)	RW		Physical address, external cacheable only, little-endian	4, 5
1D	UltraSPARC III Family	ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE (ASI_PHYS_BYPASS_EC_WITH_EBIT_L)	RW		Physical address, non-cacheable, with side-effect, little-endian	4
1E–23	—	—			Implementation-dependent, Unassigned	1
24	UltraSPARC III Family	ASI_NUCLEUS_QUAD_LDD	R		Cacheable, 128-bit atomic LDDA	6, 7
25–2B	—	—			Implementation-dependent, Unassigned	1
2C	UltraSPARC III Family	ASI_NUCLEUS_QUAD_LDD_LITTLE (ASI_NUCLEUS_QUAD_LDD_L)	R		Cacheable, 128-bit atomic LDDA, little-endian	6, 7
2D–2F	—	—			Implementation-dependent, Unassigned	1
30	UltraSPARC III Cu	ASI_PCACHE_STATUS_DATA	RW		P-cache data status RAM diagnostic access	
31	UltraSPARC III Cu	ASI_PCACHE_DATA	RW		P-cache data RAM diagnostic access	

TABLE 8-4 ASI Definitions *(3 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
32	UltraSPARC III Cu	`ASI_PCACHE_TAG`	RW		P-cache tag RAM diagnostic access	
33	UltraSPARC III Cu	`ASI_PCACHE_SNOOP_TAG`	RW		P-cache snoop tag RAM diagnostic access	
34	UltraSPARC III Cu	`ASI_ATOMIC_QUAD_LDD_PHYS`				
35–37	—	—			Implementation-dependent, Unassigned	1
38	UltraSPARC III Cu	`ASI_WCACHE_VALID_BITS`			W-cache Valid Bits diagnostic access	
39	UltraSPARC III Cu	`ASI_WCACHE_DATA`	RW		W-cache data RAM diagnostic access	
3A	UltraSPARC III Cu	`ASI_WCACHE_TAG`	RW		W-cache tag RAM diagnostic access	
3B	UltraSPARC III Cu	`ASI_WCACHE_SNOOP_TAG`	RW		W-cache snoop tag RAM diagnostic access	
3C	UltraSPARC III Cu	`ASI_ATOMIC_QUAD_LDD_PHYS_L`				1
3D–3F	—	—			Implementation-dependent, Unassigned	1
40	UltraSPARC III Cu	`ASI_SRAM_FAST_INIT`	W		Interface to clean all major SRAM arrays on chip	9
41		*Reserved.*				
42	UltraSPARC III Cu	`ASI_DCACHE_INVALIDATE`	W		D-cache Invalidate diagnostic access	
43	UltraSPARC III Cu	`ASI_DCACHE_UTAG`	RW		D-cache uTag diagnostic access	
44	UltraSPARC III Cu	`ASI_DCACHE_SNOOP_TAG`	RW		D-cache snoop tag RAM diagnostic access	
45	UltraSPARC III Family	`ASI_DCU_CONTROL_REGISTER` (`ASI_DCUCR`)	RW	0	D-cache Unit Control Register	
46	UltraSPARC III Cu	`ASI_DCACHE_DATA`	RW		D-cache data RAM diagnostic access	
47	UltraSPARC III Cu	`ASI_DCACHE_TAG`	RW		D-cache tag/valid RAM diagnostic access	

TABLE 8-4 ASI Definitions *(4 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
48	UltraSPARC III Family	ASI_INTR_DISPATCH_STATUS (ASI_MONDO_SEND_CTRL)	R	0	Interrupt vector dispatch status	6
49	UltraSPARC III Family	ASI_INTR_RECEIVE (ASI_MONDO_RECEIVE_CTRL)	RW	0	Interrupt vector receive status	
4A	UltraSPARC III Cu	ASI_FIREPLANE_CONFIG_REG	RW	0	Fireplane interconnect configuration register for the UltraSPARC III processor	
		ASI_FIREPLANE_ADDRESS_REG	RW	08	Fireplane interconnect configuration register for the UltraSPARC III processor	
4B	UltraSPARC III Cu	ASI_ESTATE_ERROR_EN_REG	RW	0	E-state error enable register	
4C	UltraSPARC III Family	ASI_ASYNC_FAULT_STATUS (ASI_AFSR)	RW	0	Asynchronous fault status register	
4D	UltraSPARC III Family	ASI_ASYNC_FAULT_ADDR (ASI_AFAR)	R	0	Asynchronous fault address register	
4E	UltraSPARC III Cu	ASI_ECACHE_TAG (ASI_EC_TAG)	RW	<22:6>	L2-cache tag state RAM data diagnostic access	
		ASI_ECACHE_FLUSH	R	[31] =1		
4F	—	–			Implementation-dependent, Unassigned	1
50	UltraSPARC III Family	ASI_IMMU_TAG_TARGET	R	0	I-MMU tag target register	6
		ASI_IMMU_SFSR	RW	18	I-MMU sync fault status register	
		ASI_IMMU_TSB_BASE	RW	28	I-MMU TSB base register	
		ASI_IMMU_TAG_ACCESS	RW	30	I-MMU TLB tag access register	
		ASI_IMMU_TSB_PEXT_REG	RW	48	I-MMU TSB primary extension register	
		ASI_IMMU_TSB_SEXT_REG	R	50	I-MMU TSB secondary extension register (hardwired to zero)	
		ASI_IMMU_TSB_NEXT_REG	RW	58	I-MMU TSB nucleus extension register	
51	UltraSPARC III Family	ASI_IMMU_TSB_8KB_PTR_REG	R	0	I-MMU TSB 8 KB pointer register	6

TABLE 8-4 ASI Definitions *(5 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
52	UltraSPARC III Family	`ASI_IMMU_TSB_64KB_PTR_REG`	R	0	I-MMU TSB 64 KB pointer register	6
53	UltraSPARC III Family	`ASI_SERIAL_ID`	R		Internal testability[7] (also known as `ASI_DEVICE_ID+SERIAL_ID`)	6, 8
54	UltraSPARC III Family	`ASI_ITLB_DATA_IN_REG`	W	0	I-MMU TLB data in register	9
55	UltraSPARC III Family	`ASI_ITLB_DATA_ACCESS_REG`	RW	0 – 2.0FF8	I-MMU TLB data access register	
	UltraSPARC III Cu	`ASI_ITLB_CAM_ADDRESS_REG`	RW	4.0000 – 6.0FF8	I-MMU TLB CAM diagnostic register	
56	UltraSPARC III Family	`ASI_ITLB_TAG_READ_REG`	R	0 – 2.0FF8	I-MMU TLB tag read register; data access port to RAM array	6
57	UltraSPARC III Family	`ASI_IMMU_DEMAP`	W		I-MMU TLB demap operations	9

TABLE 8-4 ASI Definitions *(6 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
58	UltraSPARC III Family	`ASI_DMMU_TAG_TARGET_REG`	R	0	D-MMU TSB tag target register	
		`ASI_PRIMARY_CONTEXT_REG`	RW	8	I/D-MMU primary context register (this ASI register is shared between I-MMU and MMU)	
		`ASI_SECONDARY_CONTEXT_REG`	RW	10	D-MMU secondary context register	
		`ASI_DMMU_SFSR`	RW	18	D-MMU sync fault status register (D-SFSR)	
		`ASI_DMMU_SFAR`	R	20	D-MMU sync fault address register (D-SFAR)	
		`ASI_DMMU_TSB_BASE`	RW	28	D-MMU TSB base address register	
		`ASI_DMMU_TAG_ACCESS`	RW	30	D-MMU TLB tag access register	
		`ASI_DMMU_VA_WATCHPOINT_REG`	RW	38	D-MMU VA data watchpoint register	
		`ASI_DMMU_PA_WATCHPOINT_REG`	RW	40	D-MMU PA data watchpoint register	
		`ASI_DMMU_TSB_PEXT_REG`	RW	48	D-MMU TSB primary extension register	
		`ASI_DMMU_TSB_SEXT_REG`	RW	50	D-MMU TSB secondary extension register	
		`ASI_DMMU_TSB_NEXT_REG`	RW	58	D-MMU TSB nucleus extension register	
59	UltraSPARC III Family	`ASI_DMMU_TSB_8KB_PTR_REG`	R	0	D-MMU TSB 8 KB pointer register	6
5A	UltraSPARC III Family	`ASI_DMMU_TSB_64KB_PTR_REG`	R	0	D-MMU TSB 64 KB pointer register	6
5B	UltraSPARC III Family	`ASI_DMMU_TSB_DIRECT_PTR_REG`	R	0	D-MMU TSB direct pointer register	6
5C	UltraSPARC III Family	`ASI_DTLB_DATA_IN_REG`	W	0	D-MMU TLB data in register	9

TABLE 8-4 ASI Definitions *(7 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
5D	UltraSPARC III Family	`ASI_DTLB_DATA_ACCESS_REG`	RW	0 to 2.0F F8	D-MMU TLB data access register; Data access port to RAM array	
		`ASI_TLB_CAM_ACCESS_REG`	RW	4.000 0 to 6.0F F8	D-MMU TLB CAM diagnostic registers	
5E	UltraSPARC III Family	`ASI_DTLB_TAG_READ_REG`	R	<17:0 >	D-MMU TLB tag read register	6
5F	UltraSPARC III Family	`ASI_DMMU_DEMAP`	W		D-MMU TLB demap operations	9
60	UltraSPARC III Family	`ASI_IIU_INST_TRAP`	RW	0	Instruction breakpoint register	
61–65	—	—			Implementation-dependent, Unassigned	1
66	UltraSPARC III Cu	`ASI_ICACHE_INSTR (ASI_IC_INSTR)`	RW		I-cache data RAM diagnostics access	
67	UltraSPARC III Cu	`ASI_ICACHE_TAG (ASI_IC_TAG)`	RW		I-cache tag/valid RAM diagnostics access	
68	UltraSPARC III Cu	`ASI_ICACHE_SNOOP_TAG (ASI_IC_STAG)`	RW		I-cache snoop tag RAM diagnostics access	
69–6E	—	—			Implementation-dependent, Unassigned	1
6F	Future	(Reserved for `ASI_BRANCH_PREDICTION_ARRAY`)	RW		Planned for future processor	
70	UltraSPARC III Family	`ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)`	RW		Primary address space, block load/store, user privilege	3, 10
71	UltraSPARC III Family	`ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS)`	RW		Secondary address space, block load/store, user privilege	3, 10
72	UltraSPARC III Cu	ASI_MCU_TIMING[1:4]_REG	RW	0, 8, 10, 18	Memory Control Unit (MCU) programming registers; these registers are also memory mapped	
		ASI_MCU_ADR_DEC[1:4]_REG	RW	20, 28, 30, 38		
		ASI_MCU_ADR_CNTL_REG	RW	40		

TABLE 8-4 ASI Definitions *(8 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
73	—	—			Implementation-dependent, Unassigned	
74	UltraSPARC III Cu	ASI_ECACHE_DATA	RW		L2-cache data staging register	
75	UltraSPARC III Cu	ASI_ECACHE_CONTROL (ASI_EC_CTRL)	RW	0	L2-cache control register	
76	UltraSPARC III Cu	`ASI_ECACHE_W (ASI_EC_W)`	W		L2-cache data RAM diagnostic write access	
77	UltraSPARC III Family	`ASI_INTR_DATA0_W`	W	40	Outgoing interrupt vector data Register 0 H	9
		`ASI_INTR_DATA1_W`	W	48	Outgoing interrupt vector data Register 0 L	9
		`ASI_INTR_DATA2_W`	W	50	Outgoing interrupt vector data Register 1 H	9
		`ASI_INTR_DATA3_W`	W	58	Outgoing interrupt vector data Register 1 L	9
		`ASI_INTR_DATA4_W`	W	60	Outgoing interrupt vector data Register 2 H	9
		`ASI_INTR_DATA5_W`	W	68	Outgoing interrupt vector data Register 2 L	9
		`ASI_INTR_DISPATCH_W`	W	70	Interrupt vector dispatch	9
		`ASI_INTR_DATA6_W`	W	80	Outgoing interrupt vector data Register 3 H	9
		`ASI_INTR_DATA7_W`	W	88	Outgoing interrupt vector data Register 3 L	9
		`ASI_INTR_DISPATCH_W`	W		Interrupt vector dispatch: 01.0000.0070–8F.FFFF.C070	9
78	UltraSPARC III Family	`ASI_BLOCK_AS_IF_USER_PRIMARY _LITTLE (ASI_BLK_AIUPL)`	RW	0	Primary address space, block load/store, user privilege, little-endian	3, 10
79	UltraSPARC III Family	`ASI_BLOCK_AS_IF_USER_SECONDA RY_LITTLE (ASI_BLK_AIUSL)`	RW	0	Secondary address space, block load/store, user privilege, little-endian	3, 10
7A–7D	—	—			Implementation-dependent, Unassigned	1

TABLE 8-4 ASI Definitions *(9 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
7E	UltraSPARC III Cu	ASI_ECACHE_R (ASI_EC_R)	R	0	L2-cache data RAM diagnostic read access	
7F	UltraSPARC III Family	`ASI_INTR_DATA0_R`	R	40	Incoming interrupt vector data Register 0 H	6
		`ASI_INTR_DATA1_R`	R	48	Incoming interrupt vector data Register 0 L	6
		`ASI_INTR_DATA2_R`	R	50	Incoming interrupt vector data Register 1 H	6
		`ASI_INTR_DATA3_R`	R	58	Incoming interrupt vector data Register 1 L	6
		`ASI_INTR_DATA4_R`	R	60	Incoming interrupt vector data Register 2 H	6
		`ASI_INTR_DATA5_R`	R	68	Incoming interrupt vector data Register 2 L	6
		`ASI_INTR_DATA6_R`	R	80	Incoming interrupt vector data Register 3 H	6
		`ASI_INTR_DATA7_R`	R	88	Incoming interrupt vector data Register 3 L	6
80–FF	**Non-Restricted, Accessible in Privileged or Non-privileged Mode**					
80	SPARC V9	`ASI_PRIMARY (ASI_P)`	RW		Implicit primary address space	
81	SPARC V9	`ASI_SECONDARY (ASI_S)`	RW		Secondary address space	
82	SPARC V9	`ASI_PRIMARY_NO_FAULT (ASI_PNF)`	R		Primary address space, no fault	4, 6, 11
83	SPARC V9	`ASI_SECONDARY_NO_FAULT (ASI_SNF)`	R		Secondary address space, no fault	4, 6, 11
84–87	—	—			Implementation-dependent, Unassigned	1
88	SPARC V9	`ASI_PRIMARY_LITTLE (ASI_PL)`	RW		Implicit primary address space, little-endian	
89	SPARC V9	`ASI_SECONDARY_LITTLE (ASI_SL)`	RW		Secondary address space, little-endian	
8A	SPARC V9	`ASI_PRIMARY_NO_FAULT_LITTLE (ASI_PNFL)`	R		Primary address space, no fault, little-endian	4, 6, 11

TABLE 8-4 ASI Definitions *(10 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
8B	SPARC V9	`ASI_SECONDARY_NO_FAULT_ LITTLE (ASI_SNFL)`	R		Secondary address space, no fault, little-endian	4, 6, 11
8C–BF	—	—			Implementation-dependent, Unassigned	1
C0	UltraSPARC III Family	`ASI_PST8_PRIMARY (ASI_PST8_P)`	W		Primary address space, 8x8-bit partial store	12
C1	UltraSPARC III Family	`ASI_PST8_SECONDARY (ASI_PST8_S)`	W		Secondary address space, 8x8-bit partial store	12
C2	UltraSPARC III Family	`ASI_PST16_PRIMARY (ASI_PST16_P)`	W		Primary address space, 4x16-bit partial store	12
C3	UltraSPARC III Family	`ASI_PST16_SECONDARY (ASI_PST16_S)`	W		Secondary address space, 4x16-bit partial store	12
C4	UltraSPARC III Family	`ASI_PST32_PRIMARY (ASI_PST32_P)`	W		Primary address space, 2x32-bit partial store	12
C5	UltraSPARC III Family	`ASI_PST32_SECONDARY (ASI_PST32_S)`	W		Secondary address space, 2x32-bit partial store	12
C6–C7	—	—			Implementation-dependent, Unassigned	1
C8	UltraSPARC III Family	`ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL)`	W		Primary address space, 8x8-bit partial store, little-endian	12
C9	UltraSPARC III Family	`ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL)`	W		Secondary address space, 8x8-bit partial store, little-endian	12
CA	UltraSPARC III Family	`ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL)`	W		Primary address space, 4x16-bit partial store, little-endian	12
CB	UltraSPARC III Family	`ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL)`	W		Secondary address space, 4x16-bit partial store, little-endian	12
CC	UltraSPARC III Family	`ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL)`	W		Primary address space, 2x32-bit partial store, little-endian	12
CD	UltraSPARC III Family	`ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL)`	W		Second address space, 2x32-bit partial store, little-endian	12

TABLE 8-4 ASI Definitions *(11 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
CE–CF	—	—			Implementation-dependent, Unassigned	1
D0	UltraSPARC III Family	`ASI_FL8_PRIMARY (ASI_FL8_P)`	RW		Primary address space, one 8-bit floating-point load/ store	13
D1	UltraSPARC III Family	`ASI_FL8_SECONDARY (ASI_FL8_S)`	RW		Second address space, one 8-bit floating-point load/ store	13
D2	UltraSPARC III Family	`ASI_FL16_PRIMARY (ASI_FL16_P)`	RW		Primary address space, one 16-bit floating-point load/ store	13
D3	UltraSPARC III Family	`ASI_FL16_SECONDARY (ASI_FL16_S)`	RW		Second address space, one 16-bit floating-point load/store	13
D4–D7	—	—			Implementation-dependent, Unassigned	1
D8	UltraSPARC III Family	`ASI_FL8_PRIMARY_LITTLE (ASI_FL8_PL)`	RW		Primary address space, one 8-bit floating-point load/ store, little-endian	13
D9	UltraSPARC III Family	`ASI_FL8_SECONDARY_LITTLE (ASI_FL8_SL)`	RW		Second address space, one 8-bit floating-point load/store, little-endian	13
DA	UltraSPARC III Family	`ASI_FL16_PRIMARY_LITTLE (ASI_FL16_PL)`	RW		Primary address space, one 16-bit floating-point load/store, little-endian	13
DB	UltraSPARC III Family	`ASI_FL16_SECONDARY_LITTLE (ASI_FL16_SL)`	RW		Second address space, one 16-bit floating-point load/store, little-endian	13
DC–DF	—	—			Implementation-dependent, Unassigned	1
E0	UltraSPARC III Family	`ASI_BLOCK_COMMIT_PRIMARY (ASI_BLK_COMMIT_P)`	W		Primary address space, 8x8 - byte block store commit operation	9
E1	UltraSPARC III Family	`ASI_BLOCK_COMMIT_SECONDARY (ASI_BLK_COMMIT_S)`	W		Secondary address space, 8x8 - byte block store commit operation	9
E2–EE	—	—			Implementation-dependent, Unassigned	1
EF	Future	(Reserved for `ASI_BARRIER_SYNCH`)			Planned for future processor	

TABLE 8-4 ASI Definitions *(12 of 12)*

Value (hex)	Processor	ASI Name (Suggested Macro Syntax)	Type	VA (hex)	Description	See Foot notes
F0	UltraSPARC III Family	`ASI_BLOCK_PRIMARY` (`ASI_BLK_P`)	RW		Primary address space, 8x8 - byte block load/store	9
F1	UltraSPARC III Family	`ASI_BLOCK_SECONDARY` (`ASI_BLK_S`)	RW		Secondary address space, block load/store	9
F2–F7	—	—			Implementation-dependent, Unassigned	1
F8	UltraSPARC III Family	`ASI_BLOCK_PRIMARY_LITTLE` (`ASI_BLK_PL`)	RW		Primary address space, block load/store, little endian	9
F9	UltraSPARC III Family	`ASI_BLOCK_SECONDARY_LITTLE` (`ASI_BLK_SL`)	RW		Secondary address space, block load/store, little endian	9
FA–FF	—	—			Implementation-dependent, Unassigned	1

1. Implementation-dependent, unassigned ASIs may be used in future architectures and processors.
2. Use of these ASIs causes access checks to be performed as if the memory access instruction were issued while PSTATE.PRIV = 0 (that is, in nonprivileged mode) and directed towards the corresponding address space.
3. If the memory page being accessed is privileged, then a *data_access_exception* occurs.
4. Accessible as 8-bit, 16-bit, 32-bit, and 64-bit.
5. Use in `LDSTUBA`, `SWAPA`, and `CAS(X)A` instructions.
6. Use in load instructions only; others cause a *data_access_exception*.
7. Use in `LDDA` instruction only; others cause a *data_access_exception*.
8. Do not use; reserved for factory use (part number, laser programming).
9. Use in store instructions only; others cause a *data_access_exception*.
10. Use in `LDDFA` and `STDFA` instructions only; others cause a *data_access_exception*. Align to 64-byte boundary. See Section 8.4.1, "Block Load and Block Store ASIs" for more details.
11. ASI_PRIMARY_NOFAULT{_LITTLE} and ASI_SEONDARY_NOFAULT{_LITTLE} refer to the same address spaces as ASI_PRIMARY{_LITTLE} and ASI_SECONDARY{_LITTLE}, respectively.
12. Use in `STDFA` instruction only; others cause a *data_access_exception*. Align to 8-byte boundary. See Section 8.4.2, "Partial Store ASIs" for more details.
13. Use in `LDDFA` and `STDFA` instructions only; others cause a *data_access_exception*. Align to 8-bit or 16-bit boundary as required by Section 8.4.3, "Short Floating-Point Load and Store ASIs."

8.7 Special Memory Access ASIs

This section describes special memory access ASIs that are not specified in SPARC V9 and are not described in other sections.

8.7.1 ASI 0x14 (ASI_PHYS_USE_EC)

When this ASI is specified in any memory access instructions, hardware does the following:

- VA is passed to PA, and CONTEXT values are disregarded.
- Address masking is ignored (PSTATE.AM) and the VA is used. The I-MMU passes the lower 43 bits of the VA through to create a 43-bit PA.
- Memory access behaves as if its byte order is big-endian.

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

8.7.2 ASI 0x15 (ASI_PHYS_BYPASS_EC_WITH_EBIT)

Accesses with this ASI bypass the L2-cache and behave as if the side-effect bit (E bit) is set. When this ASI is specified in any memory access instructions, hardware does the following:

- VA is passed to PA, and CONTEXT values are disregarded.
- Address masking is ignored(PSTATE.AM) and the VA is used. The I-MMU passes the lower 43 bits of the VA through to create a 43-bit PA.
- Memory access behaves as if its byte order is big-endian.

8.7.3 ASI 0x1C (ASI_PHYS_USE_EC_LITTLE)

Accesses with this ASI are cacheable. This ASI is a little-endian version of ASI 14_{16}. When this ASI is specified in any memory access instructions, hardware does the following:

- VA is passed to PA, and CONTEXT values are disregarded.
- Address masking is ignored (PSTATE.AM) and the VA is used. The I-MMU passes the lower 43 bits of the VA through to create a 43-bit PA.
- Memory access behaves as if its byte order is little-endian.

8.7.4 ASI 0x1D (ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE)

Accesses with this ASI bypass the L2-cache and behave as if the side-effect bit (E bit) is set. This ASI is a little-endian version of ASI 15_{16}. When this ASI is specified in any memory access instructions, hardware does the following:

- VA is passed to PA, and CONTEXT values are disregarded.
- Address masking is ignored (PSTATE.AM) and the VA is used. The I-MMU passes the lower 43 bits of the VA through to create a 43-bit PA.

- Memory access behaves as if its byte order is little-endian.

8.7.5 ASIs 0x24 and 0x2C (Load Quadword ASIs)

ASIs 24_{16} (`ASI_NUCLEUS_QUAD_LDD`) and $2C_{16}$ (`ASI_NUCLEUS_QUAD_LDD_LITTLE`) exist for use with the `LDDA` instruction as Load Quadword operations.

When these ASIs are used with `LDDA` for Load Quadword, a *mem_address_not_aligned* exception is generated if the operand address is not 16-byte aligned.

If these ASIs are used with any other Load Alternate instruction or any Store Alternate instruction, a *data_access_exception* is always generated and *mem_address_not_aligned* is not generated.

SECTION IV

Memory and Cache

CHAPTER **9**

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)*. UltraSPARC III Cu processors 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 III Cu 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 III Cu processor in sections organized as follows:

- UltraSPARC III Cu 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
- UltraSPARC III Cu Internal ASIs
- Store Compression
- Read-After-Write (RAW) Bypassing

9.1 UltraSPARC III Cu TSO Behavior

The UltraSPARC III Cu 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 III Cu 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 each other.
- An L2-cache or write 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.

9.2 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. See Chapter 8 “Address Space Identifiers” for details on ASIs.

9.3 Memory Accesses and Cacheability

Memory is logically divided into real memory (cached) and I/O memory (noncached with and without side-effects) spaces. Real memory 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.

9.3.1 Coherence Domains

Two types of memory operations are supported in the UltraSPARC III Cu processor: 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 III Cu processor maintains TSO ordering between memory references regardless of their cacheability.

9.3.1.1 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.

9.3.1.2 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 9.11 "Store Compression" for details.

Note – Side-effect, as indicated in `TTE.E`, does not imply noncacheability.

9.3.2 Global Visibility

A memory access is considered globally visible when one of the following events occurs:

- Read or write permission is granted for a cacheable transaction in scalable shared memory (SSM) mode.
- The transaction request is issued for a non-cacheable transaction in SSM mode.
- The transaction request is issued when not in SSM mode.

9.3.3 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 9-1 illustrates the issues involved in mixing cacheable and non-cacheable accesses.

CODE EXAMPLE 9-1 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
2   MEMBAR #LoadLoad, #LoadStore (needed in RMO for SPARC-V9
compliance)
    Load D1
    Store D2
```

CODE EXAMPLE 9-1 Memory Ordering and MEMBAR Examples *(Continued)*

```
1   MEMBAR #StoreStore (needed in PSO, RMO for SPARC-V9 compliance)
    Store F1:clear flag
}
```

9.4 Memory Synchronization

Processors always see their own normal loads and stores 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 III Cu 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 III Cu 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 9-1 lists the MEMBAR implementations.

TABLE 9-1 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.

9.4.1 MEMBAR #Sync

`Membar #Sync` forces all outstanding instructions and all deferred errors to be completed before any instructions after the `MEMBAR` are issued.

9.4.2 MEMBAR Rules

TABLE 9-2 and TABLE 9-3 summarize the cases where the programmer must insert a `MEMBAR` to ensure *ordering* between two memory operations on the UltraSPARC III Cu processor. Use TABLE 9-2 and TABLE 9-3 for ordering purposes only. Be sure not to confuse memory operation ordering with processor consistency or deterministic operation; `MEMBAR`s are required for deterministic operation of certain ASI register updates.

Caution – The `MEMBAR` requirements for the UltraSPARC III Cu processor are weaker 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 9-2 reflect the UltraSPARC III Cu processor's hardware implementation.

TABLE 9-2 `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

TABLE 9-2 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
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 9-3 reflect the UltraSPARC III Cu processor's hardware implementation.

TABLE 9-3 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	#	#	#	#

TABLE 9-3 `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**
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	#

9.4.3 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 that all instruction fetch and instruction buffering operations are also blocked.

9.5 Atomic Operations

The SPARC V9 architecture 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 ones (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 9-4. 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 9-4 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.

9.6 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.

9.7 Prefetch Instructions

The UltraSPARC III Cu 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 9-5 describes different types of software prefetch instructions.

TABLE 9-5 Types of Software Prefetch Instructions

fcn Value (hex)	**Instruction Type**	**Prefetch (64 bytes of data) into:**	**Instruction Strength** **UltraSPARC III Cu**	**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.

9.8 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 MEMBARs. Section A.4 "Block Load and Block Store (VIS I)" on page A-460 discusses block loads and stores in detail.

9.9 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 III Cu 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).
- A `MEMBAR` may be needed between side-effect and non-side-effect accesses while in PSO and RMO modes, for portability across SPARC V9 processors, as well as in some cases of TSO.

9.9.1 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, so 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.

9.9.2 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 uncorrectable error checking. Exiting RED_state with DONE, RETRY, or with the destination of the JMPL non-cacheable will avoid the problem.

9.10 UltraSPARC III Cu 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 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.

9.11 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 write 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 would overwrite a previously written entry (a valid bit is kept for each of the 16 bytes).

Caution – This behavior is unique to the UltraSPARC III Cu processor and differs from previous UltraSPARC 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`

9.12 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

9.12.1 RAW Bypassing Algorithm

The algorithm used in the UltraSPARC III Cu 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
```

9.12.2 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 **10**

Caches and Cache Coherency

This chapter describes the use of caches, and contains these sections:

- Cache Organization
- Cache Flushing
- Bypassing the D-Cache
- Controlling P-cache
- Coherence Tables

10.1 Cache Organization

In this section we describe two cache organizations: virtual indexed, physical tagged caches and physical indexed, physical tagged caches.

10.1.1 Virtual Indexed, Physical Tagged Caches (VIPT)

The Data Cache (D-cache) is virtual-indexed, physical-tagged (VIPT). Virtual addresses are used to index into the cache tag and data arrays while accessing the D-MMU, (i.e. D-TLBs). The resulting tag is compared against the translated physical address to determine cache hit.

A side-effect inherent in a virtual-indexed cache is *address aliasing*. This issue is addressed in Section 10.2.1 “Address Aliasing Flushing” on page 10-230.

10.1.1.1 Data Cache (D-Cache)

The Data Cache (D-cache) is a write-through, non-allocating on write miss, 64 KB, four-way associative cache with a 32-byte line.

Data accesses bypass the D-cache when the D-cache enable bit in the DCU Control Register is clear. If the `DM` bit in the DCU Control Register is clear, the cacheability is determined by the `CP` and `CV` bits. If the access is mapped by the D-MMU as non-virtual-cacheable, then load misses will not allocate in the D-cache.

Note – A non-virtual-cacheable access may access data in the D-cache from an earlier cacheable access to the same physical block, unless the D-cache is disabled. Software must flush the D-cache when changing a physical page from cacheable to non-cacheable (see 10.2, "Cache Flushing").

10.1.2 Physical Indexed, Physical Tagged Caches (PIPT)

10.1.2.1 Instruction Cache (I-Cache)

The Instruction Cache (I-cache) is a 32KB pseudo four-way set-associative, write-invalidate cache with 32-byte lines. Instruction fetches bypass the I-cache when any of the following occur:

- I-cache enable or I-MMU enable bits in the DCU Control Register are clear
- CP bit in the DCU Control Register is clear
- Processor is in RED mode, or
- Fetch is mapped by the I-MMU as not physically cacheable.

The I-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 I-cache, but do not flush instructions that have already been prefetched into the pipeline. A `FLUSH` instruction should be used to flush the pipeline in the case of self modifying code.

If a program changes I-cache mode to I-cache-ON from I-cache-OFF, then the next instruction fetch 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 (see CODE EXAMPLE 10-1).

CODE EXAMPLE 10-1 I-Cache Mode 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.
```

10.1.2.2 Prefetch Cache (P-Cache)

The Prefetch Cache (P-cache) is a write-invalidate, 2KB, 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 is filled by the following:

- Floating Point (FP) Load Miss of D-cache and P-cache
- Hardware or Software Prefetch

The `PREFETCH fcn`=16 instruction can be used to invalidate, or flush a P-cache entry.

The cache line size is 64 bytes with 32-byte sub-blocks. Prefetches always generate 64-byte fills. An FP Load Miss generates 32-byte fills. The P-cache is globally invalidated on context changes and MMU updates. Individual lines are invalidated on store hits. There is a diagnostic interface (ASI_PCACHE_TAG) that allows privileged software to invalidate entries in P-cache.

The P-cache is globally invalidated if any of the following conditions occur:

- The context registers are written.
- There is a demap operation in the DMU.
- The DMU 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 an autonomous hardware prefetch from the L2-cache. This cache never needs to be flushed (not even for address aliases).

Note – From a load's perspective, the P-cache is a virtually indexed, virtually tagged (VIVT) cache.

10.1.2.3 Second Level and Write Caches (L2-Cache, W-Cache)

The level-2 caches — the L2-cache and the W-cache — are physical indexed, physical 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.

Instruction fetches bypass the L2-cache in the following cases:

- The I-MMU is disabled *and* the `CP` bit in the DCU Control Register is not set.
- The processor is in `RED_state`.
- The access is mapped by the I-MMU as not physically 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 not physically cacheable (unless `ASI_PHYS_USE_EC` is used).

The system must provide a non-cacheable scratch memory for use by the booting code until the MMUs are enabled.

The L2-cache is a unified, write-back, allocating, one- or two-way set-associative cache. Its size ranges from 1 MB to 8 MB. Its line size varies from 64 bytes to 512 bytes with 64-byte sub-blocks. Fills are done at a sub-block/64-byte granularity. Allocations and evictions are done at a line granularity. See TABLE 10-1 for description on sub-block and line sizes. L2-cache uses a pseudo-random replacement policy. L2-cache cannot be disabled by software.

TABLE 10-1 External Cache Organizations

External Cache Size	Line Size	64-Byte Sublines per line
1 MB	64 bytes	1
4 MB	256 bytes	4
8 MB	512 bytes	8

Block loads and block stores, which load or store a 64-byte block of data between memory and the Floating Point Register file, do not allocate into the L2-cache, to avoid pollution. Block store commits invalidate the L2-cache. Prefetch Read Once instructions, which load a 64-byte block of data into the P-cache, do not allocate into the L2-cache. All other level-1 cache fills also allocate in L2-cache.

The W-cache is a 2-KB, 4-way associative cache, 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. The W-cache only contains those bytes of data written by stores and a per byte dirty (valid) bit is maintained. All cacheable stores allocate a 64-byte line in the W-cache. W-cache may contain newer data than in L2-cache. All data accesses to L2-cache must access the W-cache in parallel and merge at a byte granularity if the W-cache has newer data. On eviction, the W-cache writes back to the L2-cache.

10.1.2.4 L2-cache Replacement Policy

CODE EXAMPLE 10-2 reflects the cache replacement algorithm when all 4 ways of the L2-cache are active.

CODE EXAMPLE 10-2 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
```

10.2 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 10.2.1, "Address Aliasing Flushing"). 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 displacement flush (using ASI_ECACHE_TAG) or a store with `ASI_BLK_COMMIT`. Flushing the L2-cache will flush the corresponding blocks from W-cache. (See 10.2.2, "Committing Block Store Flushing").
- L2-cache, D-cache, P-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, 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 (2 places), one VA index bit (2 places), and the PA can simultaneously exist in all 4 ways of a set (4 places).

10.2.1 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 (specifically bit 13 of the address) 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 16KB. This size may increase in future designs.

When the alias boundary is violated, software must flush the D-cache if the page was virtual 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 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, since the D-cache is write-through.

10.2.2 Committing Block Store Flushing

Block store commit does a 64-byte store from Floating Point Registers directly to the memory and invalidates any caches in the system. Block store commit exists outside *Total Store Order (TSO).*

For example, 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 Write Cache must be written back to the stable storage.

Two ASIs (`ASI_BLK_COMMIT_PRIMARY` and `ASI_BLK_COMMIT_SECONDARY`) perform these writebacks efficiently when software can ensure exclusive write access to the block being flushed. These ASIs writeback 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.

10.2.3 Displacement Flushing

Cache flushing can also be accomplished by a displacement flush. This procedure reads a range of addresses that map to the corresponding cache line being flushed, forcing out modified entries in the local cache. Ensure that the range of read-only addresses is mapped in the MMU before starting a displacement flush; otherwise, the TLB miss handler may put new data into the caches.

Note – Diagnostic ASI accesses to the L2-cache can be used to invalidate a line, but they are not an alternative to displacement flushing. Modified data in the L2-cache will *not* be written back to memory when these ASI accesses are used.

10.3 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-cache only
- CP == 0, CV == 0: Non-cacheable

Only two indexes in the D-cache need be flushed for each 32-byte aligned physical address:

- {VA[13] == 0, PA[12:5]}, and
- {VA[13] == 1, PA[12:5]}

Note – The behavior of a load or store instruction using ASI 0x14 (ASI_PHYS_USE_EC) that hits on the D-cache is not defined.

10.4 Controlling P-cache

This section clarifies the use of DCR.PE, DCR.HPE, and DCR.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 trigger hardware prefetch.

Non-cacheable address space never installs in P-cache or L2-cache.

- PE = 0, HPE = x, SPE = x
  - No Hardware prefetches, no Software prefetches.
  - No FP load miss data (32 bytes) installed from L2-cache to P-cache.
  - All P-cache references treated as miss.
- PE = 1, HPE = 0, SPE = 0
  - No Hardware prefetch can work.
  - None of the software prefetches work.
  - No FP load miss data (32 bytes) installed in P-cache.
  - FP loads will be checked for P-cache hit/miss.

- PE = 1, HPE = 1, SPE = 0
  - Hardware prefetches can work
  - None of the software prefetches can work.
  - FP load miss data (32 bytes) can be installed in P-cache.
  - P-cache references (FP load) will be checked for hit/miss
- PE=1, HPE=0, SPE=1
  - No Hardware prefetches.
  - All Software prefetches can work.
  - No FP load miss data (32 bytes) installed in P-cache.
  - FP loads will be checked for P-cache hit/miss.
- PE = 1, HPE = 1, SPE = 1
  - Hardware prefetches can work.
  - All Software prefetches can work.
  - FP load miss data (32 bytes) will be installed in P-cache.
  - FP loads will checked for P-cache hit/miss.

Note – The configuration PE = 1, HPE = 1, and SPE=1 is not supported in the UltraSPARC III Cu processor. The use of this mode may cause data in the P-cache to become incoherent with the system.

10.5 Coherence Tables

The set of tables in this section describes the cache coherence protocol that governs the behavior of the processor on the interface bus.

10.5.1 Processor State Transition and the Generated Transaction

Tables in this section summarize the following:

- Hit/Miss, State Change, and Transaction Generated for Processor Action (TABLE 10-2)
- Combined Tag/MTag States (TABLE 10-3)
- Derivation of DTags, CTags, and MTags from Combined Tags (TABLE 10-4)

TABLE 10-2 Hit/Miss, State Change, and Transaction Generated for Processor Action

Combined State	MODE	Processor action					
		Load	**Store/ Swap**	**Block Load**	**Block Store**	**Block Store Commit**	**Write Prefetch**
I	~SSM	Miss: RTS	Miss: RTO	Miss : RS	Miss: WS	Miss: WS	Miss: RTO
	SSM & LPA	Miss: RTS	Miss: RTO	Miss: RS	Miss: R_WS	Miss: R_WS	Miss: RTO
	SSM & LPA & retry	Mtag miss: R_RTS	Mtag miss: R_RTO	Mtag miss: R_RS	Invalid	Invalid	MTag Miss: R_RTO
	SSM & ~LPA	Miss: R_RTS	Miss: R_RTO	Miss: R_RS	Miss: R_WS	Miss: R_WS	Miss: R_RTO
E	~SSM	Hit	Hit: E→M	Hit	Hit: E→M	Hiss: WS	Hit
	SSM & LPA	Hit	Hit: E→M	Hit	Hit: E→M	Miss: R_WS	Hit
	SSM & LPA & retry	Invalid					
	SSM & ~LPA	Hit	Hit: E→M	Hit	Hit: E→M	Miss: R_WS	Hit
S	~SSM	Hit	Miss: RTO	Hit	Miss: WS	Miss: WS	Hit
	SSM & LPA	Hit	MTag miss: RTO	Hit	Miss: R_WS	Miss: R_WS	Hit
	SSM & LPA & retry	Invalid	MTag Miss: R_RTO	Invalid	Invalid	Invalid	Invalid
	SSM & ~LPA	Hit	MTag Miss: R_RTO	Hit	Miss: R_WS	Miss: R_WS	Hit

TABLE 10-2 Hit/Miss, State Change, and Transaction Generated for Processor Action

Combined State	MODE	Processor action					
		Load	Store/ Swap	Block Load	Block Store	Block Store Commit	Write Prefetch
O	~SSM	Hit	Miss: RTO	Hit	Miss: WS	Miss: WS	Hit
	SSM & LPA	Hit	MTag Miss: RTO	Hit	Miss: R_WS	Miss: R_WS	Hit
	SSM & LPA & retry	Invalid	MTag Miss: R_RTO	Invalid	Invalid	Invalid	Invalid
	SSM & ~LPA	Hit	MTag Miss: R_RTO	Hit	Miss: R_WS	Miss: R_WS	Hit
Os (Legal only in SSM mode)	~SSM	Invalid					
	SSM & LPA	Hit	MTag Miss: R_RTO	Hit	Miss: R_WS	Miss: R_WS	Hit
	SSM & LPA & retry	Invalid	MTag Miss: R_RTO	Invalid	Invalid	Invalid	Invalid
	SSM & ~LPA	Hit	MTag Miss: R_RTO	Hit	Miss: R_WS	Miss: R_WS	Hit
M	~SSM	Hit	Hit	Hit	Hit	Miss: WS	Hit
	SSM & LPA	Hit	Hit	Hit	Hit	Miss: R_WS	Hit
	SSM & LPA & retry	Invalid					
	SSM & ~LPA	Hit	Hit	Hit	Hit	Miss: R_WS	Hit

TABLE 10-3 Combined Tag/MTag States

MTag State: CTag State	gI	gS	gM
cM	I	Os	M
cO	I	Os	O
cE	I	S	E
cS	I	S	S
cI	I	I	I

TABLE 10-4 Deriving DTags, CTags, and MTags from Combined Tags

Combined Tags (CCTags)	DTag	CTag	MTag
I	dI	cI	gI
E	dS	cE	gM
S	dS	cS	gS
O	dO	cO	gM
Os	dO	cO	gS
M	dO	cM	gM

10.5.2 Snoop Output and Input

TABLE 10-5 summarizes snoop output and DTag transition; TABLE 10-6 summarizes snoop input and CIQ operation queueing.

TABLE 10-5 Snoop Output and DTag Transition *(1 of 4)*

Snooped Request	DTag State	Shared Output	Owned Output	Error Output	Next DTag State	Action for Snoop Pipeline
own RTS (for data)	dI	0	0	0	dT	Own RTS wait data
	dS	1	0	1	dS	Error
	dO	1	0	1	dO	Error
	dT	1	0	1	dT	Error
own RTS (for instructions)	dI	0	0	0	dS	Own RTS instruction wait data
	dS	1	0	1	dS	Error
	dO	1	0	1	dO	Error
	dT	1	0	1	dT	Error

TABLE 10-5 Snoop Output and DTag Transition *(2 of 4)*

Snooped Request	DTag State	Shared Output	Owned Output	Error Output	Next DTag State	Action for Snoop Pipeline
foreign RTS	dI	0	0		dI	None
	dS	1	0		dS	None
	dO	1	1		dO	Foreign RTS copyback
	dT	1	0		dS	None
own RTO	dI	0	0		dO	Own RTO wait data
	dS & ~SSM	1	1		dO	Own RTO no data
	dS & SSM	0	0		dO	Own RTO wait data
	dO	1	1		dO	Own RTO no data
	dT	1	1	1	dO	Error
foreign RTO	dI	0	0		dI	None
	dS	0	0		dI	Foreign RTO invalidate
	dO	0	1		dI	Foreign RTO copyback-invalidate
	dT	0	0		dI	Foreign RTO invalidate
own RS	dI	0	0		dI	Own RS wait data
	dS	0	0	1	dS	Error
	dO	0	0	1	dO	Error
	dT	0	0	1	dT	Error
foreign RS	dI	0	0		dI	None
	dS	0	0		dS	None
	dO	0	1		dO	Foreign RS copyback-discard
	dT	0	0		dT	None
own WB	dI	0	1		dI	Own WB (cancel)
	dS	0	1		dI	Own WB (cancel)
	dO	0	0		dI	Own WB
	dT	0	1	1	dI	Error
foreign WB	dI	0	0	0	dI	None
	dS	0	0	0	dS	None
	dO	0	0	0	dO	None
	dT	0	0	0	dT	None

TABLE 10-5 Snoop Output and DTag Transition *(3 of 4)*

Snooped Request	DTag State	Shared Output	Owned Output	Error Output	Next DTag State	Action for Snoop Pipeline
foreign RTSM	dI	0	0		dI	None
	dS	0	0		dS	Foreign RTSM
	dO	0	1		dS	fRTSM copyback (if cacheline is not in the W-cache)
	dO	0	1		dI	fRTSM copyback (if cacheline is in the W-cache)
	dT	0	0		dS	Foreign RTSM
own RTSR (issued by SSM device)	dI	0	0		dO	Own RTSR wait data
	dS	0	0	0	dO	Own RTSR wait data
	dO	0	0	1	dO	Own RTSR wait data, Error
	dT	0	0	1	dT	Own RTSR wait data, Error
foreign RTSR	dI	0	0		dI	None
	dS	1	0		dS	None
	dO	1	1		dS	Foreign RTSR
	dT	1	0		dS	None
own RTOR (issued by SSM device)	dI	0	0		dO	Own RTOR wait data
	dS	0	0		dO	Own RTOR wait data
	dO	0	0		dO	Own RTOR wait data
	dT	0	0	1	dO	Error
foreign RTOR	dI	0	0		dI	None
	dS	0	0		dI	Foreign RTOR invalidate
	dO	0	0		dI	Foreign RTOR invalidate
	dT	0	0		dI	Foreign RTOR invalidate
own RSR	dI	0	0		dI	Own RSR wait data
	dS	0	0	1	dS	Error
	dO	0	0	1	dO	Error
	dT	0	0	1	dT	Error
foreign RSR	dI	0	0		dI	None
	dS	0	0		dS	None
	dO	0	0		dO	None
	dT	0	0		dT	None

TABLE 10-5 Snoop Output and DTag Transition *(4 of 4)*

Snooped Request	DTag State	Shared Output	Owned Output	Error Output	Next DTag State	Action for Snoop Pipeline
own WS	dI	0	0		dI	Own WS
	dS	0	0		dI	Own invalidate WS
	dO	0	0		dI	Own invalidate WS
	dT	0	0		dI	Own invalidate WS
foreign WS	dI	0	0		dI	None
	dS	0	0		dI	Invalidate
	dO	0	0		dI	Invalidate

TABLE 10-6 Snoop Input and CIQ Operation Queued

Action from Snoop Pipeline	Shared Input	Owned Input	Error (out)	Operation Queued in CIQ
own RTS wait data	1	X		RTS Shared
	0	0		RTS ~Shared
	0	1	1	RTS Shared, Error
own RTS inst wait data	X	X		RTS Shared
foreign RTS copyback	X	1		Copyback
	X	0	1	Copyback, Error
own RTO no data	1	X		RTO nodata
	0	X	1	RTO nodata, error
own RTO wait data	1	X	1	RTO data, error
	0	X		RTO data
foreign RTO invalidate	X	X		Invalidate
foreign RTO copyback-invalidate	X	0	1	Copyback-invalidate, Error
	0	1		Copyback-invalidate
	1	1		Invalidate
own RS wait data	X	X		RS data
foreign RS copyback-discard	X	0	1	Error
	X	1		Copyback-discard
foreign RTSM copyback	X	0	1	RTSM copyback, Error
	X	1		RTSM copyback
own RTSR wait data	1	X		RTSR shared

TABLE 10-6 Snoop Input and CIQ Operation Queued *(Continued)*

Action from Snoop Pipeline	Shared Input	Owned Input	Error (out)	Operation Queued in CIQ
	0	X		RTSR~shared
own RTOR wait data	X	X		RTOR data
foreign RTOR invalidate	X	X		Invalidate
own RSR	X	X		RS data
own WS	X	X		Own WS
own WB	X	X		Own WB
own invalidate WS	X	X		Own invalidate WS
invalidate	X	X		Invalidate

10.5.3 Transaction Handling

Tables in this section summarize handling of the following:

- Transactions at the head of CIQ (TABLE 10-7)
- No snoop transactions (TABLE 10-8)
- Transactions internal to the UltraSPARC III Cu processor (TABLE 10-9)

TABLE 10-7 Transaction Handling at Head of CIQ *(1 of 3)*

Operation at Head of CIQ	CCTag	MTag (in/out)	Error	Retry	Next CCTag
RTS Shared	I	gM (in)			S
		gS (in)			S
		gI (in)		1	I
	M,O, E,S,Os	X	1		
RTS ~ Shared	I	gM (in)			E
		gS (in)			S
		gI (in)		1	I
	M,O, E,S,Os	X (in)	1		
RTSR Shared	I	gM (in)			O
		gS (in)			Os
		gI (in)	1	1	I
	M,O, E,S,Os	X	1		
RTSR ~ Shared	I	gM (in)			M
		gS (in)			Os

TABLE 10-7 Transaction Handling at Head of CIQ *(2 of 3)*

Operation at Head of CIQ	CCTag	MTag (in/out)	Error	Retry	Next CCTag
		gI (in)	1	1	I
	M, O, E, S, Os	X (in)	1		
RTO nodata	I, M, E, Os	None	1		
	O, S	None			M
RTO data and SSM	M, E, O, Os, S	X (in)	1		
	I	gM (in)			M
		gS (in)		1	Os
		gI (in)		1	I
RTO data and SSM	M, E, Os, O	X(in)	1		
	I, S	gM(in)			M
		gS(in)		1	Os
		gI(in)		1	I
RTOR data	M, E	X (in)	1		
	O	gM (in)	1		O
	S, Os, I	gM (in)			M
	S, O, Os, I	gS (in)	1	1	Os
		gI (in)	1	1	I
Foreign RTSR	I	None			I
	M, O	None	1		No change
	E, Os, S	None			S
Foreign RTSM	I	X (in)	1		
	M, O, Os	None	1		S
	E, S	none			S
RTSM copyback	M, O	gM (out)			S (if cacheline is *not* in the W-cache)
	M, O	gM (out)			I (if cacheline is in the W-cache)
	Os	gS (out)			S
	E, S, I		1		
Copyback	M, O	gM (out)			O
	Os	gS (out)			Os
	I	gI (out)			I
	E, S	gM (out)	1		S
Invalidate	X				I
Copyback-invalidate	M, O	gM (out)			I

TABLE 10-7 Transaction Handling at Head of CIQ *(3 of 3)*

Operation at Head of CIQ	CCTag	MTag (in/out)	Error	Retry	Next CCTag
	Os	gS (out)			I
	I	gI (out)			I
	E, S	gM (out)	1		I
Copyback-discard	M, O	gM (out)			No change
	Os	gS (out)			Os
	I	gI (out)			I
	E, S	gM (out)	1		No change

RS data	X
	X
	X
		gM(in)
		gS(in)
		gI(in)
				
				1
					No change
					No change
					No change

Own WS	X	gM(out)			I
Own WB	M,O	gM (out)			I
	Os	gS (out)			I
	I	gI (out)			I
	S, E	gM (out)	1		I
Own invalidate WS	X	gM(out)			I

TABLE 10-8 No Snoop Transaction Handling

Combined State	Mode	Processor Action					
		Load	Store/ Swap	Block Load	Block Store	Block Store Commit	Dirty Victim
I	No snoop	Miss: `RTS_ns`	Miss: `RTO_ns`	Miss: `RS_ns`	Miss: WS	Miss: WS	None
S	No snoop	Error					
E	No snoop	Hit	Hit E→M	Hit	Hit E→M	Miss: WS	None
M	No snoop	Hit	Hit	Hit	Hit	Miss: WS	WB
O	No snoop	Error					
Os	No snoop	Error					

RTS_ns, RTO_ns, and RS_ns are transactions internal to the UltraSPARC III Cu processor and are not visible on the Sun™ Fireplane interconnect.

TABLE 10-9 Internal Transaction Handling

Operation at Head of CIQ	CCTag	MTag (in/out)	Error	Next CCTag
RTS_ns	I	gM (in)	1	E
		gS (in)	1	S
		gI (in)	1	I
	S, E, M, O, Os			
RTO_ns	I	gM (in)	1	M
		gS (in)	1	O
		gI (in)	1	I
	S, E, M, O, Os			
RS_ns	I	gM (in)	1	I
		gS (in)	1	I
		gI (in)	1	I
	S, E, M, O, Os			
WS	X	gM (out)		I
		gS, gI (out)	1	I

CHAPTER **11**

Memory Management Unit

The Memory Management Unit (MMU) conforms to the requirements set forth in the *SPARC V9 Architecture Manual*. In particular, it supports a 64-bit virtual address space, software TLB miss processing only (no hardware page table walk), simplified protection encoding, and multiple page sizes.

This chapter describes the MMU, as seen by the operating system software, in these sections:

11.1 Virtual Address Translation

The MMUs support four page sizes: 8 KB, 64 KB, 512 KB, and 4 MB. Separate Instruction and Data MMUs (I-MMU and D-MMU, respectively) are provided to enable concurrent virtual-to-physical address translations for instruction and data. A 64-bit virtual address (VA) space is supported, with 43 bits of physical address (PA). In each translation, the virtual page number is replaced by a physical page number, which is concatenated with the page offset to form the full physical address, as illustrated in FIGURE 11-1.

Each data and I-MMU consists of multiple Translation Lookaside Buffers (TLBs) that can be accessed in parallel.

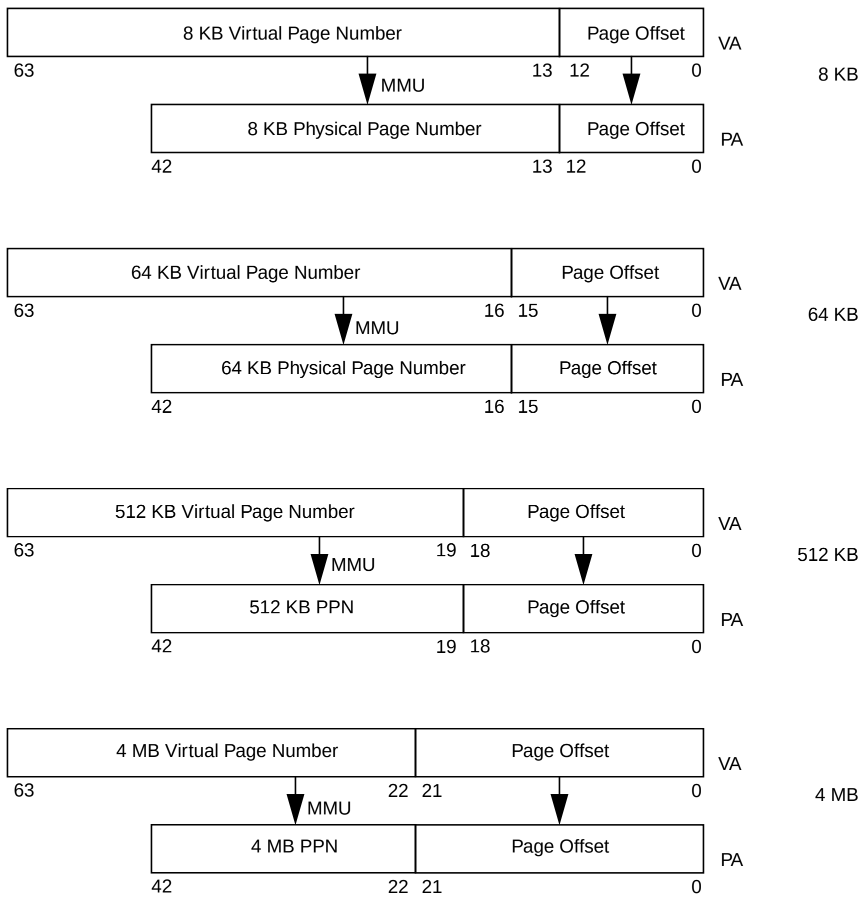


FIGURE 11-1 Virtual-to-Physical Address Translation for All Four Page Sizes

The operating system maintains translation information in an arbitrary data structure, called the *software translation table* in this chapter. The I-MMU and D-MMU TLBs act as independent, software managed caches of the software translation table, providing appropriate concurrency for virtual-to-physical address translation.

In the D-MMU, there are two TLBs each of which have 512-entry two-way associative which can be programmed to hold unlocked 8 KB, 64 KB, 512 KB, and 4 MB pages for translations. Each TLB can hold any of the 4 page sizes, but are programmed to only one page size at any given time. Each TLB can be programmed to the either same or different page sizes. Also, a 16-entry fully associative TLB is used for 8 KB, 64 KB, 512 KB and

4 MB locked and unlocked pages. Specifically, it supports locked pages of the size supported in 512-entry, 2-way TLBs and unlocked and locked pages of the other page sizes. This TLB can hold any combination of page sizes that are locked/unlocked at a time.

In the I-MMU, a 128-entry, 2-way associative TLB is used exclusively for 8 KB page translations, and a 16-entry fully associative TLB is used for 64 KB, 512 KB, and 4 MB page translations and locked pages of all four sizes.

On a TLB miss, the MMU immediately traps to software for TLB miss processing. The TLB miss handler can fill the TLB by any available means, but it is likely to take advantage of the TLB miss support features provided by the MMU, since the TLB miss handler is time-critical code.

A general software view of the MMU is shown in FIGURE 11-2. The TLBs, which are part of the MMU hardware, are small and fast. The software translation table is likely to be large and complex. The translation storage buffer (TSB), which acts like a direct-mapped cache, is the interface between the two. The TSB can be shared by all processes running on a processor or can be process specific. The hardware does not require any particular scheme.

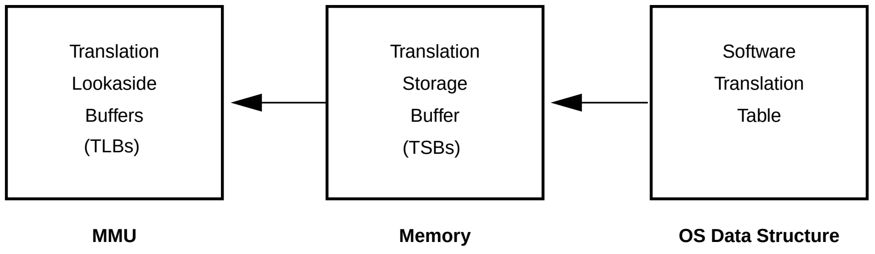

Translation Lookaside Buffers (TLBs)
Translation Storage Buffer (TSBs)
Software Translation Table
MMU
Memory
OS Data Structure

FIGURE 11-2 Software View of the MMU

Aliasing between pages of different sizes (when multiple virtual addresses map to the same physical address) can occur. However, the reverse case of multiple mappings from one virtual address to multiple physical addresses producing a multiple TLB match is not necessarily detected in hardware and may produce undefined results.

Note – The hardware ensures the physical reliability of the TLB on multiple matches.

11.2 Translation Table Entry

The Translation Table Entry (TTE) is the equivalent of a SPARC V8 page table entry; it holds information for a single page mapping. The TTE is divided into two 64-bit words representing the tag and data of the translation. Just as in a hardware cache, the tag is used to determine whether there is a hit in the TSB; if there is a hit, the data is fetched by software.

The configuration of the TTE is illustrated in FIGURE 11-3 and described in TABLE 11-1.

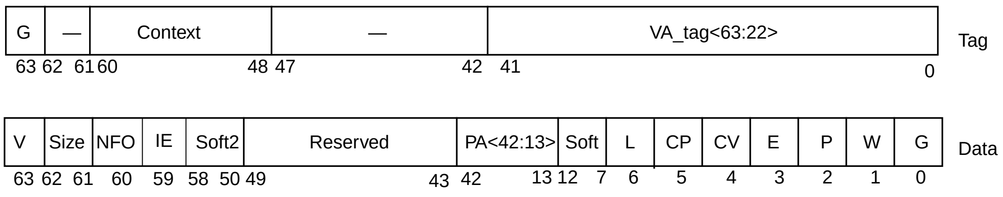


FIGURE 11-3 Translation Storage Buffer (TSB) Translation Table Entry (TTE)

TABLE 11-1 TSB and TTE Bit Description *(1 of 4)*

Bit	**Field**	**Description**
Tag – 63	`G`	Global. If the Global bit is set, the `Context` field of the TLB entry is ignored during hit detection. This behavior allows any page to be shared among all (user or supervisor) contexts running in the same processor. The Global bit is duplicated in the TTE tag and data to optimize the software miss handler.
Tag – 62:61	—	*Reserved.*
Tag – 60:48	`Context`	The 13-bit context identifier associated with the TTE.
Tag – 47:42	—	*Reserved.*
Tag – 41:0	`VA_tag` <63:22>	Virtual Address Tag. The virtual page number. Bits 21 through 13 are not maintained in the tag because these bits index the minimally sized, direct-mapped TSB of 512 entries.
Data – 63	`V`	Valid. If the Valid bit is set, then the remaining fields of the TTE are meaningful. Note that the explicit Valid bit is redundant with the software convention of encoding an invalid TTE with an unused context. The encoding of the context field is necessary to cause a failure in the TTE tag comparison, and the explicit Valid bit in the TTE data simplifies the TLB miss handler.

TABLE 11-1 TSB and TTE Bit Description *(2 of 4)*

Bit	Field	Description
Data – 62:61	`Size`	The page size of this entry, encoded as shown below. <table><tr><th>Size<1:0></th><th>Page Size</th></tr><tr><td>00</td><td>8 KB</td></tr><tr><td>01</td><td>64 KB</td></tr><tr><td>10</td><td>512 KB</td></tr><tr><td>11</td><td>4 MB</td></tr></table>
Data – 60	`NFO`	No Fault Only. If the no-fault-only bit is set, loads with ASI_PRIMARY_NO_FAULT, ASI_SECONDARY_NO_FAULT, and their *_LITTLE variations are translated. Any other access will trap with a *data_access_exception* trap (`FT` = 10_{16}). The `NFO` bit in the I-MMU is read as 0 and ignored when written. The I-TLB miss handler should generate an error if this bit is set before the `TTE` is loaded into the TLB.
Data – 59	`IE`	Invert Endianness. If this bit is set for a page, accesses to the page are processed with inverse endianness from that specified by the instruction (big for little, little for big). See page 261 for details. The `IE` bit in the I-MMU is read as 0 and ignored when written. **Note:** This bit is intended to be set primarily for non-cacheable accesses. The performance of cacheable accesses will be degraded as if the access missed the D-cache.
Data – 58:50	`Soft2`	Software-defined field, provided for use by the operating system. The `Soft2` field can be written with any value in the TSB. Hardware is not required to maintain this field in the TLB; therefore, when it is read from the TLB, it may read as zero.
Data – 49:43	*Reserved*	*Reserved*. Value on read is undefined.
Data – 42:13	`PA`	The physical page number. Page offset bits for larger page sizes (`PA`<15:13>, `PA`<18:13>, and `PA`<21:13> for 64 KB, 512 KB, and 4 MB pages, respectively) are stored in the TLB and returned for a Data Access read but are ignored during normal translation. When page offset bits for larger page sizes (`PA`<15:13>, `PA`<18:13>, and `PA`<21:13> for 64 KB, 512 KB, and 4 MB pages, respectively) are stored in the TLB on the UltraSPARC III Cu processor, the data returned from those fields by a Data Access read are the data previously written to them.
Data – 12:7	`Soft`	Software-defined field, provided for use by the operating system. The `Soft` field can be written with any value in the `TSB`. Hardware is not required to maintain this field in the TLB; therefore, when it is read from the TLB, it may read as zero.

TABLE 11-1 TSB and TTE Bit Description *(3 of 4)*

Bit	Field	Description
Data – 6	L	If the lock bit is set, then the TTE entry will be “locked down” when it is loaded into the TLB; that is, if this entry is valid, it will not be replaced by the automatic replacement algorithm invoked by an ASI store to the Data In Register. The lock bit has no meaning for an invalid entry. Arbitrary entries can be locked down in the TLB. Software must ensure that at least one entry is not locked when replacing a TLB entry; otherwise, a locked entry will be replaced. Since the 16-entry, fully associative TLB is shared for all locked entries as well as for 4 MB and 512 KB pages, the total number of locked pages is limited to less than or equal to 15. In the UltraSPARC III Cu processor, the TLB lock bit is implemented in the D-MMU 16-entry, fully associative TLB, and the I-MMU 16-entry, fully associative TLB. In the D-MMU 512-entry, 2-way associative TLB and I-MMU 128-entry, 2-way associative TLB, each TLB entry’s lock bit reads as 0 and writes to it are ignored. The lock bit set for 8 KB page translation in both I-MMU and D-MMU is read as 0 and ignored when written.
Data – 5 Data – 4	CP, CV	The cacheable-in-physically-indexed-cache bit and cacheable-in-virtually-indexed-cache bit determine the placement of data in the caches. The UltraSPARC III Cu processor fully implements the CV bit. The following table describes how CP and CV control cacheability in specific UltraSPARC III Cu processor caches.
		(see nested table below)
		The MMU does not operate on the cacheable bits but merely passes them through to the cache subsystem. The CV bit in the I-MMU is read as zero and ignored when written.
Data – 3	E	Side-effect. If the side-effect bit is set, non-faulting loads will trap for addresses within the page, non-cacheable memory accesses other than block loads and stores are strongly ordered against other E-bit accesses, and non-cacheable stores are not merged. This bit should be set for pages that map I/O devices having side-effects. Note, however, that the E bit does not prevent normal instruction prefetching. The E bit in the I-MMU is read as 0 and ignored when written. **Note:** The E bit does not force a non-cacheable access. It is expected, but not required, that the CP and CV bits will be set to 0 when the E bit is set. If both the CP and CV bits are set to 1 along with the E bit, the result is undefined. **Note Also:** The E bit and the NFO bit are mutually exclusive; both bits should never be set in any TTE.

Meaning of TTE when placed in:		
Cacheable (CP, CV)	**I-TLB (Instruction Cache PA-indexed)**	**D-TLB (Data Cache VA-indexed)**
00, 01	Non-cacheable	Non-cacheable
10	Cacheable L2-cache, I-cache	Cacheable L2-cache, W-cache, and P-cache
11	Cacheable L2-cache, I-cache	Cacheable L2-cache, D-cache, W-cache, and P-cache

TABLE 11-1 TSB and TTE Bit Description *(4 of 4)*

Bit	Field	Description
Data – 2	`P`	Privileged. If the `P` bit is set, only the supervisor can access the page mapped by the `TTE`. If the `P` bit is set and an access to the page is attempted when `PSTATE.PRIV = 0`, then the MMU signals an *instruction_access_exception* or *data_access_exception* trap (`FT` = 1_{16}).
Data – 1	`W`	Writable. If the `W` bit is set, the page mapped by this `TTE` has write permission granted. Otherwise, write permission is not granted, and the MMU causes a *fast_data_access_protection* trap if a write is attempted. The `W` bit in the I-MMU is read as 0 and ignored when written.
Data – 0	`G`	Global. This bit must be identical to the Global bit in the `TTE` tag. Like the Valid bit, the Global bit in the `TTE` tag is necessary for the TSB hit comparison, and the Global bit in the `TTE` data facilitates the loading of a TLB entry.

11.3 Translation Storage Buffer

The Translation Storage Buffer (TSB) is an array of Translation Table Entries managed entirely by software. It serves as a cache of the software translation table, used to quickly reload the TLB in the event of a TLB miss. The discussion in this section assumes the use of the hardware support for TSB access described in “Hardware Support for TSB Access” on page 253, although the operating system is not required to make use of this support hardware.

Inclusion of the TLB entries in the TSB is not required; that is, translation information that is not present in the TSB can exist in the TLB.

A bit in the TSB register allows the TSB 64 KB pointer to be computed for the case of common or split 8 KB/64 KB TSBs.

11.3.1 TSB Indexing Support

No hardware TSB indexing support is provided for the 512 KB and 4 MB page TTEs. However, since the TSB is entirely software managed, the operating system may choose to place these larger page TTEs in the TSB by forming the appropriate pointers. In addition, simple modifications to the 8 KB and 64 KB index pointers provided by the hardware allow formation of an M-way, set-associative TSB, multiple TSBs per page size, and multiple TSBs per process.

11.3.2 TSB Cacheability

The TSB exists as a normal data structure in memory and therefore can be cached. Indeed, the speed of the TLB miss handler relies on the TSB accesses hitting the level-2 cache at a substantial rate. This policy may result in some conflicts with normal instruction and data accesses, but it is hoped that the dynamic sharing of the level-2 cache resource will provide a better overall solution than that provided by a fixed partitioning.

11.3.3 TSB Organization

The TSB is arranged as a direct-mapped cache of TTEs. The MMU provides pre-computed pointers into the TSB for the 8 KB and 64 KB page TTEs. In each case, *n* least significant bits of the respective virtual page number are used as the offset from the TSB base address, with *n* equal to log base 2 of the number of TTEs in the TSB.

The TSB organization is illustrated in FIGURE 11-4. The constant *N* is determined by the `Size` field in the TSB Register. The `Size` Field is 3 bits and can have a value from 0 to 7. For the UltraSPARC III Cu processor, the `Size` Field is set to 7. The constant *N* is equal to 512×2^{size}. Therefore, for the UltraSPARC III Cu processor, the constant *N* is equal to 64K.

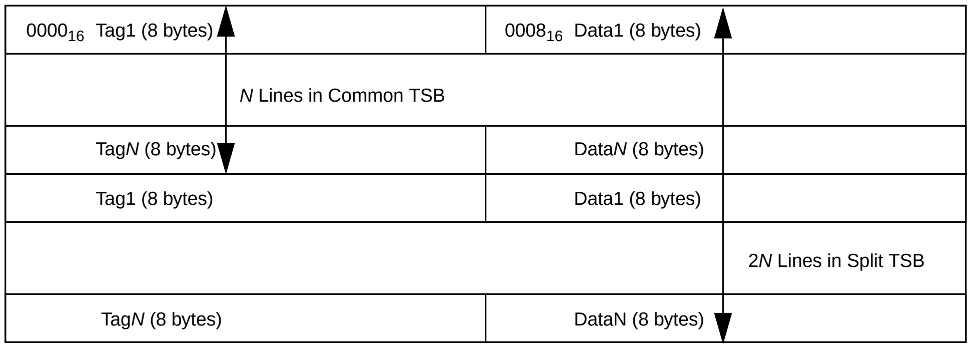


FIGURE 11-4 TSB Organization, Illustrated for Both Common and Shared Cases

11.4 Hardware Support for TSB Access

The MMU hardware provides services to allow the TLB miss handler to efficiently reload a missing TLB entry for an 8 KB or 64 KB page. These services include:

- Formation of TSB Pointers, based on the missing virtual address and address space identifier
- Formation of the TTE Tag Target used for the TSB tag comparison
- Efficient atomic write of a TLB entry with a single store ASI operation
- Alternate globals on MMU-signalled traps

11.4.1 Typical TLB Miss/Refill Sequence

The following is a typical TLB miss and TLB refill sequence:

1. A TLB miss causes either a *fast_instruction_access_MMU_miss* or a *fast_data_access_MMU_miss* exception.
2. The appropriate TLB miss handler loads the TSB Pointers and the TTE Tag Target with loads from the MMU registers.
3. Using this information, the TLB miss handler checks to see if the desired TTE exists in the TSB. If so, the TTE data is loaded into the TLB Data In Register to initiate an atomic write of the TLB entry chosen by the replacement algorithm.
4. If the TTE does not exist in the TSB, then the TLB miss handler jumps to the more sophisticated, and slower, TSB miss handler.

The virtual address used in the formation of the pointer addresses comes from the Tag Access Register, which holds the virtual address and context of the load or store responsible for the MMU exception.

Note – There are no separate physical registers in hardware for the pointer registers; rather, they are implemented through a dynamic reordering of the data stored in the Tag Access and the TSB registers.

11.4.2 TSB Pointer Formation

Hardware provides pointers for the most common cases of 8 KB and 64 KB page miss processing. These pointers give the virtual addresses where the 8 KB and 64 KB TTEs are stored if either are present in the TSB.

11.4.2.1 Input Values for TSB Pointer Formation

The pointer to the TTE in the TSB is generated from the following parameters as inputs:

- TSB base address (`TSB_Base`)
- Virtual address (`VA`)
- `TSB_size`
- `TSB_split`
- `TSB_Hash`

The TSB base address is in either of I/D Primary/Secondary (provided only for data)/Nucleus TSB Extension Registers. Depending on the context that generated the TLB miss, an appropriate TSB Extension Register is selected (which may be combined with the `TSB_Base` field from the TSB Base Register; see Note). Note that the context with the TLB miss is logged in the I/D Synchronous Fault Status Register.

`TSB_size` and `TSB_split` are supplied also from the selected TSB Extension Register.

The virtual page number to be used for TSB pointer formation is in the I/D Tag Access Register.

Note – `TSB_Base` address may be generated by exclusive-ORing `TSB_Base` register and TSB Extension Register contents, for compatibility with the UltraSPARC I and UltraSPARC II processor TSB pointer formation. In this case, if the TSB Extension Registers hold 0 as `TSB_Base`, the value in `TSB_Base` register becomes the `TSB_Base` address, thereby maintaining compatibility with the UltraSPARC I and UltraSPARC II processor TLB miss handling software. In addition, `TSB_Base` may be taken directly from an appropriate TSB Extension Register. In that case, the implementation should provide the way to maintain compatibility with the UltraSPARC I and UltraSPARC II processor TLB miss handler software.

The UltraSPARC III Cu processor generates the `TSB_Base` address by taking the `TSB_Base` field directly from the TSB Extension Register.

11.4.2.2 TSB Pointer Formation

Hardware uses the following equations to form TSB pointers for TLB misses. In the equations, n is defined to be the TSB_Size field of the TSB register; it ranges from 0 to 7. Note that TSB_Size refers to the size of each TSB when the TSB is split. The □ symbol designates concatenation of bit vectors and ⊕ indicates an exclusive-OR operation.

Exclusive-ORed TSB_Base

- ***For a shared TSB (TSB Register split field = 0):***

8K_POINTER = TSB_Base[63:13 + n] ⊕ TSB_Extension[63:13 + n] □ VA[21+n:13] □ 0000

64K_POINTER = TSB_Base[63:13 + n] ⊕ TSB_Extension[63:13 + n] □ VA[24+n:16] □ 0000

- ***For a split TSB (TSB Register split field = 1):***

8K_POINTER = TSB_Base[63:14 + n] ⊕ TSB_Extension[63:14 + n] □ 0 □ VA[21 + n:13] □ 0000

64K_POINTER = TSB_Base[63:14 + n] ⊕ TSB_Extension[63:14 + n] □ 1 □ VA[24 + n:16] □ 0000

TSB_Base from TSB Extension Registers

- ***For a shared TSB (TSB Register split field = 0):***

8K_POINTER = TSB_Extension[63:13 + n] □ (VA[21 + n:13] ⊕ TSB_Hash) □ 0000

64K_POINTER = TSB_Extension[63:13 + n] □ (VA[24 + n:16] ⊕ TSB_Hash) □ 0000

- ***For a split TSB (TSB Register split field = 1):***

8K_POINTER = TSB_Extension[63:14 + n] □ 0 □ (VA[21 + n:13] ⊕ TSB_Hash) □ 0000

64K_POINTER = TSB_Extension[63:14 + n] □ 1 □ (VA[24 + n:16] ⊕ TSB_Hash) □ 0000

The TSB Tag Target is formed by aligning the missing access VA (from the Tag Access Register) and the current context to positions found above in the description of the TTE tag, allowing a simple XOR instruction for TSB hit detection.

11.4.3 TSB Pointer Logic Hardware Description

FIGURE 11-5 illustrates the generation of the 8 KB and 64 KB pointers; CODE EXAMPLE 11-1 presents pseudocode for D-MMU pointer logic.

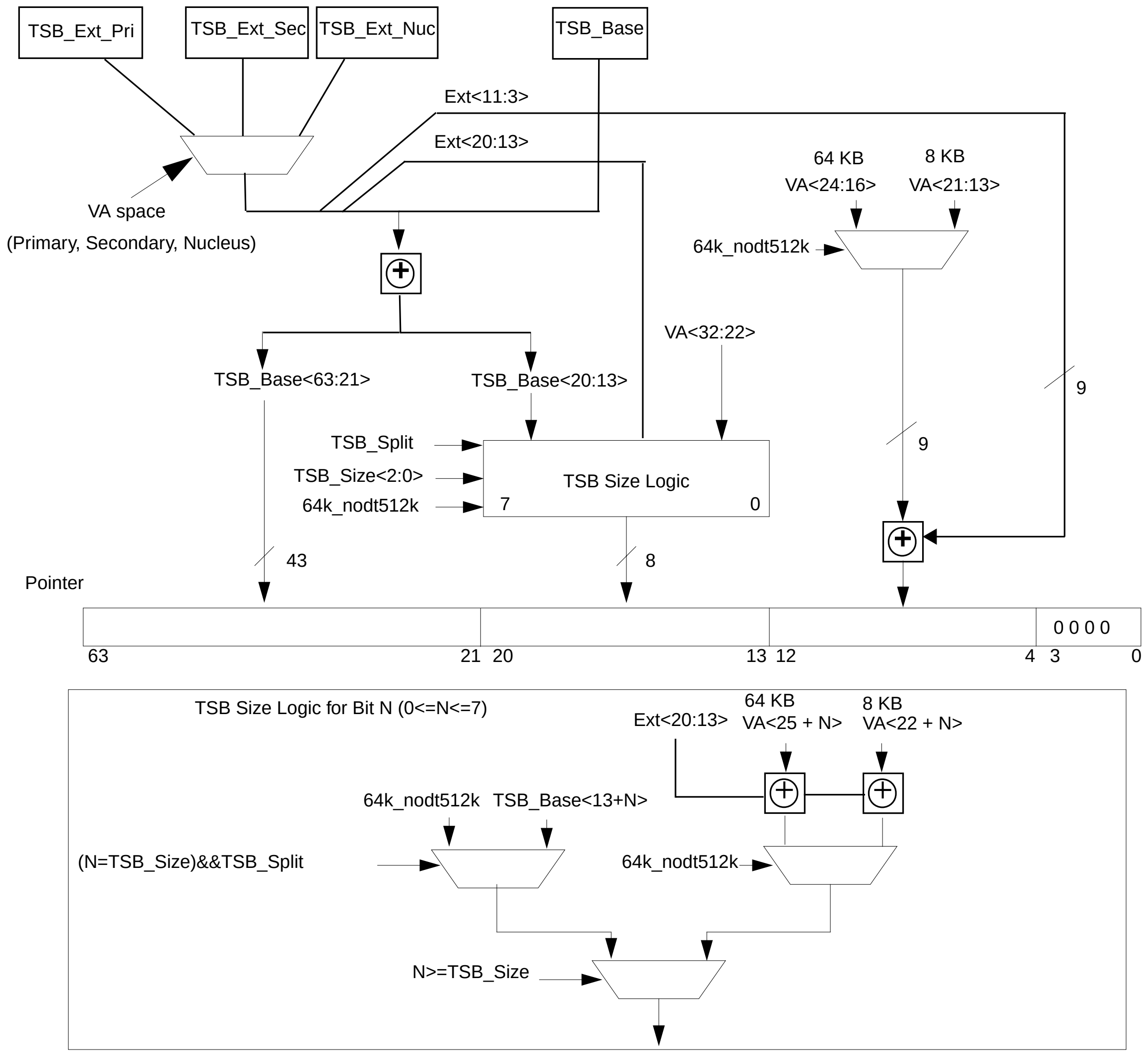


FIGURE 11-5 Formation of TSB Pointers for 8 KB and 64 KB TTE

CODE EXAMPLE 11-1 Pseudocode for D-MMU Pointer Logic

```
int64 GenerateTSBPointer(
     int64 va,         // Missing virtual address
     PointerType type, // 8K_POINTER or 64K_POINTER
     int64 TSBBase,    // TSB Register<63:13> << 13
     Boolean split,    // TSB Register<12>
     int TSBSize,      // TSB Register<2:0>
     int SpaceType space)
{
     int64 vaPortion;
     int64 TSBBaseMask;
     int64 splitMask;

     // Shift va towards lsb appropriately and
     // zero out the original va page offset
     vaPortion = (va >> ((type == 8K_POINTER)? 9: 12)) &
                0xfffffffffffffff0;

     switch (space) {
     Primary:
         TSBBASE ^=TSB_EXT_pri;
         vaPortion ^= TSB_EXT_pri<<1 & 0x1ff0;
         vaPortion ^= TSB_EXT_pri & 0x1fe000;
         break;
     Secondary:
         TSBBASE ^=TSB_EXT_sec;
         vaPortion ^= TSB_EXT_sec<<1 & 0x1ff0;
         vaPortion ^= TSB_EXT_sec & 0x1fe000;
         break;
     Nucleus:
         TSBBASE ^=TSB_EXT_nuc;
         vaPortion ^= TSB_EXT_nuc<<1 & 0x1ff0;
         vaPortion ^= TSB_EXT_nuc & 0x1fe000;
         break;
     }
     // TSBBaseMask marks the bits from TSB Base Reg
     TSBBaseMask = 0xffffffffffffe000 <<
            (split? (TSBSize + 1) : TSBSize);

     if (split) {
         // There’s only one bit in question for split
         splitMask = 1 << (13 + TSBSize);
         if (type == 8K_POINTER)
            // Make sure we’re in the lower half
            vaPortion &= ~splitMask;
         else
            // Make sure we’re in the upper half
            vaPortion |= splitMask;
     }
     return (TSBBase & TSBBaseMask) | (vaPortion & ~TSBBaseMask);
}
```

11.4.4 Required TLB Conditions

The following items must be locked in the TLB to avoid an error condition:

- TLB miss handler and data
- TSB and linked data
- Asynchronous trap handlers and data

11.4.5 Required TSB Conditions

The following items must be locked in the TSB (not necessarily the TLB) to avoid an error condition:

- TSB miss handler and data
- Interrupt-vector handler and data

11.4.6 MMU Global Registers Selection

In the SPARC V9 normal trap model, the software is presented with an alternate set of global registers in the Integer Register file. An UltraSPARC III Cu processor provides an additional feature to facilitate fast handling of TLB misses. For the following traps, the trap handler is presented with a special set of MMU globals:

- *fast_instruction_access_MMU_miss*
- *fast_data_access_MMU_miss*
- *instruction_access_exception*
- *data_access_exception*
- *fast_data_access_protection*

Trap handlers for the *privileged_action*, *mem_address_not_aligned*, and **_mem_address_not_aligned* traps use the standard alternate global registers.

Compatibility Note – The MMU does not perform hardware tablewalking. The MMU hardware never directly reads or writes the TSB.

11.5 Faults and Traps

The traps recorded by the MMU are listed in TABLE 11-2 and described below the table, by the reference number. All listed traps are precise traps.

TABLE 11-2 MMU Trap Types, Causes, and Stored State Register Update Policy

			Registers Updated (Stored State in MMU)				
Ref #	**Trap Name**	**Trap Cause**	**I-SFSR**	**I-MMU Tag Access**	**D-SFSR, SFAR**	**D-MMU Tag Access**	**Trap Type**
1.	*fast_instruction_access_MMU_miss*	I-TLB miss	X	X			64_{16}–67_{16}
2.	*instruction_access_exception*	Privilege violation for I-fetch	X	X			08_{16}
3.	*fast_data_access_MMU_miss*	D-TLB miss			X	X	68_{16}–$6B_{16}$
4.	*data_access_exception*	Several (see below)			X	X†	30_{16}
5.	*fast_data_access_protection*	Protection violation			X	X	$6C_{16}$ –$6F_{16}$
6.	*privileged_action*	Use of privileged ASI			X		37_{16}
7.	watchpoints	Watchpoint hit			X		61_{16}–62_{16}
8.	*mem_address_not_aligned*, *_mem_address_not_aligned	Misaligned mem op			X		35_{16}, 36_{16}, 38_{16}, 39_{16}

† The contents of the context field of the D-MMU Tag Register are undefined after a *data_access_exception*.

Note – In an UltraSPARC III Cu processor, *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection* traps are generated instead of SPARC V9 *instruction_access_MMU_miss*, *data_access_MMU_miss*, and *data_access_protection* traps, respectively.

1. ***fast_instruction_access_MMU_miss*** **—** Occurs when the MMU is unable to find a translation for an instruction access; that is, when the appropriate TTE is not in the I-TLB.

   In an UltraSPARC III Cu processor, the *fast_instruction_access_MMU_miss* exception is generated instead of the SPARC V8 *instruction_access_MMU_miss*.

2. ***instruction_access_exception* —** Occurs when the I-MMU is enabled and detects a privilege violation for an instruction fetch; that is, an attempted access to a privileged page when `PSTATE.PRIV` = 0.

3. ***fast_data_access_MMU_miss* —** Occurs when the MMU is unable to find a translation for a data access; that is, when the appropriate TTE is not in the D-TLB.

   In an UltraSPARC III Cu processor, the *fast_data_access_MMU_miss* exception is generated instead of the SPARC V9 *data_access_MMU_miss* trap.

4. ***data_access_exception* —** Signalled upon the detection of at least one of the following exceptional conditions:
   - The D-MMU detects a privilege violation for a data access; that is, an attempted access to a privileged page when `PSTATE.PRIV` = 0.
   - A speculative (non-faulting) load instruction issued to a page marked with the side-effect (`E` bit) set to 1, including cases in which the D-MMU is disabled.
   - An atomic instruction (including 128-bit atomic load) issued to a memory address marked non-cacheable in a physical cache; that is, with the `CP` bit set to 0, including cases in which the D-MMU is disabled.
   - An invalid `LDA/STA` ASI value, read to write-only register, or write to read-only register. Not for an attempted user access to a restricted ASI (see the *privileged_action* trap described below).
   - An access with an ASI other than "`(PRIMARY,SECONDARY)_NO_FAULT(_LITTLE)`" to a page marked with the `NFO` (no-fault-only) bit.

5. ***fast_data_access_protection* —** Occurs when the MMU detects a protection violation for a data access. A protection violation is defined to be an attempted store (including atomic load-store operations) to a page that does not have write permission.

   In an UltraSPARC III Cu processor, the *fast_data_access_protection* exception is generated instead of the SPARC V9 *data_access_protection* trap.

6. ***privileged_action* —** Occurs when an access is attempted using a *restricted* ASI while in non-privileged mode (`PSTATE.PRIV` = 0).

7. **watchpoints —** *PA_watchpoint* and *VA_watchpoint* traps are included in this category. Watchpoint traps occur when watchpoints are enabled and the D-MMU detects a load or store to the virtual or physical address specified by the watchpoint virtual or physical registers, respectively. See Section 6.10.2 "Data Watchpoint Registers" on page 6-136. The trap is precise and is signalled before the actual event, meaning that the contents of the location are not modified when the trap is invoked.

8. ***mem_address_not_aligned* —** Occurs when a load, store, atomic, `JMPL`, or `RETURN` instruction with a misaligned address is executed.

On a *mem_address_not_aligned* trap that occurs during a `JMPL` or `RETURN` instruction, the UltraSPARC III Cu processor updates the `D-SFAR` and `D-SFSR` registers with the fault address and status, respectively.

11.6 ASI Value, Context, and Endianness Selection for Translation

The selection of the context for a translation is the result of a two-step process:

1. The ASI is determined (conceptually by the Integer Unit) from the instruction, ASI register, trap level, and the processor endian mode (`PSTATE.CLE`).
2. The Context Register is determined directly from the ASI. The context value is read by the Context Register selected by the ASI.

The ASI value and endianness (little or big) are determined for the I-MMU and D-MMU, respectively, according to TABLE 11-3 through TABLE 11-5. Note that the secondary context is never used to fetch instructions. The I-MMUs and D-MMUs, when using the Primary Context identifier, use the value stored in the shared Primary Context Register.

The endianness of a data access is specified by the following three conditions:

- ASI specified in the opcode or ASI register
- `PSTATE` current little-endian bit (`CLE`)
- D-MMU invert endianness bit

  The D-MMU invert endianness bit does not affect the ASI value recorded in the `SFSR` but does invert the endianness that is otherwise specified for the access.

Note – The D-MMU invert endianness bit inverts the endianness for all accesses, including load/store/atomic alternates that have specified an ASI. That is, `LDXA` [%g1] ASI_PRIMARY_LITTLE will be _BIG if the `IE` bit is on.

TABLE 11-3 ASI Mapping for Instruction Access

Condition for Instruction Access	Resulting Action	
PSTATE.TL	**Endianness**	**ASI Value (in SFSR)**
0	BIG	ASI_PRIMARY
> 0	BIG	ASI_NUCLEUS

TABLE 11-4 ASI Mapping for Data Accesses

Condition for Data Access				**Access Processed with:**	
Opcode	**TL**	**PSTATE.CLE**	**DMMU.IE**	**Endian**	**ASI Value (Recorded in SFSR)**
Load/Store/Atomic	0	0	0	BIG	ASI_PRIMARY
			1	LITTLE	ASI_PRIMARY
		1	0	LITTLE	ASI_PRIMARY_LITTLE
			1	BIG	ASI_PRIMARY_LITTLE
	> 0	0	0	BIG	ASI_NUCLEUS
			1	LITTLE	ASI_NUCLEUS
		1	0	LITTLE	ASI_NUCLEUS_LITTLE
			1	BIG	ASI_NUCLEUS_LITTLE
Load/Store/Atomic Alternate with specified ASI *not* ending in `_LITTLE`	x	x	0	BIG	Specified ASI value from immediate field in opcode or ASI Register
			1	LITTLE	
Load/Store/Atomic Alternate with specified ASI ending in `_LITTLE`	x	x	0	LITTLE	Specified ASI value from immediate field in opcode or ASI Register
			1	BIG	
`FLUSH`	0	x	x	—	ASI_PRIMARY_*
	> 0	x	x	—	ASI_NUCLEUS

The Context Register used by the D-MMUs and I-MMUs is determined according to TABLE 11-5. The Context Register selection is not affected by the endianness of the access. For a comprehensive list of ASI values in the ASI map, see Chapter 8 “Address Space Identifiers.”

TABLE 11-5 I-MMU and D-MMU Context Register Usage

ASI Value	**Context Register**
ASI_*NUCLEUS* (any ASI name containing the string “NUCLEUS”)	Nucleus (0000_{16} hard-wired)
ASI_*PRIMARY* (any ASI name containing the string “PRIMARY”)	Primary
ASI_*SECONDARY* (any ASI name containing the string “SECONDARY”)	Secondary
All other ASI values	(Not applicable, no translation)

11.7 Reset, Disable, and RED_state Behavior

During global reset of the processor, the following statements apply:

- No change occurs in any block of the D-MMU.
- No change occurs in the datapath or TLB blocks of the I-MMU.
- The I-MMU resets its internal state machine to normal (nonsuspended) operation.
- The I-MMU and D-MMU Enable bits in the DCU Control Register are set to 0.

When the processor enters RED_state, the following statement applies:

- The I-MMU and D-MMU Enable bits in the DCU Control Register are set to 0.

Either of the MMUs is defined to be *disabled* when its respective MMU Enable bit equals 0 or, for the I-MMU only, whenever the processor is in RED_state. The D-MMU is enabled or disabled solely by the state of the D-MMU Enable bit.

When the D-MMU is disabled:

- The D-MMU passes all addresses through without translation ("bypasses" them); each address is truncated to the size of a physical address on the implementation, behaving as if the ASI_PHYS_* ASI had been used for the access.
- The processor behaves as if the TTE bits were set as:
  - TTE.IE ← 0
  - TTE.P ← 0
  - TTE.W ← 1
  - TTE.NFO ← 0
  - If DCUCR.CP and DCUCR.CV are implemented:
    - TTE.CP ← DCUCR.CP
    - TTE.CV ← DCUCR.CV
    - TTE.E ← *not* DCUCR.CP
  - If DCUCR.CP and DCUCR.CV are not implemented:
    - TTE.E ← *not* TTE.CP

However, if a bypass ASI (ASI_PHYS_*) is used while the D-MMU is disabled, the bypass operation behaves as it does when the D-MMU is enabled; that is, the access is processed with the E, CP, and CV bits as specified by the bypass ASI (see TABLE 11-26 on page 11-294).

When the I-MMU is disabled, it truncates all instruction accesses to the physical address size (43 bits) and passes the physically cacheable bit (Data Cache Unit Control Register CP bit) to the cache system. The access does not generate an *instruction_access_exception* trap.

Note – While the D-MMU is disabled and the default CV bit in the Data Cache Unit Control Register is set to 0, data in the D-cache can be accessed only through load and store alternates to the internal D-cache access ASI. Normal loads and stores bypass the D-cache. Data in the D-cache cannot be accessed by load or store alternates that use ASI_PHYS_*. Other caches are physically indexed or are still accessible.

When disabled, both the I-MMU and D-MMU correctly perform all LDXA and STXA operations to internal registers, and traps are signalled just as if the MMU were enabled. For instance, if a non-faulting load is issued when the D-MMU is disabled and DCUCR.CP is set to 0 if the implementation has the bit, then the D-MMU signals a *data_access_exception* trap (FT = 02_{16}), since E is set to 1.

Note – A reset of the TLB is not performed by a chip reset or by entry into RED_state. Before the MMUs are enabled, the operating system software must explicitly write each entry with either a valid TLB entry or an entry with the valid bit set to 0. The operation of the I-MMU or D-MMU in enabled mode is undefined if the TLB valid bits have not been set explicitly beforehand.

11.8 SPARC V9 "MMU Requirements" Annex

The MMU complies completely with the SPARC V9 "MMU Requirements" Annex. TABLE 11-6 shows how various protection modes can be achieved, if necessary, through the presence or absence of a translation in the I-MMU or D-MMU. Note that this behavior requires specialized TLB miss handler code to guarantee these conditions.

TABLE 11-6 MMU SPARC V9 Annex G Protection Mode Compliance

Condition			
TTE in D-MMU	**TTE in I-MMU**	**Writable Attribute Bit**	**Resultant Protection Mode**
Yes	No	0	Read-only
No	Yes	N/A	Execute-only
Yes	No	1	Read/Write
Yes	Yes	0	Read-only/Execute
Yes	Yes	1	Read/Write/Execute

11.9 Data Translation Lookaside Buffer

In the D-MMU, there are two TLBs (dt512_0 and dt_512_1) each of which have 512-entry two-way associative which can be programmed to hold unlocked 8 KB, 64 KB, 512 KB, and 4 MB pages for translations. Each TLB can hold any of the 4 page sizes, but are programmed to only one page size at any given time. Each TLB can be programmed to either the same or different page sizes. Also, a 16-entry fully associative TLB (dt16) is used for 8 KB, 64 KB, and 4 MB locked and unlocked pages. This TLB can hold any combination of page sizes that are locked/unlocked at a time.

Note – The UltraSPARC III Cu processor's dt16 can support *unlocked* 8K pages. This is necessary since if dt512_0 and dt512_1 were programmed to non-8K page size, then a D-TLB fill of unlocked 8K page will not get dropped.

Note – When both large D-TLBs are configured with the same page size, then they behave like a single D-TLB with 1024-entry 4-way associative (256 entries per way).

Each dt512's page size (`PgSz`) is programmable independently, one `PgSz` per context (Primary/Secondary/ Nucleus). Kernel can set the `PgSz` fields in ASI_PRIMARY_CONTEXT_REG and ASI_SECONDARY_CONTEXT_REG as illustrated in FIGURE 11-6 and FIGURE 11-7, respectively, and described in TABLE 11-7 and TABLE 11-8, respectively.

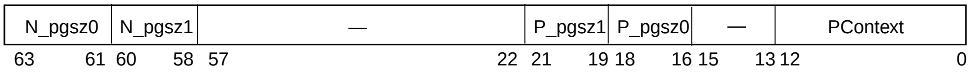


FIGURE 11-6 D-MMU Primary Context Register

TABLE 11-7 D-MMU Primary Context Register

Bit	Field	Description
63:61	`N_pgsz0`	Nucleus context's page size at the first large D-TLB (dt512_0).
60:58	`N_pgsz1`	Nucleus context's page size at the second large D-TLB (dt512_1).
57:22	—	*Reserved.*
21:19	`P_pgsz1`	Primary context's page size at the second large D-TLB (dt512_0).

TABLE 11-7 D-MMU Primary Context Register *(Continued)*

Bit	Field	Description
18:16	`P_pgsz0`	Primary context's page size at the first large D-TLB (dt512_1).
15:13	—	*Reserved.*
12:0	`PContext`	Context identifier for the primary address space.

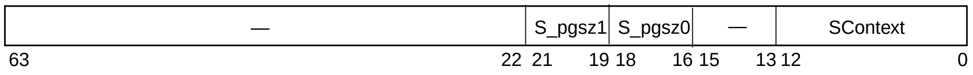


FIGURE 11-7 D-MMU Secondary Context Register

TABLE 11-8 D-MMU Primary Context Register

Bit	Field	Description
63:22	–	*Reserved.*
21:19	`S_pgsz1`	Secondary context's page size at the second large D-TLB (dt512_1).
18:16	`S_pgsz0`	Secondary context's page size at the first large D-TLB (dt512_0).
15:13	–	*Reserved.*
12:0	`SContext`	Context identifier for the secondary address space.

The following is the page size bit encoding (most significant bit is reserved to 0):

- 000 = 8 KB
- 001 = 64 KB
- 010 = 512 KB
- 011 = 4 MB

11.9.1 D-TLB Access Operation

When a memory access is issued, its VA, Context, and Page Size are presented to the D-MMU. All 3 D-TLBs (dt512_0, dt512_1, and dt16) are accessed in parallel. The fully associative dt16 only needs VA and Context to CAM-match and output an entry (1 out of 16). Proper VA bits are compared based on the page size bits of each dt16 entry (fast 3-bit encoding is used to define 8K, 64K, 512K, and 4M).

Since dt512's are not a fully associative structure, indexing the dt512 array requires knowledge of page size to properly select VA bits as index. The 2-way dt512_0 and dt512_1 needs `PgSz` to mux select proper VA bits to index the dt512 arrays, as shown below:

- 8 KB page: index = VA[20:13]
- 64 KB page: index = VA[23:16]
- 512 KB page: index = VA[26:19]
- 4 MB page: index = VA[29:22]

Context bits are used later after the indexed entry comes out of each array bank/way, to qualify the context hit.

Three possible Context numbers are active in the CPU: Primary (`PContext` field in ASI_PRIMARY_CONTEXT_REG), Secondary (`SContext` field in ASI_SECONDARY_CONTEXT_REG), and Nucleus (default to Context = 0). The Context Register to send to D-MMU is determined based on the load/store's ASI encoding of Primary/Secondary/Nucleus.

Since all 3 D-TLBs are being accessed in parallel, software must guarantee that there are no duplicate (stale) entry hits. Most of this responsibility lies in the software (operating system) with the hardware providing some assistance to support full software control. A set of rules on D-TLB replacement, demap and context switch must be followed to maintain consistent and correct behavior.

11.9.2 Same Page Size on Both dt512_0 and dt512_1

When both dt512's are programmed to have identical page size, the behavior is a "single" 4-way 1024-entry dt512. During TTE fill, if there is no invalid TLB entry to take, then dt512 selection and way selection are determined based on a new 10-bit `LFSR` (pseudo random generator, the same one used on 2-way L2-cache). For 4-way pseudo random selection, `LFSR`[1:0] bits (two least significant bits of LFSR) is described in TABLE 11-9 will be used.

TABLE 11-9 Four-way Pseudo Random Selection

LFSR[1:0]	**TTE Fill to:**
00	dt512_0, way 0
01	dt512_0, way 1
10	dt512_1, way 0
11	dt512_1, way 1

Note – All LFSRs (in D-cache, I-cache, W-cache, L2-cache, and TLBs) get initialized on Power-on Reset (Hard_POR) and System Reset (Soft_POR).

This single LFSR is also shared among both dt512_0 and dt512_1 when they have different page sizes. The least significant bit (LFSR[0]) is used for entry replacement. It selects the same bank of both dt512's, but only one dt512's write-enable is eventually asserted at TTE fill time.

Demap Context is needed when the *same* context changes Page Size. During context-in, if the operating system decides to change any Page Size setting of dt512_0 or dt512_1 *differently* from the last context-out of the *same* Context (e.g., was both 8K at context-out, now 8K and 4M at context-in), then the operating system will perform Demap Context operation first. This avoids remnant entries in dt512_0 or dt512_1, which could cause duplicate, possibly stale, hit.

11.9.3 D-TLB Automatic Replacement

A D-TLB miss fast trap handler utilizes the automatic (hardware) replacement write using store ASI_DTLB_DATA_IN_REG. Section 11.9.4 will describe the D-TLB direct write using store ASI_DTLB_DATA_ACCESS_REG, which is the kernel uses for initialization (e.g., in OBP) and diagnostic.

When D-TLB miss, or DATA_ACCESS_EXCEPTION, or FAST_DATA_ACCESS_PROTECTION is detected, hardware automatically saves the missing VA and context to the Tag Access Register (ASI_DMMU_TAG_ACCESS). To ease indexing of the dt512's when the TTE data is presented (via STXA ASI_DTLB_DATA_IN_REG), the missing page size information of dt512_0 and dt512_1 is captured into a new Extension Register, called ASI_DMMU_TAG_ACCESS_EXT.

FIGURE 11-8 illustrates and TABLE 11-10 describes the format of the Tag Access Extension Register for saving page sizes

ASI 0x58, VA<63:0> == 0x60
Name: ASI_DMMU_TAG_ACCESS_EXT
Access: RW

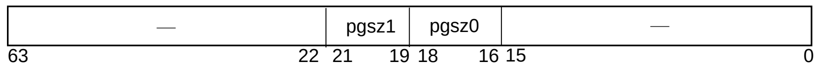


FIGURE 11-8 Tag Access Extension Register Format for Saving Page Sizes Information

TABLE 11-10 Tag Access Extension Register for Saving Page Sizes Information

Bit(s)	Field	Description
63:22	—	*Reserved.*
21:19	`pgsz1`	Page size of D-TLB miss' context (Primary/Secondary/Nucleus) in the second large D-TLB (dt512_0).
18:16	`pgsz0`	Page size of D-TLB miss' context (Primary/Secondary/Nucleus) in the first large D-TLB (dt512_1).
15:0	—	*Reserved.*

Note – Bit 21 and 18 are hardwired to zero since the most significant bit of the page size fields in Primary and Secondary registers are reserved to 0.

Note – With the saved page sizes, hardware pre-computes in the background the index to dt512_0 and dt512_1 for TTE fill. When the TTE data arrives, only one write enable to dt512_0, dt512_1, and dt16 will get activated.

CODE EXAMPLE 11-2 shows the hardware D-TLB replacement algorithm.

Note – `PgSz0` below is ASI_DMMU_TAG_ACCESS_EXT[18:16] bits; `PgSz1` below is ASI_DMMU_TAG_ACCESS_EXT[21:19] bits.

CODE EXAMPLE 11-2 D-TLB Replacement Algorithm

```
 if (TTE to fill is a locked page, i.e., L bit is set) {
    fill TTE to dt16;
} else if (both dt512's have same page size, PgSz0 == PgSz1) {
    if (TTE's Size != PgSz0) {
        fill TTE to dt16;
    } else {
        if (one of the 4 same-index entries is invalid) {
          fill TTE to an invalid entry with selection order of
          (dt512_0 way0, dt512_0 way1, dt512_1 way0, dt512_1 way1)
        } else {
            case (LFSR[1:0]) {
                00: fill TTE to dt512_0 way0;
                01: fill TTE to dt512_0 way1;
                10: fill TTE to dt512_1 way0
                11: fill TTE to dt512_1 way1;
        }
```

CODE EXAMPLE 11-2 D-TLB Replacement Algorithm *(Continued)*

```
    }
}
    } else {
        if (TTE’s Size == PgSz0) {
           if (one of the 2 same-index entries is invalid) {
            fill TTE to an invalid entry with selection order of
              (dt512_0 way0, dt512_0 way1)
        } else {
            case (LFSR[0]) {
                0: fill TTE to dt512_0 way0;
                1: fdt512ill TTE to dt512_0 way1;
            }
        }
    } else if (TTE’s Size == PgSz1) {
        if (one of the 2 same-index entries is invalid) {
          fill TTE to an invalid entry with selection order of
          (dt512_1 way0, dt512_1 way1)
        } else {
            case (LFSR[0]) {
                0: fill TTE to dt512_1 way0;
                1: fill TTE to dt512_1 way1;
            }
        }
    } else {
        fill TTE to dt16;
    }
}
```

11.9.4 D-TLB Directed Data Read/Write

Each D-TLB can be directly written using the store ASI_DTLB_DATA_ACCESS_REG instruction. The kernel typically uses this method for initialization (e.g., in OBP) and diagnostic.

Due to the addition of a second large D-TLB, in the UltraSPARC III Cu processor, a new encoding value was added to the VA of ASI_DTLB_DATA_ACCESS_REG (0x5d), as listed in TABLE 11-11.

TABLE 11-11 D-TLB Access Number

TLB #	TLB Type	Number of Entries
0	Small fully associative, all 4 page sizes (8K, 64K, 512K, 4M) and locked pages	16

TABLE 11-11 D-TLB Access Number

TLB #	TLB Type	Number of Entries
2	First large D-TLB, 2 way associative, all 4 page sizes (8K, 64K, 512K, 4M)	512
3	Second large D-TLB, 2 way associative, all 4 page sizes (8K, 64K, 512K, 4M)	512

Note – For the 2-way dt512_0 and dt512_1, bit 11 (most significant bit of "TLB Entry") of the VA of ASI 0x5d determines the way select: 0 for way 0, 1 for way 1.

Note that since hardware has to zero out proper tag/VA bits based on page size prior to writing to dt512_0/dt512_1 (to avoid tag masking delay on read access of variable page size), D-MMU will use the page size information of TTE data (store data of `STXA` ASI_DTLB_DATA_ACCESS_REG).

For diagnostic/direct ASI read from ASI_DTLB_DATA_ACCESS_REG, bits [62:61] (Page Size field of TTE data) do not reflect meaningful size of dt512_0 and dt512_1; they always read zero. This is because there is actually *no* size bits portion stored in the dt512 SRAM.

Note – Bit[47] of TTE data is hardwired to 0.

11.9.5 D-TLB Tag Read Register

The UltraSPARC III Cu processor behavior on read to ASI_DTLB_TAG_READ_REG (ASI 0x5E) is as follows:

- For dt16 (small 16-entry fully-associative D-TLB), the 64-bit data read is the same as in the UltraSPARC III processor – backward compatible.
- For dt512_0 and dt512_1, the bit positions of VPN (virtual page number) within bits[63:13] change as follows:

```
Data[12:0] = Context[12:0]
Data[63:21] = VA[63:21]                    if 8K page
Data[63:21] = VA[63:24] 3'b100'            if 64K page
Data[63:21] = VA[63:27] 6'b100000'         if 512K page
Data[63:21] = VA[63:30] 9'b100000000'      if 4M page
Data[20:13] = Hb read number
                  ^                      ^
                  |                      |
                  VPN (virtual page number)
```

11.9.6 Demap Operation

For Demap Page in the large D-TLBs, the page size used to index the D-TLBs is derived based on the Context bits (Primary/Secondary/Nucleus). Hardware will automatically select proper `PgSz` bits based on the "Context" field (Primary/Secondary/Nucleus) defined in ASI_DMMU_DEMAP (ASI 0x5f). These two `PgSz` fields are used to properly index dt512_0 and dt512_1.

Demap Context and Demap All operations with dt512_0/dt512_1/dt16 are straight forward, except that one additional dt512 is demap in parallel.

Note – When global pages are used (G = 1), any active pages in a dt512 must have the *same* Page Size (but dt512_0's `PgSz` and dt512_1's `PgSz` can be different). When pages with G = 1 in a dt512 have variety of page sizes, the dt512 cannot index and locate the page correctly when trying to match the VA tag without the context number as a qualifier. Proper demap must be done, as well as proper Page Size settings in `PContext` and `SContext` registers.

11.9.7 D-TLB Access Summary

TABLE 11-12 lists the D-MMU TLB access summary.

TABLE 11-12 D-MMU TLB Access Summary

Software Operation		**Effect on MMU Physical Registers**				
Load/ Store	**Register**	**TLB Tag Array**	**TLB Data Array**	**Tag Access**	**SFSR**	**SFAR**
Load	Tag Read	Contents returned. From entry specified by LDXA's access	No effect	No effect	No effect	No effect
	Tag Access	No effect	No effect	Contents returned	No effect	No effect
	Data In	Trap with *data_access_exception*				
	Data Access	No effect	Contents returned. From entry specified by LDXA's access	No effect	No effect	No effect
	SFSR	No effect	No effect	No effect	Contents returned	No effect
	SFAR	No effect	No effect	No effect	No effect	Contents returned

TABLE 11-12 D-MMU TLB Access Summary *(Continued)*

Software Operation		**Effect on MMU Physical Registers**				
Load/ Store	**Register**	**TLB Tag Array**	**TLB Data Array**	**Tag Access**	**SFSR**	**SFAR**
Store	Tag Read	Trap with *data_access_exception*				
	Tag Access	No effect	No effect	Written with store data	No effect	No effect
	Data In	TLB entry determined by replacement policy written with contents of Tag Access Register	TLB entry determined by replacement policy written with store data	No effect	No effect	No effect
	Data Access	TLB entry specified by STXA address written with contents of Tag Access Register	TLB entry specified by STXA address written with store data	No effect	No effect	No effect
	SFSR	No effect	No effect	No effect	Written with store data	No effect
	SFAR	No effect	No effect	No effect	No effect	Written with store data
TLB miss		No effect	No effect	Written with VA and context of access	Written with fault status of faulting instruction and page sizes at faulting context for two 2-way set associative TLB	Written with virtual address of faulting instruction

11.9.8 D-MMU Operation Summary

The behavior of the D-MMU is summarized in TABLE 11-13 on page 11-276. In each case and for all conditions, the behavior of each MMU is given by one of the following abbreviations:

Abbreviation	Meaning
OK	Normal translation
Dmiss	*fast_data_access_MMU_miss* exception
Dexc	*data_access_exception* exception
Dprot	*fast_data_access_protection* exception
Imiss	*fast_instruction_access_MMU_miss* exception
Iexc	*instruction_access_exception* exception

The ASI is indicated by one the following abbreviations:

Abbreviation	Meaning
`NUC`	ASI_NUCLEUS*
`PRIM`	Any ASI with PRIMARY translation, except *NO_FAULT
`SEC`	Any ASI with SECONDARY translation, except *NO_FAULT
`PRIM_NF`	ASI_PRIMARY_NO_FAULT*
`SEC_NF`	ASI_SECONDARY_NO_FAULT*
`U_PRIM`	ASI_AS_IF_USER_PRIMARY*
`U_SEC`	ASI_AS_IF_USER_SECONDARY*
`BYPASS`	ASI_PHYS_* and also other ASIs that require the MMU to perform a bypass operation (such as D-cache access).

Note – The `*_LITTLE` versions of the ASIs behave the same as the big-endian versions with regard to the MMU table of operations.

Other abbreviations include `W` for the writable bit, `E` for the side-effect bit, and `P` for the privileged bit.

The following cases are not covered in TABLE 11-13.

- Invalid ASIs or ASIs that have no meaning for the opcodes listed; for example, ASI_PRIMARY_NOFAULT for a store or atomic.
- Access to internal registers other than `LDXA`, `LDDFA`, `STXA`, or `STDFA`. See Chapter 8 “Address Space Identifiers.” The MMU signals a *data_access_exception* trap (`FT` = 08_{16}) for these cases.
- Attempted access using a restricted ASI in non-privileged mode. The MMU signals a *privileged_action* exception for this case.
- An atomic instruction (including 128-bit atomic load) issued to a memory address marked non-cacheable in a physical cache; that is, with the `CP` bit set to 0, including cases in which the D-MMU is disabled. The MMU signals a *data_access_exception* trap (`FT` = 04_{16}) for this case.

- A data access with an ASI other than "(PRIMARY,SECONDARY)_NO_FAULT (_LITTLE)" to a page marked with the NFO bit. The MMU signals a *data_access_exception* trap (FT = 10_{16}) for this case.

TABLE 11-13 D-MMU Table of Operations for Normal ASIs

Condition				**Behavior**				
Opcode	**PRIV mode**	**ASI**	**W**	**TLB Miss**	**E = 0 P = 0**	**E = 0 P = 1**	**E = 1 P = 0**	**E = 1 P = 1**
Load	0	PRIM, SEC	x	Dmiss	OK	Dexc	OK	Dexc
		PRIM_NF, SEC_NF	x	Dmiss	OK	Dexc	Dexc	Dexc
	1	PRIM, SEC, NUC	x	Dmiss	OK	OK	OK	OK
		PRIM_NF, SEC_NF	x	Dmiss	OK	OK	Dexc	Dexc
		U_PRIM, U_SEC	x	Dmiss	OK	Dexc	OK	Dexc
Store or Atomic	0	PRIM, SEC	0	Dmiss	Dprot	Dexc	Dprot	Dexc
			1	Dmiss	OK	Dexc	OK	Dexc
	1	PRIM, SEC, NUC	0	Dmiss	Dprot	Dprot	Dprot	Dprot
			1	Dmiss	OK	OK	OK	OK
		U_PRIM, U_SEC	0	Dmiss	Dprot	Dexc	Dprot	Dexc
			1	Dmiss	OK	Dexc	OK	Dexc
FLUSH	0		x	Dmiss	OK	Dexc	OK	Dexc
	1		x	Dmiss	OK	OK	Dexc	Desc
x	0	BYPASS	x	*privileged_action*				
x	1	BYPASS	x	Bypass				

11.9.9 Internal Registers and ASI Operations

In this section, how to access MMU registers is first described and following is the registers themselves, as follows:

- Context Registers
- D-MMU TLB Tag Access Registers
- D-TLB Data In, Data Access, and Tag Read Registers
- dTSB Tag Target Registers
- dTSB Base Registers
- dTSB Extension Registers
- dTSB 8 KB and 64 KB Pointer and Direct Pointer Registers
- Data Synchronous Fault Status Registers (D-SFSR)
- MMU Data Synchronous Fault Address Register

Following the register descriptions, the Data demap operation is described.

11.9.9.1 Accessing MMU Registers

All internal MMU registers can be accessed directly by the processor through defined ASIs. Several of the registers have been assigned their own ASI because these registers are crucial to the speed of the TLB miss handler. Allowing the use of `%g0` for the address reduces the number of instructions required to perform the access to the alternate space (by eliminating address formation).

For instance, to facilitate an access to the D-cache, the MMU performs a bypass operation.

Caution – A store to an MMU register requires a `MEMBAR #Sync`, `FLUSH`, `DONE`, or `RETRY` before the point that the effect must be visible to load/store/atomic accesses. A `FLUSH`, `DONE`, or `RETRY` is needed before the point that the effect must be visible to instruction accesses, that is, `MEMBAR #Sync` is not sufficient. In either case, one of these instructions must be executed before the next non-internal store or load of any type and on or before the delay slot of a delayed-control transfer instruction of any type. This action is necessary to avoid data corruption.

If the low-order three bits of the VA are nonzero in an `LDXA/STXA` to or from these registers, then a *mem_address_not_aligned* trap occurs. Writes to read-only, reads to write-only, illegal ASI values, or illegal VA for a given ASI can cause a *data_access_exception* trap (`FT` = 08_{16}). (The hardware detects VA violations in only an unspecified lower portion of the virtual address.) TABLE 11-14 describes MMU registers.

TABLE 11-14 MMU Internal Registers and ASI Operations

D-MMU ASI	VA<63:0>	Access	Register or Operation Name
58_{16}	0_{16}	Read-only	dTSB Tag Target Registers
58_{16}	8_{16}	Read/Write	Primary Context Register
58_{16}	10_{16}	Read/Write	Secondary Context Register
58_{16}	18_{16}	Read/Write	Data Synchronous Fault Status Registers (`D-SFSR`)
58_{16}	20_{16}	Read-only	Data Synchronous Fault Address Register (`D-SFAR`)
58_{16}	28_{16}	Read/Write	dTSB Base Registers
58_{16}	30_{16}	Read/Write	D-TLB Tag Access Registers
58_{16}	38_{16}	Read/Write	Virtual Watchpoint Address
58_{16}	40_{16}	Read/Write	Physical Watchpoint Address
58_{16}	48_{16}	Read/Write	dTSB Primary Extension Registers
58_{16}	50_{16}	Read/Write	dTSB Secondary Extension Register
58_{16}	58_{16}	Read/Write	dTSB Nucleus Extension Registers
59_{16}	0_{16}	Read-only	dTSB 8 KB Pointer Registers

TABLE 11-14 MMU Internal Registers and ASI Operations *(Continued)*

D-MMU ASI	VA<63:0>	Access	Register or Operation Name
$5A_{16}$	0_{16}	Read-only	dTSB 64 KB Pointer Registers
$5B_{16}$	0_{16}	Read-only	dTSB Direct Pointer Register
$5C_{16}$	0_{16}	Write-only	D-TLB Data In Registers
$5D_{16}$	0_{16} – $20FF8_{16}$	Read/Write	D-TLB Data Access Registers
$5D_{16}$	40000_{16} – $60FF8_{16}$	Read/Write	D-TLB CAM Diagnostic Register
$5E_{16}$	0_{16} – $20FF8_{16}$	Read-only	D-TLB Tag Read Registers
$5F_{16}$	See 11.9.9.14	Write-only	D-MMU Demap Operations

Note – `TSB_Hash` field is implemented in TSB Extension Register.

11.9.9.2 Context Registers

The Primary Context Register is shared by the I-MMU and the D-MMU and resides in the MMU. The Primary Context Register is illustrated in FIGURE 11-9; `PContext` is the context identifier for the primary address space.

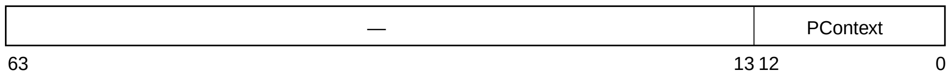


FIGURE 11-9 D-MMU Primary Context Register

The Secondary Context Register is illustrated in FIGURE 11-10; `SContext` is the context identifier for the secondary address space.

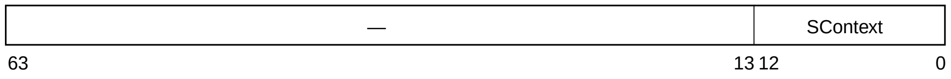


FIGURE 11-10 D-MMU Secondary Context Register

The Nucleus Context Register is hardwired to zero, as illustrated in FIGURE 11-11.

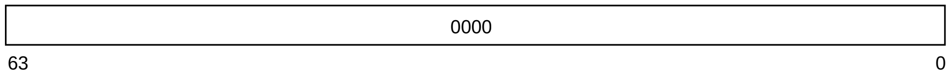


FIGURE 11-11 D-MMU Nucleus Context Register

11.9.9.3 Data MMU TLB Tag Access Registers

In each MMU, the Tag Access Register is used as a temporary buffer for writing the TLB entry tag information. The Tag Access Register holds the tag portion, and the Data In or Data Access Register holds the data being accessed.

The Tag Access Register can be updated during either of the following operations:

- When the MMU signals a trap due to a miss, exception, or protection

  The MMU hardware automatically writes the missing VA and the appropriate context into the Tag Access Register to facilitate formation of the TSB Tag Target Register. One exception is that after a *data_access_exception*, the contents of the `Context` field of the D-MMU Tag Access Register are undefined.

- An ASI write to the Tag Access Register

  Before an ASI store to the TLB Data Access Registers, the operating system must set the Tag Access Register to the values desired in the TLB entry. Note that an ASI store to the TLB Data In Register for automatic replacement also uses the Tag Access Register, but typically the value written into the Tag Access Register by the MMU hardware is appropriate.

Note – Any update to the Tag Access Registers immediately affects the data that are returned from subsequent reads of the TSB Tag Target and TSB Pointer Registers.

The TLB Tag Access Register fields are defined in TABLE 11-15 and illustrated in FIGURE 11-12.

TABLE 11-15 D-MMU TLB Tag Access Registers

Bit(s)	Field	Type	Description
63:13	`VA`	RW	The 51-bit virtual page number.
12:0	`Context`	RW	The 13-bit context identifier. This field reads 0 when there is no associated context with the access. Its contents in the D-MMU are undefined after a *data_access_exception*.

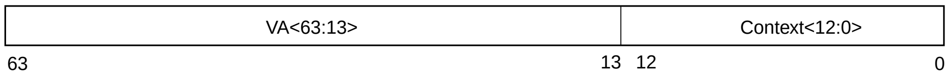


FIGURE 11-12 D-MMU TLB Tag Access Registers Format

Caution – When the D-MMU causes a trap due to a protection violation or other exception, software should use the context number from `D-SFSR.CT` instead of the `Context` field of the D-TLB Tag Access Register.

11.9.9.4 Data TLB Data In, Data Access, and Tag Read Registers

Access to the TLB is complicated because of the need to provide an atomic write of a TLB entry data item (tag and data) that is larger than 64 bits, the need to replace entries automatically through the TLB entry replacement algorithm as well as to provide direct diagnostic access, and the need for hardware assist in the TLB miss handler.

TABLE 11-2 on page 11-259 shows when loads and stores update the Tag Access Registers.

TABLE 11-16 shows how the Tag Read, Tag Access, Data In, and Data Access Registers interact to provide atomic reads and writes to the TLBs.

TABLE 11-16 MMU TLB Access Summary

Software Operation		Effect on MMU Physical Registers		
Load/Store	**Register**	**TLB Tag Array**	**TLB Data Array**	**Tag Access Register**
Load	Tag Read	Contents returned. Entry specified by STXA's access.	No effect	No effect
	Tag Access	No effect	No effect	Contents returned
	Data In	Trap with *data_access_exception*.		
	Data Access	No effect	Contents returned. Entry specified by STXA's access.	No effect
Store	Tag Read	Trap with *data_access_exception*.		
	Tag Access	No effect	No effect	Written with store data
	Data In	TLB entry determined by replacement policy written with contents of Tag Access Register	TLB entry determined by replacement policy written with store data	No effect
	Data Access	TLB entry specified by STXA address written with contents of Tag Access Register	TLB entry specified by STXA address written with store data	No effect
TLB miss		No effect	No effect	Written with VA and context of access

An ASI load from the TLB Tag Read Register initiates an internal read of the tag portion of the specified TLB entry.

11.9.9.5 Data In and Data Access Registers

The Data In and Data Access Registers are the means of reading and writing the TLB for all operations. The TLB Data In Register is used for TLB miss handler automatic replacement writes. The TLB Data Access Register is used for operating system and diagnostic directed writes (writes to a specific TLB entry).

An ASI load from the TLB Data Access Register initiates an internal read of the data portion of the specified TLB entry.

ASI loads from the TLB Data In Register are not supported.

An ASI store to the TLB Data In Register initiates an automatic atomic replacement of the TLB entry pointed to by an internal register that is updated by a proprietary replacement algorithm. The TLB data and tag are formed as in the case of an ASI store to the TLB Data Access Register.

Caution – Stores to the Data In Register are not guaranteed to replace the previous TLB entry, causing a fault. In particular, to change an entry's attribute bits, software must explicitly demap the old entry before writing the new entry; otherwise, a multiple match error condition can result.

Both the TLB Data In Register and the TLB Data Access Register use the TTE format shown in FIGURE 11-3 on page 11-248. Refer to the description of the TTE data in Section 11.2 "Translation Table Entry" on page 11-248 for a complete description of the data fields.

Writes to the TLB Data In Register require the virtual address to be set to 0.

The format of the TLB Data Access Register virtual address is illustrated in FIGURE 11-13 and described in TABLE 11-17.

TABLE 11-17 TLB Data Access Register

Bit(s)	Field	Type	Description
63:19	—	—	*Reserved.*
18	0	—	Set to zero.
17:16	TLB #	RW	The TLB to access, as defined below.

TLB #	TLB Type	# of Entries
0	Fully associative 64 KB, 512 KB, and 4 MB page size and locked pages	16
2	2-way associative	512 (D-MMU) (dt512-0)
3	2-way associative	512 (D-MMU) (dt512-1)

Bit(s)	Field	Type	Description
15:12	—	—	*Reserved.*
11:3	TLB Entry	RW	The TLB entry number to be accessed, in the range 0 – 511. Not all TLBs will have all 512 entries. All TLBs regardless of size are accessed from 0 to $N - 1$, where N is the number of entries in the TLB.
2:0	0	—	Set to zero.

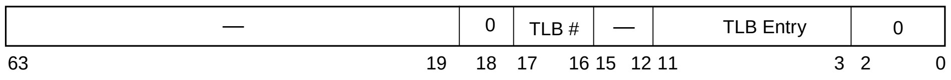


FIGURE 11-13 D-MMU TLB Data Access Address Format

11.9.9.6 Data MMU TLB Tag Read Register

The format for the Tag Read Register virtual address is described in TABLE 11-18 and illustrated in FIGURE 11-14.

TABLE 11-18 D-MMU TLB Tag Read Registers

Bit(s)	Field	Type	Description
63:13	VA	RW	The 51-bit virtual page number. In the fully associative TLB, page offset bits for larger page sizes are stored in the TLB; that is, VA<15:13>, VA<18:13>, and VA<21:13> for 64 KB, 512 KB, and 4 MB pages, respectively. These values are ignored during normal translation. When read, an implementation will return either 0 or the value previously written to them.
12:0	Data `Context`	RW	The 13-bit context identifier.

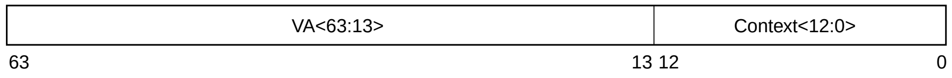


FIGURE 11-14 D-MMU TLB Tag Read Registers

11.9.9.7 Data MMU TLB Tag Access Register

An ASI store to the TLB Data Access or Data In Register initiates an internal atomic write to the specified TLB entry. The TLB entry data is obtained from the store data, and the TLB entry tag is obtained from the current contents of the TLB Tag Access Register.

11.9.9.8 Data TSB Tag Target Registers

The Data Translation Storage Buffer (dTSB) Tag Target Registers are simply bit-shifted versions of the data stored in the Data Tag Access Registers. Since the Data Tag Access Register is updated on an D-TLB miss, the Data Tag Target Registers appear to software to be updated on an D-TLB miss. The D-MMU Tag Target Register is described in TABLE 11-19 and illustrated in FIGURE 11-15.

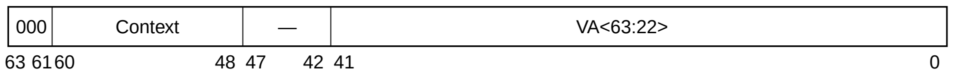


FIGURE 11-15 MMU Tag Target Registers

TABLE 11-19 MMU Tag Target Registers

Bit(s)	Field	Type	Description
63:61	000	—	Set to 000.
60:48	`Context`<12:0>	RW	The context associated with the missing virtual address.
47:42	—	—	*Reserved.*
41:0	VA<63:22>	RW	The most significant bits of the missing virtual address.

11.9.9.9 Data TSB Base Registers

The TSB registers provide information for the hardware formation of TSB pointers and tag target, to assist software in quickly handling TLB misses. If the TSB concept is not employed in the software memory management strategy and therefore the Pointer and Tag Access Registers are not used, then the TSB registers need not contain valid data.

The TSB register is illustrated in FIGURE 11-16 and described in TABLE 11-20.

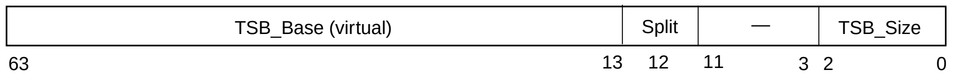


FIGURE 11-16 MMU Data TSB Registers

TABLE 11-20 TSB Register Description

Bit(s)	Field	Type	Description
63:13	Data `TSB_Base`	RW	Provides the base virtual address of the TSB. Software must ensure that the TSB base is aligned on a boundary equal to the size of the TSB or both TSBs in the case of a split TSB.

TABLE 11-20 TSB Register Description *(Continued)*

Bit(s)	Field	Type	Description
12	Split	RW	When Split = 1, the TSB 64 KB pointer address is calculated assuming separate (but abutting and equally sized) TSB regions for the 8 KB and the 64 KB TTEs. In this case, TSB_Size refers to the size of each TSB. The TSB 8 KB pointer address calculation is not affected by the value of the Split bit. When Split = 0, the TSB 64 KB pointer address is calculated assuming that the same lines in the TSB are shared by 8 KB and 64 KB TTEs, called a "common TSB" configuration. **Caution:** In the "common TSB" configuration (TSB.Split = 0), 8 KB and 64 KB page TTEs can conflict unless the TLB miss handler explicitly checks the TTE for page size. Therefore, do not use the common TSB mode in an optimized handler. For example, suppose an 8 KB page at VA = 2000_{16} and a 64 KB page at VA = 10000_{16} both exist — a legal situation. These both map to the second TSB line (line 1) and have the same VA tag of 0. Therefore, there is no way for the miss handler to distinguish these TTEs by the TTE tag alone, and unless the miss handler checks the TTE data, it may load an incorrect TTE.
11:3	—	—	Reserved.
2:0	Data TSB_Size	RW	The UltraSPARC III Cu processor implements a 3-bit TSB_Size field. The TSB_Size field provides the size of the TSB as follows: • The number of entries in the TSB (or each TSB if split) = 512×2^{TSB_Size}. • The number of entries in the TSB ranges from 512 entries at TSB_Size = 0 (8 KB common TSB, 16 KB split TSB), to 64K entries at TSB_Size = 7 (1 MB common TSB, 2 MB split TSB). **Note:** Any update to the TSB register immediately affects the data that are returned from later reads of the Tag Target and TSB Pointer Registers.

11.9.9.10 Data TSB Extension Registers

The TSB Extension Registers provide information for the hardware formation of TSB pointers and tag target, to assist software in handling TLB misses quickly. If the TSB concept is not employed in the software memory management strategy and therefore the pointer and Tag Access Registers are not used, then the TSB Extension Registers need not contain valid data.

The TSB registers are defined as follows in FIGURE 11-17.

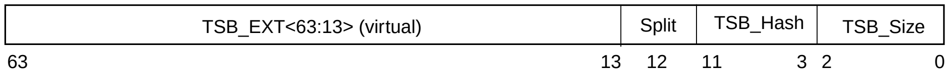


FIGURE 11-17 MMU Data TSB Extension Registers

In the UltraSPARC III Cu processor, `TSB_Hash` (bits 11:3 of the Extension Registers) are exclusive-ORed with the calculated TSB offset to provide a "hash" into the TSB. Changing the `TSB_Hash` field on a per-process basis minimizes the collision of TSB entries between different processes.

The register field definitions are the same as for Data TSB Base Registers. The field can be used either as a `TSB_Hash`, which is a representation of the context that generated the TLB miss, or as an extension to the `TSB_size` field, depending on the implementation. In the latter case, TSB pointer generation logic must incorporate the context ID into the process of TSB pointer generation, as described in "TSB Pointer Formation" on page 254.

There are three TSB Extension Registers, one for each of the virtual address spaces (Primary, Secondary, Nucleus); see TABLE 8-4 on page 8-193 for the ASI and VA of each register.

When a D-TLB miss occurs, an appropriate TSB Extension Register is selected and XORed either with the dTSB Register or with context ID, depending on the implementation. The result is then used to form a TSB pointer, as described in "TSB Pointer Formation" on page 254.

11.9.9.11 Data TSB 8 KB and 64 KB Pointer and Direct Pointer Registers

The dTSB 8 KB and 64 KB registers are provided as an aid to software in determining the location of the missing or trapping TTE in the software-maintained TSB. The TSB 8 KB and 64 KB Pointer Registers provide the possible locations of the 8 KB and 64 KB TTE, respectively.

As a fine point, the bit that controls selection of 8 KB or 64 KB address formation for the Direct Pointer Register is a state bit in the D-MMU that is updated during a *fast_data_access_protection* exception. It records whether the page that hit in the TLB was a 64 KB page or a non-64 KB page, in which case, 8 KB is assumed.

The registers are illustrated in FIGURE 11-18, *where* VA<63:4> is the full virtual address of the TTE in the TSB, as determined by the MMU hardware, and is described in "Hardware Support for TSB Access" on page 253.

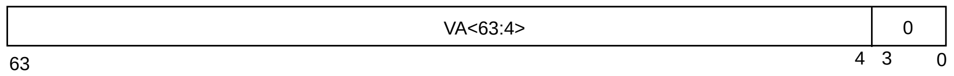


FIGURE 11-18 D-MMU TSB 8 KB/64 KB Pointer and D-MMU Direct Pointer Register

TSB 8 KB and 64 KB Pointer Registers

The TSB Pointer Registers are implemented as a recorder of the current data stored in the Tag Access Register and the TSB Extension Register. If the Tag Access Register or TSB Extension Register is updated through a direct software write (through an STXA instruction), then the values in the Pointer Registers will be updated as well.

Direct Pointer Register

The Direct Pointer Register is mapped by hardware to either the 8 KB or 64 KB Pointer Register in the case of a *fast_data_access_protection* exception according to the known size of the trapping TTE. In the case of a 512 KB or 4 MB page miss, the Direct Pointer Register returns the pointer as if the fault were from an 8 KB page.

11.9.9.12 Data Synchronous Fault Status Registers (D-SFSR)

The D-MMU maintains its own SFSR Register. The SFSR is illustrated in FIGURE 11-19 and described in TABLE 11-21.

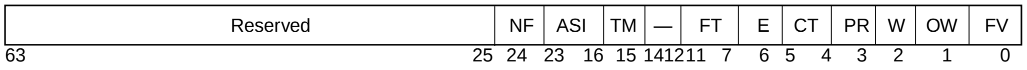


FIGURE 11-19 MMU Data Synchronous Fault Status Registers (I-SFSR, D-SFSR)

TABLE 11-21 D-SFSR Bit Description

Bit(s)	**Field**	**Type**	**Description**
63:25	—		Reserved.
24	NF	RW	Set in the D-MMU if the faulting instruction was a non-faulting load (a load to ASI_NOFAULT).
23:16	ASI	RW	Records the 8-bit ASI associated with the faulting instruction. This field is valid for both D-MMU and I-MMU SFSRs and for all traps in which the FV bit is set. A trapping alternate space load or store sets the ASI field to the ASI the instruction attempted to reference. A trapping non-alternate-space load or store sets ASI to ASI_PRIMARY if PSTATE.CLE = 0 or to ASI_PRIMARY_LITTLE if PSTATE.CLE = 1. A *mem_address_not_aligned* trap caused by a JMPL or RETURN either does not set DSFSR.ASI or sets it as would a trapping non-alternate-space load or store.
15	TM	RW	D-TLB miss.
14:12	—		Reserved.

TABLE 11-21 D-SFSR Bit Description *(Continued)*

Bit(s)	Field	Type	Description
11:7	FT	RW	Specifies the exact condition that caused the recorded fault, according to TABLE 11-22 following this table. In the D-MMU, the Fault Type field is valid only for *data_access_exception* faults; there is no ambiguity in all other MMU trap cases. Note that the hardware does not priority-encode the bits set in the fault type (FT) field; that is, multiple bits can be set. In particular, the following ASI stores could set both the 01_{16} and 08_{16} Fault Type bits (the page is privileged, as well as storing to a read-only ASI): `stda %g0, [%g4]ASI_PRIMARY_NO_FAULT` `stda %g0, [%g4]ASI_SECONDARY_NO_FAULT` `stda %g0, [%g4]ASI_PRIMARY_NO_FAULT_LITTLE` `stda %g0, [%g4]ASI_SECONDARY_NO_FAULT_LITTLE`
6	E	RW	Side-effect bit. Associated with the faulting data access or flush instruction. Set by translating ASI accesses (see Chapter 8 “Address Space Identifiers”) that are mapped by the TLB with the E bit set and bypass ASIs 15_{16} and $1D_{16}$. Other cases that update the SFSR (including bypass or internal ASI accesses) set the E bit to 0.
5:4	CT	RW	Context Register selection, as described below. The context is set to 11_2 when the access does not have a translating ASI.
3	PR	RW	Privilege bit. Set if the faulting access occurred while in privileged mode. This field is valid for all traps in which the FV bit is set.
2	W	RW	Write bit. Set if the faulting access indicated a data write operation (a store or atomic load/store instruction).
1	OW	RW	Overwrite bit. When the MMU detects a fault, the Overwrite bit is set to 1 if the Fault Valid bit has not been cleared from a previous fault; otherwise, it is set to 0.
0	FV	RW	Fault Valid bit. Set when the MMU detects a fault; it is cleared only on an explicit ASI write of 0 to the SFSR. This bit is not set on an MMU miss. Therefore, overwrites of MMU misses cannot be detected. When the Fault Valid bit is not set, the values of the remaining fields in the SFSR and SFAR are undefined for traps other than an MMU miss.

Context ID	I-MMU Context	D-MMU Context
00	Primary	Primary
01	Reserved	Secondary
10	Nucleus	Nucleus
11	Reserved	Reserved

TABLE 11-22 describes the SFSR fault type field (FT<11:7>).

TABLE 11-22 MMU Synchronous Fault Status Register FT (Fault Type) Field

Data	FT[6:0]	Fault Type
Data	01_{16}	Privilege violation.
D	02_{16}	Non-faulting load instruction to page marked with E bit. This bit is 0 for internal ASI accesses.
D	04_{16}	Atomic (including 128-bit atomic load) to page marked non-cacheable.
D	08_{16}	Illegal LDA/STA ASI value, VA, RW, or size. Does not include cases where 02_{16} and 04_{16} are set.
D	10_{16}	Access other than non-faulting load to page marked NFO. This bit is 0 for internal ASI accesses.

Note – A *fast_data_access_MMU_miss* trap causes the D-SFSR and the D-SFAR to be overwritten without setting either the OW or the FV bits.

The SFSR and the Tag Access Registers both maintain state concerning a previous translation causing an exception. The update policy for the SFSR and the Tag Access Registers is shown in TABLE 11-2 on page 11-259.

11.9.9.13 Synchronous Fault Addresses

This section describes how the D-MMU obtains a fault address.

D-MMU Fault Address

The Data Synchronous Fault Address Register contains the virtual memory address of the fault recorded in the D-MMU Synchronous Fault Status Register. The D-SFAR can be thought of as an additional field of the D-SFSR.

The D-SFAR register is illustrated in FIGURE 11-20, where *Fault Address* is the virtual address associated with the translation fault recorded in the D-SFSR; the field is set on an MMU miss fault or when the D-SFSR Fault Valid (FV) bit is set.

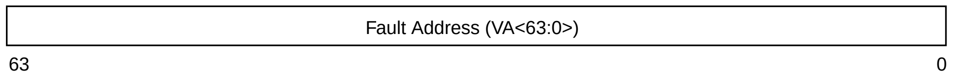


FIGURE 11-20 MMU Data Synchronous Fault Address Register (D-SFAR)

11.9.9.14 Data MMU Demap

Demap is an MMU operation, not an MMU register. Demap removes selected entries from the TLBs.

Note – A store to a D-MMU Register requires a `MEMBAR #Sync`, `FLUSH`, `DONE`, or `RETRY` before the point that the effect must be visible to load/store/atomic accesses. A `FLUSH`, `DONE`, or `RETRY` is needed before the point that the effect must be visible to instruction accesses, that is, `MEMBAR #Sync` is not sufficient. In either case, one of these instructions must be executed before the next non-internal store or load of any type and on or before the delay slot of a delayed-control transfer instruction of any type. This action is necessary to avoid data corruption.

Three types of demap operations are provided:

- **Demap page —** Removes any TLB entry that matches exactly the specified virtual page and context number. It is illegal to have more than one TLB entry per page.

  Demap page may, in fact, remove more than one TLB entry in the condition of a multiple TLB match, but this is an error condition of the TLB and has undefined results.
- **Demap context —** Removes any TLB entries that match the specified context identifier.
- **Demap all —** Removes all of the TLB entries from the TLB except for locked entries.

Demap is initiated by an `STXA` with ASI $5F_{16}$ for D-MMU demap. It removes TLB entries from an on-chip TLB. No bus-based demap is supported. The demap address format is illustrated in FIGURE 11-21 and described in TABLE 11-23.

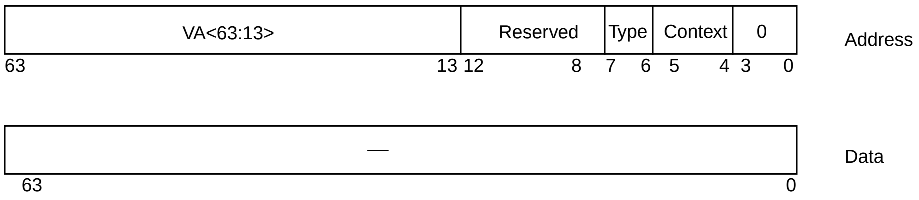


FIGURE 11-21 MMU Demap Operation Address and Data Formats

TABLE 11-23 Demap Address Format

Bit(s)	Field	Type	Description
63:13	VA<63:13>	RW	The virtual page number of the TTE to be removed from the TLB for Demap Page.
12:8	*Reserved*		This field is ignored by hardware.
7:6	`Type`	RW	The type of demap operation, as described below: **Type Field** / **Demap Operation** 0 / Demap page — see page 290 1 / Demap context — see page 290 2 / Demap all — see page 290 3 / Reserved — Ignored
5:4	`Context` ID	RW	Context Register selection, as described below. Use of the reserved value causes the demap to be ignored. **Context ID Field** / **Context Used in Demap** 00 / Primary 01 / Secondary (D-MMU only) 10 / Nucleus 11 / Reserved
3:0	0		Set to zero.

A demap operation does not invalidate the TSB in memory. Software must modify the appropriate TTEs in the TSB before initiating a demap operation.

Except for Demap All, the demap operation does not depend on the value of any entry's lock bit. A demap operation demaps both locked entries and unlocked entries.

The demap operation produces no output.

The following are Data demap page types:

- **Data Demap Page (`Type` = 0).** Demap Page removes the TTE (from the specified TLB), matching the specified virtual page number and Context Register. The match condition with regard to the global bit is the same as a normal TLB access; that is, if the global bit is set, the contexts do not need to match.

  Virtual page offset bits 15:13, 18:13, and 21:13 for 64 KB, 512 MB, and 4 MB page TLB entries, respectively, do not participate in the match for that entry. This is the same condition as for a translation match.

Note – Each Demap Page operation removes only one TLB entry. A demap of a 64 KB, 512 KB, or 4 MB page does not demap any smaller page within the specified virtual address range.

- **Data Demap Context (Type = 1).** Demap Context removes from the TLB all TTEs having the specified context. If the TTE Global bit is set, then the TTE is not removed. VA is ignored for this operation.
- **Data Demap All (Type = 2).** Demap All removes all TTEs that do not have the lock bit set. VA and `Context` are ignored for this operation.

11.9.9.15 Data TLB CAM Diagnostic Register

Accesses to the TLB Diagnostic Register require the virtual address to be set to access a TLB and TLB entry. The virtual address format of the TLB Diagnostic Register virtual address is described in TABLE 11-24 and illustrated in FIGURE 11-22.

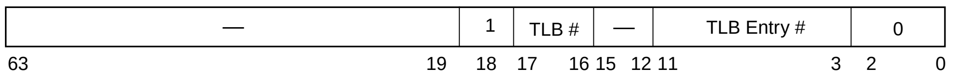


FIGURE 11-22 MMU TLB Diagnostic Access Virtual Address

TABLE 11-24 TLB Diagnostic Register Virtual Address Format

Bit(s)	Field	Type	Description
63:19	—	—	*Reserved.*
18	1	—	Set to one. Selects CAM Diagnostic mode for the t16 TLB only (the other two TLBs do not employ CAM registers).
17:16 (D)	TLB #	RW	The number of the TLB to access, as follows:

TLB	TLB Type	Entries
0	Fully associative 64 KB, 512 KB, and 4 MB page size and locked pages	16 (dt16)
2	2-way associative 8 KB, 64 KB, 512 KB, and 4 MB locked/unlocked page size	512 (dt512_0)
3	2-way associative 8 KB, 64 KB, 512 KB, and 4 MB locked/unlocked page size	512 (dt512_1)

TABLE 11-24 TLB Diagnostic Register Virtual Address Format *(Continued)*

Bit(s)	Field	Type	Description
15:12	—	—	*Reserved.*
11:3	TLB Entry #	RW	The number of the TLB entry to be accessed, in the range 0 – 511. Not all TLBs will have all 512 entries. All TLBs regardless of size are accessed from 0 to $N - 1$, where N is the number of entries in the TLB.
2:0	0	—	Set to zero.

The format for the CAM Diagnostic Register is described in TABLE 11-25 and illustrated in FIGURE 11-23.

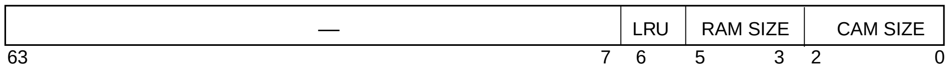


FIGURE 11-23 D-MMU TLB CAM Diagnostic Registers

TABLE 11-25 CAM Diagnostic Register

Bit(s)	Field	Type	Description
63:7	—	—	*Reserved.*
6	LRU	RW	The LRU bit in the CAM, read-write.
5:3	RAM SIZE	R	The 3-bit page size field from the RAM, read-only.
2:0	CAM SIZE	R	The 3-bit page size field from the CAM, read-only.

An ASI store to the TLB CAM Diagnostic Register initiates an internal atomic write to the specified TLB entry. The TLB RAM and CAM entry data are obtained from the store data.

An ASI load from the TLB CAM Diagnostic Register initiates an internal read of the data portion of the specified TLB RAM and CAM entry.

11.9.10 D-MMU Bypass

In a bypass access, the D-MMU sets the physical address equal to the truncated virtual address; that is, the low-order bits of the virtual address are passed through without translation as the physical address. The physical page attribute bits are set as shown in TABLE 11-26.

TABLE 11-26 Bypass Attribute Bits

	Attribute Bits							
ASI	**CP**	**IE**	**CV**	**E**	**P**	**W**	**NFO**	**Size**
ASI_PHYS_USE_EC ASI_PHYS_USE_EC_LITTLE	1	0	0	0	0	1	0	8 KB
ASI_PHYS_BYPASS_EC_WITH_EBIT ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE	0	0	0	1	0	1	0	8 KB

Compatibility Note – The virtual address is wider than the physical address; thus, there is no need to use multiple ASIs to fill in the high-order physical address bits, as is done in SPARC V8 machines.

11.10 Instruction Translation Lookaside Buffer

In the I-MMU, a 128-entry, 2-way associative TLB (it128) is used exclusively for 8 KB, unlocked page translations, and a 16-entry fully associative TLB is used for 64 KB, 512 KB, and 4 MB page translations and locked pages of all four sizes.

11.10.1 I-TLB Access Operation

When an instruction fetch access is issued, its VA and Context are presented to the I-MMU. Both I-TLBs (it128 and it16) are accessed in parallel. The two possible 8K page entries in the it128 are checked to determine a hit.

There are two possible Context numbers active in the CPU, Primary (`PContext` field in ASI_PRIMARY_CONTEXT_REG), and Nucleus (default to Context = 0). When TL = 0 primary context number is used and when TL > 0, nucleus context number is used.

Since two I-TLBs are being accessed in parallel, software must guarantee that there are no duplicate (stale) entry hits. Most of this responsibility lies in software (operating system) with the hardware providing some assistance to support full software control. A set of rules on I-TLB replacement, demap and context switch must be followed to maintain consistent and correct behavior.

11.10.2 I-TLB Automatic Replacement

An I-TLB miss fast trap handler utilizes the automatic (hardware) replacement write using store ASI_ITLB_DATA_IN_REG.

When I-TLB miss, or DATA_ACCESS_EXCEPTION, or FAST_DATA_ACCESS_PROTECTION is detected, hardware automatically saves the missing VA and context to the Tag Access Register (ASI_ IMMU_TAG_ACCESS). To ease indexing of the it128 when the TTE data is presented (via `STXA` ASI_ITLB_DATA_IN_REG), the missing page size information of it128 is captured into a new Extension Register, called ASI_ IMMU_TAG_ACCESS_EXT.

CODE EXAMPLE 11-3 shows the hardware I-TLB replacement algorithm.

CODE EXAMPLE 11-3 I-TLB Hardware Replacement Algorithm

```
if (TTE to fill is a locked page, i.e., L bit is set) {
    fill TTE to it16;
} else {
    if (TTE’s Size == 8K) {
        if (one of the 2 same-index entries is invalid) {
            fill TTE to an invalid entry
        } else if (no entry is valid |
                   both entries are valid) {
            case (LFSR[0]) {
                0: fill TTE to it128 way0;
                1: fill TTE to it128 way1;
            }
        }
    } else {
         fill TTE to it16;
    }
}
```

11.10.3 I-TLB Access Summary

TABLE 11-27 lists the I-MMU TLB access summary.

TABLE 11-27 I-MMU TLB Access Summary

Software Operation		**Effect on MMU Physical Registers**			
Load/ Store	**Register**	**TLB Tag Array**	**TLB Data Array**	**Tag Access**	**SFSR**
Load	Tag Read	Contents returned. From entry specified by LDXA's access	No effect	No effect	No effect
	Tag Access	No effect	No effect	Contents returned.	No effect
	Data In	Trap with *data_access_exception*			
	Data Access	No effect	Contents returned. From entry specified by LDXA's access	No effect	No effect
	SFSR	No effect	No effect	No effect	Contents returned
	SFAR	No effect	No effect	No effect	No effect

TABLE 11-27 I-MMU TLB Access Summary *(Continued)*

Software Operation		**Effect on MMU Physical Registers**			
Load/ Store	**Register**	**TLB Tag Array**	**TLB Data Array**	**Tag Access**	**SFSR**
Store	Tag Read	Trap with *data_access_exception*			
	Tag Access	No effect	No effect	Written with store data	No effect
	Data In	TLB entry determined by replacement policy written with contents of Tag Access Register	TLB entry determined by replacement policy written with store data	No effect	No effect
	Data Access	TLB entry specified by STXA address written with contents of Tag Access Register	TLB entry specified by STXA address written with store data	No effect	No effect
	SFSR	No effect	No effect	No effect	Written with store data
	SFAR	No effect	No effect	No effect	No effect
TLB miss		No effect	No effect	Written with VA and context of access	Written with fault status of faulting instruction and page sizes at faulting context for two 2-way set associative TLB

11.10.4 I-MMU Operation Summary

The behavior of the I-MMU is summarized in TABLE 11-28.

TABLE 11-28 I-MMU Table of Operations for Normal ASIs

Condition	**Behavior**		
PRIV mode	**TLB Miss**	**P = 0**	**P = 1**
0	Imiss	OK	Iexc
1	Imiss	OK	OK

11.10.5 Internal Registers and ASI Operations

In this section, how to access MMU registers is first described followed by the registers themselves and are described as follows:

- I-MMU TLB Tag Access Registers
- I-TLB Data In, Data Access, and Tag Read Registers
- I-TSB Tag Target Registers
- I-TSB Base Registers
- I-TSB Extension Registers
- I-TSB 8 KB and 64 KB Pointer and Direct Pointer Registers
- Instruction Synchronous Fault Status Registers (`I-SFSR`, `D-SFSR`)
- MMU Synchronous Fault Address Register

Following the register descriptions, the Instruction demap operation is described.

11.10.5.1 Instruction MMU TLB Tag Access Registers

In each MMU, the Tag Access Register is used as a temporary buffer for writing the TLB Entry tag information. The Tag Access Register holds the tag portion, and the Data In or Data Access Register holds the data being accessed.

The Tag Access Register can be updated during either of the following operations:

- When the MMU signals a trap due to a miss, exception, or protection

  The MMU hardware automatically writes the missing VA and the appropriate context into the Tag Access Register to facilitate formation of the TSB Tag Target Register. One exception is that after a *data_access_exception*, the contents of the `Context` field of the I-MMU Tag Access Register are undefined.

- An ASI write to the Tag Access Register

  Before an ASI store to the TLB Data Access Registers, the operating system must set the Tag Access Register to the values desired in the TLB Entry. Note that an ASI store to the TLB Data In Register for automatic replacement also uses the Tag Access Register, but typically the value written into the Tag Access Register by the MMU hardware is appropriate.

Note – Any update to the Tag Access Registers immediately affects the data that is returned from subsequent reads of the TSB Tag Target and TSB Pointer Registers.

The TLB Tag Access Register fields are defined in TABLE 11-29 and illustrated in FIGURE 11-12.

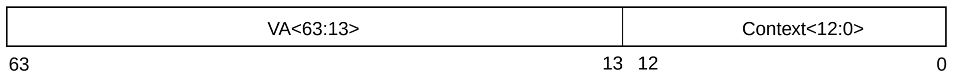


FIGURE 11-24 I-MMU TLB Tag Access Registers

TABLE 11-29 I-MMU Tag Access Register

Bit	Field	Type	Description
63:13	VA	RW	The 51-bit virtual page number.
12:0	`Context`	RW	The 13-bit context identifier. This field reads 0 when there is no associated context with the access. Its contents in the I-MMU are undefined after a *data_access_exception*.

Caution – When the I-MMU causes a trap due to a protection violation or other exception, software should use the context number from `I-SFSR.CT` instead of from the `Context` field of the I-TLB Tag Access Register.

11.10.5.2 Instruction TLB Data In, Data Access, and Tag Read Registers

Access to the TLB is complicated because of the need to provide an atomic write of a TLB entry data item (tag and data) that is larger than 64 bits, the need to replace entries automatically through the TLB entry replacement algorithm as well as to provide direct diagnostic access, and the need for hardware assist in the TLB miss handler.

TABLE 11-2 on page 11-259 shows when loads and stores update the Tag Access Registers.

TABLE 11-30 shows how the Tag Read, Tag Access, Data In, and Data Access Registers interact to provide atomic reads and writes to the TLBs.

TABLE 11-30 MMU TLB Access Summary

Software Operation		Effect on MMU Physical Registers		
Load/Store	**Register**	**TLB Tag Array**	**TLB Data Array**	**Tag Access Register**
Load	Tag Read	Contents returned. Entry specified by STXA's access.	No effect	No effect
	Tag Access	No effect	No effect	Contents returned
	Data In	Trap with *data_access_exception*.		
	Data Access	No effect	Contents returned. Entry specified by STXA's access.	No effect
Store	Tag Read	Trap with *data_access_exception*.		
	Tag Access	No effect	No effect	Written with store data
	Data In	TLB entry determined by replacement policy written with contents of Tag Access Register	TLB entry determined by replacement policy written with store data	No effect
	Data Access	TLB entry specified by STXA address written with contents of Tag Access Register	TLB entry specified by STXA address written with store data	No effect
TLB miss		No effect	No effect	Written with VA and context of access

An ASI load from the TLB Tag Read Register initiates an internal read of the tag portion of the specified TLB entry.

11.10.5.3 Data In and Data Access Registers

The Data In and Data Access Registers are the means of reading and writing the TLB for all operations. The TLB Data In Register is used for TLB miss handler automatic replacement writes. The TLB Data Access Register is used for operating system and diagnostic directed writes (writes to a specific TLB entry).

An ASI load from the TLB Data Access Register initiates an internal read of the data portion of the specified TLB entry.

ASI loads from the TLB Data In Register are not supported.

An ASI store to the TLB Data In Register initiates an automatic atomic replacement of the TLB Entry pointed to by an internal register that is updated by a proprietary replacement algorithm. The TLB data and tag are formed as in the case of an ASI store to the TLB Data Access Register.

Caution – Stores to the Data In Register are not guaranteed to replace the previous TLB entry, causing a fault. In particular, to change an entry's attribute bits, software must explicitly demap the old entry before writing the new entry; otherwise, a multiple match error condition can result.

Both the TLB Data In Register and the TLB Data Access Register use the TTE format shown in FIGURE 11-3 on page 11-248. Refer to the description of the TTE data in "Translation Table Entry" on page 248 for a complete description of the data fields.

Writes to the TLB Data In Register require the virtual address to be set to 0.

The format of the TLB Data Access Register virtual address is illustrated in FIGURE 11-25 and described in TABLE 11-31.

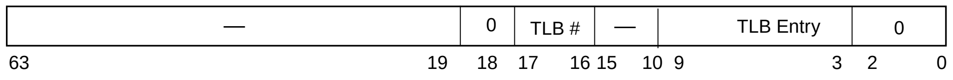


FIGURE 11-25 I-MMU TLB Data Access Address

TABLE 11-31 TLB Data Access Register

Bit(s)	Field	Type	Description
63:19	—	—	*Reserved.*
18	0	—	Set to zero.
17:16	TLB #	RW	The TLB to access, as defined below.

TLB #	TLB Type	# of Entries
0	Fully associative 64 KB, 4 MB, and 512 KB page size and locked pages	16
2	2-way associative, 8 KB page size	128 (I-MMU)

TABLE 11-31 TLB Data Access Register *(Continued)*

Bit(s)	Field	Type	Description
15:10	—	—	*Reserved.*
9:3	TLB Entry	RW	For it128, the TLB entry number to be accessed, in the range 0 – 127. Therefore, all 7 bits of the TLB entry are used. For it16, the TLB entry number to be accessed, in the range 0 – 15. Therefore, all 4 bits of the TLB entry are used.
2:0	0	—	Set to zero.

11.10.5.4 Instruction MMU TLB Tag Read Register

The format for the Tag Read Register virtual address is described in TABLE 11-32 and illustrated in FIGURE 11-26.

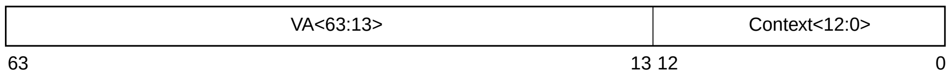


FIGURE 11-26 I-MMU TLB Tag Read Registers

TABLE 11-32 I-MMU TLB Tag Read Register

Bit	Field	Type	Description
63:13	VA	RW	The 51-bit virtual page number. In the fully associative TLB, page offset bits for larger page sizes are stored in the TLB; that is, VA<15:13>, VA<18:13>, and VA<21:13> for 64 KB, 512 KB, and 4 MB pages, respectively. These values are ignored during normal translation. When read, the UltraSPARC III Cu processor will return either 0 or the value previously written to them.
11:0	Instruction `Context`	RW	The 13-bit context identifier.

11.10.5.5 Instruction MMU TLB Tag Access Register

An ASI store to the TLB Data Access or Data In Register initiates an internal atomic write to the specified TLB Entry. The TLB entry data are obtained from the store data, and the TLB entry tag is obtained from the current contents of the TLB Tag Access Register.

11.10.5.6 Instruction TSB Tag Target Registers

The I-TSB Tag Target Registers are simply bit-shifted versions of the data stored in the Instruction Tag Access Registers, respectively. Since the Instruction Tag Access Register is updated on an I-TLB miss, respectively, the Instruction Tag Target Registers appear to software to be updated on an I-TLB miss. The MMU Tag Target Register is described in TABLE 11-33 and illustrated in FIGURE 11-27.

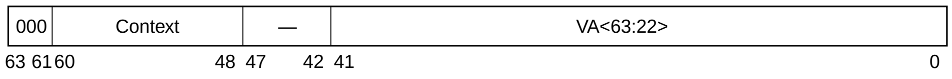


FIGURE 11-27 MMU Tag Target Registers

TABLE 11-33 MMU Tag Target Register

Bit(s)	Field	Type	Description
63:61	000	—	Set to 000.
60:48	`Context`<12:0>	RW	The context associated with the missing virtual address.
47:42	—	—	*Reserved.*
41:0	VA<63:22>	RW	The most significant bits of the missing virtual address.

11.10.5.7 Instruction TSB Base Registers

The Translation Storage Buffer (TSB) registers provide information for the hardware formation of TSB pointers and tag target, to assist software in quickly handling TLB misses. If the TSB concept is not employed in the software memory management strategy and therefore the Pointer and Tag Access Registers are not used, then the TSB registers need not contain valid data.

The TSB register is illustrated in FIGURE 11-28 and described in TABLE 11-34.

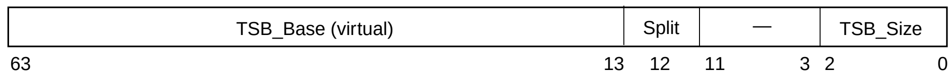


FIGURE 11-28 MMU Instruction TSB Registers

TABLE 11-34 TSB Register Description

Bit	Field	Type	Description
63:13	Instruction `TSB_Base`	RW	Provides the base virtual address of the Translation Storage Buffer. Software must ensure that the TSB base is aligned on a boundary equal to the size of the TSB or both TSBs in the case of a split TSB.
12	`Split`	RW	When `Split` = 1, the TSB 64 KB pointer address is calculated assuming separate (but abutting and equally sized) TSB regions for the 8 KB and the 64 KB TTEs. In this case, `TSB_Size` refers to the size of each TSB. The TSB 8 KB pointer address calculation is not affected by the value of the `Split` bit. When `Split` = 0, the TSB 64 KB pointer address is calculated assuming that the same lines in the TSB are shared by 8 KB and 64 KB TTEs, called a "common TSB" configuration. **Caution:** In the "common TSB" configuration (`TSB.Split` = 0), 8 KB and 64 KB page TTEs can conflict unless the TLB miss handler explicitly checks the TTE for page size. Therefore, do not use the common TSB mode in an optimized handler. For example, suppose an 8 KB page at VA = 2000_{16} and a 64 KB page at VA = 10000_{16} both exist — a legal situation. These both map to the second TSB line (line 1) and have the same VA tag of 0. Therefore, there is no way for the miss handler to distinguish these TTEs by the TTE tag alone, and unless the miss handler checks the TTE data, it may load an incorrect TTE.
11:3	—	—	Reserved.
2:0	Instruction `TSB_Size`	RW	The UltraSPARC III Cu processor implements a 3-bit `TSB_Size` field. The `TSB_Size` field provides the size of the TSB as follows: • The number of entries in the TSB (or each TSB if split) = $512 \times 2^{\text{TSB\_Size}}$. • The number of entries in the TSB ranges from 512 entries at `TSB_Size` = 0 (8 KB common TSB, 16 KB split TSB), to 64K entries at `TSB_Size` = 7 (1 MB common TSB, 2 MB split TSB). **Note:** Any update to the TSB register immediately affects the data that are returned from later reads of the Tag Target and TSB Pointer Registers.

11.10.5.8 Instruction TSB Extension Registers

The TSB Extension Registers provide information for the hardware formation of TSB pointers and tag target, to assist software in handling TLB misses quickly. If the TSB concept is not employed in the software memory management strategy and therefore the pointer and Tag Access Registers are not used, then the TSB Extension Registers need not contain valid data.

The TSB registers are defined as follows in FIGURE 11-29.

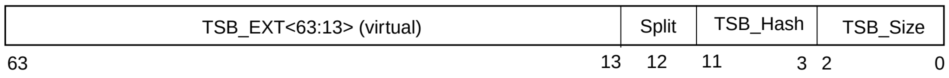


FIGURE 11-29 MMU Instruction TSB Extension Registers

In the UltraSPARC III Cu processor, `TSB_Hash` (bits 11:3 of the Extension Registers) are exclusive-ORed with the calculated TSB offset to provide a "hash" into the TSB. Changing the `TSB_Hash` field on a per-process basis minimizes the collision of TSB entries between different processes.

There are two TSB Extension Registers, one for each of the virtual address spaces (Primary, Nucleus).

When an I-TLB miss occurs, an appropriate TSB Extension Register is selected and XORed with the I-TSB Register. The result is then used to form a TSB pointer, as described in "TSB Pointer Formation" on page 254.

11.10.5.9 Instruction TSB 8 KB and 64 KB Pointer and Direct Pointer Registers

The I-TSB 8 KB and 64 KB registers are provided as an aid to software in determining the location of the missing or trapping TTE in the software-maintained TSB. The TSB 8 KB and 64 KB Pointer Registers provide the possible locations of the 8 KB and 64 KB TTE, respectively.

As a fine point, the bit that controls selection of 8 KB or 64 KB address formation for the Direct Pointer Register is a state bit in the I-MMU that is updated during a *fast_data_access_protection* exception. It records whether the page that hit in the TLB was a 64 KB page or a non-64 KB page, in which case, 8 KB is assumed.

The registers are illustrated in FIGURE 11-30, *where* VA<63:4> is the full virtual address of the TTE in the TSB, as determined by the MMU hardware, and is described in "Hardware Support for TSB Access" on page 253.

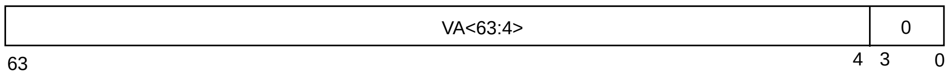


FIGURE 11-30 I-MMU TSB 8 KB/64 KB Pointer and I-MMU Direct Pointer Register

TSB 8 KB and 64 KB Pointer Registers

The TSB Pointer Registers are implemented as a reorder of the current data stored in the Tag Access Register and the TSB Extension Register. If the Tag Access Register or TSB Extension Register is updated through a direct software write (through an STXA instruction), then the values in the Pointer Registers will be updated as well.

Direct Pointer Register

The Direct Pointer Register is mapped by hardware to either the 8 KB or 64 KB Pointer Register in the case of a *fast_data_access_protection* exception according to the known size of the trapping TTE. In the case of a 512 KB or 4 MB page miss, the Direct Pointer Register returns the pointer as if the fault were from an 8 KB page.

11.10.5.10 Instruction Synchronous Fault Status Registers (I-SFSR)

The I-MMU maintains its own SFSR Register. The SFSR is illustrated in FIGURE 11-31 and described in TABLE 11-35.

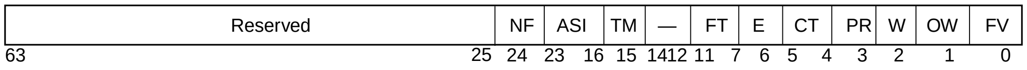


FIGURE 11-31 MMU Instruction Synchronous Fault Status Registers (I-SFSR, D-SFSR)

TABLE 11-35 SFSR Bit Description

Bit(s)	**Field**	**Type**	**Description**
63:25	—		*Reserved.*
24	NF	RW	Set in the I-MMU if the faulting instruction was a non-faulting load (a load to ASI_NOFAULT) (I-MMU = 0). NF is always 0 in I-SFSR.
23:16	ASI	RW	Records the 8-bit ASI associated with the faulting instruction. This field is valid for both I-MMU and I-MMU SFSRs and for all traps in which the FV bit is set. A trapping alternate space load or store sets the ASI field to the ASI the instruction attempted to reference. A trapping non-alternate-space load or store sets ASI to ASI_PRIMARY if PSTATE.CLE = 0 or to ASI_PRIMARY_LITTLE if PSTATE.CLE = 1. A *mem_address_not_aligned* trap caused by a JMPL or RETURN either does not set DSFSR.ASI or sets it as would a trapping non-alternate-space load or store.
15	TM	RW	I-TLB miss.
14:12	—		*Reserved.*

TABLE 11-35 SFSR Bit Description *(Continued)*

Bit(s)	Field	Type	Description

11:7	FT	RW	Specifies the exact condition that caused the recorded fault, according to TABLE 11-35 following this table. In the I-MMU, the Fault Type field is valid only for *data_access_exception* faults; there is no ambiguity in all other MMU trap cases. Note that the hardware does not priority-encode the bits set in the fault type (FT) field; that is, multiple bits can be set. In particular, the following ASI stores could set both the 01_{16} and 08_{16} Fault Type bits (the page is privileged, as well as storing to a read-only ASI):
			`stda %g0, [%g4]ASI_PRIMARY_NO_FAULT`
			`stda %g0, [%g4]ASI_SECONDARY_NO_FAULT`
			`stda %g0, [%g4]ASI_PRIMARY_NO_FAULT_LITTLE`
			`stda %g0, [%g4]ASI_SECONDARY_NO_FAULT_LITTLE`
			The FT field in the I-MMU SFSR always reads 0 for *fast_instruction_access_MMU_miss* and reads 01_{16} for *instruction_access_exception*, as all other fault types do not apply.

6	E	RW	Side-effect bit. It always reads as 0 in the I-MMU.
5:4	CT	RW	Context Register selection, as described below. The context is set to 11_2 when the access does not have a translating ASI. **Context ID** / **I-MMU Context** / **D-MMU Context**: 00 / Primary / Primary 01 / Reserved / Secondary 10 / Nucleus / Nucleus 11 / Reserved / Reserved
3	PR	RW	Privilege bit. Set if the faulting access occurred while in privileged mode. This field is valid for all traps in which the FV bit is set.
2	W	RW	Write bit. This bit always reads as 0 in the I-MMU SFSR.
1	OW	RW	Overwrite bit. When the MMU detects a fault, the Overwrite bit is set to 1 if the Fault Valid bit has not been cleared from a previous fault; otherwise, it is set to 0.
0	FV	RW	Fault Valid bit. Set when the MMU detects a fault; it is cleared only on an explicit ASI write of 0 to the SFSR. This bit is not set on an MMU miss. Therefore, overwrites of MMU misses cannot be detected. When the Fault Valid bit is not set, the values of the remaining fields in the SFSR and SFAR are undefined for traps other than an MMU miss.

TABLE 11-36 describes the SFSR fault type field (FT<11:7>).

TABLE 11-36 MMU Synchronous Fault Status Register FT (Fault Type) Field

Instruction	FT[6:0]	Fault Type
Instruction	01_{16}	Privilege violation.

Note – A *fast_instruction_MMU_miss* trap causes the SFSR and the SFAR to be overwritten without setting either the OW or the FV bits.

The SFSR and the Tag Access Registers both maintain state concerning a previous translation causing an exception. The update policy for the SFSR and the Tag Access Registers is shown in TABLE 11-2 on page 11-259.

11.10.5.11 Synchronous Fault Addresses

This section describes how the I-MMU obtains a fault address.

I-MMU Fault Address

There is no I-MMU Synchronous Fault Address Register. Instead, software must read the TPC register appropriately as discussed here.

For *fast_instruction_access_MMU_miss* traps, TPC contains the virtual address that was not found in the I-MMU TLB.

For *instruction_access_exception* traps, “privilege violation” fault type, TPC contains the virtual address of the instruction in the privileged page that caused the exception.

11.10.5.12 Instruction MMU Demap

Demap is an MMU operation, not an MMU register. Demap removes selected entries from the TLBs.

Note – A store to an I-MMU Register requires a MEMBAR #Sync, FLUSH, DONE, or RETRY before the point that the effect must be visible to load/store/atomic accesses. A FLUSH, DONE, or RETRY is needed before the point that the effect must be visible to instruction accesses, that is, MEMBAR #Sync is not sufficient. In either case, one of these instructions must be executed before the next non-internal store or load of any type and on or before the delay slot of a delayed-control transfer instruction of any type. This action is necessary to avoid data corruption.

Three types of demap operations are provided:

- **Demap page —** Removes any TLB entry that matches exactly the specified virtual page and context number. It is illegal to have more than one TLB entry per page.

  Demap page may, in fact, remove more than one TLB entry in the condition of a multiple TLB match, but this is an error condition of the TLB and has undefined results.
- **Demap context —** Removes any TLB entries that match the specified context identifier.
- **Demap all —** Removes all of the TLB entries from the TLB except for locked entries.

Demap is initiated by an STXA with ASI 57_{16} for I-MMU demap. It removes TLB entries from an on-chip TLB. No bus-based demap is supported. The demap address format is illustrated in FIGURE 11-32 and described in TABLE 11-37.

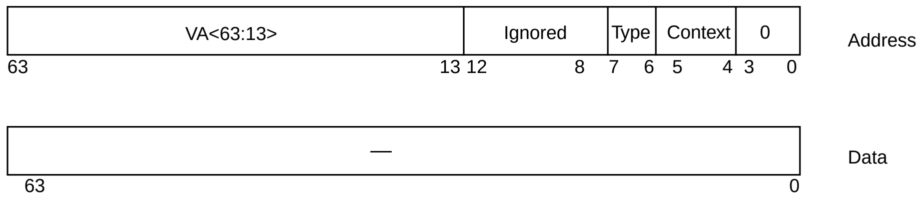


FIGURE 11-32 MMU Demap Operation Address and Data Formats

TABLE 11-37 Demap Address Format

Bit(s)	Field	Type	Description
63:13	VA<63:13>	RW	The virtual page number of the TTE to be removed from the TLB for Demap Page.
12:8	Ignored		This field is ignored by hardware.
7:6	`Type`	RW	The type of demap operation, as described below: **Type Field** / **Demap Operation** 0 / Demap page — see page 308 1 / Demap context — see page 308 2 / Demap all — see page 308 3 / *Reserved* — Ignored
5:4	`Context` ID	RW	Context Register selection, as described below. Use of the reserved value causes the demap to be ignored. **Context ID Field** / **Context Used in Demap** 00 / Primary 01 / *Reserved* 10 / Nucleus 11 / *Reserved*
3:0	0	—	Set to zero.

A demap operation does not invalidate the TSB in memory. Software must modify the appropriate TTEs in the TSB before initiating a demap operation.

Except for Demap All, the demap operation does not depend on the value of any entry's lock bit. A demap operation demaps both locked entries and unlocked entries.

The demap operation produces no output.

The following are Instruction/Data demap page types:

- **Instruction Demap Page (Type = 0).** Demap Page removes the TTE (from the specified TLB), matching the specified virtual page number and Context Register. The match condition with regard to the global bit is the same as a normal TLB access; that is, if the global bit is set, the contexts do not need to match.

  Virtual page offset bits 15:13, 18:13, and 21:13 for 64 KB, 512 KB, and 4 MB page TLB entries, respectively, do not participate in the match for that entry. This is the same condition as for a translation match.

Note – Each Demap Page operation removes only one TLB entry. A demap of a 64 KB, 512 KB, or 4 MB page does not demap any smaller page within the specified virtual address range.

- **Instruction Demap Context (Type = 1).** Demap Context removes from the TLB all TTEs having the specified context. If the TTE Global bit is set, then the TTE is not removed. VA is ignored for this operation.
- **Instruction Demap All (Type = 2).** Demap All removes all TTEs that do not have the lock bit set. VA and `Context` are ignored for this operation.

11.10.5.13 Instruction TLB Diagnostic Register

Accesses to the TLB Diagnostic Register require the virtual address to be set to access a TLB and TLB entry. The format of the TLB Diagnostic Register virtual address is described TABLE 11-38 and illustrated in FIGURE 11-33.

—	1	TLB #	—	TLB Entry #	0
63 19	18	17 16	15 10	9 3	2 0

FIGURE 11-33 MMU TLB Diagnostic Access Virtual Address Format

TABLE 11-38 MMU TLB Diagnostic Register Virtual Address

Bit	Field	Type	Description
63:19	—	—	*Reserved.*
18	1	—	Set to one.
17:16	TLB #	RW	The number of the table to access, as follows: TLB | TLB Type | Entries 0 | Fully associative 64 KB, 4 MB, and 512 KB page size and locked pages (all sizes) | 16 (it16) 2 | 2-way associative 8 KB page size | 128 (it128)
15:10	—	—	*Reserved.*
9:3	TLB Entry #	RW	For it128, the TLB entry number to be accessed, in the range 0 – 127. Therefore, all 7 bits of the TLB entry are used. For it16, the TLB entry number to be accessed, in the range 0 – 15. Therefore, all 4 bits of the TLB entry are used.
2:0	0	—	Set to zero.

11.10.6 I-MMU Bypass

The I-MMU can only be bypassed by being disabled.

SECTION V

Supervisor Programming

CHAPTER 12

Traps and Trap Handling

A trap is a vectored transfer of control to supervisor software through a trap table that contains the first eight (32 for *clean_window*, *spill*, *fill*, *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection* traps) instructions of each trap handler. The base address of the table is established by supervisor software, by writing the Trap Base Address (TBA) register. The displacement within the table is determined by the trap type and the current trap level (TL). One-half of the table is reserved for hardware traps; one-quarter is reserved for software traps generated by Tcc instructions; the remaining quarter is reserved for future use.

A trap behaves like an unexpected procedure call. It causes the hardware to do the following:

1. Save certain processors state (program counters, CWP, ASI, CCR, PSTATE, and the trap type) on a hardware register stack.
2. Enter privileged execution mode with a predefined PSTATE.
3. Begin executing trap handler code in the trap vector.

When the trap handler has finished, it uses either a DONE or RETRY instruction to return.

A trap may be caused by a Tcc instruction, an instruction induced exception, a reset, an asynchronous error, or an interrupt request not directly related to a particular instruction. The processor must appear to behave as though, before executing each instruction, it determines if there are any pending exceptions or interrupt requests. If there are pending exceptions or interrupt requests, the processor selects the highest priority exception or interrupt request and causes a trap.

Thus, an *exception* is a condition that makes it impossible for the processor to continue executing the current instruction stream without software intervention. A *trap* is the action taken by the processor when it changes the instruction flow in response to the presence of an exception, interrupt, or Tcc instruction.

A catastrophic error exception is due to the detection of a hardware malfunction, which due to the nature of the error, the state of the machine at the time of the exception cannot be restored. Since the machine state cannot be restored, execution after such an exception may not be resumable. An example of such an error is an uncorrectable bus parity error. Catastrophic errors cause deferred or disrupting traps.

Traps are described in the following sections:

- Processor States, Normal and Special Traps
- Trap Categories
- Trap Control
- Trap-Table Entry Addresses
- Trap Processing
- Exception and Interrupt Descriptions

12.1 Processor States, Normal and Special Traps

The processor is always in one of three discrete states:

- `execute_state`, which is the normal execution state of the processor
- `RED_state` (**R**eset, **E**rror, and **D**ebug state), which is a restricted execution state reserved for processing traps that occur when `TL` = `MAXTL` – 1, and for processing hardware-initiated and software-initiated resets
- `error_state`, which is a halted state that is entered as a result of a trap when `TL = MAXTL`

Traps processed in `execute_state` are called *normal traps*. Traps processed in `RED_state` are called *special traps*.

FIGURE 12-1 shows the processor state diagram.

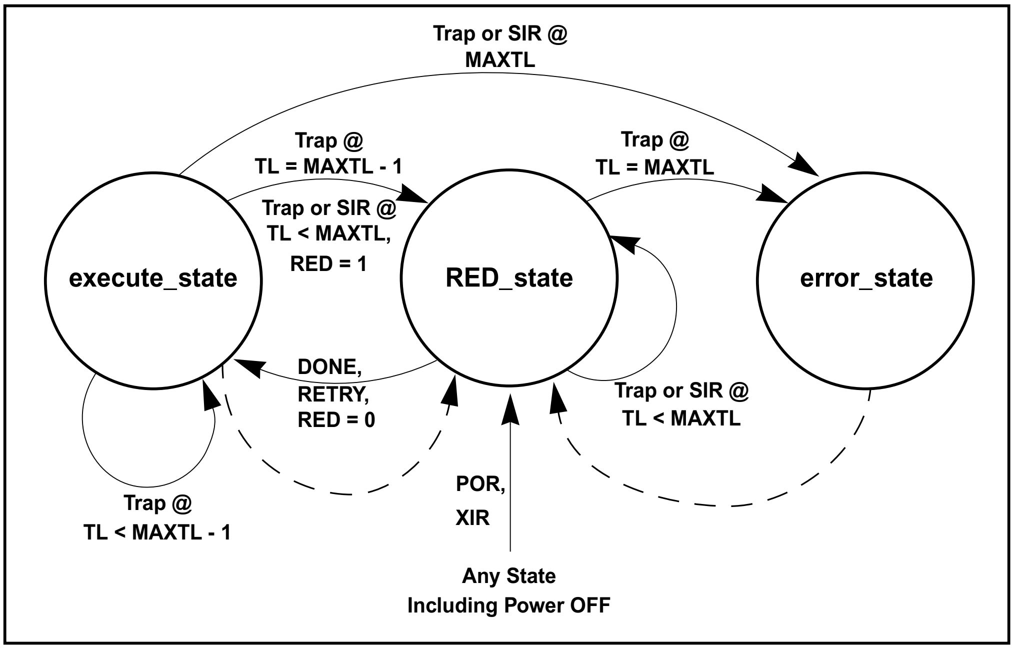


FIGURE 12-1 Processor State Diagram

12.1.1 RED_state

RED_state is an acronym for **R**eset, **E**rror, and **D**ebug state. The processor enters RED_state under any one of the following conditions:

- A trap is taken when TL = MAXTL –1.
- A POR, WDR, or XIR reset occurs.
- An SIR occurs when TL < MAXTL.
- System software sets PSTATE.RED = 1.

RED_state serves two mutually exclusive purposes:

- During trap processing, it indicates that no more trap levels are available; that is, if another nested trap is taken, the processor will enter error_state and halt. RED_state provides system software with a restricted execution environment.
- It provides the execution environment for all reset processing.

RED_state is indicated by PSTATE.RED. When this bit is set, the processor is in RED_state; when this bit is clear, the processor is not in RED_state, independent of the value of TL. Executing a DONE or RETRY instruction in RED_state restores the stacked copy of the PSTATE register, which clears the PSTATE.RED flag if the stacked copy had it cleared. System software can also set or clear the PSTATE.RED flag with a WRPR instruction, which also forces the processor to enter or exit RED_state, respectively. In this case, the WRPR instruction should be placed in the delay slot of a jump so that the program counter (PC) can be changed in concert with the state change.

Programming Notes – Setting TL = MAXTL with a WRPR instruction does not also set PSTATE.RED = 1 nor does it alter any other machine state. The values of PSTATE.RED and TL are independent.

Setting PSTATE.RED with a WRPR instruction causes the processor to execute in RED_state. However, it is different from a RED_state trap in the sense that there are no trap related changes in the machine state (for example, TL does not change).

12.1.1.1 RED_state Trap Table

Traps occurring in RED_state or traps that cause the processor to enter RED_state use an abbreviated trap vector. The RED_state trap vector is constructed so that it can overlay the normal trap vector, if necessary. TABLE 12-1 illustrates the RED_state trap vector layout.

TABLE 12-1 RED_state Trap Vector Layout

Offset	TT	Reason
00_{16}	0	*Reserved* (SPARC V8 reset)
20_{16}	1	Power-on reset (POR)
40_{16}	2†	Watchdog reset (WDR)
60_{16}	3‡	Externally initiated reset (XIR)
80_{16}	4	Software-initiated reset (SIR)
$A0_{16}$	*	All other exceptions in RED_state

†TT = 2 if a WDR occurs while the processor is not in error_state; TT = trap type of the exception that caused entry into error_state if a WDR occurs in error_state.

‡TT = 3 if an *externally_initiated_reset* (XIR) occurs while the processor is not in error_state; TT = trap type of the exception that caused entry into error_state if the externally initiated reset occurs in error_state.

*TT = trap type of the exception. See TABLE 12-3.

When the UltraSPARC III Cu 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}.

12.1.1.2 RED_state Execution Environment

In RED_state, the processor is forced to execute in a restricted environment by overriding the values of some processor controls and state registers.

The values are overridden, not set, allowing them to be switched atomically.

When RED_state is entered because of component failures, the handler should attempt to recover from potentially catastrophic error conditions or to disable the failing components. When RED_state is entered after a reset, the software should create the environment necessary to restore the system to a running state.

12.1.1.3 RED_state Entry Traps

The following traps are processed in RED_state in all cases.

- **Power-on reset (POR)** — Implemented in hardware in UltraSPARC III Cu processors; not really a trap.
- **Watchdog reset (WDR)** — Implemented in hardware in UltraSPARC III Cu processors; this trap is used as a recovery mechanism from error_state in UltraSPARC III Cu. Upon an entry to error_state, the processor automatically invokes a WDR to enter RED_state.
- **Externally initiated reset (XIR)** — Implemented in hardware in UltraSPARC III Cu processors; typically used as a nonmaskable interrupt method for debug.

In addition, the following trap is processed in RED_state if TL < MAXTL when the trap is taken. Otherwise, it is processed in error_state.

- **Software-initiated reset (SIR)**

Traps that occur when TL = MAXTL – 1 also set PSTATE.RED = 1; that is, any trap handler entered with TL = MAXTL runs in RED_state.

Any non-reset trap that sets PSTATE.RED = 1 or that occurs when PSTATE.RED = 1 branches to a special entry in the RED_state trap vector at RSTVaddr + $A0_{16}$.

12.1.1.4 RED_state Software Considerations

In effect, RED_state reserves one level of the trap stack for recovery and reset processing. Software should be designed to require only MAXTL – 1 trap levels for normal processing. That is, any trap that causes TL = MAXTL is an exceptional condition that should cause entry to RED_state.

The architected value for `MAXTL` in the UltraSPARC III Cu processor is five; typical usage of the trap levels is shown in TABLE 12-2.

TABLE 12-2 Typical Usage for Trap Levels

TL	Usage
0	Normal execution
1	System calls, interrupt handlers, instruction emulation
2	Window *spill*/*fill*
3	Page-fault handler
4	Reserved for error handling
5	`RED_state` handler

Programming Note – To log the state of the processor, `RED_state` handler software needs either a spare register or a preloaded pointer to a save area. To support recovery, the operating system might reserve one of the alternate global registers (for example, `%a7`) for use in `RED_state`.

12.1.2 Error_state

The processor enters `error_state` when a trap occurs while the processor is already at its maximum supported trap level, that is, when `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 to diagnose failures. A WDR behaves more like a trap than a reset.

WDR affects only one processor, rather than the entire system.

Note – When a window trap occurs at `TL = MAXTL`, the processor enters `RED_state` and then performs a WDR. The current window pointer (CWP) is always updated as if the window trap was taken successfully.

12.2 Trap Categories

An exception or interrupt request can cause any of the following trap types:

- Precise trap
- Deferred trap
- Disrupting trap
- Reset trap

12.2.1 Precise Traps

A *precise trap* is induced by a particular instruction and occurs before any program-visible state has been changed by the trap-inducing instructions. When a precise trap occurs, several conditions must be true:

- The `PC` saved in `TPC[TL]` points to the instruction that induced the trap, and the next program counter (`nPC`) saved in `TNPC[TL]` points to the instruction that was to be executed next.
- All instructions issued before the one that induced the trap have completed execution.
- Any instructions issued after the one that induced the trap remain not executed.

Among the actions the trap handler software might take after a precise trap are the following:

- Return to the instruction that caused the trap and re-execute it by executing a `RETRY` instruction (`PC` ← old `PC`, `nPC` ← old `nPC`).
- Emulate the instruction that caused the trap and return to the succeeding instruction by executing a `DONE` instruction (`PC` ← old `nPC`, `nPC`← old `nPC` + 4).
- Terminate the program or process associated with the trap.

12.2.2 Deferred Traps

A *deferred trap* is also induced by a particular instruction, but unlike a precise trap, a deferred trap may occur after program-visible state has been changed. Such state may have been changed by the execution of either the trap inducing instruction itself or by one or more other instructions.

Associated with a particular deferred trap implementation, the following must exist:

- An instruction that causes a potentially outstanding deferred trap exception to be taken as a trap.
- Privileged instructions that access the state information needed by the supervisor software to emulate the deferred trap inducing instruction and to resume execution of the trapped instruction stream.

Programming Note – Resuming execution may require the emulation of instructions that had not completed execution at the time of the deferred trap, that is, those instructions in the deferred trap queue.

Among the actions software can take after a deferred trap are the following:

- Emulate the instruction that caused the exception, emulate or cause to execute any other execution deferred instructions that were in an associated deferred trap state queue, and use `RETRY` to return control to the instruction at which the deferred trap was invoked.
- Terminate the program or process associated with the trap.

12.2.3 Disrupting Traps

A *disrupting trap* is neither a precise trap nor a deferred trap. A disrupting trap is caused by a *condition* (for example, an interrupt) rather than directly by a particular instruction; the cause distinguishes it from precise and deferred traps. When a disrupting trap has been serviced, trap handler software normally arranges for program execution to resume where it left off. This differentiates disrupting traps from reset traps, which trap to a unique reset address from which execution of the program that was running when the reset occurred is never expected to resume.

Disrupting traps are controlled by a combination of the Processor Interrupt Level (`PIL`) register and the Interrupt Enable (`IE`) field of `PSTATE`. A disrupting trap condition is ignored when interrupts are disabled (`PSTATE.IE` = `0`) or when the condition's interrupt level is less than or equal to that specified in `PIL`.

A disrupting trap may be due either to an interrupt request not directly related to a previously executed instruction or an exception related to a previously executed instruction. Interrupt requests may be either internal or external. An interrupt request can be induced by the assertion of a signal not directly related to any particular processor or memory state, for example, the assertion of an "I/O done" signal.

A disrupting trap related to an earlier instruction causing an exception is similar to a deferred trap in that it occurs after instructions following the trap inducing instruction have modified the processor or memory state. The difference is that the condition that caused the instruction to induce the disrupting trap may lead to unrecoverable errors, since the implementation may not preserve the necessary state. An example is an ECC data access error reported after the corresponding load instruction has completed.

Disrupting trap conditions should persist until the corresponding trap is taken.

Among the actions that trap-handler software might take after a disrupting trap are the following:

- Use `RETRY` to return to the instruction at which the trap was invoked (`PC` ← old `PC`, `nPC` ← old `nPC`).

- Terminate the program or process associated with the trap.

12.2.4 Reset Traps

A *reset trap* occurs when supervisor software or the implementation's hardware determines that the machine must be reset to a known state. Reset traps differ from disrupting traps in that trap handler software for resets is never expected to resume execution of the program that was running when the reset trap occurred.

The following reset traps are defined for the SPARC V9 architecture:

- **Software-initiated reset (SIR)** — Initiated by software by executing the SIR instruction.
- **Power-on reset (POR)** — Initiated when power is applied (or reapplied) to the processor.
- **Watchdog reset (WDR**) — Initiated in response to entry into `error_state`.
- **Externally initiated reset (XIR)** — Initiated in response to an external signal. This reset trap is normally used for critical system events, such as power failure.

12.2.5 Uses of the Trap Categories

The SPARC V9 *trap model* makes the following stipulations:

1. Reset traps, except *software_initiated_reset* traps, occur asynchronously to program execution.

2. When recovery from an exception can affect the interpretation of subsequent instructions, such exceptions shall be precise. These exceptions include:

- *software_initiated_reset*
- *instruction_access_exception*
- *privileged_action*
- *privileged_opcode*
- *trap_instruction*
- *instruction_access_error*
- *clean_window*
- *fp_disabled*
- *LDDF_mem_address_not_aligned*
- *STDF_mem_address_not_aligned*
- *tag_overflow*
- *spill_n_normal*
- *spill_n_other*

- *fill_n_normal*
- *fill_n_other*

3. An exception caused after the initial access of a multiple access load or store instruction (load/store doubleword, block load, block store, `LDSTUB`, `CASA`, `CASXA`, or `SWAP`) that causes a catastrophic exception may be precise, deferred, or disrupting. Thus, a trap due to the second memory access can occur after the processor or memory state has been modified by the first access.

4. Implementation-dependent catastrophic exceptions may cause precise, deferred, or disrupting traps.

5. Exceptions caused by external events unrelated to the instruction stream, such as interrupts, are disrupting.

A deferred trap may occur one or more instructions after the trap inducing instruction is dispatched.

12.3 Trap Control

Several registers control how any given trap is processed:

- The `IE` field in `PSTATE` and the processor interrupt level `PIL` register control interrupt processing.
- The enable floating-point unit (`FEF`) field in `FPRS`, the floating-point unit enable (`PEF`) field in `PSTATE`, and the trap enable mask (`TEM`) in the FSR control floating-point traps.
- The `TL` register, which contains the current level of trap nesting, controls whether a trap causes entry to `execute_state`, `RED_state`, or `error_state`.
- `PSTATE.TLE` determines whether implicit data accesses in the trap routine will be performed with the big-endian or little-endian byte order.

12.3.1 PIL Control

Between the execution of instructions, the IU prioritizes the outstanding exceptions and interrupt requests. At any given time, only the highest priority exception or interrupt request is taken as a trap. When there are multiple outstanding exceptions or interrupt requests, the SPARC V9 architecture assumes that lower priority interrupt requests will persist and lower priority exceptions will recur if an exception causing instruction is re-executed.

For interrupt requests, the IU compares the interrupt request level against the `PIL` register. If the interrupt request level is greater than `PIL`, then the processor takes the interrupt request trap, assuming there are no higher-priority exceptions outstanding.

12.3.2 TEM Control

The occurrence of floating-point traps of type *IEEE_754_exception* can be controlled with the user accessible TEM field of the FSR. If a particular bit of TEM is one, the associated *IEEE_754_exception* can cause a *fp_exception_ieee_754* trap.

If a particular bit of TEM is zero, the associated *IEEE_754_exception* does not cause a *fp_exception_ieee_754* trap. Instead, the occurrence of the exception is recorded in the FSR's accrued exception field (aexc).

If an *IEEE_754_exception* results in a *fp_exception_ieee_754* trap, then the destination f register, fcc*n*, and aexc fields remain unchanged. However, if an *IEEE_754_exception* does not result in a trap, then the f register, fcc*n*, and aexc fields are updated to their new values.

12.4 Trap-Table Entry Addresses

Privileged software initializes the trap base address (TBA) register to the upper 49 bits of the trap-table base address. Bit 14 of the vector address (the TL>0 field) is set based on the value of TL at the time the trap is taken, that is, to zero if TL = 0 and to one if TL > 0. Bits 13–5 of the trap vector address are the contents of the TT register. The lowest five bits of the trap address, bits 4–0, are always zero (hence, each trap-table entry is at least 2^5 or 32 bytes long). FIGURE 12-2 illustrates the trap vector address.

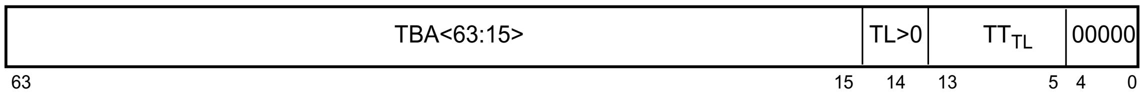


FIGURE 12-2 Trap Vector Address

12.4.1 Trap Table Organization

The trap table layout is illustrated in FIGURE 12-3.

Value of TL Before the Trap	Trap Table Contents	Trap Type
TL = 0	Hardware traps	000_{16}–$07F_{16}$
	Spill/fill traps	080_{16}–$0FF_{16}$
	Software traps	100_{16}–$7F_{16}$
	Reserved	180_{16}–$1FF_{16}$
TL > 0	Hardware traps	200_{16}–$27F_{16}$
	Spill/fill traps	280_{16}–$2FF_{16}$
	Software traps	300_{16}–$37F_{16}$
	Reserved	380_{16}–$3FF_{16}$

FIGURE 12-3 Trap Table Layout

The trap table for `TL` = 0 comprises 512 32-byte entries; the trap table for `TL` > 0 comprises 512 more 32-byte entries. Therefore, the total size of a full trap table is $512 \times 32 \times 2$, or 32 KB. However, if privileged software does not use software traps (`Tcc` instructions) at `TL` > 0, the table can be made 24 KB long.

12.4.2 Trap Type (TT)

When a normal trap occurs, a value that uniquely identifies the trap is written into the current 9-bit `TT` register (`TT[TL]`) by hardware. Control is then transferred into the trap table to an address formed by the `TBA` register (`TL>0`) and `TT[TL]`. The lowest five bits of the address are always zero; each entry in the trap table may contain the first eight instructions of the corresponding trap handler.

Programming Notes – The trap type for the *clean_window* exception is 024_{16}. Three subsequent trap vectors (025_{16}–027_{16}) are reserved to allow for an inline (branchless) trap handler. Three subsequent trap vectors are reserved for each *spill/fill* vector, to allow for an inline (branchless) trap handler.

The *spill/fill*, *clean_window*, and MMU related traps (*fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection*) trap types are spaced such that their trap-table entries are 128 bytes (32 instructions) long in the UltraSPARC III Cu processor. This length allows the complete code for one *spill/fill* routine, a *clean_window* routine, or a normal MMU miss handling routine to reside in one trap-table entry.

When a special trap occurs, the `TT` register is set as described in "RED_state" on page 317. Control is then transferred into the `RED_state` trap table to an address formed by the `RSTVaddr` and an offset depending on the condition.

TT values 000_{16}–$0FF_{16}$ are reserved for hardware traps. TT values 100_{16}–$17F_{16}$ are reserved for software traps (traps caused by execution of a Tcc instruction). TT values 180_{16}–$1FF_{16}$ are reserved for future uses.

TT values 070_{16}, 071_{16} and 072_{16} are reserved for fast_ECC_error, dcache_parity_error and icache_parity_error, respectively.

The assignment of TT values to traps is shown in TABLE 12-3. Hardware must detect and trap these exceptions and interrupts and must set the defined TT values. In the table, AG = alternate globals, MG = MMU globals, and IG = interrupt globals.

TABLE 12-3 Exception and Interrupt Requests, by TT Value *(1 of 3)*

Exception or Interrupt Request	TT	Global Register Set	Priority
Reserved	000_{16}		*NA*
power_on_reset	001_{16}	AG	0
watchdog_reset	002_{16}	AG	1
externally_initiaTrap Table Layoutted_reset	003_{16}	AG	1
software_initiated_reset	004_{16}	AG	1
RED_state_exception	005_{16}	AG	1
Reserved	006_{16}–007_{16}		*NA*
instruction_access_exception	008_{16}	MG	5
instruction_access_MMU_miss (Not used. Replaced by newer version.)	008_{16}		*NA*
instruction_access_error	$00A_{16}$	AG	3
Reserved	$00B_{16}$–$00F_{16}$		*NA*
illegal_instruction	010_{16}	AG	7
privileged_opcode	011_{16}	AG	6
Reserved	012_{16}		*NA*
Reserved	013_{16}		*NA*
Reserved	014_{16}–$01F_{16}$		*NA*
fp_disabled	020_{16}	AG	8
fp_exception_ieee_754	021_{16}	AG	11
fp_exception_other	022_{16}	AG	11
tag_overflow	023_{16}	AG	14
clean_window	024_{16}–027_{16}	AG	10
division_by_zero	028_{16}	AG	15

TABLE 12-3 Exception and Interrupt Requests, by TT Value *(2 of 3)*

Exception or Interrupt Request	TT	Global Register Set	Priority
Reserved	029_{16}–$02F_{16}$		*NA*
data_access_exception	030_{16}	MG	12
data_access_MMU_miss (Not used; replaced by newer version)	031_{16}		*NA*
data_access_error	032_{16}	AG	12
data_access_protection (Not used; replaced by newer version)	033_{16}		12
mem_address_not_aligned	034_{16}	AG	10
LDDF_mem_address_not_aligned	035_{16}	AG	10
STDF_mem_address_not_aligned	036_{16}	AG	10
privileged_action	037_{16}	AG	11
Reserved	038_{16}		*NA*
Reserved	039_{16}		*NA*
Reserved	$03A_{16}$–$03F_{16}$		*NA*
Reserved	040_{16}		*NA*
interrupt_level_n (n = 1–15)	041_{16}–$04F_{16}$	AG	32-n
Reserved	050_{16}–$05F_{16}$		*NA*
interrupt_vector	060_{16}	IG	16
PA_watchpoint	061_{16}	AG	12
VA_watchpoint	062_{16}	AG	11
ECC_error	063_{16}	AG	33
fast_instruction_access_MMU_miss	064_{16}–067_{16}	MG	2
fast_data_access_MMU_miss	068_{16}–$06B_{16}$	MG	12
fast_data_access_protection	$06C_{16}$–$06F_{16}$	MG	12
fast_ECC_error	070_{16}	AG	2
dcache_parity_error	071_{16}	AG	2
icache_parity_error	072_{16}	AG	2
spill_n_normal (n = 0–7)	080_{16}–$09F_{16}$	AG	9
spill_n_other (n = 0–7)	$0A0_{16}$–$0BF_{16}$	AG	9
fill_n_normal (n = 0–7)	$0C0_{16}$–$0DF_{16}$	AG	9

TABLE 12-3 Exception and Interrupt Requests, by TT Value *(3 of 3)*

Exception or Interrupt Request	**TT**	**Global Register Set**	**Priority**
fill_n_other (*n* = 0–7)	$0E0_{16}$–$0FF_{16}$	AG	9
trap_instruction	100_{16}–$17F_{16}$	AG	16
Reserved	180_{16}–$1FF_{16}$		*NA*

TABLE 12-4 lists the traps in priority order. Reserved and unused traps have been left out of this table.

TABLE 12-4 Exception and Interrupts Requests in Priority Order

Exception or Interrupt Request	**TT**	**Global Register Set**	**Priority**
power_on_reset	001_{16}	AG	0
watchdog_reset	002_{16}	AG	1
externally_initiated_reset	003_{16}	AG	1
software_initiated_reset	004_{16}	AG	1
RED_state_exception	005_{16}	AG	1
fast_instruction_access_MMU_miss	064_{16}–067_{16}	MG	2
fast_ECC_error	070_{16}	AG	2
dcache_parity_error	071_{16}	AG	2
icache_parity_error	072_{16}	AG	2
instruction_access_error	$00A_{16}$	AG	3
instruction_access_exception	008_{16}	MG	5
privileged_opcode	011_{16}	AG	6
illegal_instruction	010_{16}	AG	7
fp_disabled	020_{16}	AG	8
spill_n_normal (*n* = 0–7)	080_{16}–$09F_{16}$	AG	9
spill_n_other (*n* = 0–7)	$0A0_{16}$–$0BF_{16}$	AG	9
fill_n_normal (*n* = 0–7)	$0C0_{16}$–$0DF_{16}$	AG	9
fill_n_other (*n* = 0–7)	$0E0_{16}$–$0FF_{16}$	AG	9
clean_window	024_{16}–027_{16}	AG	10
mem_address_not_aligned	034_{16}	AG	10
LDDF_mem_address_not_aligned	035_{16}	AG	10

TABLE 12-4 Exception and Interrupts Requests in Priority Order *(Continued)*

Exception or Interrupt Request	TT	Global Register Set	Priority
STDF_mem_address_not_aligned	036_{16}	AG	10
fp_exception_ieee_754	021_{16}	AG	11
fp_exception_other	022_{16}	AG	11
privileged_action	037_{16}	AG	11
VA_watchpoint	062_{16}	AG	11
data_access_exception	030_{16}	MG	12
data_access_error	032_{16}	AG	12
PA_watchpoint	061_{16}	AG	12
fast_data_access_MMU_miss	068_{16}–$06B_{16}$	MG	12
fast_data_access_protection	$06C_{16}$–$06F_{16}$	MG	12
tag_overflow	023_{16}	AG	14
division_by_zero	028_{16}	AG	15
trap_instruction	100_{16}–$17F_{16}$	AG	16
interrupt_vector	060_{16}	IG	16
interrupt_level_n (n = 1–15)	041_{16}–$04F_{16}$	AG	32-*n*
ECC_error	063_{16}	AG	33

12.4.2.1 Trap Type for Spill/Fill Traps

The trap type for window *spill*/*fill* traps is determined on the basis of the contents of the OTHERWIN and WSTATE registers as described in TABLE 12-5 and shown in FIGURE 12-4.

TABLE 12-5 Trap Types for *Spill/Fill* Traps

Bit	Field	Description
8:6	SPILL_OR_FILL	010_2 for *spill* traps; 011_2 for *fill* trap
5	OTHER	(OTHERWIN ≠ 0)
4:2	WTYPE	If (OTHER) then WSTATE.OTHER; else WSTATE.NORMAL

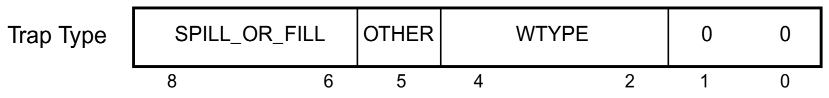


FIGURE 12-4 Trap Type Encoding for *Spill*/*Fill* Traps

12.4.3 Trap Priorities

TABLE 12-3 shows the assignment of traps to `TT` values and the relative priority of traps and interrupt requests. Priority 0 is highest, priority 31 is lowest; that is, if $X < Y$, a pending exception or interrupt request with priority X is taken instead of a pending exception or interrupt request with priority Y.

However, the `TT` values for the exceptions and interrupt requests shown in TABLE 12-3 must remain the same for every implementation.

The trap priorities given always need to be considered in light of how the processor actually issues and executes instructions. For example, if an *instruction_access_error* occurs (priority 3), it will be taken even if the instruction was an `SIR` (priority 1). This situation occurs because the processor gets the *instruction_access_error* during I-fetch and never actually issues or executes the instruction, so the `SIR` is never seen by the backend of the processor. This is an obvious case, but there are other more subtle cases.

In summary, the trap priorities are used to prioritize traps that occur in the same clock cycle. They do not take into consideration that an instruction may be alive for multiple cycles and that a trap may be detected and initiated early in the life of an instruction. Once the early trap is taken, any errors that might have occurred later in the instruction's life will not be seen.

12.4.4 Details of Supported Traps

12.4.4.1 MMU Traps

The UltraSPARC III Cu processor supports three 32-instruction traps for handling the most performance sensitive MMU traps:

- *fast_instruction_access_MMU_miss*
- *fast_data_access_MMU_miss*
- *fast_data_access_protection*

The first two traps are taken when the TLBs miss on an instruction or data access. The third type of trap is taken when a protection violation occurs. The common case of this trap occurs when a write request is made to a page marked as clean in the TLB.

Each of these trap vectors takes up four slots in the trap table; this means that each trap handler can contain up to 32 instructions before a branch is needed.

12.4.4.2 Precise Correctable Data Corruption Error Traps

The UltraSPARC III Cu processor implements three error traps listed in TABLE 12-6.

TABLE 12-6 Two New Traps Added for D-cache and I-cache Parity Error

Exception or Interrupt Request	Globals	TT	Priority
fast_ECC_error	AG	0x070	2
dcache_parity_error	AG	0x071	2
icache_parity_error	AG	0x072	2

D-Cache and I-Cache Parity Errors

Upon a *dcache_parity_error* trap taken, hardware will automatically turn off D-cache and I-cache by clearing the DC and IC bit in the DCU Control Register. The same action is done on *icache_parity_error*. The objective is to avoid any possible recursive/nested parity error of the same type, an event that is very complicated to handle by the trap handler and likely to reduce software ability to repair and recover from the parity error.

Fast ECC Errors

On *fast_ECC_error* detection during D-cache load miss fill, D-cache will still install the uncorrected data. But since the *fast_ECC_error* trap is precise, hardware can rely on software to help clean up the bad data. I-cache is different. If I-cache is filled with errors, the line will not be installed in I-cache.

A D-cache or I-cache miss request may observe an ECC error in the line it reads from the L2-cache. When this occurs, a *fast_ECC_error* precise trap is generated for the instruction that detected the error. In the case of a D-cache request, the corrupted data will be installed in the D-cache, but the trap takes effect before the data can be used.

In case of an I-cache request, the data from the L2-cache will be corrected by hardware before being installed in the I-cache. When the *fast_ECC_error* trap is taken, the I-cache and D-cache are left enabled. Software is responsible to disable any caches as a part of the recovery. Software must flush the corrupted line from the D-cache if it was filled.

Prioritization of Errors

All traps with priority 2 are of precise type. Miss/error traps with priority 2 that arrive at the same time are processed by hardware according to their "age" or program order. The oldest instruction with an error/miss will get the trap.

However, there are some cases where the same instruction (same PC) generate multiple traps. There are two cases:

1. **Case 1:** Singular trap type with highest priority.
   The processing order of which trap to take first follows the priority number (lowest number wins).

2. **Case 2:** Multiple traps having same highest priority.
   For trap priority 2, the only possible combination is simultaneous traps due to I-cache parity error and I-TLB miss. In this case, the hardware processing order is as follows:

   icache_parity_error > fast_instruction_access_MMU_miss

There are no other "simultaneous traps" combinations of Case 2. The other priority 2 traps have later arrivals. D-cache access is further down the pipeline after instruction fetch from I-cache. Thus, D-cache parity error on a load instruction (if any) will be detected after I-cache parity error (if any) and I-TLB miss (if any). The other priority 2 trap, *fast_ECC_error* can only be caused by an I-cache miss or D-cache load miss; therefore, it arrives even later.

To summarize, precise traps are processed in the following order:

program order > trap priority number > Hardware implementation order

One exception to the trap priority number order in implementation is on a non-precise ECC error trap. If the victim instruction selected as the trap point *also* has a higher priority precise trap, then the ECC error trap is taken first. This should be okay since no trap is lost. When the ECC error trap handling finishes and the victim instruction is retried, the precise trap is encountered again.

12.4.4.3 Other Traps

The UltraSPARC III Cu processor supports the following trap types in addition to those in SPARC V9:

- *interrupt_vector_trap*
- *PA_watchpoint*
- *VA_watchpoint*
- *ECC_error*
- *data_access_protection*

12.5 Trap Processing

The processor's action during trap processing depends on the trap type, the current level of trap nesting (given in the TL register), and the processor state. When a trap occurs, the global registers are replaced with one of three sets of trap global register — MMU globals, interrupt globals, or alternate globals — based on the type of trap.

All traps use normal trap processing, except those due to reset requests, catastrophic errors, traps taken when TL = MAXTL – 1, and traps taken when the processor is in RED_state. These traps use special RED_state trap processing.

During normal operation, the processor is in execute_state. It processes traps in execute_state and continues.

When a normal trap or software-initiated reset (SIR) occurs with TL = MAXTL, there are no more levels on the trap stack, so the processor enters error_state and halts. To avoid this catastrophic failure, SPARC V9 provides the RED_state processor state. Traps processed in RED_state use a special trap vector and a special trap-vectoring algorithm.

Traps that occur with TL = MAXTL – 1 are processed in RED_state. In addition, reset traps are also processed in RED_state. Finally, supervisor software can force the processor into RED_state by setting the PSTATE.RED flag to one.

Once the processor has entered RED_state, no matter how it got there, all subsequent traps are processed in RED_state until software returns the processor to execute_state or a normal or SIR trap is taken when TL = MAXTL, which puts the processor in error_state. TABLE 12-7, TABLE 12-8, and TABLE 12-9 describe the processor mode and trap-level transitions involved in handling traps.

TABLE 12-7 Trap Received While in execute_state

	New State, After Receiving Trap Type			
Original State	**Normal Trap or Interrupt**	**POR**	**XIR, WDR**	**SIR**
execute_state TL < MAXTL – 1	execute_state TL ← TL + 1	RED_state TL = MAXTL	RED_state TL ← TL + 1	RED_state TL ← TL + 1
execute_state TL = MAXTL – 1	RED_state TL = MAXTL	RED_state TL = MAXTL	RED_state TL = MAXTL	RED_state TL = MAXTL
execute_state† TL = MAXTL	error_state TL = MAXTL	RED_state TL = MAXTL	RED_state TL = MAXTL	error_state TL = MAXTL

†This state occurs when software changes TL to MAXTL and does not set PSTATE.RED, or if it clears PSTATE.RED while at MAXTL.

TABLE 12-8 Trap Received While in `RED_state`

Original State	New State, After Receiving Trap Type			
	Normal Trap or Interrupt	**POR**	**XIR, WDR**	**SIR**
`RED_state` $\mathtt{TL} < \mathtt{MAXTL} - 1$	`RED_state` $\mathtt{TL} \leftarrow \mathtt{TL} + 1$	`RED_state` `TL = MAXTL`	`RED_state` $\mathtt{TL} \leftarrow \mathtt{TL} + 1$	`RED_state` $\mathtt{TL} \leftarrow \mathtt{TL} + 1$
`RED_state` $\mathtt{TL} = \mathtt{MAXTL} - 1$	`RED_state` `TL = MAXTL`	`RED_state` `TL = MAXTL`	`RED_state` `TL = MAXTL`	`RED_state` `TL = MAXTL`
`RED_state` $\mathtt{TL} = \mathtt{MAXTL}$	`error_state` `TL = MAXTL`	`RED_state` `TL = MAXTL`	`RED_state` `TL = MAXTL`	`error_state` `TL = MAXTL`

TABLE 12-9 Reset Received While in `error_state`

Original State	New State, After Receiving Trap Type			
	Normal Trap or Interrupt	**POR**	**XIR, WDR**	**SIR**
`error_state` $\mathtt{TL} < \mathtt{MAXTL} - 1$	—	`RED_state` `TL = MAXTL`	`RED_state` $\mathtt{TL} \leftarrow \mathtt{TL} + 1$	—
`error_state` $\mathtt{TL} = \mathtt{MAXTL} - 1$	—	`RED_state` `TL = MAXTL`	`RED_state` `TL = MAXTL`	—
`error_state` $\mathtt{TL} = \mathtt{MAXTL}$	—	`RED_state` `TL = MAXTL`	`RED_state` `TL = MAXTL`	—

Implementation Note – The processor does not recognize interrupts while it is in `error_state`.

12.5.1 Normal Trap Processing

A trap other than a fast MMU trap (see Section 12.5.2, "Fast MMU Trap Processing") or an interrupt vector trap (see Section 12.5.3, "Interrupt Vector Trap Processing") causes the following state changes to occur[1]:

- If the processor is already in `RED_state`, the new trap is processed in `RED_state` unless $\mathtt{TL} = \mathtt{MAXTL}$.

1. Please note that these state transitions include the minimum set covered in the SPARC V9 architecture. There are other non-SPARC V9 transitions that are implementation-dependent and are not described here.

- If the processor is in `execute_state` and the trap level is one less than its maximum value, that is, `TL` = `MAXTL`−1, then the processor enters `RED_state`.
- If the processor is in either `execute_state` or `RED_state` and the trap level is already at its maximum value, that is, `TL` = `MAXTL`, then the processor enters `error_state`.

Otherwise, the trap uses normal trap processing, and the following state changes occur:

- The trap level is set. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack.

  `TL` ← `TL` + 1

- Existing state is preserved.

`TSTATE[TL].CCR`	← `CCR`
`TSTATE[TL].ASI`	← `ASI`
`TSTATE[TL].PSTATE`	← `PSTATE`
`TSTATE[TL].CWP`	← `CWP`
`TPC[TL]`	← `PC`
`TNPC[TL]`	← `nPC`

- The trap type is preserved.

`TT[TL]`	← the trap type

- The `PSTATE` register is updated to a predefined state.

`PSTATE.MM`	is unchanged
`PSTATE.RED`	← `0`
`PSTATE.PEF`	← `1` (FPU is present)
`PSTATE.AM`	← `0` (address masking is turned off)
`PSTATE.PRIV`	← `1` (the processor enters privileged mode)
`PSTATE.IE`	← `0` (interrupts are disabled)
`PSTATE.AG`	← `1` (global registers are replaced with alternate globals)
`PSTATE.MG`	← `0` (MMU globals are disabled)
`PSTATE.IG`	← `0` (interrupt globals are disabled)
`PSTATE.CLE`	← `PSTATE.TLE` (set endian mode for traps)
`PSTATE.TLE`	is unchanged

- For a register-window trap only, `CWP` is set to point to the register window that must be accessed by the trap-handler software, that is:
  - If `TT[TL]` = 024_{16} (a *clean_window* trap), then `CWP` ← `CWP` + 1.
  - If (080_{16} ≤ `TT[TL]` ≤ $0BF_{16}$) (window *spill* trap), then `CWP` ← `CWP` + `CANSAVE` + 2.
  - If ($0C0_{16}$ ≤ `TT[TL]` ≤ $0FF_{16}$) (window *fill* trap), then `CWP` ← `CWP` – 1.

  For non-register-window traps, `CWP` is not changed.

- Control is transferred into the trap table:

  `PC` ← `TBA`<63:15> □ (`TL` > 0) □ `TT[TL]` □ 0 0000

  `nPC` ← `TBA`<63:15> □ (`TL` > 0) □ `TT[TL]` □ 0 0100

  where "(`TL` > 0)" is zero if `TL` = 0, and one if `TL` > 0.

Interrupts are ignored as long as PSTATE.IE = 0.

Programming Note – State in TPC[*n*], TNPC[*n*], TSTATE[*n*], and TT[*n*] is only changed autonomously by the processor when a trap is taken while TL = *n* – 1; however, software can change any of these values with a WRPR instruction when TL = *n*.

12.5.2 Fast MMU Trap Processing

Fast MMU traps (*fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection*) cause the following state changes to occur[1]:

- If the processor is already in RED_state, the new trap is processed in RED_state unless TL = MAXTL.
- If the processor is in execute_state and the trap level is one less than its maximum value, that is, TL = MAXTL – 1, then the processor enters RED_state.
- If the processor is in either execute_state or RED_state and the trap level is already at its maximum value, that is, TL = MAXTL, then the processor enters error_state.

Otherwise, the trap uses normal trap processing, and the following state changes occur:

- The trap level is set. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack.

  ```
  TL ← TL + 1
  ```

- Existing state is preserved:

  ```
  TSTATE[TL].CCR     ← CCR
  TSTATE[TL].ASI     ← ASI
  TSTATE[TL].PSTATE  ← PSTATE
  TSTATE[TL].CWP     ← CWP
  TPC[TL]            ← PC
  TNPC[TL]           ← nPC
  ```

- The trap type is preserved.

  ```
  TT[TL]             ← the trap type
  ```

- The PSTATE register is updated to a predefined state.

  ```
  PSTATE.MM          is unchanged
  PSTATE.RED         ← 0
  PSTATE.PEF         ← 1 (FPU is present)
  PSTATE.AM          ← 0 (address masking is turned off)
  PSTATE.PRIV        ← 1 (the processor enters privileged mode)
  PSTATE.IE          ← 0 (interrupts are disabled)
  PSTATE.AG          ← 0 (alternate globals are disabled)
  ```

1. Please note that these state transitions include the minimum set covered in the SPARC V9 architecture. There are other non-SPARC V9 transitions that are implementation-dependent and are not described here.

PSTATE.MG	← 1 (global registers are replaced with MMU globals)
PSTATE.IG	← 0 (interrupt globals are disabled)
PSTATE.CLE	← PSTATE.TLE (set endian mode for traps)
PSTATE.TLE	is unchanged

- For non-register-window traps, CWP is not changed.
- Control is transferred into the trap table:

  PC ← TBA<63:15> ▯ (TL > 0) ▯ TT[TL] ▯ 0 0000

  nPC ← TBA<63:15> ▯ (TL > 0) ▯ TT[TL] ▯ 0 0100

  where "(TL > 0)" is zero if TL = 0, and one if TL > 0.

  Interrupts are ignored as long as PSTATE.IE = 0.

Programming Note – State in TPC[*n*], TNPC[*n*], TSTATE[*n*], and TT[*n*] is only changed autonomously by the processor when a trap is taken while TL = *n* – 1; however, software can change any of these values with a WRPR instruction when TL = *n*.

12.5.3 Interrupt Vector Trap Processing

An *interrupt_vector* trap causes the following state changes to occur[1]:

- If the processor is already in RED_state, the new trap is processed in RED_state unless TL = MAXTL.
- If the processor is in execute_state and the trap level is one less than its maximum value, that is, TL = MAXTL – 1, the processor enters RED_state.
- If the processor is in either execute_state or RED_state and the trap level is already at its maximum value, that is, TL = MAXTL, then the processor enters error_state.

Otherwise, the trap uses normal trap processing, and the following state changes occur:

- The trap level is set. This provides access to a fresh set of privileged trap-state registers used to save the current state, in effect, pushing a frame on the trap stack.

  TL ← TL + 1

- Existing state is preserved.

TSTATE[TL].CCR	← CCR
TSTATE[TL].ASI	← ASI
TSTATE[TL].PSTATE	← PSTATE
TSTATE[TL].CWP	← CWP
TPC[TL]	← PC
TNPC[TL]	← nPC

- The trap type is preserved.

1. Please note that these state transitions include the minimum set covered in the SPARC V9 architecture. There are other non-SPARC V9 transitions that are implementation-dependent and are not described here.

TT[TL]	← the trap type

- The PSTATE register is updated to a predefined state.

PSTATE.MM	is unchanged
PSTATE.RED	← 0
PSTATE.PEF	← 1 (FPU is present)
PSTATE.AM	← 0 (address masking is turned off)
PSTATE.PRIV	← 1 (the processor enters privileged mode)
PSTATE.IE	← 0 (interrupts are disabled)
PSTATE.AG	← 0 (alternate globals are disabled)
PSTATE.MG	← 0 (MMU globals are disabled)
PSTATE.IG	← 1 (global registers are replaced with interrupt globals)
PSTATE.CLE	← PSTATE.TLE (set endian mode for traps)
PSTATE.TLE	is unchanged

- For non-register-window traps, CWP is not changed.
- Control is transferred into the trap table:

PC	← TBA<63:15> ▯ (TL > 0) ▯ TT[TL] ▯ 0 0000
nPC	← TBA<63:15> ▯ (TL > 0) ▯ TT[TL] ▯ 0 0100

where "(TL > 0)" is zero if TL = 0, and one if TL > 0.

Interrupts are ignored as long as PSTATE.IE = 0.

Programming Note – State in TPC[*n*], TNPC[*n*], TSTATE[*n*], and TT[*n*] is only changed autonomously by the processor when a trap is taken while TL = *n* – 1; however, software can change any of these values with a WRPR instruction when TL = *n*.

12.5.4 Special Trap Processing

The following conditions invoke special trap processing:

- Traps taken with TL = MAXTL – 1
- Power-on reset traps
- Watchdog reset traps
- Externally initiated reset traps
- Software-initiated reset traps
- Traps taken when the processor is already in RED_state

12.5.4.1 Normal Traps with TL = MAXTL – 1

Normal traps that occur when TL = MAXTL – 1 are processed in RED_state. The following state changes occur[1]:

- The trap level is advanced.

  `TL` ← `MAXTL`

- Existing state is preserved.

`TSTATE[TL].CCR`	←	`CCR`
`TSTATE[TL].ASI`	←	`ASI`
`TSTATE[TL].PSTATE`	←	`PSTATE`
`TSTATE[TL].CWP`	←	`CWP`
`TPC[TL]`	←	`PC`
`TNPC[TL]`	←	`nPC`

- The trap type is preserved.

  `TT[TL]` ← the trap type

- The `PSTATE` register is set as follows:

`PSTATE.MM`	←	00_2 (TSO)
`PSTATE.RED`	←	`1` (enter `RED_state`)
`PSTATE.PEF`	←	`1` (FPU is present)
`PSTATE.AM`	←	`0` (address masking is turned off)
`PSTATE.PRIV`	←	`1` (the processor enters privileged mode)
`PSTATE.IE`	←	`0` (interrupts are disabled)
`PSTATE.AG`	←	`1` (global registers are replaced with alternate globals)
`PSTATE.MG`	←	`0` (MMU globals are disabled)
`PSTATE.IG`	←	`0` (interrupt globals are disabled)
`PSTATE.CLE`	←	`PSTATE.TLE` (set endian mode for traps)
`PSTATE.TLE`	←	undefined[1]

- For a register-window trap only, `CWP` is set to point to the register window that must be accessed by the trap-handler software, that is:
  - If `TT[TL]` = 024_{16} (a *clean_window* trap), then `CWP` ← `CWP` + 1.
  - If (080_{16} ≤ `TT[TL]` ≤ $0BF_{16}$) (window *spill* trap), then `CWP` ← `CWP` + `CANSAVE` + 2.
  - If ($0C0_{16}$ ≤ `TT[TL]` ≤ $0FF_{16}$) (window *fill* trap), then `CWP` ← `CWP` – 1.

  For non-register-window traps, `CWP` is not changed.

- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the `RED_state` trap table.

  `PC` ← `RSTVaddr`<63:8> ▯ $1010\ 0000_2$

  `nPC` ← `RSTVaddr`<63:8> ▯ $1010\ 0100_2$

1. Please note that these state transitions include the minimum set covered in the SPARC V9 architecture. There are other non-SPARC V9 transitions that are implementation-dependent and are not described here.

1. Note that this differs from SPARC V9.

12.5.4.2 Power-On Reset (POR) Traps

POR traps occur when power is applied to the processor. If the processor is in `error_state`, a POR brings the processor out of `error_state` and places it in `RED_state`. Processor state is undefined after POR, except for the following[1]:

- The trap level is set.

  `TL` ← `MAXTL`

- The trap type is set.

  `TT[TL]` ← 001_{16}

- The `PSTATE` register is set as follows:

`PSTATE.MM`	← 00_2 (TSO)
`PSTATE.RED`	← `1` (enter `RED_state`)
`PSTATE.PEF`	← `1` (FPU is present)
`PSTATE.AM`	← `0` (address masking is turned off)
`PSTATE.PRIV`	← `1` (the processor enters privileged mode)
`PSTATE.IE`	← `0` (interrupts are disabled)
`PSTATE.AG`	← `1` (global registers are replaced with alternate globals)
`PSTATE.MG`	← `0` (MMU globals are disabled)
`PSTATE.IG`	← `0` (interrupt globals are disabled)
`PSTATE.CLE`	← `0` (big-endian mode for nontraps)
`PSTATE.TLE`	← `0` (big-endian mode for traps)

- The `TICK` register is protected.

  `TICK.NPT` ← `1` (`TICK` unreadable by non-privileged software)

- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the `RED_state` trap table.

  `PC` ← `RSTVaddr`<63:8> ▯ $0010\ 0000_2$

  `nPC` ← `RSTVaddr`<63:8> ▯ $0010\ 0100_2$

For any reset when `TL` = `MAXTL`, for all n < `MAXTL`, the values in `TPC`[n], `TNPC`[n], and `TSTATE`[n] are undefined.

12.5.4.3 Watchdog Reset (WDR) Traps

The WDR reset in the UltraSPARC III Cu processor provide automatic recovery from `error_state`.

Processor state is undefined after WDR, except for the following[1]:

- The trap level is set.

  `TL` ← **min** (`TL` + 1, `MAXTL`)

1. Please note that these state transitions include the minimum set covered in the SPARC V9 architecture. There are other non-SPARC V9 transitions that are implementation-dependent and are not described here.

- Existing state is preserved.

TSTATE[TL].CCR	←	CCR
TSTATE[TL].ASI	←	ASI
TSTATE[TL].PSTATE	←	PSTATE
TSTATE[TL].CWP	←	CWP
TPC[TL]	←	PC
TNPC[TL]	←	nPC

- The trap type is set.

  TT[TL] ← 002_{16}

- The PSTATE register is set as follows:

PSTATE.MM	←	00_2 (TSO)
PSTATE.RED	←	1 (enter RED_state)
PSTATE.PEF	←	1 (FPU is present)
PSTATE.AM	←	0 (address masking is turned off)
PSTATE.PRIV	←	1 (the processor enters privileged mode)
PSTATE.IE	←	0 (interrupts are disabled)
PSTATE.AG	←	1 (global registers are replaced with alternate globals)
PSTATE.MG	←	0 (MMU globals are disabled)
PSTATE.IG	←	0 (interrupt globals are disabled)
PSTATE.CLE	←	PSTATE.TLE (set endian mode for traps)
PSTATE.TLE	←	undefined[1]

- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the RED_state trap table.

PC	←	RSTVaddr<63:8> ▯ $0100\ 0000_2$
nPC	←	RSTVaddr<63:8> ▯ $0100\ 0100_2$

For any reset when TL = MAXTL, for all n < MAXTL, the values in TPC[n], TNPC[n], and TSTATE[n] are undefined.

12.5.4.4 Externally Initiated Reset (XIR) Traps

XIR traps are initiated by an external signal. They behave like an interrupt that cannot be masked by IE = 0 or PIL. Typically, XIR is used for critical system events, such as power failure, reset button pressed, failure of external components, that does not require a WDR (which aborts operations), or systemwide reset in a multiprocessor. The following state changes occur[2]:

- Existing state is preserved.

1. Note that this differs from SPARC V9.

2. Please note that these state transitions include the minimum set covered in the SPARC V9 architecture. There are other non-SPARC V9 transitions that are implementation-dependent and are not described here.

TSTATE[TL].CCR ← CCR
TSTATE[TL].ASI ← ASI
TSTATE[TL].PSTATE ← PSTATE
TSTATE[TL].CWP ← CWP
TPC[TL] ← PC
TNPC[TL] ← nPC

- The trap type is set.

  TT[TL] ← 003_{16}

- The PSTATE register is set as follows:

  PSTATE.MM ← 00_2 (TSO)
  PSTATE.RED ← 1 (enter RED_state)
  PSTATE.PEF ← 1 (FPU is present)
  PSTATE.AM ← 0 (address masking is turned off)
  PSTATE.PRIV ← 1 (the processor enters privileged mode)
  PSTATE.IE ← 0 (interrupts are disabled)
  PSTATE.AG ← 1 (global registers are replaced with alternate globals)
  PSTATE.MG ← 0 (MMU globals are disabled)
  PSTATE.IG ← 0 (interrupt globals are disabled)
  PSTATE.CLE ← PSTATE.TLE (set endian mode for traps)
  PSTATE.TLE ← undefined[1]

- Implementation-specific state changes, for example, disabling an MMU.
- Control is transferred into the RED_state trap table.

  PC ← RSTVaddr<63:8> ▯ $0110\ 0000_2$
  nPC ← RSTVaddr<63:8> ▯ $0110\ 0100_2$

For any reset when TL = MAXTL, for all *n* < MAXTL, the values in TPC[*n*], TNPC[*n*], and TSTATE[*n*] are undefined.

12.5.4.5 Software-Initiated Reset (SIR) Traps

SIR traps are initiated by execution of an SIR instruction in privileged mode. Supervisor software uses the SIR trap as a panic operation or a metasupervisor trap.

The following state changes occur[1]:

- If TL = MAXTL, then enter error_state. Otherwise, do the following:
- The trap level is set.

  TL ← TL + 1

- Existing state is preserved.

1. Please note that these state transitions include the minimum set covered in theSPARC V9 architecture. There are other non-SPARC V9 transitions that are implementation-dependent and are not described here.

`TSTATE[TL].CCR`	← `CCR`
`TSTATE[TL].ASI`	← `ASI`
`TSTATE[TL].PSTATE`	← `PSTATE`
`TSTATE[TL].CWP`	← `CWP`
`TPC[TL]`	← `PC`
`TNPC[TL]`	← undefined[1]

- The trap type is set.

  `TT[TL]` ← 04_{16}

- The `PSTATE` register is set as follows:

`PSTATE.MM`	← 00_2 (TSO)
`PSTATE.RED`	← `1` (enter `RED_state`)
`PSTATE.PEF`	← `1` (FPU is present)
`PSTATE.AM`	← `0` (address masking is turned off)
`PSTATE.PRIV`	← `1` (the processor enters privileged mode)
`PSTATE.IE`	← `0` (interrupts are disabled)
`PSTATE.AG`	← `1` (global registers are replaced with alternate globals)
`PSTATE.MG`	← `0` (MMU globals are disabled)
`PSTATE.IG`	← `0` (interrupt globals are disabled)
`PSTATE.CLE`	← `PSTATE.TLE` (set endian mode for traps)
`PSTATE.TLE`	← undefined[2]

- Implementation-specific state changes, for example, disabling an MMU.
- Control is transferred into the RED_state trap table.

`PC`	← `RSTVaddr`<63:8> ▯ $1000\ 0000_2$
`nPC`	← `RSTVaddr`<63:8> ▯ $1000\ 0100_2$

For any reset when `TL` = `MAXTL`, for all *n* < `MAXTL`, the values in `TPC`[*n*], `TNPC`[*n*], and `TSTATE`[*n*] are undefined.

12.5.4.6 Normal Traps When the Processor Is in RED_state

Normal traps taken when the processor is already in `RED_state` are also processed in `RED_state`, unless `TL` = `MAXTL`, in which case the processor enters `error_state`.

Assuming that `TL` < `MAXTL`, the processor state shall be set as follows[1]:

- The trap level is set.

  `TL` ← `TL` + 1

- Existing state is preserved.

1. Please note that these state transitions include the minimum set covered in the SPARC V9 architecture. There are other non-SPARC V9 transitions that are implementation-dependent and are not described here.

2. Note that this differs from SPARC V9.

TSTATE[TL].CCR	← CCR
TSTATE[TL].ASI	← ASI
TSTATE[TL].PSTATE	← undefined[1]
TSTATE[TL].CWP	← CWP
TPC[TL]	← PC
TNPC[TL]	← nPC

- The trap type is preserved.

  TT[TL] ← trap type

- The PSTATE register is set as follows:

PSTATE.MM	← 00_2 (TSO)
PSTATE.RED	← 1 (enter RED_state)
PSTATE.PEF	← 1 (FPU is present)
PSTATE.AM	← 0 (address masking is turned off)
PSTATE.PRIV	← 1 (the processor enters privileged mode)
PSTATE.IE	← 0 (interrupts are disabled)
PSTATE.AG	← 1 (global registers are replaced with alternate globals)
PSTATE.MG	← 0 (MMU globals are disabled)
PSTATE.IG	← 0 (interrupt globals are disabled)
PSTATE.CLE	← PSTATE.TLE (set endian mode for traps)
PSTATE.TLE	← undefined[1]

- For a register-window trap only, CWP is set to point to the register window that must be accessed by the trap-handler software, that is:
  - If TT[TL] = 024_{16} (a *clean_window* trap), then CWP ← CWP + 1.
  - If (080_{16} ≤ TT[TL] ≤ $0BF_{16}$) (window *spill* trap), then CWP ← CWP + CANSAVE + 2.
  - If ($0C0_{16}$ ≤ TT[TL] ≤ $0FF_{16}$) (window *fill* trap), then CWP ← CWP – 1.
- For non-register-window traps, CWP is not changed.
- Implementation-specific state changes; for example, disabling an MMU.
- Control is transferred into the RED_state trap table.

  PC ← RSTVaddr<63:8> ▯ $1010\ 0000_2$

  nPC ← RSTVaddr<63:8> ▯ $1010\ 0100_2$

12.6 Exception and Interrupt Descriptions

This section describes the various exceptions and interrupt requests and the conditions that cause them. Each exception and interrupt request describes the corresponding trap type as defined by the trap model. On the UltraSPARC III Cu processor, all traps are precise except for the deferred traps and disrupting traps.

1. Note that this differs from SPARC V9

- ***data_access_exception*** [`tt` = 030_{16}] (Precise) — An exception occurred on an attempted data access. Detailed information regarding the error is logged into the `FTYPE` field of Data Synchronous Fault Status Register (ASI 58_{16}, VA = 18_{16}). Below is a list of exceptions that cause a *data_access_exception* exception.
  - **Invalid ASI** — An attempt to perform a load or store with undefined or reserved ASI or a disallowed instruction/ASI combination.
  - **Illegal Access to Strongly Ordered Page** — An attempt to access a strongly ordered page by any type of load instruction with non-faulting ASI or by `FLUSH` instruction.
  - **Illegal Access to Non-Faulting-Only Page** — An attempt to access a non-faulting-only page by any type of load or store instruction or `FLUSH` instruction with ASI other than non-faulting ASI.
  - **Illegal Access to Non-cacheable Page** —An attempt to access a non-cacheable page by atomic instructions (`CASA`, `CASXA`, `SWAP`, `SWAPA`, `LDSTUB`, `LDSTUBA`), atomic quad load instructions (`LDDA` with ASI = 24_{16}, $2C_{16}$), or by `FLUSH` instruction.
- ***division_by_zero*** [`tt` = 028_{16}] (Precise) — An integer divide instruction attempted to divide by zero**.**
- ***fill_n_normal*** [`tt` = $0C0_{16}$–$0DF_{16}$] (Precise)
- ***fill_n_other*** [`tt` = $0E0_{16}$–$0FF_{16}$] (Precise)

  A `RESTORE` or `RETURN` instruction has determined that the contents of a register window must be restored from memory.

Compatibility Note – The SPARC V9 *fill_n_** exceptions supersede the SPARC V8 *window_underflow* exception.

- ***fp_disabled*** [`tt` = 020_{16}] (Precise) — An attempt was made to execute a Floating-point operate (FPop), a floating-point branch, or a floating-point load/store instruction while an FPU was not present, `PSTATE.PEF` = 0, or `FPRS.FEF` = 0.
- ***illegal_instruction*** [`tt` = 010_{16}] (Precise) — An attempt was made to execute an instruction with an unimplemented opcode, an `ILLTRAP` instruction, an instruction with invalid field usage, instruction breakpoints, or an instruction that would result in illegal processor state.

Note – Unimplemented FPop instructions generate *fp_exception_other* traps.

An *illegal_instruction* is generated in the following cases:

- An instruction encoding does not match any of the opcode map definitions.
- An instruction is not implemented in hardware (if the `op` and `op3` fields of the instruction decode as an FPop, then a *fp_exception_other* exception, with `ftt` = 3, will be generated instead of *illegal_instruction*).
- An illegal value is present in an instruction `i` field.
- An illegal value is present in a field that is explicitly defined for an instruction, such as `cc2`, `cc1`, `cc0`, `fcn`, `impl`, `op2` (`IMPDEP2A`, `IMPDEP2B`), `rcond`, or `opf_cc`.

- Illegal register alignment (such as odd `rd` value in a doubleword load instruction).
- `RDASR` instruction with `rs1` = 1, 7–14, 20–21, or 26–31.
- `RDASR` with `rs1` = 15 and nonzero `rd`.
- `RDPR` with `rs1` = 16–30.
- `RDPR` with `rs1` ≤ 3 when `TL` = 0.
- `WRPR` with `rd` = 15–31.
- `WRPR` with `rd` ≤ 3 when `TL` = 0.
- `WRPR` to `PSTATE` register that attempts to set more than one of bits `IG`, `MG`, and `AG`.
- Illegal `rd` value for `LDXFSR`, `STXFSR`, or the deprecated instructions `LDFSR` or `STFSR`.
- Illegal `rd` value for `WRPR`.
- Illegal `rs1` value for `RDPR`.
- `WRASR` instruction with `rd` = 1, 4, 5, 7–14, 26-31.
- `WRASR` with `rd` = 15 and nonzero `rs1`.
- `WRASR` with `rd` = 15 and `i` = 0.
- `DONE` or `RETRY` when `TL` = 0.
- `ILLTRAP` instruction.
- Instruction breakpoint occurred.
- A reserved instruction field in `Tcc` instruction is nonzero.

If a reserved instruction field in an instruction other than `Tcc` is nonzero, an *illegal_instruction* exception should be generated.[1]

Note – If an instruction breakpoint triggers an *illegal_instruction* trap, then the *illegal_instruction* trap has a higher priority than does a *privileged_opcode* trap.

- ***instruction_access_exception*** [`tt` = 008_{16}] (Precise) — A protection exception occurred on an instruction access, typically as a result of an attempt to access a privileged page while the processor was executing in non-privileged mode.
- ***interrupt_level_n*** [`tt` = 041_{16}–$04F_{16}$] (Disrupting) — An interrupt request level of *n* was presented to the IU, while `PSTATE.IE` = 1 and (interrupt request level > `PIL`).
- ***mem_address_not_aligned*** [`tt` = 034_{16}] (Precise) — A load/store instruction generated a memory address that was not properly aligned according to the instruction, or a `JMPL` or `RETURN` instruction generated a non-word-aligned address.
- ***power_on_reset*** (POR) [`tt` = 001_{16}] (Reset) — An external signal was asserted. This trap is issued to bring a system reliably from the power-off to the power-on state.
- ***privileged_action*** [`tt` = 037_{16}] (Precise) — An action defined to be privileged has been attempted while `PSTATE.PRIV` = 0. Some examples include: a data access by non-privileged software using an ASI value with its most significant bit = 0 (a restricted ASI), or an attempt to read the `TICK` register by non-privileged software when `TICK.NPT` = 1.
- ***privileged_opcode*** [`tt` = 011_{16}] (Precise) — An attempt was made to execute a privileged instruction while `PSTATE.PRIV` = 0.

1. It is required that reserved fields have zero value.

Compatibility Note – The trap type for *privileged_opcode* is identical to that of the SPARC V8 *privileged_instruction* trap. The name was changed to distinguish it from the new *privileged_action* trap type.

- ***RED_state_exception*** [`tt` = 005_{16}] — Caused when `TL` = `MAXTL` – 1 and a trap occurs, an event that brings the processor into `RED_state`.
- ***software_initiated_reset*** (SIR) [`tt` = 004_{16}] (Precise/Reset) — Caused by the execution of the `WRSIR`, write to `SIR` register, instruction. It allows system software to reset the processor.
- ***spill_n_normal*** [`tt` = 080_{16}–$09F_{16}$] (Precise)
- ***spill_n_other*** [`tt` = $0A0_{16}$–$0BF_{16}$] (Precise)

  A `SAVE` or `FLUSHW` instruction has determined that the contents of a register window must be saved to memory.

Compatibility Note – The SPARC V9 *spill_n_** exceptions supersede the SPARC V8 *window_overflow* exception.

- ***tag_overflow*** [`tt` = 023_{16}] (Precise) — A `TADDccTV` or `TSUBccTV` instruction was executed, and either 32-bit arithmetic overflow occurred or at least one of the tag bits of the operands was nonzero.
- ***trap_instruction*** [`tt` = 100_{16}–$17F_{16}$] (Precise) — A `Tcc` instruction was executed and the trap condition evaluated to `TRUE`.
- ***clean_window*** [`tt` = 024_{16}–027_{16}] (Precise) — A `SAVE` instruction discovered that the window about to be used contains data from another address space; the window must be cleaned before it can be used.
- ***data_access_error*** [`tt` = 032_{16}] (Precise or Deferred) — An error occurred on a data access.
- ***externally_initiated_reset*** (XIR) [`tt` = 003_{16}] (Reset) — An external signal was asserted. This trap is used for catastrophic events, such as power failure, reset button pressed, and systemwide reset, in multiprocessor systems.
- ***fp_exception_ieee_754*** [`tt` = 021_{16}] (Precise) — An FPop instruction generated an *IEEE_754_exception* and its corresponding trap enable mask (`TEM`) bit was one. The floating-point exception type, *IEEE_754_exception*, is encoded in the `FSR.ftt`, and specific *IEEE_754_exception* information is encoded in `FSR.cexc`.
- ***fp_exception_other*** [`tt` = 022_{16}] (Precise) — An FPop instruction generated an exception other than an *IEEE_754_exception*. Some examples include: the FPop is unimplemented, or there was a sequence or hardware error in the FPU. The floating-point exception type is encoded in the `FSR`'s `ftt` field.
- ***instruction_access_error*** [`tt` = $00A_{16}$] (Precise) — An error occurred on an instruction access.

- ***LDDF_mem_address_not_aligned*** [`tt` = 035_{16}] (Precise) — An attempt was made to execute an `LDDF` instruction and the effective address was not doubleword aligned.
- ***STDF_mem_address_not_aligned*** [`tt` = 036_{16}] (Precise) — An attempt was made to execute an `STDF` instruction and the effective address was not doubleword aligned.
- ***Watchdog reset*** (WDR) [`tt` = 002_{16}] (Reset) — This trap occurs as a transition from `error_state` to `RED_state`.
- ***ECC_error*** [`tt` = 063_{16}] (Disrupting) — The trap to signal the detection of hardware errors asynchronous to the instruction execution, or to request to save the information logged for the error that was detected and corrected by the processor.

Implementation Note – Some implementations may refer to this trap by the name "*corrected_ECC_error*."

- ***fast_data_access_MMU_miss*** [`tt` = 068_{16} –$06B_{16}$] (Precise) — During an attempted data access, the MMU detected that a translation lookaside buffer (TLB) did not contain a translation for the virtual address (that is, a TLB miss occurred). Four trap vectors are allocated for this trap, allowing a TLB miss handler of up to 32 instructions to fit within the trap vector area.
- ***fast_data_access_protection*** [`tt` = $06C_{16}$–$06F_{16}$] (Precise) — During an attempted data write access (by a store or load-store instruction), the instruction had appropriate access privilege but the MMU signalled that the location was write-protected (write to a read-only location). Note that on an UltraSPARC III Cu processor, an attempt to read or write to a privileged location while in non-privileged mode causes the higher priority *data_access_exception* instead of this exception. Four trap vectors are allocated for this trap, allowing a trap handler of up to 32 instructions to fit within the trap vector area.
- ***fast_instruction_access_MMU_miss*** [`tt` = 064_{16} –067_{16}] (Precise) — During an attempted instruction access, the MMU detected a TLB miss. Four trap vectors are allocated for this trap, allowing a trap handler of up to 32 instructions to fit within the trap vector area.
- ***interrupt_vector_trap*** [`tt` = 060_{16}] (Disrupting) — The processor has received an interrupt request.
- ***PA_watchpoint*** [`tt` = 061_{16}] (Precise) — The processor has detected a physical address breakpoint.
- ***VA_watchpoint*** [`tt` = 062_{16}] (Precise) — The processor has detected a virtual address breakpoint.
- ***fast_ECC_error*** [`tt` = 070_{16}] (Precise) — A single-bit or multiple-bit ECC error is detected.

  This trap is taken on ECC errors from the L2-cache. The trap handler is required to flush the cache line containing the error from both the D-cache and L2-cache since incorrect data would have already been written into the D-cache. The UltraSPARC III Cu hardware will automatically correct single-bit ECC errors on the L2-cache write back when the trap handler performs the L2-cache flush. After the caches are flushed, the instruction that encountered the error should be retried; the corrected data will then be brought back in from memory and reinstalled in the D-cache and L2-cache.

On *fast_ECC_error* detection during D-cache load miss fill, D-cache installs the uncorrected data. But since the fast_ECC_error trap is precise, hardware can rely on software to help clean up the bad data. I-cache is different. If I-cache is filled with errors, the line will not be installed in I-cache.

A D-cache or I-cache miss request may observe an ECC error in the line it reads from the L2-cache. When this occurs, a *fast_ECC_error* precise trap is generated for the instruction that detected the error. In the case of a D-cache request, the corrupted data will be installed in the D-cache, but the trap takes effects before the data can be used.

In case of an I-cache request, the data from the L2-cache will be corrected by hardware before being installed in the I-cache. When the *fast_ECC_error* trap is taken, the I-cache and D-cache are left enabled. Software is responsible to disable any caches as a part of the recovery. Software must flush the corrupted line from the D-cache if it was filled.

CHAPTER 13

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 the interrupter 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.

In the following sections, we describe different aspects of interrupt handling:

- Interrupt Vector Dispatch
- Interrupt Vector Receive
- Interrupt Global Registers
- Interrupt ASI Registers
- Software Interrupt Register (SOFTINT)

13.1 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`'s BUSY and NACK bits. A `MEMBAR #Sync` should be used before polling begins to ensure that earlier stores are completed. If both `NACK` and `BUSY` are cleared, the interrupt has been successfully delivered to the target processor. With the `NACK` bit cleared and `BUSY` bit set, the interrupt delivery is pending. Finally, if the delivery cannot be completed (if it is rejected by the target processor), the `NACK` bit is set. The pseudocode sequence in CODE EXAMPLE 13-1 sends an interrupt.

The `ASI_INTR_DISPATCH_STATUS` register contains 32 pairs of BUSY/NACK bit pairs enabling interrupts to be pipelined. Specifying a unique pair of BUSY/NACK bits to be used for each interrupt when writing the Interrupt Dispatch Register enables up to 32 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 13-1 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)
    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)
```

CODE EXAMPLE 13-1 Code Sequence for Interrupt Dispatch *(Continued)*

```
    Poll state of ASI_INTR_DISPATCH_STATUS (BUSY, NACK)
        Loop if BUSY
    End atomic sequence(PSTATE.IE ← 1)
```

Note – To avoid deadlocks, enable interrupts for some period before retrying the atomic sequence. Alternatively, implement the atomic sequence with locks without disabling interrupts.

13.2 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 (implementation-dependent 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 re-prioritized as lower priority interrupts in the software handler. CODE EXAMPLE 13-2 illustrates interrupt receive handling.

CODE EXAMPLE 13-2 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)
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
```

13.3 Interrupt Global Registers

A separate set of global registers is implemented to expedite interrupt processing. As described in Section 13.2, "Interrupt Vector Receive," the processor takes an implementation-dependent 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.

13.4 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.

13.4.1 Outgoing Interrupt Vector Data<7:0> Register

`ASI_INTR_W` (data 0): ASI = 77_{16}, VA<63:0> = 40_{16}
`ASI_INTR_W` (data 1): ASI = 77_{16}, VA<63:0> = 48_{16}
`ASI_INTR_W` (data 2): ASI = 77_{16}, VA<63:0> = 50_{16}
`ASI_INTR_W` (data 3): ASI = 77_{16}, VA<63:0> = 58_{16}
`ASI_INTR_W` (data 4): ASI = 77_{16}, VA<63:0> = 60_{16}
`ASI_INTR_W` (data 5): ASI = 77_{16}, VA<63:0> = 68_{16}
`ASI_INTR_W` (data 6): ASI = 77_{16}, VA<63:0> = 80_{16}
`ASI_INTR_W` (data 7): ASI = 77_{16}, VA<63:0> = 88_{16}

Name: `ASI_INTR_W`: Outgoing Interrupt Vector Data Registers (privileged, write-only)

TABLE 13-1 describes the register field of the eight Outgoing Interrupt Vector Data Registers.

TABLE 13-1 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.

13.4.2 Interrupt Vector Dispatch Register

ASI 77_{16}

VA<63:39> = 0
VA<38:29> = SID<9:0>
VA<28:24> = BUSY/NACK bit pair # (BN)
VA<23:14> = interrupt target identifier (ITID)
VA<13:0> = 70_{16}

Name: ASI_INTR_DISPATCH_W (interrupt dispatch) (Privileged, write-only)

TABLE 13-2 describes the fields of the Interrupt Vector Dispatch Register.

TABLE 13-2 Interrupt Vector Dispatch Register Format

Bits	Field	Type	Description
VA<28:24>	BN	W	Specifies which of the BUSY/NACK bit pairs to use for the interrupt. A 016 in this field (which current software is using) selects BUSY/NACK bits ASI_INTR_DISPATCH_STATUS<1:0> for backward compatibility. A 116 in this field selects BUSY/NACK bits ASI_INTR_DISPATCH_STATUS<3:2>.
VA<23:14>	ITID	W	Interrupt Target ID. Specifies the interrupt target CPU using the BUSY/NACK bit pair BN, along with the contents of the eight Interrupt Vector Data Registers.

A write to this ASI triggers an interrupt vector dispatch to the target CPU 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. The upper bits VA<28:14> steer the interrupt.

The UltraSPARC III Cu processor interprets all ten bits of VA<38:39> when the Interrupt Vector Dispatch Register is written.

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.

13.4.3 Interrupt Vector Dispatch Status Register

ASI 48_{16}

VA<63:0> = 0

Name: `ASI_INTR_DISPATCH_STATUS` (Privileged, read-only)

TABLE 13-3 describes the fields of the Interrupt Vector Dispatch Status Register.

TABLE 13-3 Interrupt Dispatch Status Register Format

Bits	Field	Type	Description
Odd	NACK	R	Set if interrupt dispatch has failed. Cleared at the start of every interrupt dispatch attempt; set when a dispatch has failed.
Even	BUSY	R	Set when there is an outstanding dispatch.

In the UltraSPARC III Cu processor, 32 BUSY/NACK pairs are implemented in the Interrupt Vector Dispatch Status Register.

The status of up to 32 outgoing interrupts can be read from `ASI_INTR_DISPATCH_STATUS` BUSY/NACK bits. This register contains up to 32 pairs of BUSY/NACK bit pairs: the pair at <1:0> is referred to as pair 0, <3:2> as pair 1, and so on up to pair 31 at bits <63:62>. The VA<28:24> 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.

13.4.4 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`

TABLE 13-4 describes the register field of the eight Incoming Interrupt Vector Data Registers.

TABLE 13-4 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.

13.4.5 Interrupt Vector Receive Register

ASI 49_{16}

VA<63:0> = 0

Name: `ASI_INTR_RECEIVE` (Privileged)

TABLE 13-5 describes the fields of the Interrupt Receive Register.

TABLE 13-5 Interrupt Receive Register Format

Bits	Field	Type	Description
63:11		R	Reserved.
10:6	`SID_U`	R	Most significant (upper) 5 bits of the physical module ID (`MID`) of the interrupter. Source ID bits <9:5> of interrupter.
5	`BUSY`	RW	Set when an interrupt vector is received. The `BUSY` bit must be cleared by software writing zero.
4:0	`SID_L`	R	Least significant (lower) 5 bits of the physical module ID (`MID`) of the interrupter.

The status of an incoming interrupt can be read from `ASI_INTR_RECEIVE`. The `BUSY` bit is cleared by writing zero to this register.

The UltraSPARC III Cu processor sets all ten physical module ID (`MID`) bits in the `SID_U` and `SID_L` fields of the Interrupt Vector Receive Register. These bits correspond to the interrupt ID of the interrupter.

Non-privileged access to the Interrupt Vector Receive Register causes a *privileged_action* trap.

13.5 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}) is used for conication 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*>.

Non-privileged access to this register causes a *privileged_opcode* trap.

13.5.1 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.

13.5.2 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 13-6 summarizes the `SOFTINT` ASRs.

TABLE 13-6 SOFTINT ASRs

ASR Value	ASR Name	Type	Description
1416	`SET_SOFTINT`	W	Set bit(s) in Soft Interrupt Register.
1516	`CLEAR_SOFTINT`	W	Clear bit(s) in Soft Interrupt Register.
1616	`SOFTINT`	RW	Per-processor Soft Interrupt Register.

SECTION VI

Performance Programming

CHAPTER 14

Performance Instrumentation

Performance instrumentation consists of processor event counters that can be used to gather statistics during program execution and calls that start and stop the gathering process. Many events can be monitored, two at a time, to gain information about the performance of the processor. Memory access and stall times, for example, can be measured using two, 32-bit Performance Instrumentation Counters (`PIC`s). The `PCR` and `PIC` are accessed through read /write Ancillary State Register instructions.

This chapter describes the performance instrumentation features in the following sections:

- Section 14.1, "Performance Control Register (PCR)"
- Section 14.2, "Performance Instrumentation Counter (PIC) Register"
- Section 14.3, "Performance Instrumentation Operation"
- Section 14.4, "Pipeline Counters"
- Section 14.5, "Cache Access Counters"
- Section 14.6, "Memory Controller Counters"
- Section 14.7, "Data Locality Counters for Scalable Shared Memory Systems"
- Section 14.8, "Miscellaneous Counters"
- Section 14.9, "PCR.SL and PCR.SU Encodings"

Supervisor/User Mode

Access to the PCR is privileged. Non-privileged accesses cause a *privileged_opcode* trap. Software can restrict non-privileged access to PICs by setting the `PCR.PRIV` field while in privileged mode. When `PCR.PRIV` = 1 (supervisor access only), an attempt by User Software to access the `PIC` register causes a *privileged_action* trap. Software can control event measurements in non-privilege or privileged modes by setting the `PCR.UT` (user trace) and `PCR.ST` (system trace) fields.[1]

1. The PCR has mode bits to enable the counters in privileged mode, non-privilege mode, or to count when in either mode. The mode setting affects both counters.

14.1 Performance Control Register (PCR)

The Performance Control Register (PCR) is used to select the events to monitor and provides control for counting in privileged and/or non-privileged modes.

The 64-bit PCR is accessed through read/write Ancillary State Register (ASR) instructions (RDASR/WRASR). PCR is located at ASRs 16 (10_{16}).

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 14-1 shows the details of the PCR. TABLE 14-1 describes the various fields of the PCR. Counts are collected in the PIC register (see Section 14.2, "Performance Instrumentation Counter (PIC) Register").

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

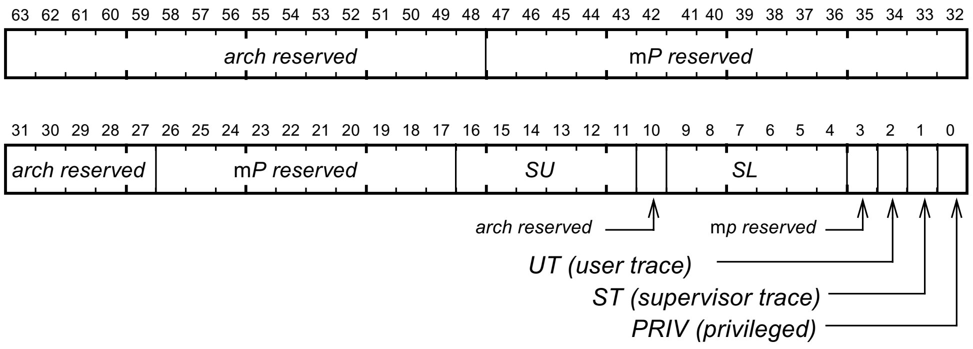


FIGURE 14-1 Performance Control Register

TABLE 14-1 PCR Bit Description

Bit	Field	Description
63:48	—	Reserved by SPARC architecture. Read zero, write zero, or write value read previously (read-modify-write).
47:32	—	Unused UltraSPARC III Cu processor. Read zero, write zero, or write value read previously (read-modify-write).
31:27	—	Reserved by SPARC architecture. Read zero, write zero, or write value read previously (read-modify-write).
26:17	—	Unused UltraSPARC III Cu processor. Read zero, write zero, or write value read previously (read-modify-write).
16:11	SU	Selects 1of up to 64 counters accessible in the upper half (bits <63:32>) of the PIC register.
10	—	Reserved by SPARC architecture. Read zero, write zero, or write value read previously (read-modify-write).
9:4	SL	Selects 1 of up to 64 counters accessible in the lower half (bits <31:0>) of the PIC register.
3	—	Unused UltraSPARC III Cu processor. Read zero, write zero, or write value read previously (read-modify-write).
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.

14.2 Performance Instrumentation Counter (PIC) Register

The 64-bit PIC is accessed through read/write Ancillary State Register (ASR) instructions (RDASR/WRASR). PIC is located at ASRs 17 (11_{16}).

The PIC 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 pair of 32-bit counters can accumulate over four billion events each prior to wrapping. Overflow of PICL or PICU causes a disrupting trap and SOFTINT. Active monitoring will allow the gathering software to extend the data range by periodically reading the contents of the PICs to detect and avoid overflow; an interrupt can be enabled on a counter overflow[1].

Each of the two 32-bit PICs can accumulate over 4 billion events before wrapping around. Overflow of PICL or PICU causes a disrupting trap and SOFTINT register bit 15 to be set to 1; then, if ((PSTATE.IE = 1) and (PIL < 15)), causes an *interrupt_level_15* trap. Extended event logging can be accomplished by periodic reading of the contents of the PICs before each overflows. Additional statistics can be collected by use of the two PICs over multiple passes of program execution.[2]

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. FIGURE 14-2 shows the details of the PIC. TABLE 14-2 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

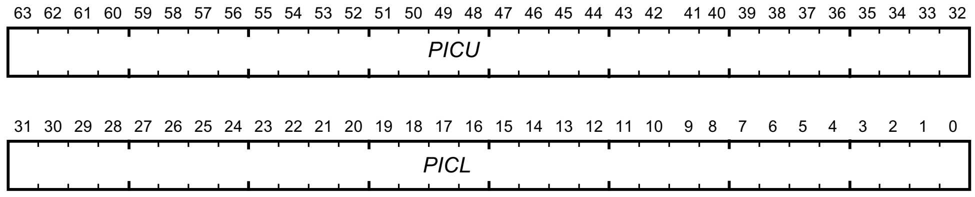


FIGURE 14-2 Performance Instrumentation Counter Register

1. The point at which the interrupt due to a PIC overflow is delivered may be several instructions after the instruction responsible for the overflow event. This situation is known as a "skid". The degree of skid depends on the event that caused the overflow and the type of instructions being executed in the pipeline at the time the overflow occurred. It may not be possible to associate a counter overflow with the particular instruction that caused it due to the skid problem.

2. Two events can simultaneously be measured by setting PCR.SU / PCR.SL fields along with the PCR.UT and PCR.ST fields. The selected statistics are reflected during subsequent accesses to the PICs.

TABLE 14-2 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)

14.2.1 PIC Counter Overflow Trap Operation

When a PIC counter overflows, an interrupt is generated as described in TABLE 14-3.

TABLE 14-3 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.”

14.3 Performance Instrumentation Operation

FIGURE 14-3 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[1] 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.

1. When more than two events need to be monitored, the program, code sequence, or code 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.

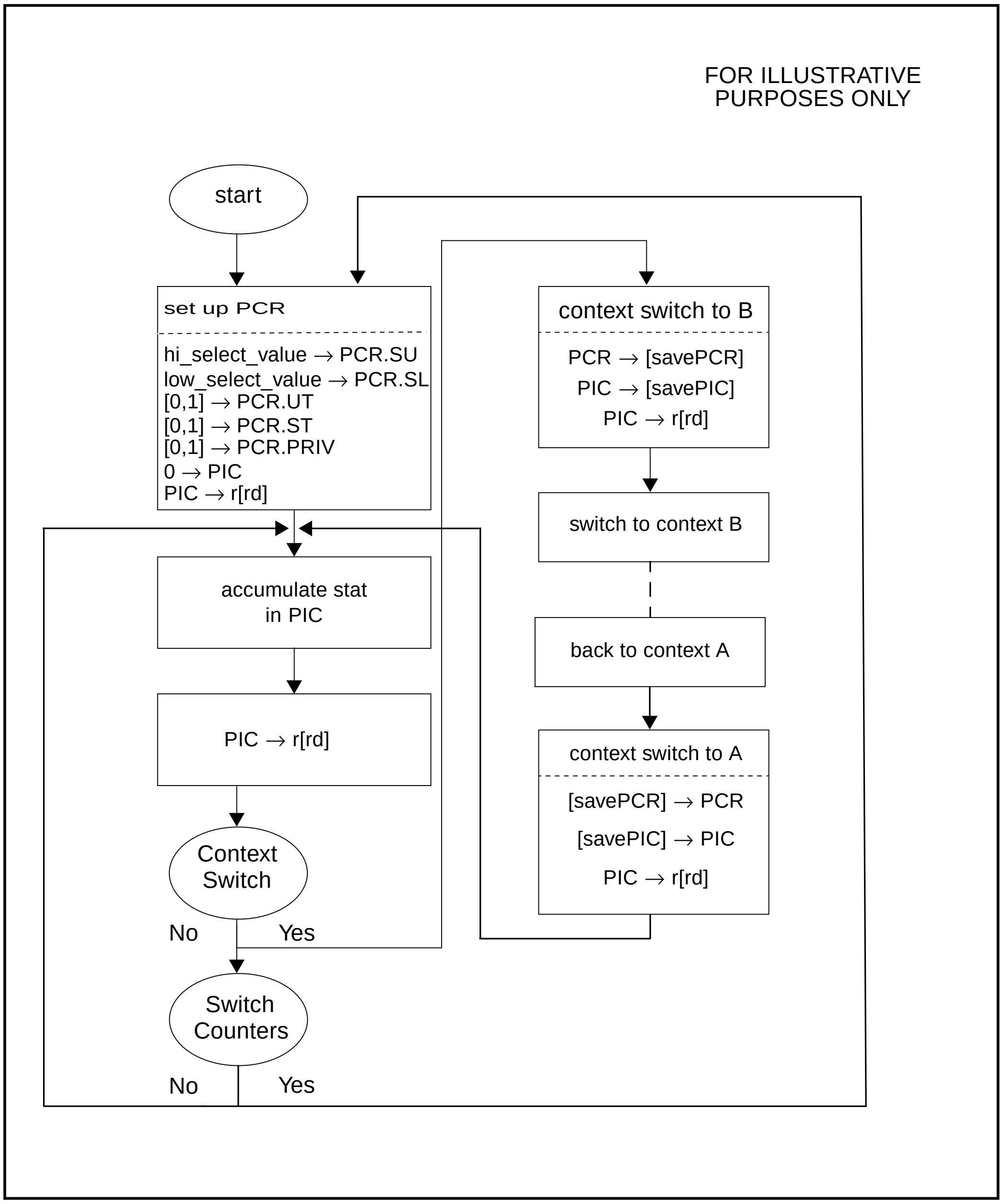


FIGURE 14-3 Operational Flow Diagram for Controlling Event Counters

14.3.1 Performance Instrumentation Implementations

Counting events and cycle stalls is sometimes complex because of the dynamic conditions and cancelled activities[1].

14.3.2 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 pin-point 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.

14.4 Pipeline Counters

14.4.1 Instruction Execution and CPU Clock Counts

The instruction execution count monitors are described in TABLE 14-4 for clock and instruction execution counts.

TABLE 14-4 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 (retired)[1].

1. This count does not include annulled, mispredicted, trapped, or helper instructions.

1. Specifics of each event and cycle stall are covered.

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`.

14.4.2 IIU Statistics

The counters listed in TABLE 14-5 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 14-5 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.

14.4.3 IIU Stall Counts

IIU stall counts, listed in TABLE 14-6 on page 14-371, 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 14-4 on page 14-371 illustrates the first two considerations described below.

14.4.3.1 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 to the reason it is empty. Multiple reasons 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 in Section 14.3.2, “Performance Instrumentation Accuracy” for all counters.

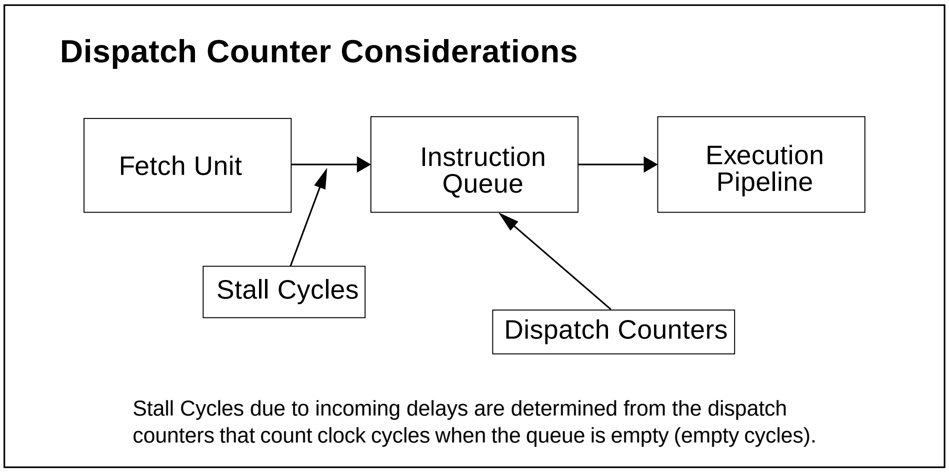

Dispatch Counter Considerations
Fetch Unit
Instruction Queue
Execution Pipeline
Stall Cycles
Dispatch Counters
Stall Cycles due to incoming delays are determined from the dispatch counters that count clock cycles when the queue is empty (empty cycles).

FIGURE 14-4 Dispatch Counters

TABLE 14-6 Counters for IIU Stalls

Counter	Description
`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.[1]
`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.[1]
`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.[1]
`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 re-fetched, delaying its entrance into the I-queue.[1]
`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]

1. See “Dispatch Counter Considerations” on page 370 for important information.

14.4.4 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 14-7 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 14-7 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.

14.4.5 Recirculation Counts

Recirculation instrumentation is implemented through the counters listed in TABLE 14-8.

TABLE 14-8 Counters for Recirculation

Counter	Description
`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 9.12, "Read-After-Write (RAW) Bypassing" for a description of the RAW hazard and causes of recirculation.
`Re_FPU_bypass`	`[PICU 00.0101]` Counts stall cycles due to recirculation when an 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.
`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 prefetch cache miss occurs on a prefetch predicted second load.

14.5 Cache Access Counters

Instruction, data, prefetch, write, and L2-cache access events can be collected through the counters listed in TABLE 14-9. Counts are updated by each cache access, regardless of whether the access will be used.

14.5.1 Instruction Cache Events

TABLE 14-9 describes the counters for instruction cache events.

TABLE 14-9 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.

14.5.2 Data Cache Events

TABLE 14-10 describes the counters for data cache events.

TABLE 14-10 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.

TABLE 14-10 Counters for Data Cache Events *(Continued)*

Counter	Description
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 D-TLB miss.

14.5.3 Write Cache Events

TABLE 14-11 describes the counters for write cache events.

TABLE 14-11 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.

14.5.4 Prefetch Cache Events

TABLE 14-12 describes the counters for prefetch cache events.

TABLE 14-12 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).

14.5.5 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 14-13 describes the counter for L2-cache events.[1]

TABLE 14-13 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.
`EC_write_hit_RTO`	`[PICL 00.1101]` Counts W-cache exclusive requests that hit L2-cache in S, O, or O_S state and thus, do a read-to-own (RTO) bus transaction.
`EC_wb`	`[PICU 00.1101]` Counts dirty sub-blocks 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 00.1110]` 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.

1. A block load or store access is counted as eight (8) references. For atomics, the read and write events are counted individually.

14.5.6 Separating D-cache Stall Counts

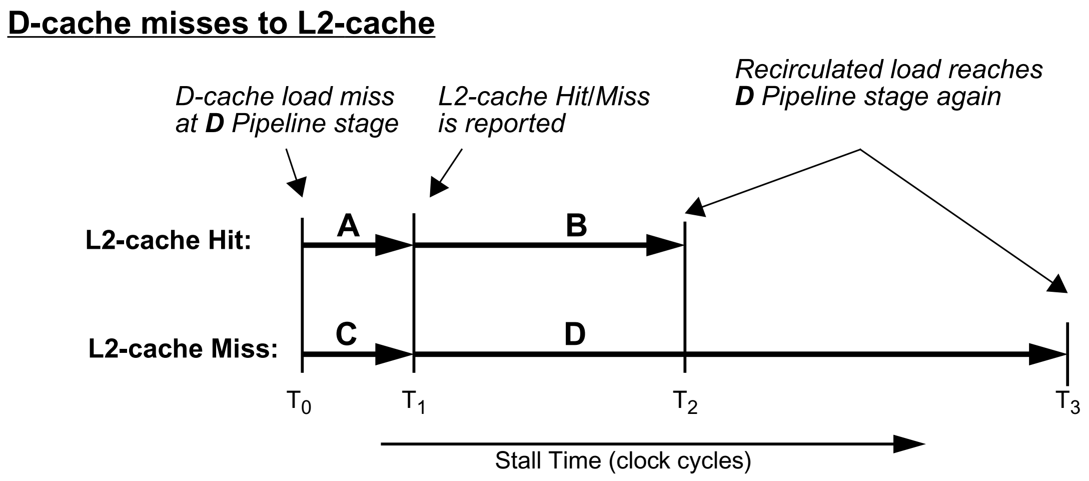
The D-cache stall cycle counts can be measured separately for L2-cache hits and misses by using the *Re_DC_missovhd* counter. The *Re_DC_missovhd* stall cycle counter is used with the recirculation and cache access events to separately calculate the D-cache loads that hit and miss the L2-cache. TABLE 14-14 describes the *Re_DC_missovhd* stall cycle counter processor compatibility.

TABLE 14-14 *Re_DC_missovhd* Stall Counter Processor Compatibility

Function	**Description**
Re_DC_missovhd	`[PICL 00.0100]`The *Re_DC_missovhd* cycle stall counter.

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 14-5 and the definitions and calculations that follow.


D-cache misses to L2-cache
D-cache load miss at D Pipeline stage
L2-cache Hit/Miss is reported
Recirculated load reaches D Pipeline stage again
L2-cache Hit:
L2-cache Miss:
A
B
C
D
T0
T1
T2
T3
Stall Time (clock cycles)

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

$$\text{Fraction of D-cache misses that miss L2-cache} = \frac{\text{miss L2}}{\text{miss D-cache}} = \frac{\textit{EC\_rd\_miss}\text{ (events)}}{\textit{DC\_rd\_miss}\text{ (events)}} = \text{Miss L2 Ratio}$$

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)

FIGURE 14-5 D-Cache Load Miss Stall Regions

14.6 Memory Controller Counters

This section describes the memory controller counters in the UltraSPARC III Cu processor.

14.6.1 Memory Controller Read Request Events

If the snoop indicates that the line exists in some other device, the memory read request is cancelled. The cancelled read requests are not counted. TABLE 14-15 describes the counters for memory controller read events.

TABLE 14-15 Counters for Memory Controller Read Events

Counter	Description
`MC_reads_0`	`[PICL 10.0000]` Counts read requests completed to memory bank 0.
`MC_reads_1`	`[PICL 10.0001]` Counts read requests completed to memory bank 1.
`MC_reads_2`	`[PICL 10.0010]` Counts read requests completed to memory bank 2.
`MC_reads_3`	`[PICL 10.0011]` Counts read requests completed to memory bank 3.

14.6.1.1 Memory Controller Write Request Events

TABLE 14-16 describes the counters for memory controller write events.

TABLE 14-16 Counters for Memory Controller Write Events

Counter	Description
`MC_writes_0`	`[PICU 10.0000]` Counts write requests completed to memory bank 0.
`MC_writes_1`	`[PICU 10.0001]` Counts write requests completed to memory bank 1.
`MC_writes_2`	`[PICU 10.0010]` Counts write requests completed to memory bank 2.
`MC_writes_3`	`[PICU 10.0011]` Counts write requests completed to memory bank 3.

14.6.1.2 Memory Request Stall Cycles

The stall cycles may be generated due to bus contention, a bank being busy, data availability for a write, etc. TABLE 14-17 describes the counters for memory stall cycles.

TABLE 14-17 Counters for Memory Controller Stall Cycles

Counter	Description
`MC_stalls_0`	`[PICL 10.0100]` Counts clock cycles that requests were stalled in the MCU queues because bank 0 was busy with a previous request.
`MC_stalls_1`	`[PICU 10.0100]` Counts clock cycles that requests were stalled in the MCU queues because bank 1 was busy with a previous request.
`MC_stalls_2`	`[PICL 10.0101]` Counts clock cycles that requests were stalled in the MCU queues because bank 2 was busy with a previous request.
`MC_stalls_3`	`[PICU 10.0101]` Counts clock cycles that requests were stalled in the MCU queues because bank 3 was busy with a previous request.

14.7 Data Locality Counters for Scalable Shared Memory Systems

There are four data locality performance event counters in the UltraSPARC III Cu processor. These event counters are provided to improve the ability to monitor and exploit performance in Scalable Shared Memory (SSM) systems where there are multiprocessor system clusters using Shared Memory Protocol (SMP) that are tied to other clusters using fabric interconnect utilizing the SSM architecture. TABLE 14-18 describes the counters for data locality events.

TABLE 14-18 Counters for Data Locality Events

Counter	Description
`EC_miss_local`	`[PICL 01.1010]` Counts any transaction to an LPA for which the processor issues an RTS/RTO/RS transaction.
`EC_miss_mtag_remote`	`[PICL 01.1011] and [PICU 10.1000]` Counts any transaction to an LPA address in which the processor is required to generate a retry transaction.

TABLE 14-18 Counters for Data Locality Events *(Continued)*

Counter	Description
`EC_miss_remote`	`[PICU 10.1001]` Counts the events triggered whenever the processor generates a remote (R_*) transaction and the address is to a non-LPA portion (remote) of the physical address space, or an R_WS transaction due to block store (BST)/block store commit (BSTC) to any address space (LPA or non-LPA), or an R_RTO due to Store/Swap request on O_S state to LPA space.
`EC_wb_remote`	`[PICL 01.1001]` Counts the retry event when any victimization for which the processor generates an R_WB transaction to non-LPA address region. In practice, these are all NUMA cases, since for Coherence Memory Replication (CMR or COMA), the protocol insures that the processor only generates WB transactions.

SSM Systems

Typically, four to six local processors are in a system cluster and have their own local memory subsystem(s). They use an SMP to maintain data coherency amongst themselves. Data coherency is maintained between system clusters using a directory based SSM data coherency mechanism to insure data coherency across systems with a large number of processors.

The data locality event counters are only valid for Shared System Memory architectures in SSM mode.

14.7.1 Event Tree

The event and cycle stall counters are illustrated in FIGURE 14-6 on page 14-381. The diagram includes actual counters and synthesized counts.

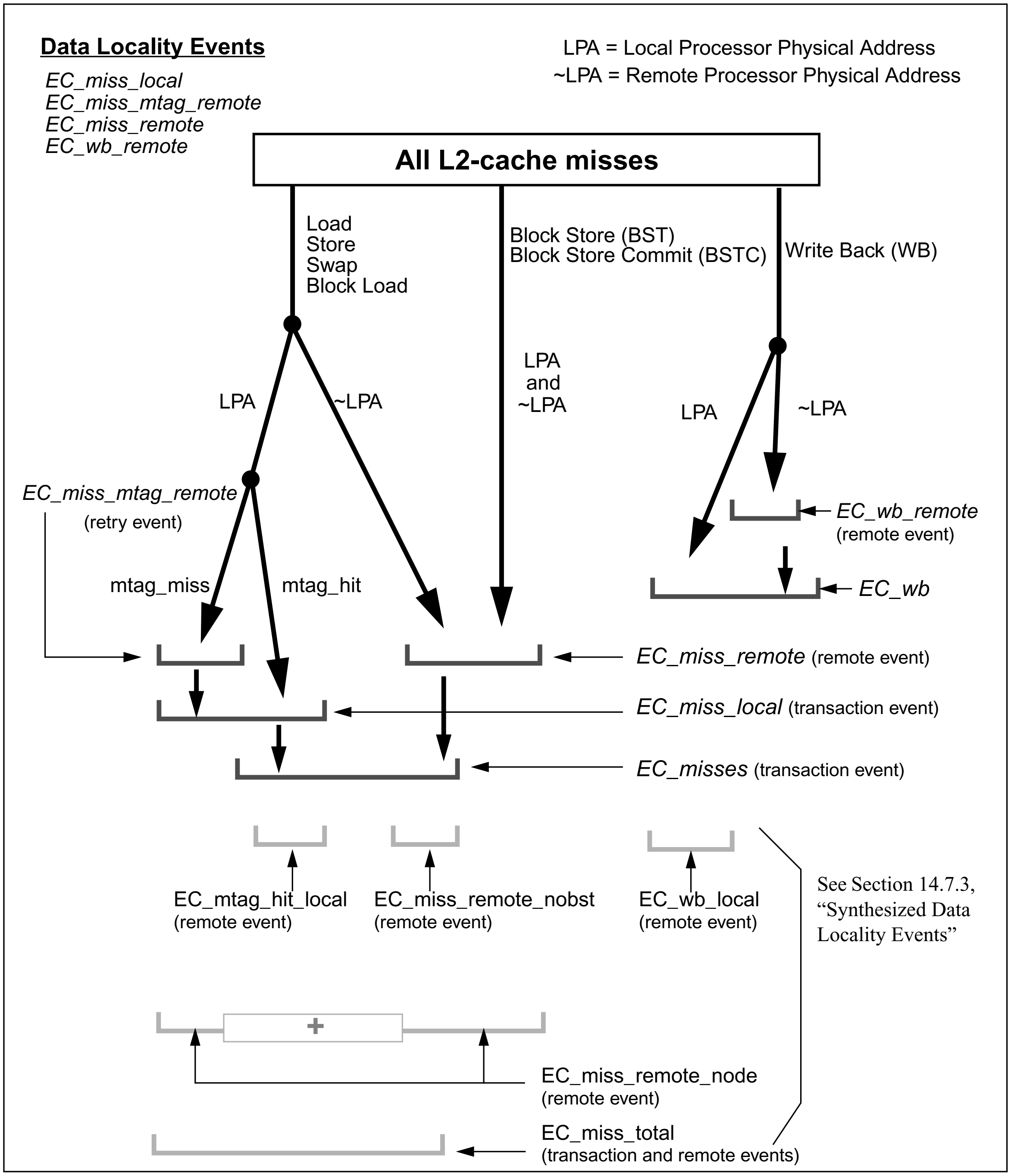


FIGURE 14-6 Data Locality Event Tree for L2-cache Misses

14.7.2 Data Locality Event Matrix

TABLE 14-19 shows the data locality event matrix.

TABLE 14-19 Data Locality Events

MODE	**Combined State**	**Processor action**					
		Load	**Store/ Swap**	**Block Load**	**Block Store**	**Block Store with Commit**	**Write Back**
LPA	I	miss: RTS	miss: RTO	miss: RS	miss: R_WS	miss: R_WS	none
	E	hit	hit: E→M	hit	hit: E→M		
	S		mtag miss: RTO		miss: R_WS		
	O						WB
	Os		mtag miss: R_RTO				WB
	M	hit					WB
LPA Retried	I	mtag miss: R_RTS	mtag miss: R_RTO	mtag miss: R_RS	invalid		none
	E	invalid					
	S	invalid	mtag miss: R_RTO	invalid			none
	O						
	Os						
	M	invalid					
~LPA	I	miss: R_RTS	miss: R_RTO	miss: R_RS	miss: R_WS	miss: R_WS	none
	E	hit	hit: E→M	hit	hit: E→M		
	S	hit	mtag miss: R_RTO	hit	miss: R_WS		
	O						miss: R_WB
	Os						
	M	hit					

LPA Retried Events

Retry is to issue a R_* transaction for an RTS/RTO/RS transaction that gets unexpected *MTag* from the SSM system interconnect (for example, cache state = O and mtag state = gS). A retry takes place in LPA.

14.7.3 Synthesized Data Locality Events

The Data Locality event assignments allow the software to create synthetic events based on arithmetic combinations of events assigned to PICL and PICU, as shown in TABLE 14-20.

TABLE 14-20 Synthesized Data Locality Events

Synthesized Event		PICL	PICU
EC_wb_local	=	+ EC_wb	– EC_wb_remote
EC_miss_remote_nobst (remote misses excluding BST/BSTC)	=	+ EC_misses	– EC_miss_local
EC_miss_total (L2-cache misses + retries)	=	+ EC_misses	+ EC_miss_mtag_remote
EC_miss_remote_node (CMR, NUMA)	=	+ EC_miss_remote	+ EC_miss_mtag_remote
EC_mtag_hit_local	=	– EC_miss_mtag_remote	+ EC_miss_local

EC_miss_total

EC_miss_total counts all EC misses, which is EC_misses plus all retries. Retry means you have two transactions for each miss (first it misses MTag and then reissued).

14.8 Miscellaneous Counters

14.8.1 System Interface Events and Clock Cycles

System interface statistics are collected through the counters listed in TABLE 14-21.

TABLE 14-21 Counters for System Interface Statistics

Counter	Description
`SI_snoop`	`[PICL 01.0001]` Counts snoops from remote processor(s) including RTS, RTSR, RTO, RTOR, RS, RSR, RTSM, and WS.
`SI_ciq_flow`	`[PICL 01.0010]` Counts system clock cycles when the flow control (`PauseOut`) signal is asserted.
`SI_owned`	`[PICL 010011]` Counts events where `owned_in` is asserted on bus requests from the local processor.

14.8.2 Software Events

Software statistics are collected through the counters listed in TABLE 14-22.

TABLE 14-22 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.

14.8.3 Floating-Point Operation Events

Floating-point operation statistics are collected through the counters listed in TABLE 14-23.

TABLE 14-23 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`	`[PICL 10.0111]` Counts instructions that complete execution on the Floating-point/Graphics Multiply pipelines.

14.9 PCR.SL and PCR.SU Encodings

TABLE 14-24 lists `PCR.SL` and `PCR.SL` selection bit field encoding. Shaded blocks show SL and SU field duplications with light shading.

TABLE 14-24 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_RTO	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 14-24 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	EC_wb_remote	PC_snoop_inv
01.1010	EC_miss_local	PC_hard_hit
01.1011	EC_miss_mtag_remote	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_reads_0	MC_writes_0
10.0001	MC_reads_1	MC_writes_1
10.0010	MC_reads_2	MC_writes_2
10.0011	MC_reads_3	MC_writes_3
10.0100	MC_stalls_0	MC_stalls_1
10.0101	MC_stalls_2	MC_stalls_3
10.0110	*Reserved*	Re_RAW_miss
10.0111	*Reserved*	FM_pipe_completion
10.1000	*Reserved*	EC_miss_mtag_remote
10.1001	*Reserved*	EC_miss_remote
10.1010 - 11.1111	*Reserved*	*Reserved*

CHAPTER **15**

Processor Optimization

15.1 Introduction

A significantly larger performance gain can be obtained if the code is re-compiled using a compiler specifically designed for a specific processor. Several features are provided on the UltraSPARC processors that can only be taken advantage of by using modern compiler technology. This technology was previously unavailable primarily due to insufficient hardware support to justify its development.

The front end of the processor consists of the following components:

- Prefetch Unit
- Instruction cache
- Next Field RAM
- The Branch and Set Prediction Logic
- Return address stack

The role of the front end in processor performance optimization is to supply as many valid instructions as possible to the grouping logic and eventually to the functional units, including the ALUs, the floating-point adder, the branch unit, the load/store pipe, to name a few.

Optimization is also about prefetching data, as cache misses are cycle intensive.

15.2 Instruction Stream Issues

The following section addresses the following issues:

- Instruction Alignment

- Instruction Cache Timing
- Executing Code Out of the Level 2 Cache
- TLB Misses
- Instruction Cache Utilization
- Handling of CTI Couples
- Mispredicted Branches
- Return Address Stack

15.2.1 Instruction Alignment

15.2.1.1 Instruction Cache Organization

The I-cache is organized as a 4-way set associative cache, with each set containing a multiple of eight-instruction lines (see FIGURE 15-1). Depending on its address, for each line of 8 instructions, up to 4 instructions are sent to the instruction buffer. If the address points to one of the last three instructions in the line, only this last instruction and instructions (0-2) from the end of the line are selected. Consequently, on average for random accesses, 3.25 instructions are fetched from the I-cache. For sequential accesses, the fetching rate (4 instructions per cycle) matches the consuming rate of the pipeline (up to 4 instructions per cycle).

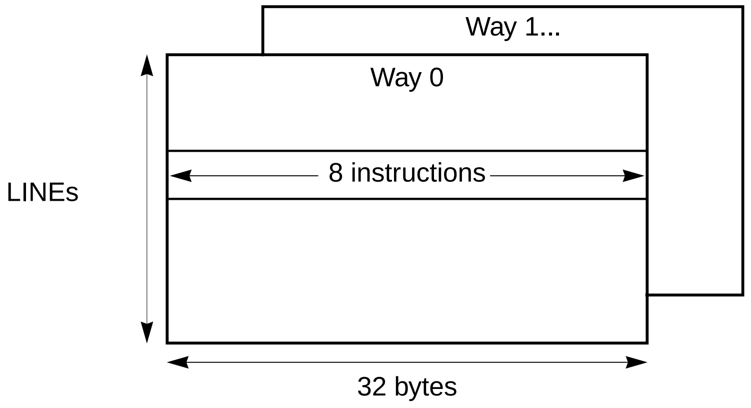


FIGURE 15-1 Instruction Cache Organization

15.2.1.2 Branch Target Alignment

Given the restriction mentioned above regarding the number of instructions fetched from an I-cache access, it is desirable to align branch targets so that enough instructions are fetched to match the number of instructions issued in the first group of the branch target. The following examples highlight the logic behind branch target alignment:

- If the compiler scheduler indicates that the target can be grouped with only one more instruction, the target should be placed anywhere in the line except in the last slot, since only one instruction would be fetched in that case. It may be beneficial to fetch more instructions, if possible.
- If the target is accessed from more than one place, it should be aligned so that it accommodates the largest possible group (first 5 instructions of a line).
- If accesses to the I-cache are expected to miss, it may be desirable to align targets on a 32-byte boundary, or at least the front end of a block, so that 4 instructions are forwarded to the next stage. Such an alignment helps assure that the maximum number of instructions can be processed between cache misses, assuming that the code does not branch out of the sequence of instructions. In fact 64-byte alignment can help instruction prefetch.

In general, it is best to align for maximum fetching by aligning branch targets on 4 instruction (16-byte) boundaries. This can help ensure that the fetch bandwidth matches the issue width, which is a maximum of 4 instructions in the case of the UltraSPARC III processor.

15.2.1.3 Branch Optimization

The UltraSPARC processors favor branch not taken conditionals. Regardless of this preference, the instruction issue remains the same and the fetch is optimized.

15.2.1.4 Impact of the Delay Slot on Instruction Fetch

Most Control Transfer Instructions (CTIs) are actually *delayed* CTIs taking effect 1 instruction after the CTI, with the resulting lag known as the *delay slot*. That is the instruction following the branch, or after CTI, is always executed regardless of where the CTI directs execution (unless annulling is used). If the last instruction of a line is a branch, the next sequential line in the I-cache must be fetched even if the branch predicted is taken, since the delay slot must be sent to the grouping logic. This leads to inefficient fetches, since an entire L2-cache access must be dedicated to fetching the missing delay slot. Therefore one should take care not to place delayed CTIs at the end of a cache line.

15.2.1.5 Instruction Alignment for the Grouping Logic

See Chapter 4 “Grouping Rules” for a description of grouping logic.

15.2.1.6 Impact of Instruction Alignment on Instruction Dispatch

It is important that no two branches are in the same fetch group. If there are two branches in the same group the second branch will end the group and will cause a re-fetch taking 2 cycles. To guarantee this does not happen in the case of uncertain instruction alignment, ensure that no 2 branches are within 4 instructions of each other.

15.2.2 Instruction Cache Timing

While accesses to the I-cache hit successfully, the pipeline rarely starves for instructions. In rare cases however, the Instruction Dispatch is unable to provide a sufficient number of instructions to keep the functional units busy. For example, a taken branch to a taken branch sequence without any instructions between the branches (except for the delay slot) could only be executed at a peak rate of two instructions per cycle.

An I-cache miss does not necessarily result in bubbles being inserted into the pipeline. Part of the I-cache miss processing, or even all of it, can be overlapped with the execution of instructions that are already in the instruction buffer and are waiting to be grouped and executed. Moreover, since the operation of the Instruction Dispatch is somewhat separated from the rest of the pipeline, the I-cache miss may have occurred when the pipeline was already stalled (for example, due to a multi-cycle integer divide, floating-point divide dependency, dependency on load data that missed the D-cache, etc.). This means that the miss (or part of it) may be transparent to the pipeline.

Because of the possibility of stalling the processor when the pipeline is waiting for new instructions, it is desirable to try to make code routines fit in the I-cache and avoid cache misses. The UltraSPARC processor provides instrumentation to profile a program and detect if instruction accesses generate a cache miss or a cache hit. By checkpointing the counters before and after a large section of code, combined with profiling the section of code, one can determine if the frequently executed functions generally hit or miss the I-cache.

15.2.3 Executing Code Out of the Level 2 Cache

Executing out of the L2-cache is much better than executing from main memory. From main memory, hardware can prefetch the next 8 instructions if the initial fetch was from the lower 32 of a 64-byte aligned boundary.

15.2.4 TLB Misses

The TLB contains the virtual page number and the associated physical page number of the most recently accessed pages.

A TLB miss is handled by software through the use of the TSB, and takes a large number of cycles. In order to minimize the frequency of TLB misses, the UltraSPARC processor provides a large number of entries in the TLB. Nonetheless, techniques are encouraged to further decrease the TLB miss cost.

15.2.4.1 Impact of the Annulled Slot

Grouping rules in Chapter 4 "Grouping Rules" describes how the UltraSPARC processor handles instructions following an annulling branch. In connection with these instructions, pay regard to the rules:

- Avoid scheduling `WR(PR, ASR)`, `SAVE`, `SAVED`, `RESTORE`, `RESTORED`, `RETURN`, `RETRY`, and `DONE` in the delay slot and in the first three groups following an annulling branch.

15.2.5 Conditional Moves vs. Conditional Branches

The `MOVcc` and `MOVR` instructions provide an alternative to conditional branches for executing short code segments. The UltraSPARC processor differentiates the two as follows:

- Conditional branches: Distancing the `SETcc` from `Bicc` does not gain any performance. The penalty for a mispredicted branch is always 8-cycles. `SETcc`, `Bicc`, and the delay slot can be grouped together (FIGURE 15-2).

```
setcc G   E   C   N1  N2  N3  W
bicc  G   E   C   N1  N2  N3  W
delay G   E   C   N1  N2  N3  W
```

FIGURE 15-2 Handling of Conditional Branches

- Conditional moves: A use of the destination register for the `MOVcc` follows the same rule as a load-use. FIGURE 15-3 shows a typical example.

```
setcc G   E   C   N1  N2  N3  W
movcc     G   E   C   N1  N2  N3  W
use               G   E   C   N1  N2  N3  W
```

FIGURE 15-3 Handling of MOVCC

If a branch is correctly predicted, the issue rate can be higher than that of a branch that is replaced by a conditional move. If a branch is not predictable the mis-predicted penalty is significantly higher than the extra latency of a conditional move.

15.2.6 Instruction Cache Utilization

Grouping blocks that are executed frequently can effectively increase the apparent size of the I-cache. Case studies show that, often, half of the I-cache entries are never executed. Placing rarely executed code out of a line containing a frequently executed block (identified by profiling) achieves a better I-cache utilization.

15.2.7 Handling of CTI couples

Placing a CTI into the delay slot of another CTI will disrupt the fetch and cost many cycles and should be avoided.

15.2.8 Mispredicted Branches

Correctly predicted conditional branches allow the processor to group instructions from subsequent basic blocks and continue to progress speculatively until the branch is resolved. The capability of executing instructions speculatively is a significant performance boost for the UltraSPARC processor.

15.2.9 Return Address Stack (RAS)

In order to speed up returns from subroutines invoked through `CALL` instructions, UltraSPARC processor dedicates a 8-deep stack to store the return address. Each time a `CALL` is detected, the return address is pushed onto this RAS (Return Address Stack). Each time a return is encountered, the address is obtained from the top of the stack and the stack is popped.The UltraSPARC III Cu processor considers a return to be a `JMPL` or `RETURN` with *rs1* equal to `%o7` (normal subroutine) or `%i7` (leaf subroutine). The RAS provides a guess for the target address, so that prefetching can continue even though the address calculation has not yet been performed. `JMPL` or `RETURN` instructions using *rs1* values other than `%o7` or `%i7` use the value on the top of the RAS for continuing prefetching, but they do not pop the stack.

To take full advantage of the RAS, one should follow the standard call and return conventions so that hardware can correctly predict the return addresses.

15.3 Data Stream Issues

The following section addresses the following issues:

- Data Cache Organization
- Data Cache Timing
- Data Alignment
- Using LDDF to Load Two Single-Precision Operands/Cycle
- Store Considerations
- Read-After-Write Hazards

15.3.1 Data Cache Organization

The D-cache is a mapped, virtually indexed, physically tagged (VIPT), write-through, non-write-allocating cache. It is logically organized as lines of 32 bytes.

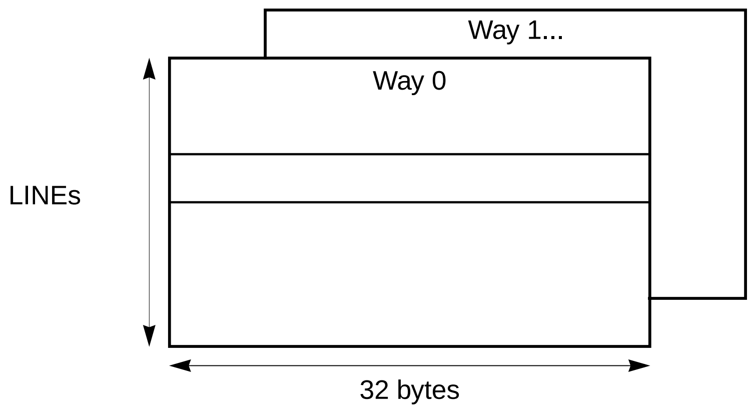


FIGURE 15-4 Logical Organization of Data Cache

15.3.2 Data Cache Timing

The latency of a load to the D-cache depends on the opcode. `LDX` and `LDUW` have 2-cycle load-to-use latency while all other loads have 3. For instance, if the first two instructions in the instruction buffer are a load and an instruction dependent on that load, the grouping logic will break the group after the load and a bubble will be inserted in the pipeline. It is very important to separate loads from their use.

15.3.3 Data Alignment

SPARC V9 requires that all accesses be aligned on an address equal to the size of the access. Otherwise a *mem_address_not_aligned* trap is generated. This is especially important for double precision floating-point loads, which should be aligned on an 8-byte boundary. If misalignment is determined to be possible at compile time, it is better to use two `LDF` (load floating-point, single precision) instructions and avoid the trap. The UltraSPARC processor supports single-precision loads mixed with double-precision operations, so that the case above can execute without penalty (except for the additional load). If a trap does occur, the UltraSPARC processor dedicates a trap vector for this specific misalignment, which reduces the overall penalty of the trap.

Grouping load data is desirable, since a D-cache sub-block can contain either four properly aligned single-precision operands or two properly aligned double-precision operands (eight and four respectively for a D-cache line). This is desirable not only for improving the D-cache hit rate (by increasing its utilization density), but also for D-cache misses where, for sequential accesses, one out of two requests to the L2-cache can be eliminated.

15.3.3.1 Using LDDF to Load Two Single-Precision Operands/Cycle

The UltraSPARC processor supports single cycle 8-byte data transfers into the floating-point register file for LDDF. Wherever possible, applications that use single-precision floating-point arithmetic heavily should organize their code and data to replace two LDFs with one LDDF. This reduces the load frequency by approximately one half, and cuts execution time considerably.

15.3.4 Store Considerations

The store on the UltraSPARC processor is designed so that stores can be issued even when the data is not ready. More specifically, a store can be issued in the same group as the instruction producing the result. The address of a store is buffered until the data is eventually available. Once in the store buffer, the store data is buffered until it can be completed.

The write cache can be used to exploit locality (both temporal and spatial) in the write stream. Try to organize data to exploit the locality by not having more than 4 data streams because of the 4-way set associative aliasing.

15.3.5 Read-After-Write Hazards

See Chapter 9 “Read-After-Write (RAW) Bypassing” for rules on RAW hazards.

CHAPTER **16**

Prefetch

This chapter contains the following sections:

- Prefetch Cache
- Software Prefetch
- Hardware Prefetch
- FP/VIS Load Instruction Miss Fetch

Prefetches can be used to hide memory latency, increase memory-level parallelism (overlap memory accesses), and hide L2 latency for floating-point loads.

The Prefetch mechanism contains an eight entry prefetch request queue to hold up to eight outstanding prefetch requests to L2-cache and beyond. Prefetches are generated by software prefetch instructions and by an autonomous hardware prefetch engine. Prefetches may bring data from memory into L2-cache. Prefetches may also bring data (either from L2-cache or memory) all the way to the Level-1 P-cache. Only floating-point loads may get data from the P-cache.

16.1 Prefetch Cache

The P-cache is virtually indexed and virtually tagged for CPU cache reads and never contains modified data. It is physically indexed and physically tagged for system snooping of the cache. The prefetch cache is 2 KB, 4-way set associative with 64 bytes per line.

The P-cache is accessed in parallel with the D-cache for floating-point/VIS loads. If the P-cache contains the data and the D-cache does not, the data will be provided by the P-cache. Integer loads cannot get data for the P-cache.

16.1.1 P-Cache Data Flow

Data is put into the P-cache under the following conditions:

- Hardware Prefetch (64 bytes)
- FP Load Miss (32 bytes)
- Some variants of Software Prefetch (64 bytes)

The hardware prefetch mechanism and the software prefetch instructions fetch data into the P-cache but not the D-cache. The FP/VIS Load Instruction Miss Fetch puts data in both the D-cache and P-cache.

Block load data is never loaded into the P-cache. The P-cache can hold cacheable data that is not in the D-cache, and vice versa. The P-cache does not contain modified data. Stores that hit the P-cache invalidate the 64-byte P-cache line.

16.1.2 Fetched_Mode Tag Bit

Each P-cache line contains a fetched_mode bit that indicates how the P-cache line entry was installed: software prefetch instruction or hardware prefetch mechanism.

Successful software prefetches set this bit to indicate the cache line was installed using a software prefetch instruction. This will inhibit the hardware prefetch mechanism from operating when it might otherwise operate using the cache line information. The hardware prefetch logic will not prefetch when the fetch mode bit of the current cache line is set to software prefetched data. This helps keep the prefetching from interfering with successful software prefetching.

16.2 Software Prefetch

The SPARC V9 instruction set contains data prefetch instructions. These instructions allow software to give warning to hardware that it will be using data in the future. This gives hardware the opportunity to potentially bring data closer to hide the latency of the memory subsystem when the data is actually needed. There are different types of prefetches that enable software to tell how the data will be used.

The UltraSPARC III Cu processor has extensive support for making use of prefetch directives from software. The prefetching can be used to either hide latency or memory operations or to overlap memory operations. In the UltraSPARC III Cu processor, different types of prefetches are handled differently. Prefetches may do one of the following:

- Bring data into L2-cache only

- Bring data into P-cache only
- Bring data into both P-cache and L2-cache
- Try to acquire ownership of a line for future writes

The software prefetch instructions are enabled by setting both the pcache_enable and the software_prefetch_enable (SPE) (bits 45 and 43, DCUCR register). If the P-cache is disabled or the SPE is zero, then all software prefetch instructions are ignored as they go through the pipeline.

The P-cache responds to software prefetches by fetching the requested data into the P-cache and/or L2-cache, as specified by the prefetch instruction.

16.2.1 Software Prefetch Instructions

Software prefetch instructions can prefetch data into the P-cache, L2-cache, or both, depending on the type of prefetch instruction executed. TABLE 16-1 lists the types of software prefetch instructions.

TABLE 16-1 Types of Software Prefetch Instructions

`fcn` Value (hex)	Instruction Type	Prefetch (64 bytes of data) into:	Instruction Strength UltraSPARC III Cu	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.

Instruction Strength. For each of the prefetch types, the software instruction can either be a weak or a strong software prefetch instruction. This allows software to tell hardware how confident it is that a prefetch will be helpful. Hardware may try harder to make sure that a strong prefetch gets executed. In the case of the UltraSPARC III Cu processors, all software prefetch instructions are weak, allowing the processor to more easily cancel their execution.

16.2.2 Software Prefetch Instruction Uses the MS Pipeline

All software prefetch instructions are executed by the MS pipeline because the MMU/TLB structure is required and it is dedicated to the load and store instructions that are executed in the MS pipeline.

The software prefetch instruction can be grouped with a second FP/VIS load instruction that executes in the A0 or A1 pipeline, but the software prefetch instruction will always execute in the MS pipeline. This means the processor never executes two software prefetch instructions within one instruction group.

16.2.3 Cancelling Software Prefetch

The processor will cancel a prefetch instruction under various conditions that depend on the instruction strength. The following four conditions can cancel a software prefetch instruction.

Regardless of software prefetch instruction strength, the prefetch is cancelled if:

- The data already exists in the appropriate cache, *or*
- The prefetch address is the same as one of the outstanding prefetches.

Weak software prefetch instruction strengths are also cancelled when:

- The prefetch instruction misses the D-TLB, *or*
- The prefetch queue is full (there are already too many outstanding prefetches).

16.2.4 Prefetch Instruction Variants

Prefetch for Several Reads

The data installs in the P-Cache and L2-cache. In cases that a memory request is required to get the line, a Read to Share (RTS) is issued to the system. This is typically used when the data will be read more than once.

Prefetch for One Read

The data installs in the P-Cache only. It does not install in the L2-cache if the line is not already there. In cases that a memory request is required to get the line, an RTS is issued to the system. This is typically used for streaming data that is only used for a short period.

Note – Prefetch for one read can be used to stream special non-coherent data. A Prefetch Invalidate should be used to flush such data to maintain correct behavior.

Prefetch for Several Writes

The data installs a 64-byte line in L2-cache, if not present. In the case that a memory request is required, a Read to Own (RTO) is issued. This variant is typically used to prefetch for one or more writes.

If the prefetch line is already in the L2-cache, then the prefetch instruction will be cancelled even if the line is in a shared state. No RTO request will be sent out and no state change of the 64-byte sub-block in the L2-cache is made.

Prefetch for One Write

The data installs a 64-byte line in L2-cache, if not present. In the case that a memory request is required, an RTS is issued. This is typically used for prefetching integer data from memory or for a two stage prefetch that separates prefetch to L2-cache and prefetch from L2-cache to P-cache.

16.2.5 General Comments

- *Block Load (BLD) data bypasses the P-cache and D-cache.*

  The block load data is always supplied by the L2-cache and written into FP/VIS register file. The P-cache is used as a staging area utilizing eight 64-bit registers to hold the data that is forwarded to the FP/VIS register file. Using BLD can corrupt the P-cache. It is recommended that you use the Solaris^{TM} bcopy routine for transferring large amounts of data.

- *P-cache data is not included in L2-cache.*

  The prefetch-read-once software prefetch instruction puts data into the P-cache from the L2-cache or memory, but does not install it in L2-cache.

- *Stores invalidate prefetch cache lines.*

  Stores invalidate P-cache lines that reference the same store address to maintain cache consistency. There is no mechanism to modify data in the P-cache, so code loops must be careful to place any required store instructions after the P-cache data line is no longer needed.

16.2.6 Code Example

CODE EXAMPLE 16-1 is an example of two memory operations per cycle, double-precision, vector-multiply code.

CODE EXAMPLE 16-1 Two Memory Operations Per Cycle Code

```
    ! %l0 -> j
    ! %l1 -> c
    ! %l2 -> a
    ! %l3 -> b
    ! %l4 -> N
    ! %f32 -> K

    ! prefetch fcn=3 - one write, used to prefetch “c” into the E$
    ! prefetch fcn=1 - one read, used to prefetch “a” and “b” into
the P$

loop:
    prefetch [%l1+64],3                ! prefetch in first memory
slot
    lddf     [%l2],%f0                 ! 2nd load from P$
    fmuld    %f32,%f24,%f24
    faddd    %f16,%f18,%f16

    stdf     %f8,[%l1+16]              ! stf in first memory slot
    add      %l1,64,%l1
    lddf     [%l3],%f2                 ! 2nd load from P$

    prefetch [%l2+64],1
    lddf     [%l2+8],%f4
    fmuld    %f32,%f28,%f28
    faddd    %f20,%f22,%f20

    stdf     %f12,[%l1-40]
    lddf     [%l3+8],%f6

    prefetch [%l3+64],1
    lddf     [%l2+16],%f8
    fmuld    %f32,%f0,%f0
```

CODE EXAMPLE 16-1 Two Memory Operations Per Cycle Code *(Continued)*

```
     faddd    %f24,%f28,%f24

     stdf     %f16, [%l1-32]
     lddf     [%l3+16],%f10

     lddf     [%l2+24],%f12
     fmul     %f32,%f4,%f4
     fadd     %f28,%f30,%f28
     stdf     %f20,[%l1-24]

     lddf     [%l3+24],%f14
     lddf     [%l2+32],%f16
     fmuld    %f32,%f8,%f8
     faddd    %f0,%f2,%f0

     stdf     %f24,[%l1-16]
     lddf     [%l3+32],%f18

     lddf     [%l2+40],%f20
     fmuld    %f32,%f12,%f12
     faddd    %f4,%f6,%f4
     stdf     %f28,[%l1-8]

     lddf     [%l3+40],%f22
     subcc    %l0,8,%l0

     lddf     [%l2+48],%f24
     fmuld    %f32,%f16,%f16
     faddd    %f8,%f10,%f8
     stdf     %f0,[%l1]

     lddf     [%l3+48],%f26
     lddf     [%l2+56],%f28

     lddf     [%l3+56],%f30
     fmuld    %f32,%f20,%f20
     faddd    %f14,%f12,%f12
     stdf     %f4,[%l1+8]

     add      %l3,64,%l3
     bgt      loop
     add      %l2,64,%l2
```

16.3 Hardware Prefetch

Note – Some versions of the UltraSPARC III Cu processor may have a hardware prefetch mechanism as described in this section.

The hardware prefetch mechanism monitors the dispatching of floating-point load instructions. If there is a P-cache hit and the state of the line indicates that it was not brought in by a software prefetch and that a hardware prefetch for the next line has not already been generated, then the hardware attempts to prefetch the next 64-byte cache line from the L2-cache. If the request misses in the L2-cache, then the prefetch request is cancelled. The hardware prefetch mechanism triggers regardless of which word(s) position within the 64 bytes is loaded; they do not need to be sequentially accessed to trigger the prefetch mechanism.

The hardware prefetch mechanism is enabled when the pcache_enable and hardware_prefetch_enable (bits 45 and 44, DCUCR register) are both set.

The hardware prefetch from the L2-cache to the P-cache can get needed data to the P-cache well in advance of its use and hence reduce the L2-cache latency.

16.3.1 Cancelling Hardware Prefetches

Hardware prefetches are cancelled when they encounter one of the following conditions:

- The data already exists in the P-cache.
- The prefetch queue is full, *or*
- The hardware prefetch address is the same as one of the outstanding prefetches.
- The prefetch address is not in the same 8 KB boundary as the line that initiated the prefetch.
- The request misses the L2-cache.

Hardware prefetch will return 64 bytes of data to P-cache only if it hits the L2-cache. No more than one speculative prefetch is allowed to be generated from each prefetch cache line. Once a cache hit has caused a speculative prefetch, subsequent hits to that line will not cause redundant speculative prefetches.

16.3.2 FP/VIS Load Instruction Miss Fetch

When an FP/VIS load instruction misses in the D-cache and P-cache, the hardware fetches 32 bytes of data from the L2-cache and fills it into the D-Cache, regardless of the load instruction data size. If hardware prefetch is enabled, then the 32 bytes are also installed into the P-cache. This is how hardware prefetch can get initiated. If the request misses the L2-cache, then 64 bytes are fetched from main memory and installed in the L2-cache first.

CHAPTER 17

IEEE 754-1985 Standard

The implementation of the floating-point unit for standard and nonstandard operating modes are described in this chapter.

This chapter defines debug and diagnostics support in these sections:

- Introduction
- Floating-point Numbers
- IEEE Operations
- Traps and Exceptions
- IEEE Traps
- Underflow Operation
- IEEE NaN Operations
- Subnormal Operations

17.1 Introduction

17.1.1 Floating-point Operations

Floating-point Operations (FPops) include the algebraic operations and usually do not include the specially treated FP Load/store, FB*fcc*, or the VIS instructions. The `FABS`, `FNEG`, and `FMOV` instructions are also treated separately from the algebraic operations.

17.1.2 Rounding Mode

The rounding mode of the FPU is determined either by the `FSR.RD` bit while in standard rounding mode or by the `GSR.IRND` bit when in interval arithmetic rounding mode. The rounding direction effects the result after any under or overflow condition is detected. Underflow is detected before rounding. The `FSR.RD` bit options are shown in TABLE 17-1.

TABLE 17-1 FSR.RD bit options

FSR.RD	*Round Toward*
0	Nearest (even, if tie)
1	0
2	$+\infty$
3	$-\infty$

17.1.3 Nonstandard FP Operating Mode

The processor supports a nonstandard FP mode to facilitate in the handling of Subnormals by the hardware, avoiding a software trap to supervisor software. The FP operating mode is controlled by the `FSR.NS` bit. When `FSR.NS` = 1, nonstandard mode is selected. However, when `GSR.IM` = 1, interval arithmetic rounding mode is selected, then regardless of the `FSR.NS` bit the processor will be in standard mode.

17.1.4 Memory and Register Data Images

The floating-point values are represented in the `f` registers in the same way that they are represented in memory. Any conversions for ALU operations are completed within the FP execution unit. Load and store operations do not modify the register value.

VIS instructions (logical and move/copy operations) can be used with values generated by the FPU.

17.1.5 Subnormal Operations

Subnormal operations include operations with Subnormal number operands and situations where an operation without Subnormal number operands generate a Subnormal number result. The FPU response to Subnormal numbers is described in section 17.8, *Subnormal Operations,* on page 428.

17.1.6 FSR.CEXC and FSR.AEXC Updates

The current exception (cexc) and accrued exception (aexc) fields in the FSR register are described in section 17.5, *IEEE Traps,* on page 422.

In general:

- Only floating-point operations (FPops) will update cexc and only when an exceptional condition is detected. All other instructions will leave cexc unchanged.
- When an exception is detected, but the trap is masked, then the FPop will update the appropriate aexc field of the FSR register.

17.1.7 Prediction Logic

Prediction of overflow, underflow, and inexact traps is used in the hardware. Prediction always errors on the side of providing correct results when the hardware can and generating an exception when it cannot or is not sure.

Prediction of inexact occurs unless one of the operands is a Zero, NaN, or Infinity. When prediction occurs and the exception is enabled, system software will properly handle these cases and resume program execution. If the exception is not enabled, the result status is used to update the FSR.aexc and FSR.cexc bits of the FSR register.

17.2 Floating-point Numbers

The floating-point number types and their abbreviations are shown in TABLE 17-2. In general the IEEE 754-1985 Standard reserves exponent field values of all 0s and all 1s to represent special values in the standard's floating-point scheme.

TABLE 17-2 Floating-point Numbers

Number Type	**Abbreviation**	**Data Representation**		
		Sign	**Exponent**	**Fraction**
Zero	0	0 or 1	000...000	000...000
Subnormal	SbN	0 or 1	000...000	000...001 to 111...111
Normal	Normal	0 or 1	000...001 to 111...110	000...000 to 111...111
Infinity	Infinity	0 or 1	111...111	000...000
Signalling NaN	SNaN	0 or 1	111...111	0xx...xxx
Quiet NaN	QNaN	0 or 1	111...111	1xx...xxx

Zero

Zero is not directly representable if the straight format is followed, this is due to the assumption of a leading 1. To allow the number zero to yield a value of zero, the fraction (or mantissa) must be exactly zero. Therefore the number zero is special cased with exponent and fraction fields of zero. It is also important to note that -0 and +0 are considered to be distinct values, though they both compare as equal.

SubNormal

If the exponent field is all 0s and the fraction field is non-zero then the value is a subnormal (denormalized) number. These numbers do not have an assumed leading 1 before the binary point. For single precision, these numbers are represented as $(-1)^s$ x 0.*f* x 2^{-126}, in double precision the representation is $(-1)^s$ x 0.*f* x 2^{-1022}. In both cases s is the sign bit and *f* is the fraction. Note that exponent and fraction fields of all 0s is the special representation of the number zero. From this point of view, the number zero can be considered a subnormal.

Infinity

The values -infinity and +infinity are represented with an exponent field of all 1s and a fraction field of all 0s. The sign bit distinguishes between positive and negative infinities. The infinity representation is important as it allows operations to continue past overflow. Operations dealing with infinities are well defined by the IEEE 754-1985 Standard.

Not a Number

The value NaN (Not a Number) is used to represent values that do not represent real numbers. The NaN exponent field is all 1s and the fraction field is non-zero. There are two categories of NaN; the QNaN (quiet NaN) and the SNaN (signalling NaN). A QNaN is a NaN with the most significant fraction field bit set. QNaN is allowed to freely propagate through most arithmetic operations; this NaN tends to appear when an operation produced mathematically undefined results. A SNaN fraction field significant bit is clear. The SNaN is used to signal an exception when it appears out of an operation being executed. Semantically QNaN can be considered to denote indeterminate operations, while SNaN indicates invalid operations.

17.2.1 Floating-point Number Line

The floating-point number line in FIGURE 17-1 represents the floating-point numbers used in the processor.

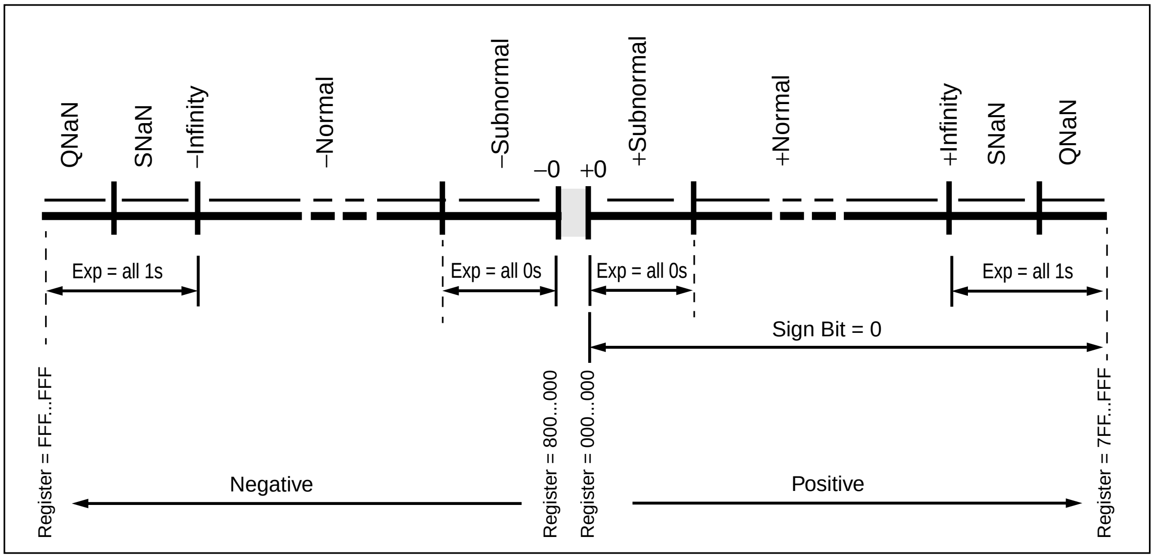


FIGURE 17-1 Floating-point Number Line

17.3 IEEE Operations

The response of each operation to operands with 0, Normal, Infinite, and NaN numbers are described in this section. The response to Subnormal numbers are described in section 17.8, *Subnormal Operations,* on page 428.

The result of each operation is concluded by one of the following:

- A number is written to the destination `f` register (rd).
- A number is written to the destination register **and** an IEEE flag is set.
- An IEEE flag is set **and** an IEEE trap is generated (rd is unchanged).

Each instruction is defined with one or more operands. Most instructions generate a result. The `FCMP{E}` instruction does not generate a result, instead it set the fccN bits.

17.3.1 Addition

The floating-point addition instruction is shown in TABLE 17-3.

TABLE 17-3 Floating-point Addition

ADDITION Instruction **FADD** rs_1, rs_2 [rs_2, rs_1] $\rightarrow$ *rd*	**RESULT from the operation includes one or more of the following:** • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
	Masked Exception, TEM=0		**Enabled Exception, TEM=1**	
	Destination register written (rd)	**Flag(s)**	**Destination register written (rd)**	**Flag(s), Trap**
+0, +0	+0	no	+0	no
+0, -0	+0 (FSR.RD=0,1,2) -0 (FSR.RD=3)	no	+0 (FSR.RD=0,1,2) -0 (FSR.RD=3)	no
-0, -0	-0	no	-0	no
0, +Normal	+Normal	no	+Normal	no
0, -Normal	-Normal	no	-Normal	no
0, +Infinity	+Infinity	no	+Infinity	no
0, -Infinity	-Infinity	no	-Infinity	no
Normal, +Infinity	+Infinity	set ofc, set ofa, set nvc, set nva	no	set ofc, set nvc, ieee trap
Normal, -Infinity	-Infinity	set ofc, set ofa, set nvc, set nva	no	set ofc, set nvc, ieee trap
+Normal, +Normal	May overflow, see 17.5.3		May overflow, see 17.5.3	
+Normal, -Normal	Normal		Normal	
-Normal, +Normal	Normal		Normal	
-Normal, -Normal	May underflow, see 17.5.4		May underflow, see 17.5.4	
+Infinity, +Infinity	+Infinity	no	+Infinity	no
+Infinity, -Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap
-Infinity, +Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap
-Infinity, -Infinity	-Infinity	no	-Infinity	no

17.3.2 Subtraction

The floating-point subtraction instruction is shown in TABLE 17-4.

TABLE 17-4 Floating-point Subtraction

SUBTRACTION Instruction rs_1 - rs_2 **FSUB** rs_1, $rs_2 \rightarrow rd$	**RESULT from the operation includes one or more of the following:** • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
	Masked Exception, TEM=0		**Enabled Exception, TEM=1**	
	Destination register written (rd)	**Flag(s)**	**Destination register written (rd)**	**Flag(s), Trap**
+0, +0	+0	no	+0	no
+0, -0	-0	no	-0	no
-0, +0	-0	no	-0	no
-0, -0	+0	no	+0	no
0, +Normal	-Normal	no	-Normal	no
0, -Normal	+Normal	no	+Normal	no
0, +Infinity	-Infinity	no	-Infinity	no
0, -Infinity	+Infinity	no	+Infinity	no
Normal, +Infinity	-Infinity	set ufc, set nvc, set ufa, set nva	no	set ufc, set nvc, ieee trap
Normal, -Infinity	+Infinity	set ufc, set nvc, set ufa, set nva	no	set ofc, set nvc, ieee trap
+Normal, -Normal	May overflow, see 17.5.3		May overflow, see 17.5.3	
Normal, +Normal	Normal	no	Normal	no
-Normal, -Normal	May underflow, see 17.5.4		May underflow, see 17.5.4	
+Infinity, [*0, Normal*]	+Infinity	no	+Infinity	no
-Infinity, [*0, Normal*]	-Infinity	no	-Infinity	no
+Infinity, +Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap
+Infinity, -Infinity	+Infinity	no	+Infinity	no
-Infinity, +Infinity	-Infinity	no	-Infinity	no
-Infinity, -Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap

17.3.3 Multiplication

The floating-point multiplication instruction is shown in TABLE 17-5.

TABLE 17-5 Floating-point Multiplication

MULTIPLICATION Instruction	**RESULT from the operation includes one or more of the following:** • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
FMUL *rs_1, rs_2 [rs_2, rs_1] $\rightarrow$ rd*	**Masked Exception, TEM=0**		**Enabled Exception, TEM=1**	
	Destination register written (rd)	**Flag(s)**	**Destination register written (rd)**	**Flag(s), Trap**
+0, [+0|+Normal]	+0	no	+0	no
+0, [-0|-Normal]	-0	no	-0	no
-0, [+0|+Normal]	-0	no	-0	no
-0, [-0|-Normal]	+0	no	+0	no
+0, +Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap
+0, -Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap
-0, +Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap
-0, -Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap
Normal, Normal	May underflow/ overflow, see 17.5		May underflow/ overflow, see 17.5	
[+Normal|+Infinity], +Infinity	+Infinity	no	+Infinity	no
[+Normal|+Infinity], -Infinity	-Infinity	no	-Infinity	no
[-Normal|-Infinity], +Infinity	-Infinity	no	-Infinity	no
[-Normal|-Infinity], -Infinity	+Infinity	no	+Infinity	no

17.3.4 Division

The floating-point division instruction is shown in TABLE 17-6.

TABLE 17-6 Floating-point Division

DIVISION Instruction *rs_1 rs_2* **FDIV** *rs_1, rs_2 $\rightarrow$ rd*	**RESULT from the operation includes one or more of the following:** • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
	Masked Exception, TEM=0		**Enabled Exception, TEM=1**	
	Destination register written (rd)	**Flag(s)**	**Destination register written (rd)**	**Flag(s), Trap**
0, 0	sign=0, expo=111...111, frac=111...111 (QNaN)	set nvc, set nva	no	set nvc, ieee trap
0, Normal	0	no	0	no
0, Infinity	0	no	0	no
+Normal, +0	+Infinity	set nvc, set nva	no	set dzc, set nvc, ieee trap
+Normal, -0	-Infinity	set nvc, set nva	no	set dzc, set nvc, ieee trap
-Normal, +0	-Infinity	set nvc, set nva	no	set dzc, set nvc, ieee trap
-Normal, -0	+Infinity	set nvc, set nva	no	set dzc, set nvc, ieee trap
Normal, Normal	May underflow/overflow, see 17.5		May underflow/overflow, see 17.5	
Infinity, Infinity	QNaN	set nvc, set nva	no	set nvc, ieee trap
+Infinity, +Normal	+Infinity	no	+Infinity	no
+Infinity, -Normal	-Infinity	no	-Infinity	no
-Infinity, +Normal	-Infinity	no	-Infinity	no
-Infinity, -Normal	+Infinity	no	+Infinity	no

17.3.5 Square Root

The floating-point square root instruction is shown in TABLE 17-7.

TABLE 17-7 Floating-point Square Root

SQUARE ROOT Instruction *sq root of* rs_2 **FSQRT** $rs_2 \rightarrow rd$	**RESULT from the operation includes one or more of the following:** • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
	Masked Exception, TEM=0		**Enabled Exception, TEM=1**	
	Destination register written (rd)	**Flag(s)**	**Destination register written (rd)**	**Flag(s), Trap**
+0	+0	no	+0	no
-0	-0	set nvc, set nva	no	set nvc, ieee trap
+Normal	May underflow/overflow, see 17.5		May underflow/overflow, see 17.5	
[*-Normal*|*-Infinity*]	QNaN (sign=0, expo=111...111, frac=111...111)	set nvc, set nva	no	set nvc, ieee trap
+Infinity	+ Infinity	no	+ Infinity	no

17.3.6 Compare

Two f registers are compared. The result of the compare is reflected in the fccN bits of the FSR registers, shown in TABLE 17-8.

The FCMPE version of the instruction relates to Subnormal operations, see TABLE 17-16, *Results from NaN Operands,* on page 427.

TABLE 17-8 Number Compare

FP NUMBER COMPARE Instruction **FCMP{E}** rs_1, rs_2	**RESULT from the operation includes one or more of the following:** • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • The fcc bit set.			
	Masked Exception, TEM=0		**Enabled Exception, TEM=1**	
	FP Condition Code Setting (*fccN*)	**Flag(s)**	**FP Condition Code Setting (*fccN*)**	**Flag(s), Trap**
+0, +0	fcc=0 ($rs_1 = rs_2$)	no	fcc=0 ($rs_1 = rs_2$)	no
-0, -0	fcc=0 ($rs_1 = rs_2$)	no	fcc=0 ($rs_1 = rs_2$)	no
+0, [+Normal|+Infinity]	fcc=1 ($rs_1 < rs_2$)	no	fcc=1 ($rs_1 < rs_2$)	no
-0, [-Normal|-Infinity]	fcc=0 ($rs_1 = rs_2$)	no	fcc=0 ($rs_1 = rs_2$)	no
-0, [+0|+Normal|+Infinity]	fcc=1 ($rs_1 < rs_2$)	no	fcc=1 ($rs_1 < rs_2$)	no

TABLE 17-8 Number Compare *(Continued)*

FP NUMBER COMPARE Instruction	**RESULT from the operation includes one or more of the following:** • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • The fcc bit set.			
	Masked Exception, TEM=0		**Enabled Exception, TEM=1**	
FCMP{E} *rs_1, rs_2*	**FP Condition Code Setting (*fccN*)**	**Flag(s)**	**FP Condition Code Setting (*fccN*)**	**Flag(s), Trap**
+0, [-0|-Normal|-Infinity]	fcc=2 ($rs_1 > rs_2$)	no	fcc=2 ($rs_1 > rs_2$)	no
Normal, Normal	=, >, or <	no	=, >, or <	no

17.3.7 Precision Conversion

Details of the precision conversion operations are shown in TABLE 17-9.

TABLE 17-9 Precision Conversion

PRECISION CONVERSION Operations single operand	**RESULT from the operation includes one or more of the following:** • Number in `f` register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
	Masked Exception, TEM=0		**Enabled Exception, TEM=1**	
FsTOd *$rs_2 \rightarrow rd$* **FdTOs** *$rs_2 \rightarrow rd$*	**Destination register written (rd)**	**Flag(s)**	**Destination register written (rd)**	**Flag(s), Trap**
FsTOd ±*0* **FdTOs** ±*0*	±*0*	no	±*0*	no
FsTOd ±*Normal*	±Normal	no	±Normal	no
FdTOs ±*Normal*	May underflow / overflow, see 17.4.		May underflow / overflow, see 17.4.	
FsTOd ±*Infinity* **FdTOs** ±*Infinity*	±Infinity	no	±Infinity	no

Examples:

- FsTOd (`7FD1.0000`) = `7FFA.2000.0000.0000`
- FsTOd (`FDD1.0000`) = `FFFA.2000.0000.0000`
- FdTOs (`7FFA.2000.0000.0000`) = `7FD1.0000`
- FdTOs (`FFFA.2000.0000.0000`) = `FFD1.0000`

17.3.8 Floating-point to Integer Number Conversion

The floating-point to integer number conversion instruction is shown in TABLE 17-10.

TABLE 17-10 Floating-point to Integer Number Conversion

FP to Int NUMBER CONVERSION Instruction single operand **FsTOi** *$rs_2 \rightarrow rd$* **FsTOx** *$rs_2 \rightarrow rd$* **FdTOi** *$rs_2 \rightarrow rd$* **FdTOx** *$rs_2 \rightarrow rd$*		**RESULT from the operation includes one or more of the following:** • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
		Masked Exception, TEM.NVM=0		**Enabled Exception, TEM.NVM=1**	
		Destination register written (rd)	**Flag(s)**	**Destination register written (rd)**	**Flag(s), Trap**
SP/DP Int	*+0*	000...000	no	000...000	no
	-0	111...111	no	111...111	no
	+Infinity	011...111	no	no	set nvc, ieee trap
	-Infinity	100...000	no	no	set nvc, ieee trap
SP Int	*+Normal* $< 2^{31}$	Integer representation of the Normal number	no	Integer representation of the Normal number	no
	+Normal $\geq 2^{31}$	011...111	set nvc, set nva	no	set nvc, ieee trap
	-Normal $> -[2^{31} + 1]$	Integer representation of the Normal number	no	Integer representation of the Normal number	no
	-Normal $\leq -[2^{31} + 1]$	100...000	set nvc, set nva	no	set nvc, ieee trap
DP Int	*+Normal* $< 2^{63}$	Integer representation of the Normal number	no	Integer representation of the Normal number	no
	+Normal $\geq 2^{63}$	011...111	set nvc, set nva	no	set nvc, ieee trap
	-Normal $> -[2^{63} + 1]$	Integer representation of the Normal number	no	Integer representation of the Normal number	no
	-Normal $\leq -[2^{63} + 1]$	100...000	no	100...000	no

17.3.9 Integer to Floating-point Number Conversion

The integer to floating-point number conversion instruction is shown in TABLE 17-11.

TABLE 17-11 Integer to Floating-point Number Conversion

Int to FP NUMBER CONVERSION Instruction single operand **FiTOs** $rs_2 \rightarrow rd$ **FiTOd** $rs_2 \rightarrow rd$ **FxTOs** $rs_2 \rightarrow rd$ **FxTOd** $rs_2 \rightarrow rd$		**RESULT from the operation includes one or more of the following:** • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
		Masked Exception, TEM.NXM=0		**Enabled Exception, TEM.NXM=1**	
		Destination register written (rd)	**Flag(s)**	**Destination register written (rd)**	**Flag(s), Trap**
SP/DP FP	*0*	±0	no	±0	no
SP FP	*+Integer* $< 2^{23}$	FP representation of +Normal	no	FP representation of +Normal	no
	+Integer $\geq 2^{23}$	Integer is rounded to 23 msb and converted.	set nvc, set nxc	no	set nvc, ieee trap
	-Integer $> -[2^{23} + 1]$	FP representation of +Normal	no	FP representation of +Normal	no
	-Integer $\leq -[2^{23} + 1]$	Integer is rounded to 23 msb and converted.	set nvc, set nxc	no	set nvc, ieee trap
DP FP	*+Integer* $< 2^{52}$	FP representation of +Normal	no	FP representation of +Normal	no
	+Integer $\geq 2^{52}$	Integer is rounded to 52 msb and converted.	set nvc, set nxc	no	set nvc, ieee trap
	-Integer $> -[2^{52} + 1]$	FP representation of +Normal	no	FP representation of +Normal	no
	-Integer $\leq -[2^{52} + 1]$	Integer is rounded to 52 msb and converted.	set nvc, set nxc	no	set nvc, ieee trap

17.3.10 Copy/Move Operations

Floating-point numbers are not modified by the copy and move instructions: `FMOV`, `FABS`, and `FNEG`. The copy/move instructions will not generate an *unfinished_FPop* or *unimplemented_FPop* exception, but they will generate the *fp_disabled* exception if the FPU is disabled.

The processor performs the appropriate sign bit transformation but will not cause an invalid exception and will not perform a QNaN to SNaN transformation.

These are single operand instructions that use the rs_2 register as the source operand.

FMOV

- f register-to- f register move.
  - No change to any bit, regardless of register content.
  - Useful with VIS instructions.

FABS

- Changes the FP/Int sign bit to positive, if needed.
  - No change to any other bit, regardless of register content.

FNEG

  - Changes the FP/Int sign bit (If 0, then 1. If 1, then 0.)
  - No change to any other bit, regardless of register content.

17.3.11 f Register Load/Store Operations

A load single floating-point (LDF) instruction writes to a 32-bit register. This must be converted to a 64-bit value (FsTOd) for use with double precision instructions.

A load double floating-point (LDDF) instruction writes to a pair of adjacent, 32-bit f registers aligned to an even boundary, and it can write to a 64-bit register. This must be converted to a 32-bit value (FdTOs) for use with single precision instructions.

Two LDF instructions can be used to load a 64-bit value when the memory address alignment to 64-bits is not guaranteed. Similarly, two STF instructions can be used to store a 64-bit value when the memory address alignment to 64-bits is not guaranteed.

17.3.12 VIS Operations

VIS instructions are unaffected by floating-point models. However, the FPU must be enabled. VIS instructions do not generate interrupts unless the FPU is disabled.

17.4 Traps and Exceptions

There are 3 trap vectors defined for floating-point operations:

- *fp_disabled*
- *fp_exception_ieee_754* (see section 17.5, *IEEE Traps,* on page 422)

- *fp_exception_other*

fp_disabled Trap

The floating-point unit can be enabled and disabled.

fp_exception_other Trap

The *fp_exception_other* trap occurs when a floating-point operation cannot be completed by the processor (*unfinished_FPop*) or an operation is requested that is not implemented by the processor (*unimplemented_FPop*).

17.4.1 Summary of Exceptions

The floating-point unit exceptions are shown in TABLE 17-12.

TABLE 17-12 Floating-point Unit Exceptions

Description	IEEE Flag	Trap Abbreviation	Fault Trap Type	Exception/Trap Vector
FPU disabled	none	disable trap	none	*fp_disabled* (020_{16})
FP operation **invalid** (IEEE)	**nv**	**ieee trap**	*IEEE_745_exception* (FSR.FTT = 1)	*fp_exception_ieee_754* (021_{16})
FP operation **overflow** (IEEE)	**of**			
FP operation **underflow** (IEEE)	**uf**			
FP operation **division by zero** (IEEE)	**dz**			
FP operation **inexact** (IEEE)	**nx**			

17.4.2 Trap Event

When a floating-point exception causes a trap, the trap is precise. The response to traps is described in TABLE 17-13.

TABLE 17-13 Response to Traps

Exception Event → / Resulting Action ↓	fp_disabled	fp_exception_other		fp_exception_ieee_754
		unimplemented_FPop	unfinished_FPop	
Address of instruction that caused the trap is put in the PC and pushed onto the trap stack.	✓	✓	✓	✓
The destination f register (rd) is unchanged from its state prior to the execution of the instruction that caused the trap.	✓	✓	✓	✓
The floating-point condition codes (fcc*N*) are unchanged.	✓	✓	✓	✓
The FSR.aexc field is unchanged.	✓	✓	✓	✓
The FSR.cexc field is unchanged.	✓	✓	✓	Appropriate bit is set to 1.
The FSR.ftt field is set to:	nc	3	2	1

17.4.3 Trap Priority

The traps generated by floating-point exceptions (*fp_disabled*, *fp_exception_ieee_754*, and *fp_exception_other*) are prioritized.

17.5 IEEE Traps

The Underflow, Overflow, Inexact, Division-by-zero, and Invalid IEEE traps are supported in standard and nonstandard modes. They are listed in TABLE 17-12, *Floating-point Unit Exceptions,* on page 421 and operate according to the IEEE 754-1985 Standard.

17.5.1 IEEE Trap Enable Mask (TEM)

Individual IEEE traps (*nv*, *of*, *uf*, *dz*, and *nx*) are masked by the `FSR.TEM` bits.

When a trap is masked and an exception is detected, then the appropriate `FSR.cexc` bit(s) are set and the destination register is written with data shown in TABLE 17-3, TABLE 17-4, TABLE 17-5, TABLE 17-6, TABLE 17-7, TABLE 17-8, and TABLE 17-9.

17.5.2 IEEE Invalid (nv) Trap

The IEEE invalid exception (`nv`) is generated when the source operand is a NaN (signalling or quiet), or the result cannot fit in the integer format.

The `nv` trap for an invalid case can be masked using the `FSR` register.

17.5.3 IEEE Overflow (of) Trap

When an overflow occurs the inexact flag is also set.

If an overflow occurs *and* the IEEE Overflow (of) and Invalid (`nv`) traps are enabled (`FSR.TEM.NVM` = 1), then a *fp_exception_IEEE_754* is generated. If the Overflow trap is masked and the operation is valid, then the destination register (rd) receives Infinity.

The Overflow Trap is caused when the result of an arithmetic operation exceeds the range supported by the floating-point or integer number precision. This can happen in many different cases as listed in the tables of this section.

17.5.4 IEEE Underflow (uf) Trap

When a Normal number underflows the inexact flag is also set. Underflow is detected before rounding.

The Underflow condition leads to a Subnormal result unless gross underflow is detected. In that case the result is 0 and the inexact flag is raised.

Underflow is discussed in detail in section 17.6, *Underflow Operation,* on page 424.

17.5.5 IEEE Divide-by-Zero (DZ) Trap

When a number is divided by zero, the Divide-by-zero flag is asserted and an *ieee_exception* is generated, if enabled. The *dz* flag and trap can only be generated by the `FDIV` instruction.

17.5.6 IEEE Inexact (nx) Trap

When an inexact condition occurs, the processor sets the `FSR.aexc.nxa` and/or the `FSR.cexc.nxc` bits whenever the rounded result of an operation differs from the precise result. The floating-point/integer conversions that generate inexact exceptions are shown in TABLE 17-14.

The Inexact (nx) flag is asserted for most an overflow or underflow conditions.

The Inexact trap is caused when the ideal result cannot fit into the destination format:

- most square root operations
- some add, subtract, multiply, and divide operations
- some number and precision conversion operations

TABLE 17-14 FP ↔ Integer Conversions that Generate Inexact Exceptions

Instruction	Conversion Description	Unmasked Exception, TEM=0	Masked Exception, TEM=1
FsTOi FdTOi	FP to 32-bit integer when the source operand is not between $-(2^{31} - 1)$ and 2^{31}, then the result is inexact.	Integer number, `nx`	`nx` ieee trap
FsTOx FdTOx	FP to 64-bit integer when the source operand is not between $-(2^{63} - 1)$ and 2^{63}, then the result is inexact.	Integer number, `nx`	`nx` ieee trap
FiTOs	Integer to FP when the 32-bit integer source operand magnitude is not exactly representable in single precision (23-bit fraction).[1]	Single Precision Normal, `nx`	`nx` ieee trap
FxTOs	Integer to FP when the 64-bit integer source operand magnitude is not exactly representable in single precision (23-bit fraction).[1]	Single Precision Normal, `nx`	`nx` ieee trap
FxTOd	Integer to FP when the 64-bit integer source operand magnitude is not exactly representable in double precision (52-bit fraction).[2]	Double Precision Normal, `nx`	`nx` ieee trap

1. Even if the operand is $> 2^{24} - 1$, if enough of its trailing bits are zeros, it may still be exactly representable.
2. Even if the operand is $> 2^{53} - 1$, if enough of its trailing bits are zeros, it may still be exactly representable.

17.6 Underflow Operation

Underflow occurs when the result of an operation (before rounding) is less than that representable by a Normal number.

After rounding, the tiny number (underflow) is usually represented by a Subnormal number, but may equal the smallest Normal number if the unrounded result is just below the range of Normal numbers and the rounding mode (specified in `FSR.RD`) moves it into the Normal number range. The underflow result will be zero, Subnormal, or the smallest Normal value.

Compatibility Note – The FPU does not support exponent wrapping for underflow or overflow.

17.6.1 Trapped Underflow

The FPU will trap on underflow if the `FSR.TEM.UFM` bit is set to 1. Since tininess is detected before rounding, trapped underflow occurs when the exact unrounded result has a magnitude between zero and the smallest representable Normal number in the precision of the destination format.

When underflow is trapped, the destination and other registers are left unchanged, see section 17.4.2, *Trap Event,* on page 422.

17.6.2 Untrapped Underflow

The FPU will not generate an underflow trap when an underflow occurs, if the `FSR.TEM.UFM` bit is set to 0.

If the result causes an underflow and the result after rounding is exact, then the FPU will not generate an inexact trap.

Tininess detection before rounding is summarized in TABLE 17-15.

Define a few terms:

- ***u*** is the unrounded (exact) value of the result.
- ***r*** is the rounded value of ***u*** (occurs when there is no trap generated)
- Underflow is when: $0 < |u| <$ *smallest Normal number.*

TABLE 17-15 Underflow Exception Summary

Underflow : **Inexact :**		**enabled** (UFM = 1) **don't care** (NXM = x)	**masked** (UFM = 0) **enabled** (NXM = 1)	**masked** (UFM = 0) **masked** (NXM = 0)
u = ***r***	***r*** is minimum Normal	none	none	none
exact result	***r*** is Subnormal	set ufc, ieee trap	none	none
	r is Zero	none	none	none

TABLE 17-15 Underflow Exception Summary *(Continued)*

Underflow :		**enabled** (UFM = 1)	**masked** (UFM = 0)	**masked** (UFM = 0)
Inexact :		**don't care** (NXM = x)	**enabled** (NXM = 1)	**masked** (NXM = 0)
u r	***r*** is minimum Normal	set ufc, ieee trap	set nxc, ieee trap	set ufc, set ufa
	r is Subnormal	set ufc, ieee trap	set nxc, ieee trap	set ufc, set ufa
inexact result	***r*** is Zero	set ufc, ieee trap	set nxc, ieee trap	set ufc, set ufa
		set nxc means FSR.cexc.nxc set to 1 **set ufc** means FSR.cexc.ufc set to 1 **set ufa** means FSR.aexc.ufa set to 1 **ieee trap** means *fp_exception_ieee_754*		

17.7 IEEE NaN Operations

When a NaN operand appears or a NaN result is generated, and the invalid (`nv`) trap is enabled (`FSR.TEM.NVM` = 1), then the *fp_exception_ieee_754* occurs.

If the invalid (*nv*) trap is masked (`FSR.TEM.NVM` = 0), then a signalling NaN operand is transformed into a quiet NaN. A quiet NaN operand will propagate to the destination register. Subnormals operations are described in TABLE 17-16, *Results from NaN Operands,* on page 427.

Whenever a NaN is created from non NaN operands, the *nv* flag is set.

17.7.1 Signaling and Quiet NaNs

SNaN and QNaN numbers are unsigned, the sign bit is an extension of the NaN's fraction field.

SNaN operands propagate to the destination register as a QNaN result when the `nv` exception is masked. All operations with NaN operands keep the sign bit unchanged including a FSQRT operation.

NaNs are generated for the conditions shown in section 17.7.4, *NaN Results from Operands without NaNs,* on page 428.

17.7.2 SNaN to QNaN Transformation

The signalling to quiet NaN transformation causes:

- The most significant bits of the operand fraction are copied to the most significant bits of the result's fraction. In conversion to a narrower format, excess low-order bits of the operand fraction are discarded. In conversion to a wider format, unwritten low-order bits of the result fraction are set to 0.
- The quiet bit (the most significant bit of the result fraction) is set to 1 (the NaN transformation produces a QNaN).
- The sign bit is copied from the operand to the result without modification.

17.7.3 Operations with NaN Operands

Operations with NaN operands may assert the IEEE invalid trap flag (*nv*). These operations are listed in TABLE 17-16.

If the Invalid Trap is enabled (`FSR.TEM.NVM` = 1), then a trap event occurs as described in section 17.4.2, *Trap Event,* on page 422.

TABLE 17-16 Results from NaN Operands

Operation		**RESULT from the operation includes one or more of the following:** • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
		Masked Exception, TEM.NVM=0		**Enabled Exception, TEM.NVM=1**	
		rd or fcc register written	**flag set**	**rd or fcc register written**	**flag set**
One Operand $rs_2 \rightarrow rd$					
Any	*QNaN*	QNaN, see note[1]	no	QNaN, see note[1]	no
Any	*SNaN*	SNaN → QNaN, see note[1]	set nvc, set nva	no	set nvc, ieee trap
Two Operand $rs_1, rs_2\ [rs_2, rs_1] \rightarrow rd$					
FADD, FSUB, FMUL, FDIV	*QNaN, QNaN*	QNaN_{rs2}	no	QNaN_{rs2}	no
	QNaN, anything except SNaN and QNan	QNaN	no	QNaN	no
	SNaN, SNaN	SNaN_{rs2} → QNaN, see note[1]	set nvc, set nva	no	set nvc, ieee trap
	SNaN, anything except SNaN	SNaN → QNaN, see note[1]	set nvc, set nva	no	set nvc, ieee trap
FCMPEs,d	*[SNaN or QNaN], anything*	fcc=3 (*unordered*)	set nvc, set nva	no	set nvc, ieee trap

TABLE 17-16 Results from NaN Operands *(Continued)*

Operation		RESULT from the operation includes one or more of the following: • Number in f register, see *Trap Event* note, page 422. • Exception bit set, see TABLE 17-12. • Trap occurs, see abbreviations in TABLE 17-12. • Underflow/Overflow may occur.			
		Masked Exception, TEM.NVM=0		**Enabled Exception, TEM.NVM=1**	
		rd or fcc register written	**flag set**	**rd or fcc register written**	**flag set**
FCMPs,d	*SNaN, anything*	fcc=3 (*unordered*)	set nvc, set nva	no	set nvc, ieee trap
FCMPs,d	*QNaN, anything except SNaN*	fcc=3 (*unordered*)	no	fcc=3 (*unordered*)	no

1. For the Fs,dTOs,d and other instructions, see section 17.7.2, *SNaN to QNaN Transformation,* on page 426.

Note – Notice from TABLE 17-16 that the *compare and cause exception if unordered* instruction (`FCMPEs,d`) will cause an invalid (*nv*) exception if either operand is a quiet or signalling NaN. The `FCMP` instruction causes an exception for signalling NaNs only.

17.7.4 NaN Results from Operands without NaNs

The following operations generate NaNs, see section 17.3, *IEEE Operations,* on page 411, for details.

- `FSQRT` [*–Normal*, or *–0*]
  - `FDIV` ±*0*

17.8 Subnormal Operations

The handling of Subnormals is different for standard and nonstandard floating-point modes. The handling of operands and results are described separately in the following sections.

17.8.1 Response to Subnormal Operands

The FPU responds to Subnormal operands and results in either hardware or by generating an *fp_exception_other (*with `FSR.ftt` = 2*, unfinished_FPop*).

The response of the FPU depends on the operating mode of the floating-point unit. This is controlled by the `FSR.NS` bit.

Standard Mode

In Standard mode, the FPU generally traps when a Subnormal operand is detected or a Subnormal result is generated. In this situation, the system software must perform or complete the operation.

The FPU supports the following in Standard mode:

- Some cases of Subnormal operands are handled in hardware.
- Gross underflow results are supported in hardware for `FdTOs`, `FMULs,d`, and `FDIVs,d` instructions.

Nonstandard Mode

In Nonstandard mode the FPU generally flushes Subnormal operands to 0 (with the same sign as the SbN number) and proceeds to use the value in the operation. Subnormal results (those that would otherwise cause an *unfinished_FPop*) are also flushed to 0 in Nonstandard mode.

If the higher priority invalid operation (`nv`) or divide-by-zero (`dz`) condition occurs, then the corresponding condition(s) are flagged in the `FSR.cexc` register field. If the trap is enabled (`FSR.TEM` register), then an *fp_exception_ieee_754* trap occurs. If the trap is disabled, then the corresponding condition(s) are also flagged in the `FSR.aexc` register field.

If neither the invalid nor divide-by-zero conditions occur, then an inexact condition plus any other detected floating-point exception conditions are flagged in the `FSR.cexc` register field. If an IEEE trap is enabled (`FSR.TEM` register), then an *fp_exception_ieee_754* trap occurs. If the trap is disabled, then the corresponding condition(s) are also flagged in the `FSR.aexc` register field.

17.8.2 Subnormal Number Generation

Handling of the `FMULs`, `FMULd`, `FDIVs`, `FDIVd`, and `FdTOs` instructions requires further explanation.

Define:

- Sign_r = sign of result,
- RT_{Eff} = round nearest effective truncate or round truncate,
- RP = round to +Infinity,
- RM = round to –Infinity,
- RND = `FSR.RD,`

- Er = biased exponent result,
- E_{rb} = the biased exponent result before rounding,
- E(`rs`$_1$) = biased exponent of `rs`$_1$ operand, and
- `P_rs`$_1$ = precision of the `rs`$_1$ operand.

The value of the constants dependent on precision type, see TABLE 17-17.

TABLE 17-17 Subnormal Handling Constants per Destination Register Precision

Destination Register Precision (P)	Number of Bits in Exponent Field	Exponent Bias (E_{BIAS})	Exponent Max (E_{MAX})	Exponent Gross Underflow (E_{GUF})
Single	8	127	255	-24
Double	11	1023	2047	-53

- For `FMULs` and `FMULd`: E_r = E(`rs`$_1$) + E(`rs`$_2$) – E_{BIAS}.
- For `FDIVs` and `FDIVd`: E_r = E(`rs`$_1$) – E(`rs`$_2$) + E_{BIAS} – 1.

When two Normal operands of `FMULs,d` and `FDIVs,d` generate a Subnormal result, the Erb is calculated using the algorithm shown in code example 17-1.

CODE EXAMPLE 17-1 Normal Operands Generating a Subnormal Result Pseudocode

```
If (fraction_msb overflows)   // i.e., fraction_msb >= 1'd2
{
   $E_{rb}$ = $E_r$ + 1
}
ELSE
{
   $E_{rb}$ = $E_r$
}
```

- For `FdTOs`, E_r = E(`rs`$_2$) – E_{BIAS}(P_rs$_2$) + E_{BIAS}(P_rd), where P_rs$_2$ is the larger precision of the source and P_rd is the smaller precision of the destination.

Even though 0 ≤ [E(`rs`$_1$) or E(`rs`$_2$)] ≤ 255 for each single precision biased operand exponent, the *computed* biased exponent result (E_r) can be 0 ≤ E_r ≤ 255 or can even be negative. For example, for the `FMULs` instruction:

- If E(`rs`$_1$) = E(`rs`$_2$) = +127, then E_r = +127 (127 + 127 – 127)
- If E(`rs`$_1$) = E(`rs`$_2$) = 0, then E_r = –127 (0 + 0 – 127)

Overflow Result

- If the appropriate trap enable masks are not set (`FSR.OFM` = 0 *and* `FSR.NXM` = 0), then set `FSR.aexc` and `FSR.cexc` overflow and inexact flags: `FSR.ofa` = 1, `FSR.nxa` = 1, `FSR.ofc` = 1, and `FSR.nxc` = 1. No trap is generated.
- If any or both of the appropriate trap enable masks are set (`FSR.OFM` = 1 *or* `FSR.NXM` = 1), then only an IEEE overflow trap is generated: `FSR.ftt` = 1. The particular `FSR.cexc` bit that is set follows the SPARC V9 architecture:
  - If `FSR.OFM` = 0 *and* `FSR.NXM` = 1, then `FSR.nxc` = 1.
  - If `FSR.OFM` = 1 (independent of `FSR.NXM`), then `FSR.ofc` = 1 *and* `FSR.nxc` = 0.

Gross Underflow Zero result

- Result = 0 (with correct sign).
- If the appropriate trap enable masks are not set (`FSR.UFM` = 0 *and* `FSR.NXM` = 0), then set the `FSR.aexc` and `FSR.cexc` underflow and inexact flags: `FSR.ufa` = 1, `FSR.nxa` = 1, `FSR.ufc` = 1, and `FSR.nxc` = 1. A trap is not generated.
- If either or both of the appropriate trap enable masks are set (`FSR.UFM` = 1 *or* `FSR.NXM` = 1), then only an IEEE underflow trap is generated: `FSR.ftt` = 1 *and* `FSR.cexc.uf` = 1. The particular `FSR.cexc` bit that is set diverges from previous UltraSPARC implementations to follow the SPARC V9 architecture:
  - If `FSR.UFM` = 0 **and** `FSR.NXM` = 1, then `FSR.nxc` = 1.
  - If `FSR.UFM` = 1, independent of `FSR.NXM`, then `FSR.ufc` = 1 and `FSR.nxc` = 0.

Subnormal Handling Override

- Result is an QNaN or SNaN
  - Subnormal + SNaN = QNaN, invalid exception generated
    - Standard mode: No *unfinished_FPop*
    - Nonstandard mode: No FSR.NX
  - Subnormal + QNaN = QNaN, no exception generated
    - Standard mode: No *unfinished_FPop*
    - Nonstandard mode: No FSR.NX
- Result already generates an exception (Divide-by-zero or Invalid operation)
  - `FSQRT`(number less than zero) = invalid
- Result is Infinity:

    - Subnormal + Infinity = Infinity, no exception generated
        - Standard mode: No *unfinished_FPop*
        - Nonstandard mode: No FSR.nx
    - Standard mode: Subnormal × Infinity = Infinity
    - Nonstandard mode: Subnormal × Infinity = QNaN with nv exception (Subnormal is flushed to zero)
- Result is zero:
    - Subnormal × 0 = 0, no exception generated
        - Standard mode: No *unfinished_FPop*
        - Nonstandard mode: No FSR.nx

SECTION VII

Special Topics

CHAPTER **18**

Reset and RED_state

This chapter examines RED_state (Reset, Error, and Debug state) in the following sections:

- RED_state Characteristics
- Resets
- RED_state Trap Vector
- Machine States

18.1 RED_state Characteristics

A reset or trap that sets PSTATE.RED (including a trap in RED_state) will clear the DCU Control Register, including enable bits for I-cache, D-cache, I-MMU, D-MMU, and virtual and physical watchpoints. The characteristics of RED_state include the following:

- The default access in RED_state is non-cacheable; therefore, there must be non-cacheable scratch memory somewhere in the system.
- The D-cache, watchpoints, and D-MMU can be enabled by software in RED_state, but any trap will disable them again.
- The I-MMU and consequently the I-cache are always disabled in RED_state. Disabling overrides the enable bits in the DCU Control Register.

When PSTATE.RED is explicitly set by a software write, there are no side-effects other than the I-MMU is disabled. Software must create the appropriate state itself.

A trap when TL = MAXTL – 1 immediately brings the processor into RED_state. In addition, a trap when TL = MAXTL immediately brings the processor into error_state. Upon error_state entry, the processor automatically recovers through watchdog reset (WDR) into RED_state. A trap to error_state immediately triggers WDR. A Signal

Monitor (SIGM) instruction generates a software-initiated reset (SIR) trap on the local processor. A trap to software-initiated reset causes an SIR trap on the processor and brings the processor into RED_state.

During the RED_state, the caches continue to snoop and maintain coherence if DVMA or other processors are still issuing cacheable accesses.

Note – A recommended way to exit RED_state is with a DONE or RETRY. Exiting RED_state by writing 0 to PSTATE.RED in the delay slot of a JMPL is not recommended and may result in an instruction access error.

18.2 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).

18.2.1 Power-on Reset (POR)

A Power-on Reset (POR) occurs when the POK pin is activated and stays asserted until the processor is within its specified operating range. When the POK pin is active, all other resets and traps are ignored. POR has a trap type of 1 at physical address offset 20_{16}. Any pending external transactions are canceled.

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

- In the I-cache, valid bits must be cleared and Microtag (Utag) bits must be set so that each way within a set has a unique Utag value.
- In the D-cache, valid bits must be cleared and Utag bits must be set so that each way within a set has a unique Utag 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.

The MCU Refresh Control Register, as well as the Fireplane Configuration Register, must be initialized after a POR.

In SSM systems, the Utags contained in memory must be initialized before any Fireplane transactions are generated.

Caution – Executing a DONE or RETRY instruction when TSTATE is not initialized 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.

18.2.2 System Reset

A System Reset occurs when the Reset pin is activated. When the Reset pin is active, all other resets and traps are ignored. System Reset has a trap type of 1 at physical address offset 20_{16}. Any pending external transactions are canceled.

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

18.2.3 Externally Initiated Reset (XIR)

An externally initiated reset (XIR) is sent to the processor through an external hardware pin. It causes a SPARC V9 XIR, which has a trap type 3_{16} at physical address offset 60_{16}. XIR has higher priority than all other resets except POR and System Reset.

XIR affects only one processor rather than the entire system. Memory state, cache state, and most Control Status Register states 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.

18.2.4 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.

18.2.5 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 80_{16}. SIR affects only one processor rather than the entire system.

18.3 RED_state Trap Vector

When the UltraSPARC III Cu 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`$_{16}$.

18.4 Machine States

TABLE 18-1 shows the machine states created as a result of any reset or after `RED_state` is entered. `RSTVaddr` is often abbreviated as `RSTV` in the table.

TABLE 18-1 Machine State After Reset and in RED_state *(1 of 6)*

Name	Fields	Hard_POR	System Reset	WDR	XIR	SIR	RED_state‡
Integer Registers		Unknown	Unchanged	Unchanged			
Floating-Point Registers		Unknown	Unchanged	Unchanged			
L2-cache Control Register		0	0	Unchanged			
`RSTVaddr` value		VA = `FFFF FFFF F000 0000`$_{16}$ PA = `7FF F000 0000`$_{16}$					
`PC` `nPC`		RSTV | 20_{16} RSTV | 24_{16}	RSTV | 20_{16} RSTV | 24_{16}	RSTV | 40_{16} RSTV | 44_{16}	RSTV | 60_{16} RSTV | 64_{16}	RSTV | 80_{16} RSTV | 84_{16}	RSTV | $A0_{16}$ RSTV | $A4_{16}$

TABLE 18-1 Machine State After Reset and in RED_state *(2 of 6)*

Name	Fields	Hard_POR	System Reset	WDR	XIR	SIR	RED_state[‡]
PSTATE	MM	0 (TSO)	0 (TSO)	0 (TSO)			
	RED	1 (RED_state)	1 (RED_state)	1 (RED_state)			
	PEF	1 (FPU on)	1 (FPU on)	1 (FPU on)			
	AM	0 (Full 64-bit address	0 (Full 64-bit address	0 (Full 64-bit address)			
	PRIV	1 (Privileged mode)	1 (Privileged mode)	1 (Privileged mode)			
	IE	0 (Disable interrupts)	0 (Disable interrupts)	0 (Disable interrupts)			
	AG	1 (Alternate globals selected)	1 (Alternate globals selected)	1 (Alternate globals selected)			
	CLE	0 (Current little-endian)	0 (Current little-endian)	PSTATE.TLE			
	TLE	0 (Trap little-endian)	0 (Trap little-endian)	Unchanged			
	IG	0 (Interrupt globals not selected)	0 (Interrupt globals not selected)	0 (Interrupt globals not selected)			
	MG	0 (MMU globals not selected)	0 (MMU globals not selected)	0 (MMU globals not selected)			
TBA<63:15>		Unknown	Unchanged	Unchanged			
Y		Unknown	Unchanged	Unchanged			
PIL		Unknown	Unchanged	Unchanged			
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 & ~$1F_{16}$ nPC = PC + 4	PC nPC	

TABLE 18-1 Machine State After Reset and in RED_state *(3 of 6)*

Name	Fields	Hard_POR	System Reset	WDR	XIR	SIR	RED_state[‡]
TSTATE	CCR	Unknown	Unchanged	CCR			
	ASI	Unknown	Unchanged	ASI			
	PSTATE	Unknown	Unchanged	PSTATE			
	CWP	Unknown	Unchanged	CWP			
	PC	Unknown	Unchanged	PC			
	nPC	Unknown	Unchanged	nPC			
TICK	NPT	1	1	Unchanged	Unchanged	Unchanged	
	counter	Restart at 0	Restart at 0	Count	Restart at 0	Count	
CANSAVE		Unknown	Unchanged	Unchanged			
CANRESTORE		Unknown	Unchanged	Unchanged			
OTHERWIN		Unknown	Unchanged	Unchanged			
CLEANWIN		Unknown	Unchanged	Unchanged			
WSTATE	OTHER	Unknown	Unchanged	Unchanged			
	NORMAL	Unknown	Unchanged	Unchanged			
VER	MANUF	$003E_{16}$					
	IMPL	0015_{16}					
	MASK	Mask dependent					
	MAXTL	5					
	MAXWIN	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	1	1	Unchanged			
	counter	0	0	Count			
STICK_COMPARE	INT_DIS	1 (off)	1 (off)	Unchanged			
	TICK_CMPR	0	0	Unchanged			

TABLE 18-1 Machine State After Reset and in RED_state *(4 of 6)*

Name	Fields	Hard_POR	System Reset	WDR	XIR	SIR	RED_state‡
PCR	S1	Unknown	Unchanged	Unchanged			
	S0	Unknown	Unchanged	Unchanged			
	UT (trace user)	Unknown	Unchanged	Unchanged			
	ST (trace system)	Unknown	Unchanged	Unchanged			
	PRIV (privileged access)	Unknown	Unchanged	Unchanged			
PIC	All	Unknown	Unknown	Unknown			
GSR	IM	0	0	Unchanged			
	others	Unknown	Unchanged	Unchanged			
DCR	MS	0	0	Unchanged			
	SI	0	0	Unchanged			
	RPE	0	0	Unchanged			
	BPE	0	0	Unchanged			
	OBS	0	0	Unchanged			
	IFPOE	0	0	Unchanged			
	IPE	0	0	Unchanged			
	DPE	0	0	Unchanged			
Non-SPARC V9 ASIs							
Fireplane Information							
DCUCR	WE	0 (off)	0 (off)0	Unchanged			
	All others	0 (off)	0 (off)	0 (off)			
INSTRUCTION_TRAP	All	0 (off)	0 (off)	Unchanged			
VA_WATCHPOINT		Unknown	Unchanged	Unchanged			
PA_WATCHPOINT		Unknown	Unchanged	Unchanged			

TABLE 18-1 Machine State After Reset and in RED_state *(5 of 6)*

Name	Fields	Hard_POR	System Reset	WDR	XIR	SIR	RED_state‡
I-SFSR, D-SFSR	ASI	Unknown	Unchanged	Unchanged			
	FT	Unknown	Unchanged	Unchanged			
	E	Unknown	Unchanged	Unchanged			
	CTXT	Unknown	Unchanged	Unchanged			
	PRIV	Unknown	Unchanged	Unchanged			
	W	Unknown	Unchanged	Unchanged			
	OW (overwrite)	Unknown	Unchanged	Unchanged			
	FV (SFSR valid)	0	0	Unchanged			
	NF	Unknown	Unchanged	Unchanged			
	TM	Unknown	Unchanged	Unchanged			
DMMU_SFAR		Unknown	Unchanged	Unchanged			
INTR_DISPATCH	All	0	0	Unchanged			
INTR_RECEIVE	BUSY	0	0	Unchanged			
	MID	Unknown	Unchanged	Unchanged			
ESTATE_ERR_EN	All	0 (all off)	0 (all off)	Unchanged			
AFAR	PA	Unknown	Unchanged	Unchanged			
AFSR	All	0	Unchanged	Unchanged			
Rfr_CSR	All	Unknown	Unchanged	Unchanged			
Mem_Timing_CSR	All	Unknown	Unchanged	Unchanged			
Mem_Addr_Dec	All	Unknown	Unchanged	Unchanged			
Mem_Addr_Cntl	All	Unknown	Unchanged	Unchanged			
Other Processor-Specific States							
Processor and L2-cache tags, Microtags and data (includes data, instruction, prefetch, and write caches)		Unknown	Unchanged	Unchanged			
Cache snooping		Enabled					
Instruction Queue		Empty					
Store Queue		Empty	Empty	Unchanged			

TABLE 18-1 Machine State After Reset and in RED_state *(6 of 6)*

Name	Fields	Hard_POR	System Reset	WDR	XIR	SIR	RED_state‡
I-TLB, D-TLB[1]	Mappings in #2 (2-way set-associative)	Unknown	Unchanged	Unchanged			
	Mappings in #0 (fully set-associative)	Unknown	Unknown and invalid	Unchanged			
	E (side-effect) bit	1	1	1			
	NC (non-cacheable) bit	1	1	1			

1. The V, L, U, and G bits are cleared at reset.

‡ Processor states are only updated according to the table if RED_state is entered because of a reset or trap. If RED_state is entered because the PSTATE.RED bit was explicitly set to 1, then software must create the appropriate states itself.

Appendix

APPENDIX **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	Ext. to V9?
`ADD`, `ADDcc`	Add (and modify condition codes)	page 454	
`ADDC`, `ADDCcc`	Add with carry (and modify condition codes)	page 454	
`ALIGNADDRESS{_LITTLE}`	Calculate address for misaligned data	page 455	3
`AND`, `ANDcc`	And (and modify condition codes)	page 517	
`ANDN`, `ANDNcc`	And not (and modify condition codes)	page 517	
`ARRAY`(8,16,32)	Three-Dimensional array addressing instructions	page 457	3
`BPcc`	Branch on integer condition codes with prediction	page 474	
`Bicc`^{D}	Branch on integer condition codes	page 605	
`BMASK`	Set the `GSR.MASK` field	page 468	3
`BPr`	Branch on contents of integer register with prediction (also known as BRr)	page 469	
`BSHUFFLE`	Permute bytes as specified by `GSR.MASK`	page 468	3
`CALL`	Call and link	page 476	
`CASA`$^{P_{ASI}}$	Compare and swap word in alternate space	page 477	
`CASXA`$^{P_{ASI}}$	Compare and swap doubleword in alternate space	page 477	
`DONE`^{P}	Return from trap	page 479	
`EDGE`(8,16,32)`{,L,N,LN}`	Edge handling instructions	page 480	3
`FABS`(s,d,q)	Floating-point absolute value	page 493	
`FADD`(s,d,q)	Floating-point add	page 483	
`FALIGNDATA`	Perform data alignment for misaligned data	page 455	3
`FAND`{`S`}	Logical AND operation	page 514	3
`FANDNOT`(1,2){`S`}	Logical AND operation with one inverted source	page 514	3
`FBfcc`^{D}	Branch on floating-point condition codes	page 603	
`FBPfcc`	Branch on floating-point condition codes with prediction	page 471	

TABLE A-2 Instruction Set *(2 of 6)*

Operation	Name	Page	Ext. to V9?
FCMP(s,d,q)	Floating-point compare	page 486	
FCMPE(s,d,q)	Floating-point compare (exception if unordered)	page 486	
FCMP(GT,LE,NE,EQ)(16,32)	Pixel compare operations	page 550	3
FDIV(s,d,q)	Floating-point divide	page 494	
FdMULq	Floating-point multiply double to quad	page 494	
FEXPAND	Pixel expansion	page 558	3
FiTO(s,d,q)	Convert integer to floating-point	page 491	
FLUSH	Flush instruction memory	page 497	
FLUSHW	Flush register windows	page 499	
FMOV(s,d,q)	Floating-point move	page 493	
FMOV(s,d,q)cc	Move floating-point register if condition is satisfied	page 524	
FMOV(s,d,q)r	Move floating-point register if integer register contents satisfy condition	page 529	
FMUL(s,d,q)	Floating-point multiply	page 494	
FMUL8x16	8x16 partitioned product	page 545	3
FMUL8x16(AU,AL)	8x16 upper/lower α partitioned product	page 545	3
FMUL8(SU,UL)x16	8x16 upper/lower partitioned product	page 546	3
FMULD8(SU,UL)x16	8x16 upper/lower partitioned product	page 548	3
FNAND{S}	Logical NAND operation	page 514	3
FNEG(s,d,q)	Floating-point negate	page 493	
FNOR{S}	Logical NOR operation	page 514	3
FNOT(1,2){S}	Copy negated source	page 514	3
FONE{S}	One fill	page 514	3
FOR{S}	Logical OR operation	page 514	3
FORNOT(1,2){S}	Logical OR operation with one inverted source	page 514	3
FPACK(16,32, FIX)	Pixel packing	page 554, page 556, page 557	3
FPADD(16,32){S}	Pixel add (single) 16- or 32-bit	page 542	3
FPMERGE	Pixel merge	page 559	3
FPSUB(16,32){S}	Pixel subtract (single) 16- or 32-bit	page 542	3
FsMULd	Floating-point multiply single to double	page 494	
FSQRT(s,d,q)	Floating-point square root	page 496	
FSRC(1,2){S}	Copy source	page 514	3
F(s,d,q)TOi	Convert floating-point to integer	page 488	
F(s,d,q)TO(s,d,q)	Convert between floating-point formats	page 489	
F(s,d,q)TOx	Convert floating-point to 64-bit integer	page 488	

TABLE A-2 Instruction Set *(3 of 6)*

Operation	Name	Page	Ext. to V9?
FSUB(s,d,q)	Floating-point subtract	page 483	
FXNOR{S}	Logical XNOR operation	page 514	3
FXOR{S}	Logical XOR operation	page 514	3
FxTO(s,d,q)	Convert 64-bit integer to floating-point	page 491	
FZERO{S}	Zero fill	page 514	3
ILLTRAP	Illegal instruction	page 500	
JMPL	Jump and link	page 501	
LDD^{D}	Load integer doubleword	page 612	
LDDA$^{D,\ P_{ASI}}$	Load integer doubleword from alternate space	page 614	
LDDA ASI_NUCLEUS_QUAD*	Atomic quad load	page 510	3
LDDF	Load double floating-point	page 502	
LDDFA$^{P_{ASI}}$	Load double floating-point from alternate space	page 460	
LDDFA ASI_BLK*	Block loads	page 460	3
LDDFA ASI_FL*	Short floating-point loads (VIS I)	page 580	3
LDF	Load floating-point	page 502	
LDFA$^{P_{ASI}}$	Load floating-point from alternate space	page 502	
LDFSR^{D}	Load floating-point state register lower	page 611	
LDQF	Load quad floating-point	page 502	
LDQFA$^{P_{ASI}}$	Load quad floating-point from alternate space	page 502	
LDSB	Load signed byte	page 506	
LDSBA$^{P_{ASI}}$	Load signed byte from alternate space	page 508	
LDSH	Load signed halfword	page 506	
LDSHA$^{P_{ASI}}$	Load signed halfword from alternate space	page 508	
LDSTUB	Load-store unsigned byte	page 512	
LDSTUBA$^{P_{ASI}}$	Load-store unsigned byte in alternate space	page 513	
LDSW	Load signed word	page 506	
LDSWA$^{P_{ASI}}$	Load signed word from alternate space	page 508	
LDUB	Load unsigned byte	page 506	
LDUBA$^{P_{ASI}}$	Load unsigned byte from alternate space	page 508	
LDUH	Load unsigned halfword	page 506	
LDUHA$^{P_{ASI}}$	Load unsigned halfword from alternate space	page 508	
LDUW	Load unsigned word	page 506	
LDUWA$^{P_{ASI}}$	Load unsigned word from alternate space	page 508	
LDX	Load extended	page 506	
LDXA$^{P_{ASI}}$	Load extended from alternate space	page 508	
LDXFSR	Load floating-point state register	page 502	
MEMBAR	Memory barrier	page 519	

TABLE A-2 Instruction Set *(4 of 6)*

Operation	Name	Page	Ext. to V9?
MOVcc	Move integer register if condition is satisfied	page 524	
MOVr	Move integer register on contents of integer register	page 536	
MULScc^{D}	Multiply step (and modify condition codes)	page 616	
MULX	Multiply 64-bit integers	page 537	
NOP	No operation	page 538	
OR, ORcc	Inclusive OR (and modify condition codes)	page 517	
ORN, ORNcc	Inclusive OR not (and modify condition codes)	page 517	
PDIST	Pixel component distance	page 552	3
POPC	Population Count	page 559	
PREFETCH	Prefetch data	page 560	
PREFETCHA$^{P_{ASI}}$	Prefetch data from alternate space	page 560	
RDASI	Read ASI register	page 568	
RDASR$^{P_{ASR}}$	Read ancillary state register	page 568	
RDCCR	Read condition codes register	page 568	
RDDCR^{P}	Read dispatch control register	page 568	
RDFPRS	Read floating-point registers state register	page 568	
RDGSR	Read graphic status register	page 568	
RDPC	Read program counter	page 568	
RDPCR^{P}	Read performance control register	page 568	
RDPIC$^{P_{PIC}}$	Read performance instrumentation counters	page 568	
RDPR^{P}	Read privileged register	page 566	
RDSOFTINT^{P}	Read per-processor soft interrupt register	page 568	
RDSTICK$^{P_{NPT}}$	Read system TICK register	page 568	
RDSTICK_CMPR	Read system TICK compare register	page 568	
RDTICK$^{P_{NPT}}$	Read TICK register	page 568	
RDTICK_CMPR^{P}	Read TICK compare register	page 568	
RDY^{D}	Read Y register	page 619	
RESTORE	Restore caller's window	page 572	
RESTORED^{P}	Window has been restored	page 574	
RETRY^{P}	Return from trap and retry	page 479	
RETURN	Return	page 570	
SAVE	Save caller's window	page 572	
SAVED^{P}	Window has been saved	page 574	
SDIV^{D}, SDIVcc^{D}	32-bit signed integer divide (and modify condition codes)	page 608	
SDIVX	64-bit signed integer divide	page 537	
SETHI	Set high 22 bits of low word of integer register	page 577	
SHUTDOWN	Shut down the processor	page 582	3

TABLE A-2 Instruction Set *(5 of 6)*

Operation	Name	Page	Ext. to V9?
SIAM	Set Interval Arithmetic Mode (VIS II)		
SIR	Software-initiated reset	page 583	
SLL	Shift left logical (IU)	page 578	
SLLX	Shift left logical, extended (IU)	page 578	
SMUL^{D}, SMULcc^{D}	Signed integer multiply (and modify condition codes)	page 616	
SRA	Shift right arithmetic (IU)	page 578	
SRAX	Shift right arithmetic, extended (IU)	page 578	
SRL	Shift right logical (IU)	page 578	
SRLX	Shift right logical, extended (IU)	page 578	
STB	Store byte (IU)	page 588	
STBA$^{P_{ASI}}$	Store byte into alternate space (IU)	page 589	
STBAR^{D}	Store barrier	page 619	
STD^{D}	Store doubleword	page 622	
STDA$^{D,\ P_{ASI}}$	Store doubleword into alternate space	page 623	
STDF	Store double floating-point (FP)	page 584	
STDFA$^{P_{ASI}}$	Store double floating-point into alternate space (FP)	page 586	
STDFA ASI_BLK*	Block stores	page 460	3
STDFA ASI_FL*	Short floating-point stores (VIS I)	page 580	3
STDFA ASI_PST*	Partial Store instructions	page 540	3
STF	Store floating-point (FP)	page 584	
STFA$^{P_{ASI}}$	Store floating-point into alternate space (FP)	page 586	
STFSR^{D}	Store floating-point state register (FP)	page 621	
STH	Store halfword (IU)	page 588	
STHA$^{P_{ASI}}$	Store halfword into alternate space (IU)	page 589	
STQF	Store quad floating-point (FP)	page 584	
STQFA$^{P_{ASI}}$	Store quad floating-point into alternate space (FP)	page 586	
STW	Store word (IU)	page 588	
STWA$^{P_{ASI}}$	Store word into alternate space (IU)	page 589	
STX	Store extended (IU)	page 588	
STXA$^{P_{ASI}}$	Store extended into alternate space (IU)	page 589	
STXFSR	Store extended floating-point state register (MS)	page 584	
SUB, SUBcc	Subtract (and modify condition codes)	page 591	
SUBC, SUBCcc	Subtract with carry (and modify condition codes)	page 591	
SWAP^{D}	Swap integer register with memory	page 625	
SWAPA$^{D,\ P_{ASI}}$	Swap integer register with memory in alternate space	page 626	
TADDcc, TADDccTV^{D}	Tagged add and modify condition codes (trap on overflow)	page 592, page 628	

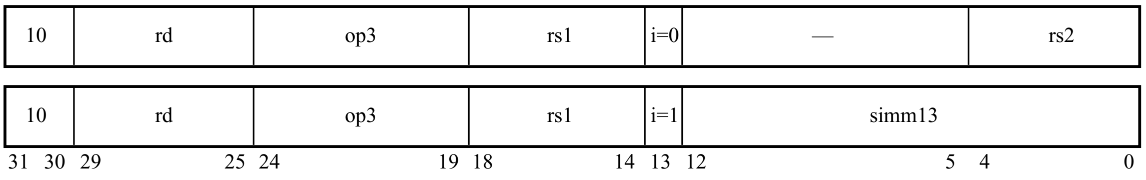
TABLE A-2 Instruction Set *(6 of 6)*

Operation	**Name**	**Page**	**Ext. to V9?**
`Tcc`	Trap on integer condition codes	page 595	
`TSUBcc`, `TSUBccTV`^{D}	Tagged subtract and modify condition codes (trap on overflow)	page 593, page 629	
`UDIV`^{D}, `UDIVcc`^{D}	Unsigned integer divide (and modify condition codes)	page 608	
`UDIVX`	64-bit unsigned integer divide	page 537	
`UMUL`^{D}, `UMULcc`^{D}	Unsigned integer multiply (and modify condition codes)	page 616	
`WRASI`	Write ASI register	page 600	
`WRASR`$^{P_{ASR}}$	Write ancillary state register	page 600	
`WRCCR`	Write condition codes register	page 600	
`WRDCR`^{P}	Write dispatch control register	page 600	
`WRFPRS`	Write floating-point registers state register	page 600	
`WRGSR`	Write graphic status register	page 600	
`WRPCR`^{P}	Write performance control register	page 600	
`WRPIC`$^{P_{PIC}}$	Write performance instrumentation counters register	page 600	
`WRPR`^{P}	Write privileged register	page 597	
`WRSOFTINT`^{P}	Write per-processor soft interrupt register	page 600	
`WRSOFTINT_CLR`^{P}	Clear bits of per-processor soft interrupt register	page 600	
`WRSOFTINT_SET`^{P}	Set bits of per-processor soft interrupt register	page 600	
`WRTICK_CMPR`^{P}	Write TICK compare register	page 600	
`WRSTICK`^{P}	Write System TICK register	page 600	
`WRSTICK_CMPR`^{P}	Write System TICK compare register	page 600	
`WRY`^{D}	Write Y register	page 630	
`XNOR`, `XNORcc`	Exclusive NOR (and modify condition codes)	page 517	
`XOR`, `XORcc`	Exclusive OR (and modify condition codes)	page 517	

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 the SPARC V8 architecture.

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)

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

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)

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

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]`.

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

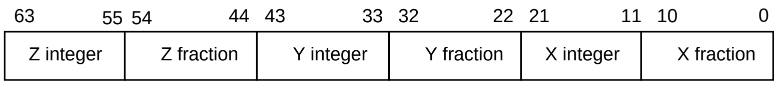


FIGURE A-1 Three-Dimensional Array Fixed-Point Address Format

The integer parts of X, Y, and Z are converted to the following blocked-address formats illustrated in FIGURE A-2 through FIGURE A-4.

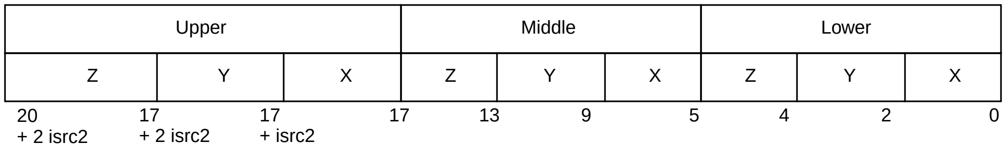


FIGURE A-2 Three-Dimensional Array Blocked-Address Format (`Array8`)

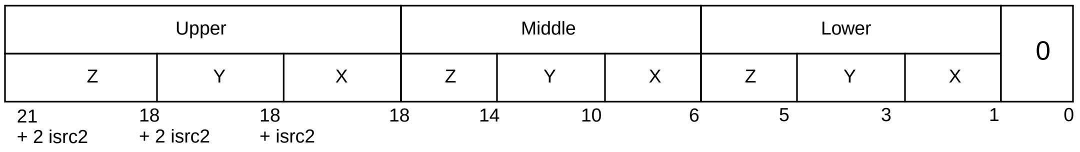


FIGURE A-3 Three-Dimensional Array Blocked-Address Format (`Array16`)

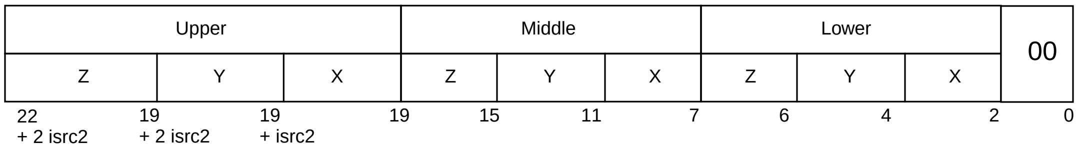


FIGURE A-4 Three-Dimensional Array Blocked-Address Format (`Array32`)

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.

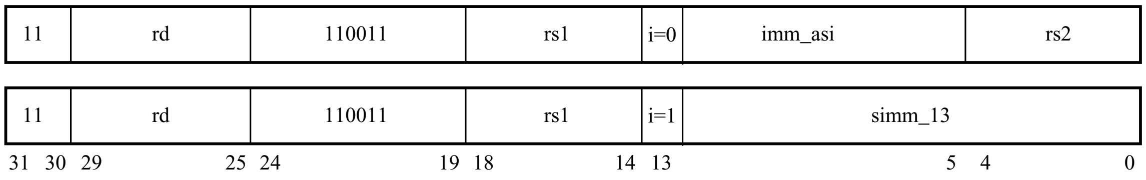
Exceptions

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

11	rd	110111	rs1	i=0	imm_asi	rs2

11	rd	110111	rs1	i=1	simm_13

31 30 29 25 24 19 18 14 13 5 4 0

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 six 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 III Cu processor, BLD does not offer a performance advantage over normal loads. For high performance, we recommend the use of prefetch instructions and 8-byte loads. BST and BST commit can offer performance advantage and are used in high performance UltraSPARC III Cu 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 operate (FPop) 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 FPop instruction only.

The second lowest numbered destination register of the first BLD may be referenced in the second instruction group containing an FPop instruction, and so on.

If this block-load/block-load synchronization mechanism is used, the initial reference to the BLD data must be an FPop instruction (not an FP store), and only instruction groups with FPop 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.

FPop instructions can be issued in a single instruction group with FP stores. If block-load/block-load synchronization is used, FPops and FP stores can be interlaced. This allows an FPop 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 III Cu 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, so 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 follow:

- 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, so 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 BLD/BST

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
    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 `bcopy()` loop to use PREFETCH and floating-point load instructions instead of BLDs.

CODE EXAMPLE A-4 High-Performance UltraSPARC III Cu `bcopy()` Preamble Code *(1 of 2)*

```
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

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
```

CODE EXAMPLE A-4 High-Performance UltraSPARC III Cu `bcopy()` Preamble Code *(2 of 2)*

```
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)

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

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)

00	a	0	rcond	011	d16hi	p	rs1	d16lo
31 30	29	28	27 25	24 22	21 20	19	18 14	13 0

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 III Cu processor does not implement this instruction by tagging each register value. The UltraSPARC III Cu 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)

00	a	cond	101	cc1	cc0	p	disp19
31 30	29	28 25	24 22	21	20	19	18 0

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.

- **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's 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, a *fp_disabled* exception is generated.

Compatibility Note – Unlike the SPARC V8 architecture, the SPARC V9 architecture does not require an instruction between a floating-point compare operation and a floating-point branch (FBfcc, FBPfcc).

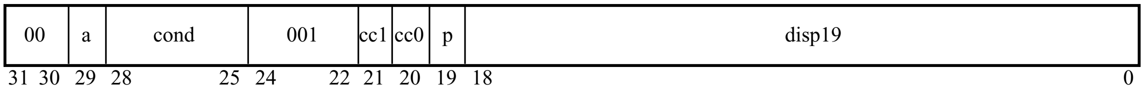
Exceptions

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's 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)

01	disp30
31 30	29 … 0

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 $+2^{31}$ minus four 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)

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

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 (nonlocking) protocols.

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

The `CASA` instruction compares the low-order 32 bits of register `r[rs2]` with a word in memory pointed to by the word address in `r[rs1]`. If the values are equal, then the low-order 32 bits of register `r[rd]` are swapped with the contents of the memory word pointed to by the address in `r[rs1]` and the high-order 32 bits of register `r[rd]` are set to 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: a 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)

10	fcn	op3	—
31 30	29 25	24 19	18 0

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 CC
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 CC
EDGE16	0 0000 0100	Four 16-bit edge boundary processing
EDGE16N	0 0000 0101	Four 16-bit edge boundary processing, no CC
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 CC
EDGE32	0 0000 1000	Two 32-bit edge boundary processing
EDGE32N	0 0000 1001	Two 32-bit edge boundary processing, no CC
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 CC

Format (3)

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

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
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)

10	rd	op3	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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 a *fp_exception_other* exception with FSR.ftt = *unfinished_FPop*, system software emulates the instruction, and the correct numerical result is calculated. The UltraSPARC II and UltraSPARC III Cu class of processors operate identically in this case.

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 UltraSPARC II class of processors, the result is replaced by zero in hardware, without trapping. On the UltraSPARC III Cu 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).

So 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, a *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)

10	000	cc1	cc0	op3	rs1	opf	rs2
31 30	29 27	26	25	24 19	18 14	13 5	4 0

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 the SPARC V8 architecture, the SPARC V9 architecture 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)

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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 ≥ 2^{31}, Inf, NaN
`FsTOx`	$\lvert\text{result}\rvert \geq 2^{52}$, Inf, NaN
`FdTOi`	result < -2^{31}, result ≥ 2^{31}, Inf, NaN
`FdTOx`	$\lvert\text{result}\rvert \geq 2^{52}$, 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)

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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 III Cu 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 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`	$\lvert result \rvert \geq 2^{52}$, $\lvert result \rvert < 2^{-31}$, 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)

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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 III Cu 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)

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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)

10	rd	op3	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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)

10	rd	op3	—	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

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.

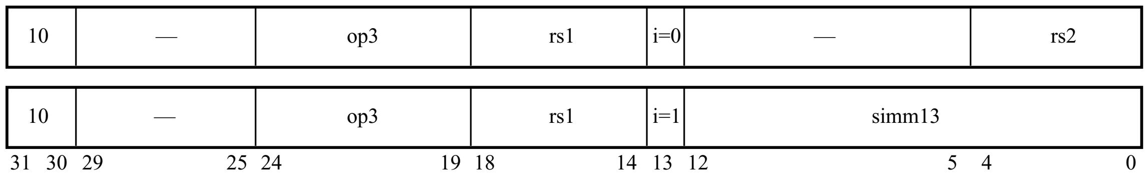
Exceptions

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-call 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 III Cu processor:

- A FLUSH instruction flushes the processor pipeline and synchronizes the processor.
- The instruction cache is kept coherent so 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)

10	—	op3	—	i=0	—
31 30	29 25	24 19	18 14	13	12 0

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.

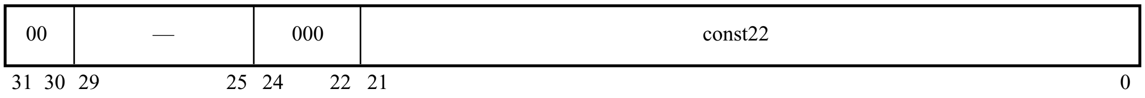
Exceptions

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.

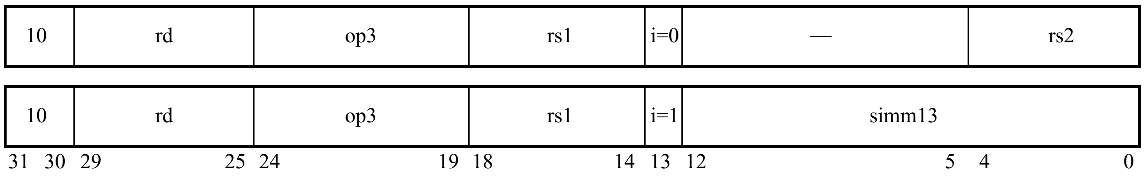
Exceptions

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

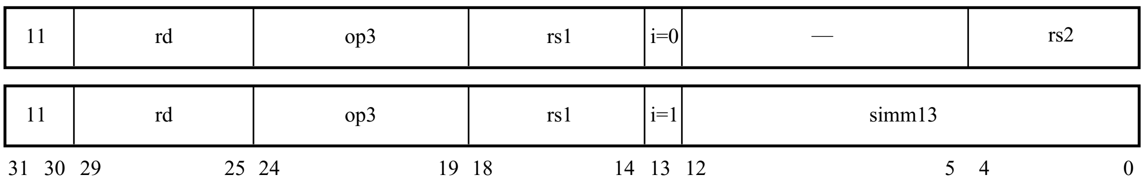
mem_address_not_aligned

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 a *fp_disabled* exception.

`LDDF` requires doubleword aligned. If word alignment is used, then the `LDDF` causes a *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, we recommend that compilers 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 an UltraSPARC III Cu processor, an `LDDF` instruction causes a *LDDF_mem_address_not_aligned* trap if the effective address is 32-bit aligned but not 64-bit (doubleword) aligned.

Exceptions

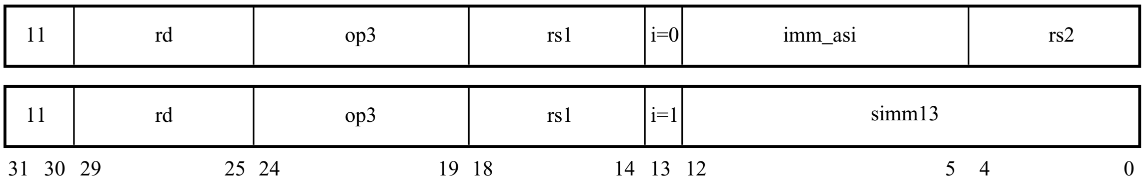
illegal_instruction (`op3`=21_{16} and `rd` = 2–31)
fp_disabled
LDDF_mem_address_not_aligned (`LDDF` only)
mem_address_not_aligned
data_access_exception
PA_watchpoint
VA_watchpoint
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.26 Load Floating-Point from Alternate Space

Opcode	**op3**	**rd**	**Operation**
`LDFA`$^{P_{ASI}}$	11 0000	0–31	Load Floating-Point Register from Alternate Space
`LDDFA`$^{P_{ASI}}$	11 0011	†	Load Double Floating-Point Register from Alternate Space
`LDQFA`$^{P_{ASI}}$	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 a *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 a *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 an UltraSPARC III Cu 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)

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

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.

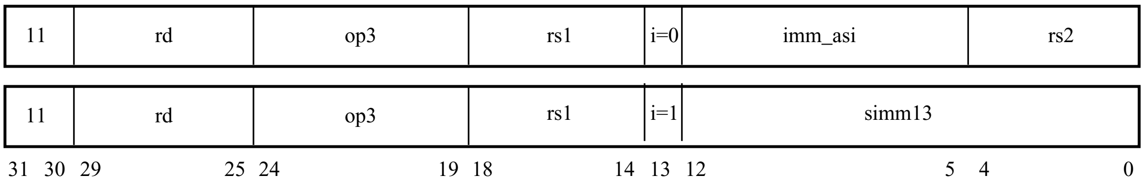
Exceptions

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

11	rd	010011	rs1	i=0	imm_asi	rs2
11	rd	010011	rs1	i=1	simm_13	
31 30	29 25	24 19	18 14	13	5	4 0

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 today's 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 = 8 K
```

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.

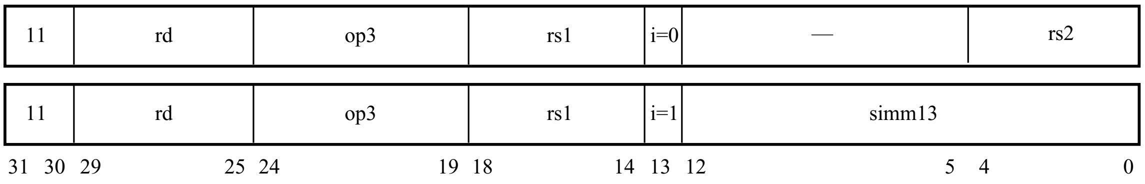
Exceptions

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

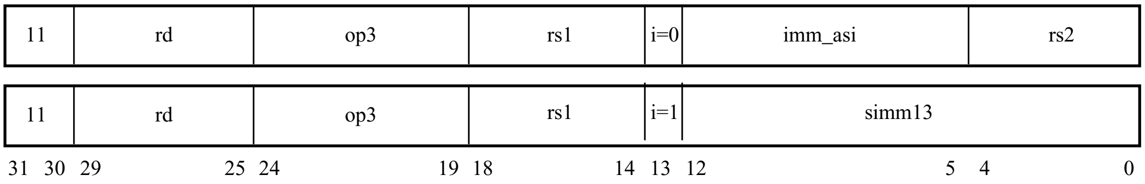
data_access_exception
data_access_error
fast_data_access_MMU_miss

fast_data_access_protection
VA_watchpoint
PA_watchpoint

A.31 Load-Store Unsigned Byte to Alternate Space

Opcode	op3	Operation
LDSTUBA$^{\text{PASI}}$	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

Opcode	opf	Operation
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
FANDNOT2	0 0110 0100	`src1` AND negated `src2`
FANDNOT2S	0 0110 0101	`src1` AND negated `src2`, single-precision

Format (3)

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

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$_{rd}$*

Assembly Language Syntax	
`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*$_{rd}$
`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 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.

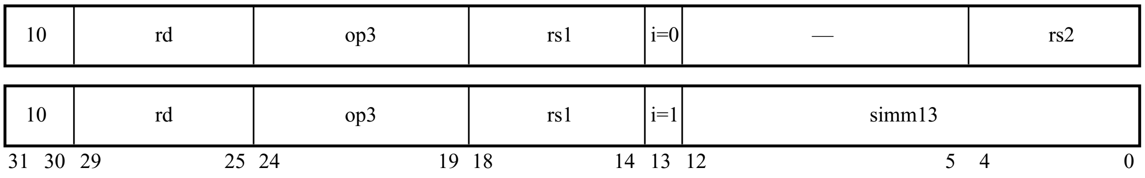
Exceptions

fp_disabled

A.33 Logical Operations

Opcode	op3	Operation
AND	00 0001	AND
ANDcc	01 0001	AND and modify cc's
ANDN	00 0101	AND Not
ANDNcc	01 0101	AND Not and modify cc's
OR	00 0010	Inclusive OR
ORcc	01 0010	Inclusive OR and modify cc's
ORN	00 0110	Inclusive OR Not
ORNcc	01 0110	Inclusive OR Not and modify cc's
XOR	00 0011	Exclusive OR
XORcc	01 0011	Exclusive OR and modify cc's
XNOR	00 0111	Exclusive NOR
XNORcc	01 0111	Exclusive NOR and modify cc's

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)

10	0	op3	0 1111	i=1	—	cmask	mmask
31 30	29 25	24 19	18 14	13	12 7	6 4	3 0

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; 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.
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 an UltraSPARC III Cu processor are generally more ordered with respect to other operations, compared to UltraSPARC I and UltraSPARC II processors. Code written and found to "work" on the UltraSPARC III Cu processor may not work on UltraSPARC I and UltraSPARC II processors if it does not follow the rules for BLD/BST specified for those processors. Code that happens to work on UltraSPARC I and UltraSPARC II processors may not work on the UltraSPARC III Cu processor if it did not meet the coding guidelines specified for those processors. In no case is the coding requirement for the UltraSPARC III Cu processor more restrictive than that of the UltraSPARC I and 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 III Cu rules are less restrictive than SPARC V9, UltraSPARC I, and UltraSPARC II rules and are never more restrictive.

MEMBAR Rules

The UltraSPARC III Cu 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 III Cu 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 UltraSPARC III Cu'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
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 III Cu processor's hardware implementation.

TABLE A-12 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	#

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)

10	rd	op3	0	cond	opf_cc	opf_low	rs2
31 30	29 25	24 19	18	17 14	13 11	10 5	4 0

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)

10	rd	op3	rs1	0	rcond	opf_low	rs2
31 30	29 25	24 19	18 14	13	12 10	9 5	4 0

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 III Cu processor does not implement this instruction by tagging each register value; it 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)

10	rd	op3	cc2	cond	i=0	cc1	cc0	—	rs2
10	rd	op3	cc2	cond	i=1	cc1	cc0	simm11	
31 30	29 25	24 19	18	17 14	13	12	11	10 5	4 0

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 `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.

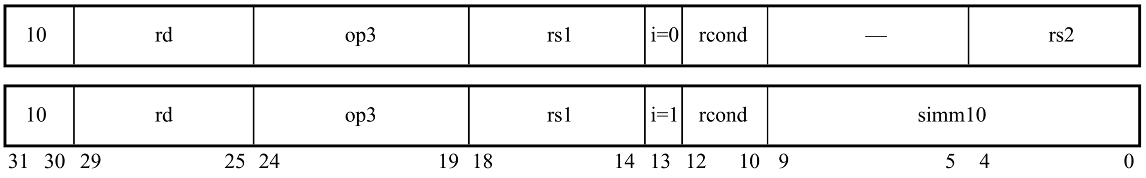
Exceptions

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]` $\neq 0$
`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 III Cu processor does not implement this instruction by tagging each register value; it looks at the full 64-bit register to determine a negative or zero.

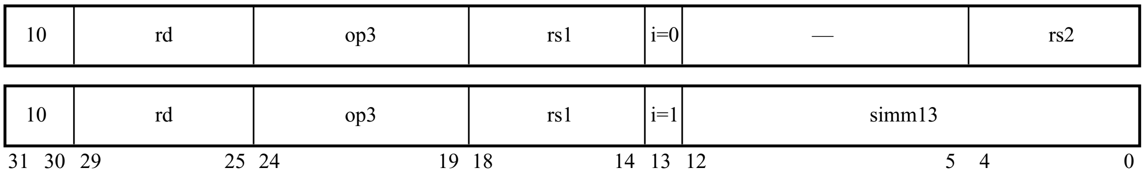
Exceptions

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:

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

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)

00	op	op2	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
31 30	29 25	24 22	21 0

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.

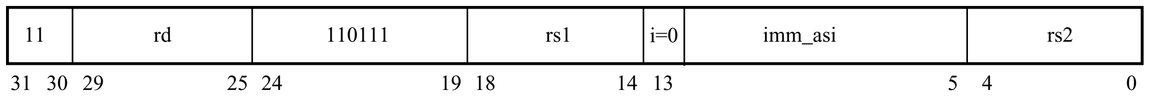
Exceptions

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
`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
`FPSUB32S`	0 0101 0111	One 32-bit Subtract

Format (3)

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

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)

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

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

Notes – 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.

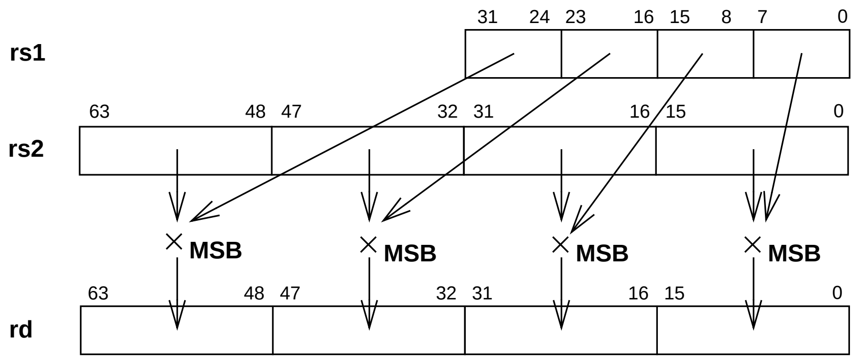


FIGURE A-5 `FMUL8x16` Operation

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.

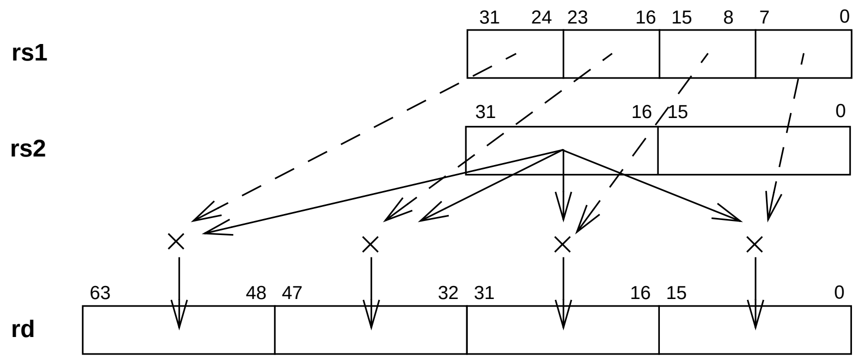


FIGURE A-6 `FMUL8x16AU` Operation

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]` are used as a proportional value. FIGURE A-7 illustrates the operation.

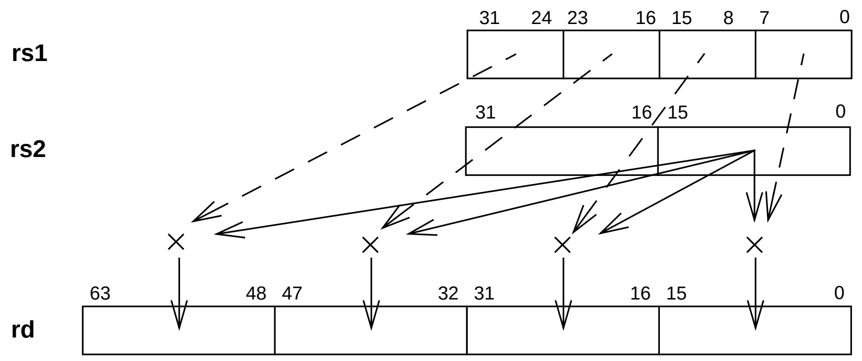


FIGURE A-7 `FMUL8x16AL` Operation

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.

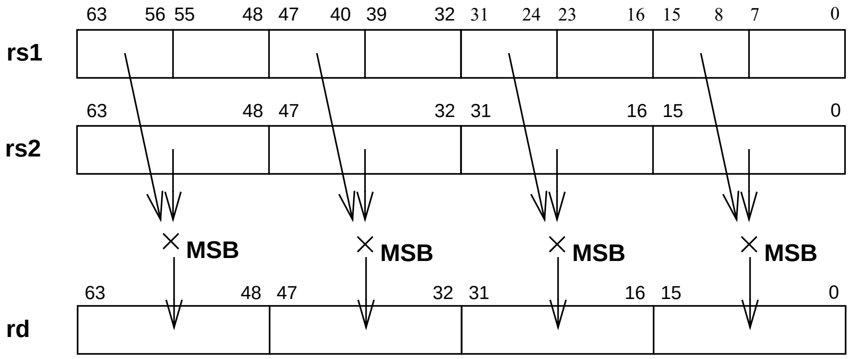


FIGURE A-8 `FMUL8SUx16` Operation

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.

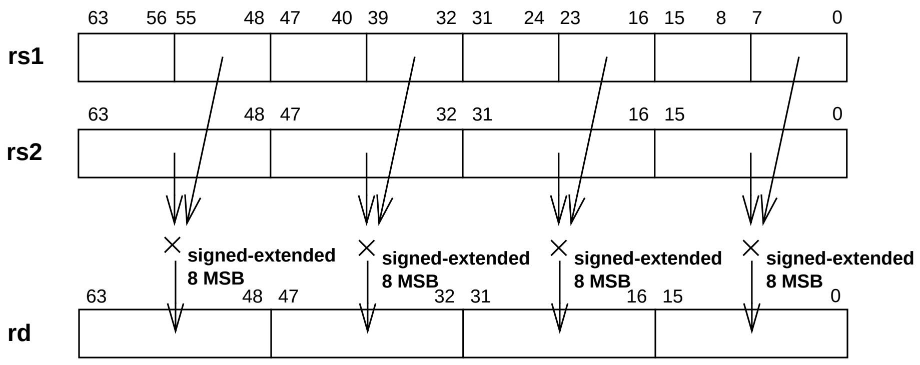


FIGURE A-9 `FMUL8LUx16` Operation

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.

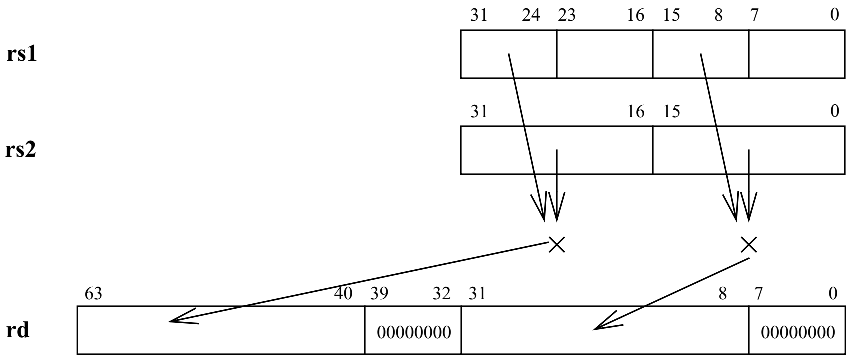


FIGURE A-10 `FMULD8SUx16` Operation

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.

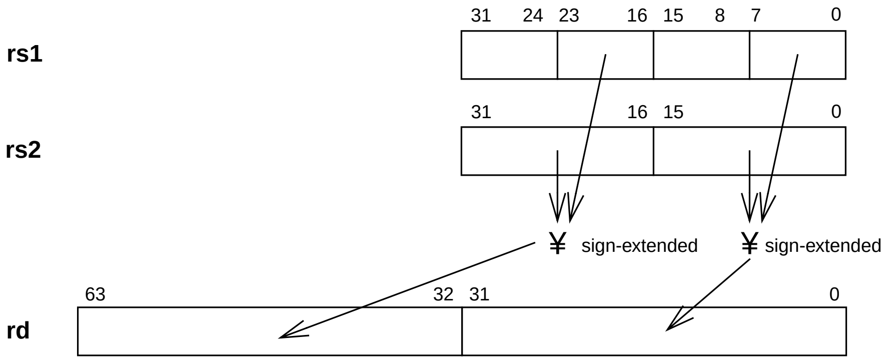


FIGURE A-11 `FMULD8ULx16` Operation

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)

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

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)

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

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)

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

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.

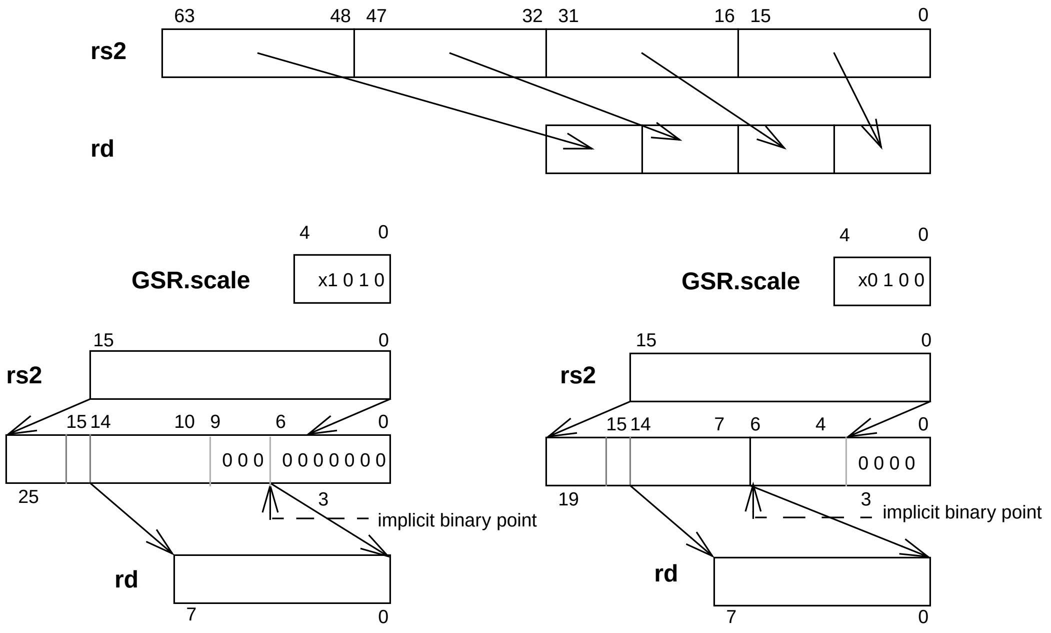


FIGURE A-12 `FPACK16` Operation

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.

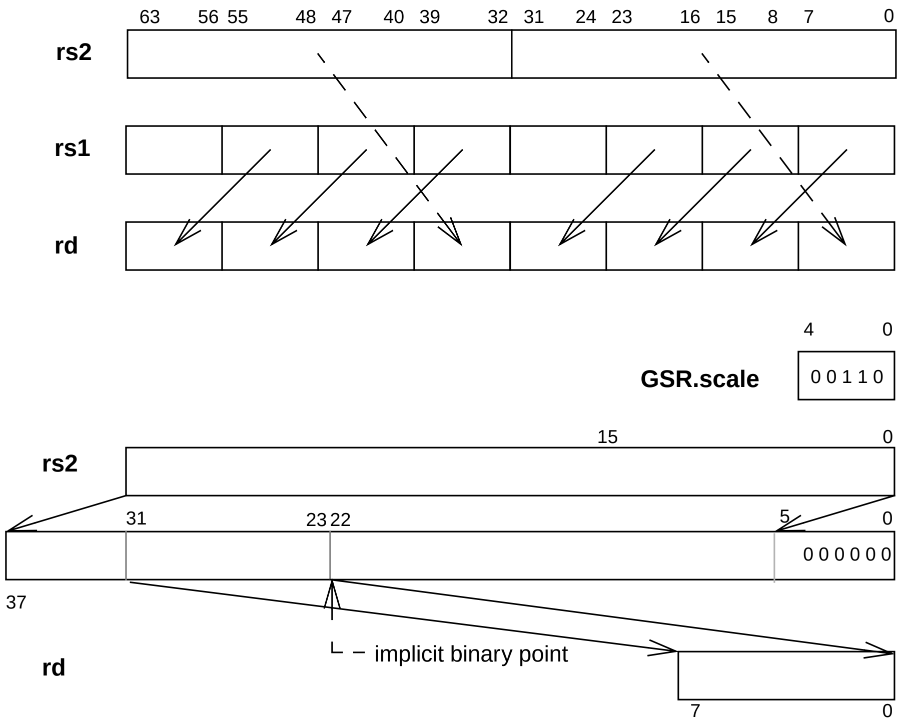


FIGURE A-13 FPACK32 Operation

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.

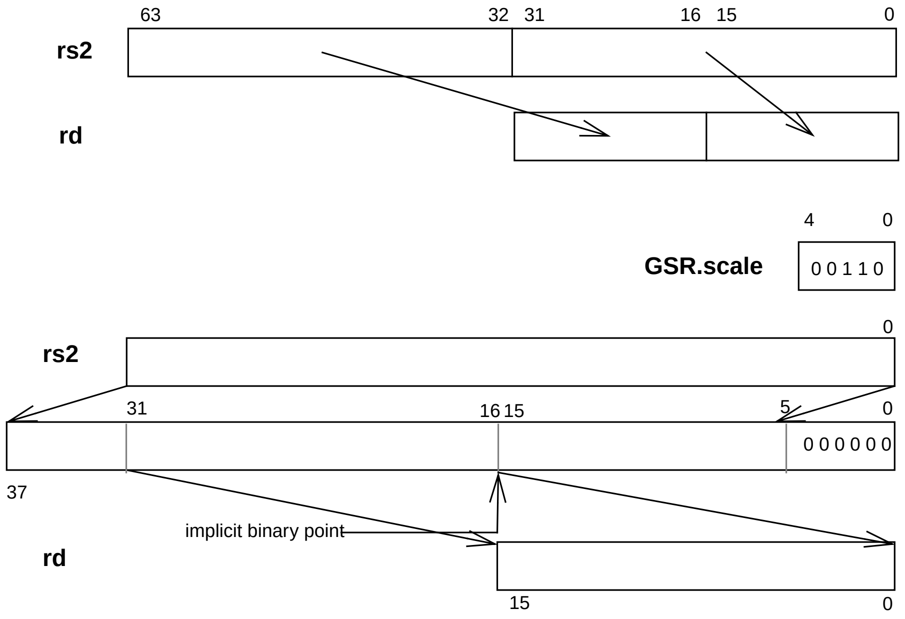


FIGURE A-14 `FPACKFIX` Operation

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.

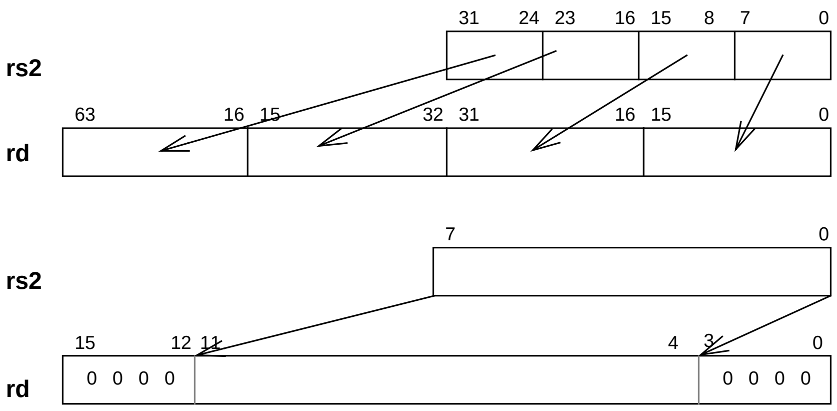


FIGURE A-15 `FEXPAND` Operation

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.

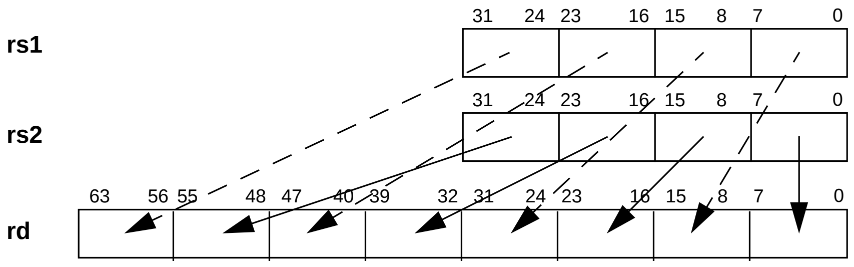


FIGURE A-16 `FPMERGE` Operation

Back-to-back `FPMERMGEs` cannot be done on adjacent cycles.

A.47 Population Count

Opcode	**op3**	**Operation**
`POPC`	10 1110	Population Count

Format (3)

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	0 0000	i=0	—	rs2
10	rd	op3	0 0000	i=1	simm13	

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 III Cu 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$^{\text{PASI}}$	11 1101	Prefetch Data from Alternate Space

Implementation Note – The PREFETCH{A} instructions are supported in the UltraSPARC III Cu processor.

Format (3) PREFETCH{A}

11	fcn	op3	rs1	i=0	PREFETCH: — PREFETCHA: imm_asi	rs2
11	fcn	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

See Chapter 16 “Prefetch” for complete details on the Prefetch Instructions and a definition of the `fcn` field.

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. Chapter 9 “Memory Models” describes non-prefetchable locations. Noncachable 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. Chapter 16 “Prefetch” further discusses the prefetch instructions.

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*, then 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.

See Chapter 8 "Address Space Identifiers" for acceptable ASI values used with the PREFETCH(A) instruction.

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 map to `fcn` 0–3 and are a new feature of the UltraSPARC III Cu processor.

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 III Cu	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

TABLE A-13 Types of Software Prefetch Instructions

`fcn` Value (hex)	Instruction Type	Prefetch into:	Instruction Strength UltraSPARC III Cu	Request Exclusive Ownership
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.

A.48.2 New Error Handling of 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 III Cu processor 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)

10	rd	op3	rs1	—
31 30	29 25	24 19	18 14	13 0

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 an 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 an illegal_instruction exception.*
See description below	10 1000	15	STBAR, MEMBAR, or *Reserved*; see description below.
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 an 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 an illegal_instruction exception.*

Format (3)

10	rd	op3	rs1	i=0	—
31 30	29 25	24 19	18 14	13	12 0

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)

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

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
```

```
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”
```

Programming Note – A routine that uses a register window may be structured as either:

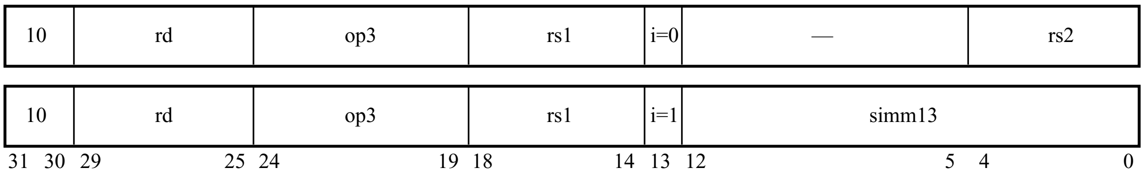
Exceptions

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 re-execute 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 $\neq$ 0, the SAVE instruction checks whether the new window needs to be cleaned. It causes a *clean_window* trap if the number of unused clean windows is zero, that is, (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.

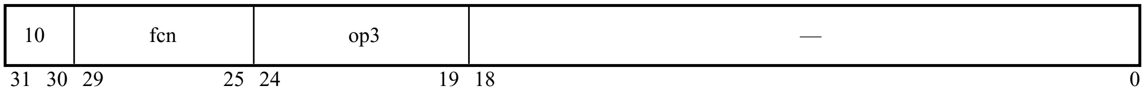
Exceptions

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)

10	—	110110	—	opf	—	mode
31 30	29 25	24 19	18 14	13 5	4 3	2 0

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)

00	rd	op2	imm22
31 30	29 25	24 22	21 0

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 14 "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 following code 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
```

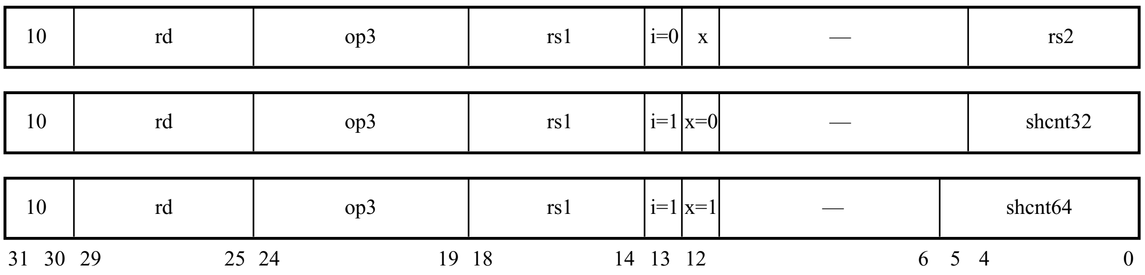
Exceptions

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`

11	rd	110011	rs1	i=0	imm_asi	rs2
11	rd	110011	rs1	i=1	simm_13	
31 30	29 25	24 19	18 14	13	5	4 0

Format (3) `STDFA`

11	rd	110111	rs1	i=0	imm_asi	rs2
11	rd	110111	rs1	i=1	simm_13	
31 30	29 25	24 19	18 14	13	5	4 0

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)

10	—	110110	—	opf	—
31 30	29 25	24 19	18 14	13 5	4 0

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)

10	0 1111	op3	0 0000	i=1	simm13
31 30	29 25	24 19	18 14	13	12 0

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

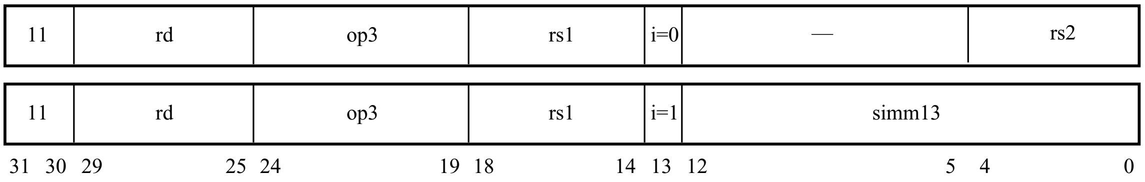
software_initiated_reset

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 (FPU) 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

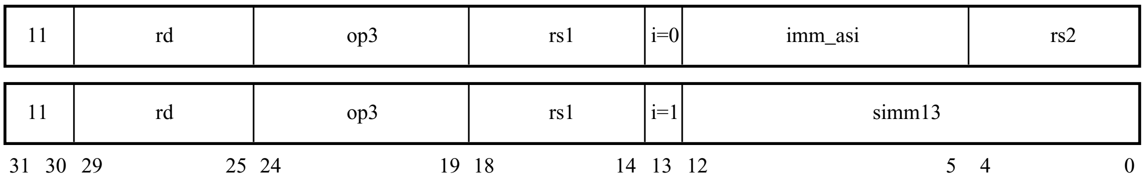
illegal_instruction (op3 = 25_{16} and *r*d = 2–31)
fp_disabled
mem_address_not_aligned
STDF_mem_address_not_aligned (STDF only)
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

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*$freg_{rd}$, *[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 Notes – 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, we recommend that compilers issue sets of single-precision stores only when they can determine that doubleword or quadword operands are *not* properly aligned.

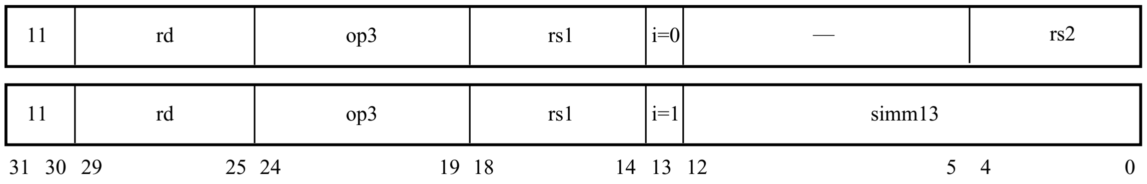
Exceptions

illegal_instruction
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.

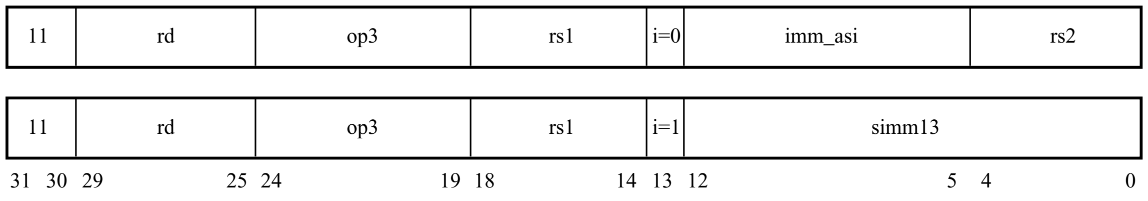
Exceptions

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 cc's
`SUBC`	00 1100	Subtract with Carry
`SUBCcc`	01 1100	Subtract with Carry and modify cc's

Format (3)

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

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 cc's

Format (3)

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

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.

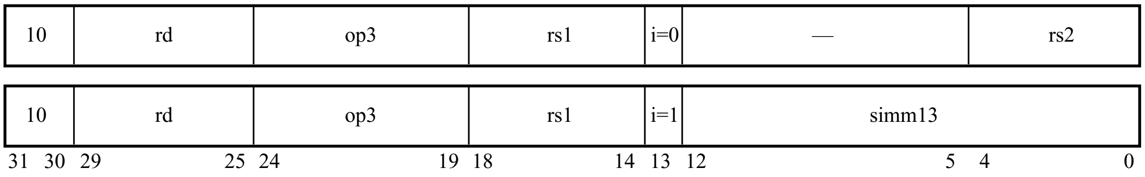
Exceptions

None

A.66 Tagged Subtract

Opcode	**op3**	**Operation**
TSUBcc	10 0001	Tagged Subtract and modify cc’s

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.

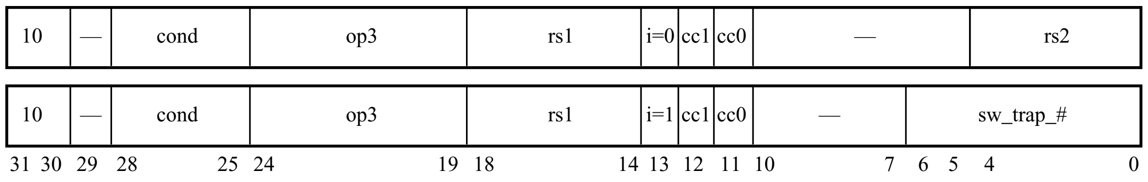
Exceptions

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]. As described in Chapter 12 "Traps and Trap Handling" 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)

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

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, take care that traps do not occur while the `TL` register is modified.

The `WRPR` instruction is a *non*-delayed write instruction. The instruction immediately following the `WRPR` observes any changes made to 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.

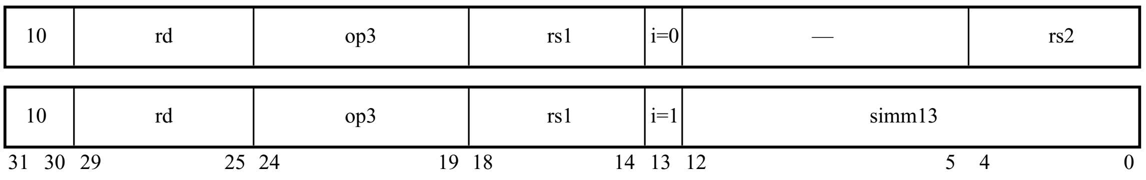
Exceptions

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, we recommend the instruction 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)

00	a	cond	110	disp22
31 30	29	28 25	24 22	21 0

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

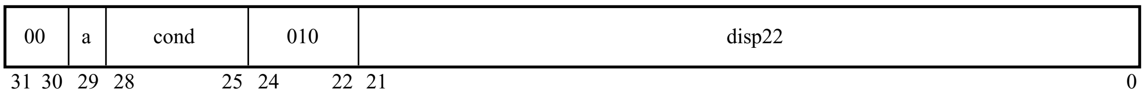
fp_disabled

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

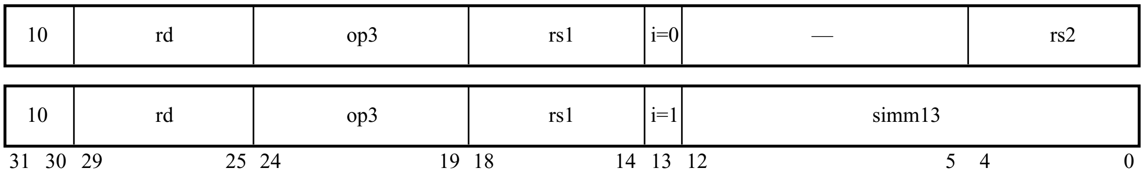
None

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 cc's
`SDIVcc`^{D}	01 1111	Signed Integer Divide and modify cc's

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 III Cu 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 ≥ 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

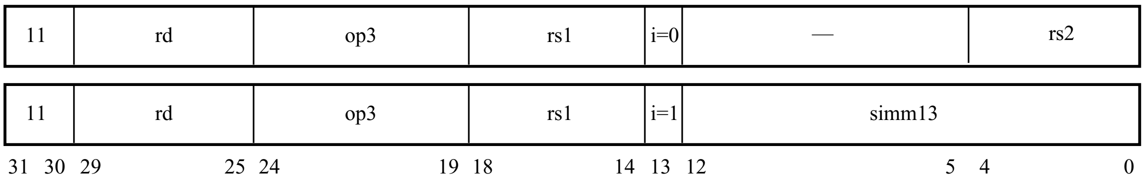
division_by_zero

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)

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

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

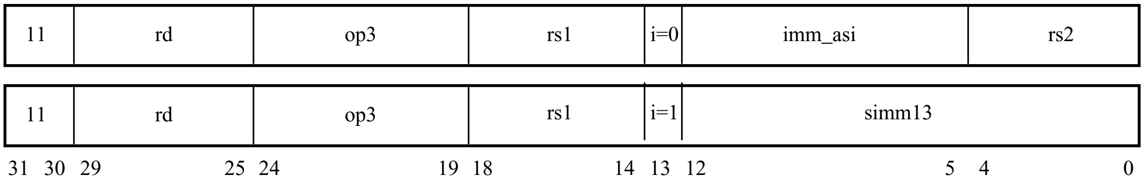
A.70.6 Load Integer Doubleword from Alternate Space

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**
`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 III Cu 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

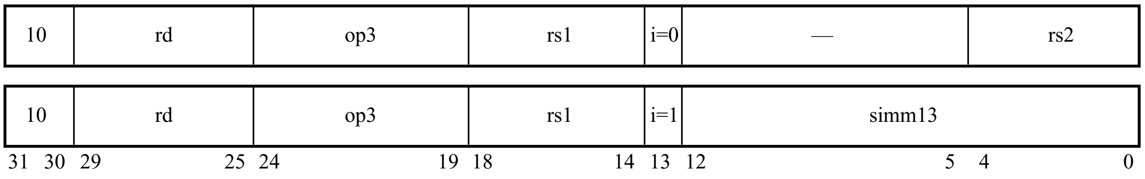
data_access_exception
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

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 cc's
`SMULcc`^{D}	01 1011	Signed Integer Multiply and modify cc's

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:** 32-bit negative (icc.N) and zero (icc.Z) condition codes are set according to the *less* significant word of the product, not according to the full 64-bit result.

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 Note – 32-bit overflow after UMUL/UMULcc is indicated by Y ≠ 0.

Thirty-two bit overflow after SMUL/SMULcc is indicated by Y ≠ (r[rd] >> 31), 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 cc's

Format (3)

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

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 31of 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. We recommend that all instructions that reference the Y register be avoided.

Opcode	**op3**	**rs1**	**Operation**
RDY^D	10 1000	0	Read Y Register

Format (3)

10	rd	op3	rs1	i=0	—
31 30	29 25	24 19	18 14	13	12 0

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)

10	0	op3	0 1111	0	—
31 30	29 25	24 19	18 14	13	12 0

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

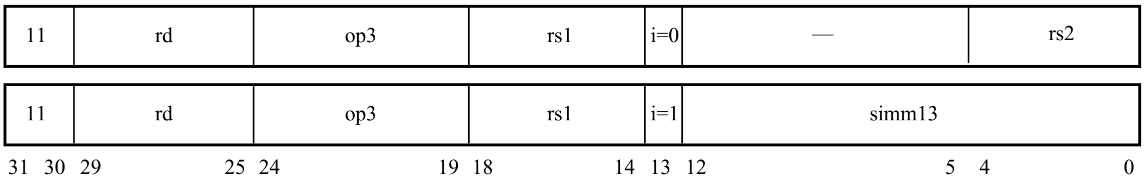
None

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)

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

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 III Cu processor, STD is implemented in hardware.

Programming Note – 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

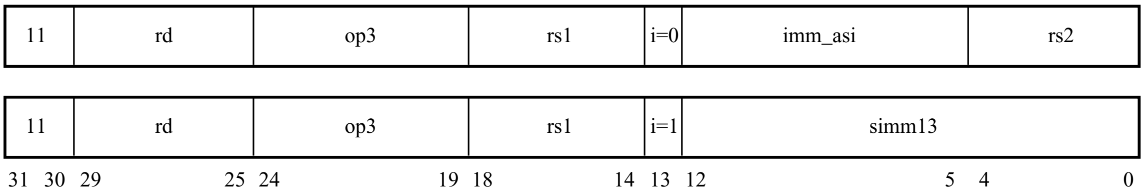
illegal_instruction (STD with odd `rd`)
mem_address_not_aligned (all except STB)
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

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 III Cu 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

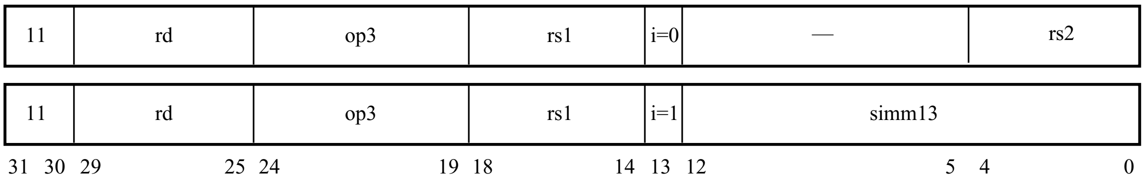
illegal_instruction (STDA with odd rd)
privileged_action
mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

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)

11	rd	op3	rs1	i=0	imm_asi	rs2

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

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

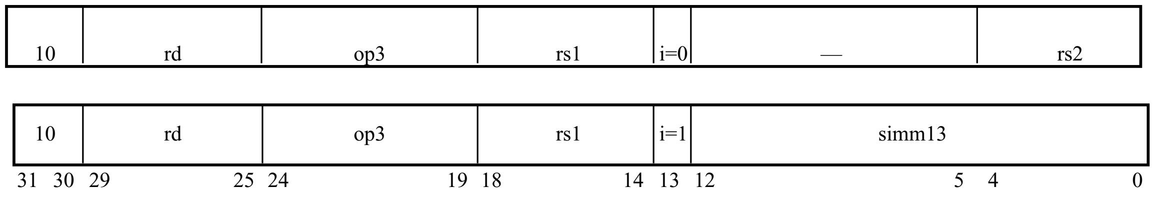
mem_address_not_aligned
privileged_action
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection
PA_watchpoint
VA_watchpoint

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 cc's, 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

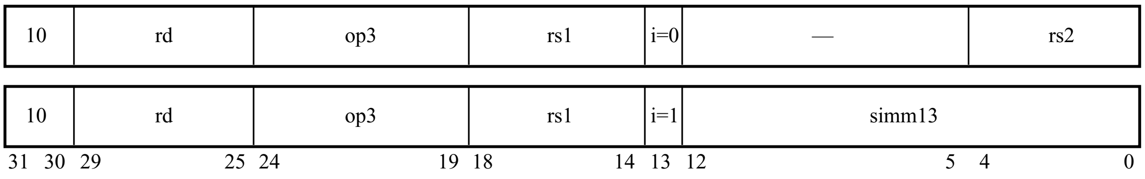
tag_overflow

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 cc's, 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)

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

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

APPENDIX **B**

Assembly Language Syntax

This appendix supports Appendix A "Instruction Definitions." Each instruction description in Appendix A includes a table that describes the suggested assembly language format for that instruction. This appendix describes the notation used in those assembly language syntax descriptions and lists some synthetic instructions provided by the SPARC V9 assemblers for the convenience of assembly language programmers.

The appendix contains these sections:

- Notation Used
- Syntax Design
- Synthetic Instructions

B.1 Notation Used

The notations defined here are also used in the assembly language syntax descriptions in Appendix A "Instruction Definitions."

Items in `typewriter font` are literals to be written exactly as they appear. Items in *italic font* are metasymbols that are to be replaced by numeric or symbolic values in actual SPARC V9 assembly language code. For example, "*imm_asi*" would be replaced by a number in the range 0 to 255 (the value of the *imm_asi* bits in the binary instruction) or by a symbol bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, reg_{rs2} is a *reg* (register name) whose binary value will be placed in the `rs2` field of the resulting instruction.

B.1.1 Register Names

See *The SPARC Architecture Manual, Version 9* regarding notational conventions for register names.

B.1.2 Special Symbol Names

Certain special symbols appear in the syntax table in `typewriter font`. They must be written exactly as they are shown, including the leading percent sign (%). See *The SPARC Architecture Manual, Version 9* regarding special symbol names.

Additional UltraSPARC III Cu symbol names and the registers or operators to which they refer are as follows:

`%ccr`	Condition Codes Register
`%clear_softint`	Soft Interrupt Register (clear selected bits)
`%dcr`	Dispatch Control Register
`%fprs`	Floating-Point Registers State Register
`%gsr`	Graphics Status Register
`%pcr`	Performance Control Register
`%pic`	Performance Instrumentation Counters
`%set_softint`	Soft Interrupt Register (set selected bits)
`%softint`	Soft Interrupt Register
`%sys_tick`	System Timer (`TICK`) Register
`%sys_tick_cmpr`	System TImer (`STICK`) Compare Register
`%tba`	Trap Base Address Register
`%tick_cmpr`	Timer (TICK) Compare Register

The following special symbol names are unary operators that perform the functions described:

`%uhi`	Extracts bits 63:42 (high 22 bits of upper word) of its operand
`%ulo` or `%hm`	Extracts bits 41:32 (low-order 10 bits of upper word) of its operand
`%hi` or `%lm`	Extracts bits 31:10 (high-order 22 bits of low-order word) of its operand
`%lo`	Extracts bits 9:0 (low-order 10 bits) of its operand

Certain predefined value names appear in the syntax table in `typewriter font`. They must be written exactly as they are shown, including the leading sharp sign (#). The value names and the values to which they refer are listed in TABLE B-1.

TABLE B-1 Value Names and Values *(1 of 2)*

Name	Value	Description
#*n*_reads	00_{16}	for PREFETCH instruction, Strong version = 14_{16}
#one_read	01_{16}	for PREFETCH instruction, Strong version = 15_{16}
#*n*_writes	02_{16}	for PREFETCH instruction, Strong version = 16_{16}
#one_write	03_{16}	for PREFETCH instruction, Strong version = 17_{16}
#page	04_{16}	for PREFETCH instruction
#Sync	40_{16}	for MEMBAR instruction cmask field
#MemIssue	20_{16}	for MEMBAR instruction cmask field
#Lookaside	10_{16}	for MEMBAR instruction cmask field
#StoreStore)	08_{16}	for MEMBAR instruction cmask field
#LoadStore	04_{16}	for MEMBAR instruction cmask field
#StoreLoad	02_{16}	for MEMBAR instruction cmask field
#LoadLoad	01_{16}	for MEMBAR instruction cmask field
#ASI_AIUP	10_{16}	ASI_AS_IF_USER_PRIMARY
#ASI_AIUS	11_{16}	ASI_AS_IF_USER_SECONDARY
#*n*_reads_strong	14_{16}	Strong version for PREFETCH instruction (00_{16})
#one_read_strong	15_{16}	Strong version for PREFETCH instruction (01_{16})
#*n*_writes_strong	16_{16}	Strong version for PREFETCH instruction (02_{16})
#one_write_strong	17_{16}	Strong version for PREFETCH instruction (03_{16})
#ASI_AIUP_L	18_{16}	ASI_AS_IF_USER_PRIMARY_LITTLE
#ASI_AIUS_L	19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE
#ASI_PHYS_USE_EC_L	$1C_{16}$	ASI_PHYS_USE_EC_LITTLE
#ASI_PHYS_BYPASS_EC_WITH_EBIT_L	$1D_{16}$	ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE
#ASI_NUCLEUS_QUAD_LDD_L	$2C_{16}$	ASI_NUCLEUS_QUAD_LDD_LITTLE
#ASI_MONDO_SEND_CTRL	48_{16}	ASI_INTR_DISPATCH_STATUS
#ASI_MONDO_RECEIVE_CTRL	49_{16}	ASI_INTR_RECEIVE
#ASI_AFSR	$4C_{16}$	ASI_ASYNC_FAULT_STATUS
#ASI_AFAR	$4D_{16}$	ASI_ASYNC_FAULT_ADDR
#ASI_BLK_AIUP	70_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY
#ASI_BLK_AIUS	71_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY
#ASI_BLK_AIUPL	78_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE
#ASI_BLK_AIUSL	79_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE
#ASI_P	80_{16}	ASI_PRIMARY
#ASI_S	81_{16}	ASI_SECONDARY
#ASI_PNF	82_{16}	ASI_PRIMARY_NOFAULT

TABLE B-1 Value Names and Values *(2 of 2)*

Name	Value	Description
#ASI_SNF	83_{16}	ASI_SECONDARY_NOFAULT
#ASI_P_L	88_{16}	ASI_PRIMARY_LITTLE
#ASI_S_L	89_{16}	ASI_SECONDARY_LITTLE
#ASI_PNF_L	$8A_{16}$	ASI_PRIMARY_NOFAULT_LITTLE
#ASI_SNF_L	$8B_{16}$	ASI_SECONDARY_NOFAULT_LITTLE
#ASI_PST8_P	$C0_{16}$	ASI_PST8_PRIMARY
#ASI_PST8_S	$C1_{16}$	ASI_PST8_SECONDARY
#ASI_PST16_P	$C2_{16}$	ASI_PST16_PRIMARY
#ASI_PST16_S	$C3_{16}$	ASI_PST16_SECONDARY
#ASI_PST32_P	$C4_{16}$	ASI_PST32_PRIMARY
#ASI_PST32_S	$C5_{16}$	ASI_PST32_SECONDARY
#ASI_PST8_PL	$C8_{16}$	ASI_PST8_PRIMARY_LITTLE
#ASI_PST8_SL	$C9_{16}$	ASI_PST8_SECONDARY_LITTLE
#ASI_PST16_PL	CA_{6}	ASI_PST16_PRIMARY_LITTLE
#ASI_PST16_SL	CB_{16}	ASI_PST16_SECONDARY_LITTLE
#ASI_PST32_PL	CC_{16}	ASI_PST32_PRIMARY_LITTLE
#ASI_PST32_SL	CD_{16}	ASI_PST32_SECONDARY_LITTLE
#ASI_FL8_P	$D0_{16}$	ASI_FL8_PRIMARY
#ASI_FL8_S	$D1_{16}$	ASI_FL8_SECONDARY
#ASI_FL16_P	$D2_{16}$	ASI_FL16_PRIMARY
#ASI_FL16_S	$D3_{16}$	ASI_FL16_SECONDARY
#ASI_FL8_PL	$D8_{16}$	ASI_FL8_PRIMARY_LITTLE
#ASI_FL8_SL	$D9_{16}$	ASI_FL8_SECONDARY_LITTLE
#ASI_FL16_PL	DA_{16}	ASI_FL16_PRIMARY_LITTLE
#ASI_FL16_SL	DB_{16}	ASI_FL16_SECONDARY_LITTLE
#ASI_BLK_COMMIT_P	$E0_{16}$	ASI_BLOCK_COMMIT_PRIMARY
#ASI_BLK_COMMIT_S	$E1_{16}$	ASI_BLOCK_COMMIT_SECONDARY
#ASI_BLK_P	$F0_{16}$	ASI_BLOCK_PRIMARY
#ASI_BLK_S	$F1_{16}$	ASI_BLOCK_SECONDARY
#ASI_BLK_PL	$F8_{16}$	ASI_BLOCK_PRIMARY_LITTLE
#ASI_BLK_SL	$F9_{16}$	ASI_BLOCK_SECONDARY_LITTLE

The full names of the ASIs, listed in the Description column of TABLE B-1 can also be defined.

B.1.3 Values

See *The SPARC Architecture Manual, Version 9* regarding notational conventions for values.

B.1.4 Labels

See *The SPARC Architecture Manual, Version 9* regarding notational conventions for labels.

B.1.5 Other Operand Syntax

See *The SPARC Architecture Manual, Version 9* regarding notational conventions for general operand syntax. Additional operand syntax is listed below.

membar_mask

Operand syntax is as follows:

const7 — A constant that can be represented in 7 bits. Typically, this is an expression involving the logical OR of some combination of `#Lookaside`, `#MemIssue`, `#Sync`, `#StoreStore`, `#LoadStore`, `#StoreLoad`, and `#LoadLoad`.

prefetch_fcn (Prefetch Function)

Operand syntax is summarized in the TABLE B-2.

TABLE B-2 Prefetch Function Values

Prefetch Function Value (Decimal)	Operand
0	#n_reads (Strong version = 20)
1	#one_read (Strong version = 21)
2	#n_writes (Strong version = 22)
3	#one_write (Strong version = 23)
4	#page
5 – 15	Reserved and Traps
16	P-cache Invalidate
17-19	NOP
20	#n_reads_strong (Strong version of 0)
21	#one_read_strong (Strong version of 1)

TABLE B-2 Prefetch Function Values *(Continued)*

Prefetch Function Value (Decimal)	Operand
22	#n_writes_strong (Strong version of 2)
23	#one_write_strong (Strong version of 3)
24 – 31	NOP

B.1.6 Comments

See *The SPARC Architecture Manual, Version 9* regarding notational conventions for comments.

B.2 Syntax Design

See *The SPARC Architecture Manual, Version 9* regarding syntax design.

B.3 Synthetic Instructions

See *The SPARC Architecture Manual, Version 9* regarding synthetic instructions.

Index

Revision History

Date	Comments	Version
May 2002	Initial release	1.0
February 2003	Additional chapters included	2.0
April 2003	Font modifications	2.1
May 2003	Modification to IEEE 754 and Performance Instrumentation chapters	2.2
December 2003	Additional clarifications on pages 11-290, 11-291, and 18-441. Revised figures 6-12 and 6-29.	2.2
January 2004	Updated table 6-9 and added note on page 6-102.	2.2.1