SPARC64-III User's Guide

HAL Computer Systems, Inc.

Campbell, California

May 1998

COMPUTER SYSTEMS

A Fujitsu Company

Contents

1   Overview

The *SPARC64-III User's Guide* describes Revision 3 of HAL Computer Systems' 64-bit SPARC-V9 compliant processor module. It documents the instruction set, register model, data types, instruction opcodes, trap model, and virtual address translation algorithms.

HAL Computer Systems currently supports three CPU architectures:

1. The first-generation CPU-1 (also called SPARC64) is used in the HALstation models 330 and 350.

2. The second generation CPU-2 is used in the HALstation models 375 and 385.

3. The third generation CPU-3, intended for use by Fujitsu and HAL. This Guide documents the architecture of CPU-3, or SPARC64-III.

In this book, any references to "the CPU" refers to the third generation CPU-3, or SPARC64-III.

For the purposes of this document 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 include details of the implementation that are not visible or easily observable by software.

1.1   Notes About This Book

1.1.1   Audience

The audience for this guide includes developers of SPARC64-III system software (simulators, compilers, debuggers, and operating systems, for example) and SPARC64-III assembly language programmers.

1.1.2   Where to Start

1.1.2.1   Background

The *SPARC64-III User's Guide* was derived directly from the source text of *The SPARC Architecture Manual-Version 9*, which is abbreviated throughout this guide as *V9*. We have deleted some of the more theoretical material contained in *V9*, but our goal has been to create a book that can stand alone. For some implementors, however, this theoretical infor-

mation is very important. In particular, 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 should obtain and use *V9*. Any reader of this guide could profit from using *V9* as a companion text.

Whenever a chapter, subsection, or appendix in this guide matches the parallel number or letter in *V9*, the information contained herein is directly related. In some instances, these parallel sections contain HAL-specific information; in other instances, the original information from *V9* has been duplicated so that this guide can stand alone. Because we have added and deleted a significant number of tables and figures, the table and figure numbers in this guide are not parallel with the numbers in *V9*. Where this guide's tables and figures are identical to or based on those in *V9*, we have included the *V9* number within the table and/or figure title. We also have included a list of tables and a list of figures in this guide; these lists also contain the *V9* cross references, which helps to relate the material in this guide back to the original.

One new chapter has been added: Chapter 9, "Guidelines for Instruction Scheduling". Finally, some entire appendixes found in *V9* have been eliminated from the *SPARC64-III User's Guide*. In these cases, and whenever we refer to information contained only in *V9*, we place an icon in the margin, as at left.

1.1.2.2 Navigating the *SPARC64-III User's Guide*

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

If you are familiar with SPARC-V8 but not SPARC-V9, you should review the list of changes in Appendix K, "Changes From SPARC-V8 to SPARC-V9," in *V9*. For additional details of architectural changes, review the following chapters:

■ Chapter 4, "Data Formats," for a description of the supported data formats

■ Chapter 5, "Registers," for a description of the register set

■ Chapter 6, "Instructions," for a description of the new instructions

■ Chapter 7, "Traps," for a description of the trap model

■ Chapter 8, "Memory Models," both here and in *V9*, for a description of the memory models

■ Appendix A, "Instruction Definitions," for descriptions of the instructions

Finally, if you are familiar with the SPARC-V9 architecture and wish to familiarize yourself with the SPARC64-III-specific implementation, study the following:

■ Chapter 2, "Definitions," the unnumbered subsection entitled "SPARC64-III Implementation-Specific Terms" beginning on page 26

■ Chapter 9, "Guidelines for Instruction Scheduling," for detailed information about low-level instruction scheduling for compiler optimizer writers

■   Appendix A, "Instruction Definitions," for descriptions of the SPARC64-III-specific instruction extensions

■   Appendix C, "SPARC-V9 Implementation Dependencies," for descriptions of SPARC64_III's resolution of all SPARC-V9 implementation dependencies

■   Appendix E, "Opcode Maps," to see how the SPARC64-III-specific opcode extensions fit into the SPARC-V9 opcode maps

■   Appendix F, "MMU Architecture," to see the requirements that SPARC-V9 systems impose upon an MMU, and how SPARC64-III fulfills those requirements

■   Appendix G, "Assembly Language Syntax," to see how SPARC64-III has extended the SPARC-V9 assembly language syntax; in particular, SPARC64-III-specific synthetic instructions are documented in this appendix

1.1.3 Contents Compared with *V9*

Table 1 on page 15 describes the contents of every chapter and appendix, comparing *The SPARC Architecture Manual-Version 9* with *SPARC64-III User's Guide*.

1.1.4 Editorial Conventions

1.1.4.1 Fonts and Notational Conventions

Fonts are used as follows:

■   *Italic* font is used for register names, instruction fields, and read-only register fields. For example: "The *rs1* field contains...."

■   `Typewriter` font is used for literals and for software examples.

■   **Bold** font is used for emphasis and the first time a word is defined. For example: "A **precise trap** is induced...."

■   UPPER-CASE items are acronyms, instruction names, or writable register fields. Some common acronyms appear in the glossary in Chapter 2. **Note:** Names of some instructions contain both upper- and lower-case letters.

■   *Italic sans serif* font is used for exception and trap names. For example, "The *privileged_action* exception...."

■   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. For example: "This is true whenever the integer_condition_code field...."

■   Reduced-size font is used in informational notes. See 1.1.4.4, "Informational Notes."

■   The marginal icon at left indicates that more information is available in *The SPARC Architecture Manual-Version 9* (*V9*), which is available from SPARC International and at many technical bookstores.

The following notational conventions are used:

■ Square brackets '[ ]' indicate a numbered register in a register file. For example: "r[0] contains...."

■ Angle brackets '< >' indicate a bit number or colon-separated range of bit numbers within a field. For example: "Bits FSR<29:28> and FSR<12> are...."

■ Curly braces '{ }' are used to indicate textual substitution. For example, the string "ASI_PRIMARY{_LITTLE}" expands to "ASI_PRIMARY" and "ASI_PRIMARY_ LITTLE".

■ 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

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

results in X = 0, Y = 1, and Z = 1.

Table 1: Contents of *V9* vs. *SPARC64-III User's Guide*

Ref	Title	The SPARC Architecture Manual-Version 9	SPARC64-III User's Guide
1	Overview	Describes the background, design philosophy, and high-level features of the architecture. Also defines SPARC-V9 compliance levels and implementation dependencies.	Describes differences between *The SPARC Architecture Manual-Version 9* and *SPARC64-III User's Guide*. Deletes information about SPARC-V9 compliance.
2	Definitions	Defines some of the terms used in the specification.	Adds SPARC64-III-specific definitions.
3	Architectural Overview	Contains an overview of the architecture: its organization, instruction set, and trap model.	Particularizes the entire architectural discussion to the SPARC64-III. This chapter contains the primary description of the SPARC64-III CPU architecture.
4	Data Formats	Describes the supported data types.	Notes formats not supported by the SPARC64-III hardware.
5	Registers	Describes the register set.	Particularizes the register set for the SPARC64-III.
6	Instructions	Describes the instruction set.	Adds descriptions of the SPARC64-III-specific instructions.
7	Traps	Describes the trap model.	Adds SPARC64-III-specific information.
8	Memory Models	Describes the memory models.	Contains only information about SPARC64-III-specific memory models.
9	Guidelines for Instruction Scheduling	*- Not present -*	Contains detailed, low-level information about instruction scheduling, for compiler writers and anyone needing to write highly optimized code.
A	Instruction Definitions	Contains definitions of all SPARC-V9 instructions, including tables showing the recommended assembly language syntax for each instruction.	Includes information about SPARC64-III-specific instructions.
B	IEEE Std 754-1985 Requirements for SPARC-V9	Contains information about the SPARC-V9 implementation of the IEEE 754 floating-point standard.	Duplicates Appendix B for completeness.
C	SPARC-V9 Implementation Dependencies	Contains information about features that may differ among conforming implementations. **Note**: Implementation dependency definitions and references are scattered throughout this document.	Describes how the SPARC64-III resolves each implementation-dependency. **Note**: Implementation dependency definitions and references have been *deleted* from the other sections of this document.

Table 1: Contents of *V9* vs. *SPARC64-III User's Guide* (Continued)

Ref	Title	The SPARC Architecture Manual-Version 9	SPARC64-III User's Guide
D	Formal Specification of the Memory Models	Contains formal descriptions of the memory models.	- Not present -
E	Opcode Maps	Contains tables showing the encoding of all opcodes.	Includes SPARC64-III-specific opcodes.
F	MMU Architecture	Describes the requirements that SPARC-V9 imposes on memory management units.	Includes SPARC64-III-specific information.
G	Assembly Language Syntax	Defines the syntactic conventions used in the appendixes for the suggested SPARC-V9 assembly language. It also lists synthetic instructions that may be supported by SPARC-V9 assemblers for the convenience of assembly language programmers.	Lists SPARC64-III-specific synthetic instructions.
H	Software Considerations	Contains general SPARC-V9 software considerations.	- Not present -
I	Extending the SPARC-V9 Architecture	Contains information on how an implementation can extend the instruction set or register set.	- Not present -
J	Programming With the Memory Models	Contains information on programming with the SPARC-V9 memory models.	- Not present -
K	Changes From SPARC-V8 to SPARC-V9	Describes the differences between the SPARC-V8 and SPARC-V9 architectures.	- Not present -
L	ASI Assignments	- Not present -	Defines the SPARC64-III ASI assignments.
M	Cache Organization	- Not present -	Describes the SPARC64-III cache organization.
N	Interrupt Handling	- Not present -	Describes SPARC64-III interrupt handling.
O	Reset, RED_state, and Error_state	- Not present -	Describes the SPARC64-III implementation of Reset, RED-state, and Error_state.
P	Error Handling	- Not present -	Describes SPARC64-III error handling.
Q	Performance Monitoring	- Not present -	Defines the SPARC64-III performance monitoring extensions.
R	UPA Programmer's Monitor	- Not present -	Defines the SPARC64-III UPA Programmer's Monitor extensions.

1.1.4.2 Implementation Dependencies

The implementors of SPARC-V9-compliant processors are allowed to resolve some aspects of the architecture in machine-dependent ways. Each possible implementation dependency is indicated in *V9* by the notation "**IMPL. DEP. #nn**: Some descriptive text." The number **nn** is used to enumerate the dependencies in Appendix C, "SPARC-V9 Implementation Dependencies." References to SPARC-V9 implementation dependencies are indicated in *V9* by the notation "(impl. dep. #nn)." In *SPARC64-III User's Guide*, we have replaced all definitions of and references to SPARC-V9 implementation dependencies with descriptions of the SPARC64-III implementation. Appendix C in this document describes the HAL-specific implementation decisions in detail. Refer to *V9* for more information about implementation dependencies.

1.1.4.3 Notation for Numbers

Numbers throughout this guide are decimal (base-10) unless otherwise indicated. Numbers in other bases are followed by a numeric subscript indicating their base (for example, 1001_2, $FFFF\ 0000_{16}$). Long binary and hex numbers within the text have spaces inserted every four characters to improve readability. Within C or assembly language examples, numbers may be preceded by "0x" to indicate base-16 (hexadecimal) notation (for example, `0xffff0000`).

1.1.4.4 Informational Notes

This guide provides several different types of information in notes; the information appears in a reduced-size font. The following examples illustrate the various note types:

Programming Note:
> Programming notes contain incidental information about programming HAL's SPARC64-III implementation.

Implementation Note:
> Implementation notes contain information that is specific to HAL's SPARC64-III implementation. Such information may not pertain to other SPARC-V9 implementations.

1.2 SPARC64-III Architecture

1.2.1 Features

HAL's SPARC64-III includes the following principal features:

■ A linear 64-bit address space with 64-bit addressing.

■ 32-bit wide instructions, which are aligned on 32-bit boundaries in memory. Only load and store instructions access memory and perform I/O.

■ Few addressing modes: A memory address is given as either "register + register" or "register + immediate."

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

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

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

■ Fast trap handlers: Traps are vectored through a table.

■ Multiprocessor synchronization instructions: One instruction performs an atomic read-then-set-memory operation; another performs an atomic exchange-register-with-memory operation; another compares the contents of a register with a value in memory and exchanges memory with the contents of another register if the comparison was equal (compare and swap); two others are used to synchronize the order of shared memory operations as observed by processors.

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

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

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

■ Relaxed memory order (RMO) model: This weak memory model allows the hardware to schedule memory accesses in almost any order, as long as the program computes the correct result.

1.2.2 Attributes

SPARC-V9 is a CPU **instruction set architecture** (ISA) derived from SPARC-V8; both architectures come from a reduced instruction set computer (RISC) lineage. As architectures, SPARC-V9 and SPARC-V8 allow for a spectrum of chip and system **implementations** at a variety of price/performance points for a range of applications, including scientific/engineering, programming, real-time, and commercial.

1.2.2.1 Design Goals

The CPU is designed to be a target for optimizing compilers and high-performance hardware implementations. The CPU provides exceptionally high execution rates and short time-to-market development schedules.

1.2.2.2 Register Windows

The CPU is derived from SPARC, which was formulated at Sun Microsystems in 1985. SPARC is based on the RISC I and II designs engineered at the University of California at Berkeley from 1980 through 1982. SPARC's "register window" architecture, pioneered in the UC Berkeley designs, allows for straightforward, high-performance compilers and a reduction in memory load/store instructions.

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

One major difference between SPARC64-III and the Berkeley RISC I and II is that SPARC64-III provides greater flexibility to a compiler in its assignment of registers to program variables. SPARC64-III is more flexible because register window management is not tied to procedure call and return instructions, as it is on the Berkeley machines. Instead, separate instructions (SAVE and RESTORE) provide register window management. The management of register windows by privileged software is very different too, as discussed in Appendix H, "Software Considerations" in *V9*.

1.2.3 System Components

The SPARC-V9 architecture allows for a spectrum of I/O, memory-management unit (MMU), and cache system subarchitectures.

1.2.3.1 SPARC64-III MMU

The SPARC-V9 ISA does not mandate a single MMU design for all system implementations. Rather, designers are free to use the MMU that is most appropriate for their application, or no MMU at all, if they wish. The SPARC64-III MMU implementation and virtual address translation are described in Appendix F, "MMU Architecture".

1.2.3.2 Privileged Software

SPARC-V9 does not assume that all implementations must execute identical privileged software. Thus, certain traits of the SPARC64-III that are visible to privileged software have been tailored to the requirements of the system.

1.2.4 Binary Compatibility

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

Although different SPARC-V9 systems may execute nonprivileged programs at different rates, they will generate the same results, as long as they are run under the same memory model. See Chapter 8, "Memory Models," for more information.

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

1.2.5 Architectural Definition

The SPARC-V9 architecture is defined by the chapters and normative appendixes of *The SPARC Architecture Manual-Version 9*. A correct implementation of the architecture interprets a program strictly according to the rules and algorithms specified in the chapters and normative appendixes.

This guide defines a conforming implementation of the SPARC-V9 architecture named the SPARC64-III.

1.2.6 SPARC-V9 Compliance

SPARC International is responsible for certifying that implementations comply with the SPARC-V9 Architecture. Two levels of compliance are distinguished: Level 1 and Level 2. The SPARC64-III is Level-2-compliant. See *V9* for a definition of the SPARC-V9 compliance levels.

Appendix C, "SPARC-V9 Implementation Dependencies," describes the manner in which the SPARC64-III has resolved all implementation dependencies.

2 Definitions

The following subsections define some of the most important words and acronyms used in this guide.

SPARC-V9 Terms

2.1 **address space identifier (ASI):** 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**.

2.2 **application program:** A program executed with the processor in **nonprivileged mode**. Note: Statements made in this guide 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).

2.3 **ASI:** Abbreviation for **address space identifier**.

2.4 **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.

2.5 **byte:** Eight consecutive bits of data.

2.6 **clean window:** A register window in which all of the registers contain either zero, a valid address from the current address space, or valid data from the current address space.

2.7 **completed:** A memory transaction is said to be 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.

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

2.9 **dispatch:** Issue a fetched instruction to one or more functional units for execution.

2.10   **doublet:** Two bytes (16 bits) of data.

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

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

2.13   **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.

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

2.15   *fccn:* One of the floating-point condition code fields: *fcc0*, *fcc1*, *fcc2*, or *fcc3*.

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

2.17   **floating-point IEEE-754 exception:** A floating-point exception, as specified by IEEE Std 754-1985. Listed within this guide as *IEEE_754_exception*.

2.18   **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**.

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

2.20   **floating-point unit:** A processing unit that contains the floating-point registers and performs floating-point operations, as defined by this guide.

2.21   **FPU:** Abbreviation for **floating-point unit**.

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

2.23   **hexlet:** Sixteen bytes (128 bits) of data.

2.24   **implementation:** Hardware and/or software that conforms to all of the specifications of an **instruction set architecture (ISA)**.

2.25   **implementation-dependent:** An aspect of the architecture that may legitimately vary among implementations. In many cases, the permitted range of variation is specified in the standard. When a range is specified, compliant implementations shall not deviate from that range.

2.26   **implicit ASI:** The **address space identifier** 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.

2.27   **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**.

2.28   **initiated:** *Synonym*: **issued**.

2.29   **instruction field:** A bit field within an instruction word.

2.30   **instruction set architecture (ISA):** An ISA defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. An ISA does not define clock cycle times, cycles per instruction, data paths, and other implementation-dependent characteristics. This guide defines the SPARC-V9 ISA and also contains details about HAL's implementation of the ISA.

2.31   **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 guide.

2.32   **interrupt request:** A request for service presented to the processor by an external device.

2.33   **ISA:** Abbreviation for **instruction set architecture**.

2.34   **issued:** In reference to memory transaction, a load, store, or atomic load-store is said to be 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**.

2.35   **IU:** Abbreviation for **integer unit**.

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

2.37   **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.

2.38   **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.

2.39   **must:** *Synonym*: **shall**.

2.40   **next program counter (nPC):** A register that contains the address of the instruction to be executed next, if a trap does not occur.

2.41 **nonfaulting load:** A load operation that either completes correctly (in the absence of any faults) or returns a value (nominally zero) if a fault occurs. *See* **speculative load**.

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

2.43 **nonprivileged mode:** The processor mode when PSTATE.PRIV = 0. *See also*: **nonprivileged**.

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

2.45 **NWINDOWS:** The number of register windows present in an implementation.

2.46 **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.

2.47 **opcode:** A bit pattern that identifies a particular instruction.

2.48 **prefetchable:** An attribute of a memory location that indicates to an MMU that PREFETCH operations to that location may be applied. Normal memory is prefetchable. Nonprefetchable locations include those that, when read, change state or cause external events to occur. *See also*: **side effect**.

2.49 **privileged:** An adjective that describes (1) the state of the processor when PSTATE.PRIV = 1, that is, **privileged mode**; (2) processor state information that is accessible to software only 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**.

2.50 **privileged mode:** The processor mode when PSTATE.PRIV = 1. *See also*: **nonprivileged**.

2.51 **processor:** The combination of the **integer unit** and the **floating-point unit**.

2.52 **program counter (PC):** A register that contains the address of the instruction currently being executed by the **IU**.

2.53 **quadlet:** Four bytes (32 bits) of data.

2.54 **quadword:** Aligned **hexlet**. Note: The definition of this term is architecture-dependent and may be different from that used in other processor architectures.

2.55   *r* **register:** An integer register. Also called a general purpose register or working register.

2.56   **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.

2.57   **reserved:** Used to describe 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-compliant 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-compliant processors shall decode and trap on these reserved combinations. **Reserved register fields** should be written only to zero by software; they should read as zero in hardware. Software intended to run on future versions of SPARC-V9 should not assume that these field will read as zero or any other particular value. Throughout this guide, figures and tables illustrating registers and instruction encodings indicate reserved fields and combinations with an em dash '—'.

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

2.59   **restricted:** An adjective used to describe an **address space identifier** (ASI) that can be accessed only while the processor is operating in **privileged mode**.

2.60   *rs1, rs2, rd*: The integer register operands of an instruction, where *rs1* and *rs2* are the source registers and *rd* is the destination register.

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

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

2.63   **side effect:** A secondary effect induced by an operation in addition to its primary effect. For example, access to an I/O location may cause a register value in an I/O device to change state or initiate an I/O operation. A memory location is deemed to have side effects if additional actions beyond the reading or writing of data may occur when a memory operation on that location is allowed to succeed. *See also*: **prefetchable**.

2.64   **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. Contrast with **nonfaulting load**, which is an explicit load

that always completes, even in the presence of faults. **Note:** Some authors confuse speculative loads with nonfaulting loads.

2.65   **supervisor software:** Software that executes when the processor is in **privileged mode**.

2.66   **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 Trap Base Address (TBA) register.

2.67   **unassigned:** A value (for example, an **address space identifier**) the semantics of which are not architecturally mandated and may be determined independently by each implementation within any guidelines given.

2.68   **undefined:** An aspect of the architecture that has deliberately been left unspecified. Software should have no expectation of, nor make any assumptions about, an undefined feature or behavior. Use of such a feature may deliver random results, may or may not cause a trap, may vary among implementations, and may 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 put the processor into an unrecoverable state.

2.69   **unrestricted:** An adjective used to describe an **address space identifier** that may be used regardless of the processor mode, that is, regardless of the value of PSTATE.PRIV.

2.70   **user application program:** *Synonym*: **application program**.

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

SPARC64-III Implementation-Specific Terms

The following terms define concepts unique to HAL's implementation.

2.72   **checkpoint:** SPARC64-III checkpoints the CPU at certain intervals to ensure that it can recover from mispredicted branches, exceptions, interrupts, and so on. The checkpoint can be used to return the machine to a known correct state.

2.73   **committed:** An instruction can be committed only when it has completed without error and *all* prior instructions have completed without error *and have been committed.* When an instruction is committed the state of the machine is permanently changed to reflect the result of the instruction; the previously existing state is no longer needed and can be discarded.

2.74   **completed:** After an instruction has **finished** and has sent a nonerror status to the Issue Unit, it is considered completed. **Note:** Although the state of the machine has

been temporarily altered by completion of an instruction, the state has not yet been permanently changed and the old state can be recovered until the instruction has been **committed**.

2.75 **executed:** An instruction is executed by an execution unit such as a Floating-point Multiply Adder (FMA). An instruction is in execution as long as it is still being processed by an execution unit.

2.76 **fetched:** Instructions are fetched from the external U2 instruction cache, the internal I0 instruction cache, the internal I1 instruction cache, or from the instruction prefetch buffers and sent to the Issue Unit.

2.77 **finished:** An instruction is finished when it has completed execution in a functional unit and has written its results onto a result bus. Results on the result busses go the register files and to waiting instructions in the instruction queues.

2.78 **initiated:** An instruction is initiated when it has all of the resources that it needs (for example, source operands) and it has been selected for execution (for example, it enters an FMADD unit).

2.79 **Instruction Dispatch:** The act of issuing an instruction to a reservation station.

2.80 **Instruction Issued:** An instruction is issued when it has been assigned a serial number in the active instruction ring. For example, an add instruction is considered "issued" when it has been assigned a serial number and decided which reservation station to be sent.

2.81 **Instruction Retired:** An instruction is retired when all machine resources (serial numbers, renamed registers) have been reclaimed and are available for use by other instructions. An instruction can only be retired after it has been committed.

2.82 **Instruction Stall:** Not every instruction can be issued in a given cycle. The CPU imposes certain issue constraints based on resource availability and program requirements. Instructions which may not be issued in this cycle are said to have stalled.

2.83 **issue-stalling instruction:** An instruction that prevents new instructions from being issued until it has committed.

2.84 **issue window:** This window holds the instructions to be issued in one clock cycle. SPARC64-III can issue a maximum of four instructions per clock cycle; thus, the issue window holds up to four instructions.

2.85 **machine sync:** The machine is synced when all previously executing instructions have committed; that is, there are no issued but uncommitted instructions in the machine.

2.86 **Memory Management Unit (MMU):** This term is used to refer to the address translation hardware in SPARC64-III that translates 64-bit Virtual Address into

Physical Addresses. The MMU is composed of the µITLB, µDTLB, MTLB, and the ASR and ASI registers used to manage address translation.

2.87 **MTLB:** Main TLB. Contains address translations for the µITLB and µDTLB. When the µITLB or µDTLB do not contain a translation they ask the MTLB for the translation. If the MTLB contains the translation, it sends the translation to the respective micro TLB. If it does not contain the translation it causes a fast access exception to a software translation trap handler which will load the translation information (PTE) into the MTLB and retry the access. *See also*: **TLB.**

2.88 **PTE:** Page Table Entry. An entry in the µITLB, µDTLB, or MTLB. The PTE contains all the information necessary to translate a virtual address into a physical address. If none of the TLB's contain a translation for a virtual address then a trap is taken to kernel software which will load the correct PTE into the MTLB. *See also*: **TLB.**

2.89 **reclaimed:** All instruction-related resources that were held until **commit** have been released and are available for subsequent instructions. Instruction resources are usually reclaimed a few cycles after they are committed.

2.90 **register renaming:** The CPU implements a large set of hardware registers that are invisible to the programmer. Before instructions are **issued**, source and destination registers are mapped onto this set of rename registers. This allows instructions that normally would be blocked, waiting for an architected register, to proceed in parallel. When instructions are **committed**, results in rename registers are posted to the architected registers in the proper sequence to produce the correct program results.

2.91 **scan:** A method used to initialize all of the machine state within a chip. In a chip that has been designed to be scannable, all of the machine state is connected in one or several loops called "scan rings." Initialization data can be scanned into the chip using the scan rings. The state of the machine also can be scanned out via the scan rings. The SPARC64-III chip is initialized by scanning in the initialization data before execution begins.

2.92 **serializing instruction:** *Synonym*: **syncing instruction**.

2.93 **superscalar:** An implementation that allows several instructions to be issued, executed, and committed in one clock cycle. The CPU **issues** up to four instructions per clock cycle. Up to eight can be committed, and up to 64 can be active per clock cycle.

2.94 **sync:** *Synonym*: **machine sync**.

2.95 **syncing instruction:** An instruction that causes a **machine sync**. Thus, before a syncing instruction is issued, all previous instructions (in program order) must have been committed. At that point, the syncing instruction is issued, executed, completed, and committed by itself.

2.96 **Reservation Station:** The CPU implements dataflow execution based on operand availability. Dispatched instructions are sent to reservation stations where they are buffered until all input operands become available. When operands are available, the instruction is scheduled for execution. Reservation stations also contain special tag matching logic which is used to capture the appropriate operand data. The reservation stations are sometimes referred to as queues (for example, the integer queue).

2.97 **Serial Number:** Every issued instruction is assigned a serial number (also sometimes called a sequence number) which provides a unique tag for identifying the instruction. The serial number accompanies the instruction throughout the processor until eventual retirement.

2.98 **TLB:** Translation Lookaside Buffer. A cache within the MMU that contains recent partial translations. These speed up closely following translations by eliminating the need to reread the Page Table Entry from memory.

2.99 μ**DTLB:** Micro Data TLB. A small fully associative buffer that contains address translations for data accesses. Misses in the μDTLB are handled by the MTLB. *See also*: **TLB.**

2.100 μ**ITLB:** Micro Instruction TLB. A small fully associative buffer that contains address translations for instruction accesses. Misses in the μITLB are handled by the MTLB. *See also*: **TLB.**

3   Architectural Overview

SPARC64-III architecture supports 32- and 64-bit integer and 32- and 64-bit floating-point as its principal data types. It also supports 128-bit floating-point operations by software emulation. The 32- and 64-bit floating-point types conform to IEEE Std 754-1985. The 128-bit floating-point type conforms to IEEE Std 1596.5-1992. The CPU defines general-purpose integer, floating-point, and special state/status register instructions, all encoded in 32-bit-wide instruction formats. The load/store instructions address a linear, 2^{64}-byte virtual address space.

3.1   SPARC-V9 Processor Architecture

Note:

> This section and its subsections are repeated from *V9*. Even though the SPARC64-III processor architecture is beginning to differ more significantly from this earlier, more simple model, these sections still provide some useful background for the implementation-specific discussion of the SPARC64-III processor architecture in 3.4.

A SPARC-V9 processor logically consists of an integer unit (**IU**) and a floating-point unit (**FPU**), each with its own registers. This organization allows for implementations with concurrent integer and floating-point instruction execution. Integer registers are 64-bits wide; floating-point registers are 32-, 64-, or 128-bits wide. Instruction operands are single registers, register pairs, register quadruples, or immediate constants.

The processor can be in either of two modes: **privileged** or **nonprivileged**. In privileged mode, the processor can execute any instruction, including privileged instructions. In nonprivileged mode, an attempt to execute a privileged instruction causes a trap to privileged software.

3.1.1   Integer Unit (IU)

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

An implementation of the SPARC-V9 IU may contain from 64 to 528 general-purpose 64-bit *r* registers. This corresponds to a grouping of the registers into 8 global *r* registers, 8 alternate global *r* registers, plus a circular stack of from 3 to 32 sets of 16 registers each, known as register windows. The number of register windows present (NWINDOWS) is implementation-dependent; on SPARC64-III, NWINDOWS = 5.

At a given time, an instruction can access the 8 *global*s (or the 8 *alternate globals*) and a register window into the *r* registers. The 24-register window consists of a 16-register set — divided into 8 *in* and 8 *local* registers — together with the 8 *in* registers of an adjacent register set, addressable from the current window as its *out* registers. See <u>Figure 22</u> on <u>page 64</u>.

The current window is specified by the current window pointer (CWP) register. The processor detects window spill and fill exceptions via the CANSAVE and CANRESTORE registers, respectively, which are controlled by hardware and supervisor software. The actual number of windows in a SPARC-V9 implementation is invisible to a user application program.

Whenever the IU accesses an instruction or datum in memory, it appends an **address space identifier** (**ASI**), to the address. All instruction accesses and most data accesses append an **implicit ASI**, but some instructions allow the inclusion of an explicit ASI, either as an immediate field within the instruction, or from the ASI register. The ASI determines the byte order of the access. All instructions are accessed in big-endian byte order; data can be referenced in either big- or little-endian order. See <u>5.2.1, "Processor State Register (PSTATE)"</u>, for information about changing the default byte order.

3.1.2 Floating-point Unit (FPU)

The FPU has thirty-two 32-bit (single-precision) floating-point registers, thirty-two 64-bit (double-precision) floating-point registers, and sixteen 128-bit (quad-precision) floating-point registers, some of which overlap. Double-precision values occupy an even-odd pair of single-precision registers, and quad-precision values occupy a quad-aligned group of four single-precision registers. The 32 single-precision registers, the lower half of the double-precision registers, and the lower half of the quad-precision registers overlay each other. The upper half of the double-precision registers and the upper half of the quad-precision registers overlay each other but do not overlay any of the single-precision registers. Thus, the floating-point registers can hold a maximum of 32 single-precision, 32 double-precision, or 16 quad-precision values. The floating-point registers are described in more detail in <u>5.1.4, "Floating-point Registers"</u>.

Floating-point load/store instructions are used to move data between the FPU and memory. The memory address is calculated by the IU. Floating-point **operate** (FPop) instructions perform the floating-point arithmetic operations and comparisons.

The floating-point instruction set and 32- and 64-bit data formats conform to the IEEE Standard for Binary Floating-point Arithmetic, IEEE Std 754-1985. The 128-bit floating-point data type conforms to the IEEE Standard for Shared Data Formats, IEEE Std 1596.5-1992.

If an FPU is not enabled, an attempt to execute a floating-point instruction generates an *fp_disabled* trap. In either case, privileged-mode software must:

■ Enable the FPU and reexecute the trapping instruction, or

■ Emulate the trapping instruction.

3.2 Instructions

Instructions fall into the following basic categories:

■ Memory access

■ Integer arithmetic / logical / shift

■ Control transfer

■ State register access

■ Floating-point operate

■ Conditional move

■ Register window management

These classes are discussed in the following subsections.

3.2.1 Memory Access

Load and store instructions, PREFETCHes, and the atomic operations, CASX, SWAP, and LDSTUB, are the only instructions that access memory. They use two *r* registers or an *r* register and a signed 13-bit immediate value to calculate a 64-bit, byte-aligned memory address. The IU appends an ASI to this address.

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

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

CAS, SWAP, and LDSTUB are special atomic memory access instructions that are used for synchronization and memory updates by concurrent processes.

3.2.1.1 Memory Alignment Restrictions

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 the possible exception of cases described in 6.3.1.1, "Memory Alignment Restrictions".

3.2.1.2 Addressing Conventions

The CPU 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 CPU 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. See 5.2.1, "Processor State Register (PSTATE)", for information about changing the implicit byte order to little-endian.

Addressing conventions are illustrated in Figure 65 on page 119 on Figure 66 on page 121.

3.2.1.3 Load/Store Alternate

Versions of load/store instructions, the **load/store alternate** instructions, can specify an arbitrary 8-bit address space identifier for the load/store data access. Access to alternate spaces $00_{16}..7F_{16}$ is restricted, and access to alternate spaces $80_{16}..FF_{16}$ is unrestricted. Some of the ASIs are available for implementation-dependent uses. Supervisor software can use the implementation-dependent ASIs to access special protected registers, such as MMU, cache control, and processor state registers, and other processor- or system-dependent values. See 6.3.1.3, "Address Space Identifiers (ASIs)", for more information.

Alternate space addressing is also provided for the atomic memory access instructions, LDSTUB, SWAP, and CASX.

3.2.1.4 Separate I and D Memories

The CPU has separate level-1 instruction and data caches. For this reason, programs that modify their own code (self-modifying code) must issue FLUSH instructions, or a system call with a similar effect, to bring the instruction and data caches into a consistent state.

3.2.1.5 Input/Output (I/O)

SPARC-V9 assumes that input/output registers are accessed via load/store alternate instructions, normal load/store instructions, or read/write Ancillary State Register instructions (RDASR, WRASR).

This document does not contain information about SPARC64-III-specific I/O registers. In particular, it does not discuss:

■ The semantic effect of accessing I/O locations

■ Nonprivileged access to I/O registers

■ The addresses and contents of I/O registers

3.2.1.6 Memory Synchronization

Two instructions are used for synchronization of memory operations: FLUSH and MEM-BAR. Their operation is explained in <u>A.20, "Flush Instruction Memory"</u>, and <u>A.32, "Memory Barrier"</u>, respectively. **Note**: STBAR is also available, but it is deprecated and should not be used in newly developed software.

3.2.2 Arithmetic / Logical / Shift Instructions

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

Shift instructions are used to shift the contents of an *r* register left or right by a given count. The shift distance is specified by a constant in the instruction or by the contents of an *r* register.

The integer multiply instruction performs a $64 \times 64 \rightarrow$ 64-bit operation. The integer division instructions perform $64 \div 64 \rightarrow$ 64-bit operations. In addition, for compatibility with SPARC-V8, $32 \times 32 \rightarrow$ 64-bit multiply, $64 \div 32 \rightarrow$ 32-bit divide, and multiply step instructions are included. Division by zero causes a trap. Some versions of the 32-bit multiply and divide instructions set the condition codes.

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

3.2.3 Control Transfer

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

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

Compatibility Note:

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

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 as either the sum of two *r* registers, or the sum of an *r* register and a 13-bit signed immediate value. The branch on condition codes without prediction instruction provides a displacement of ±8 Mbytes; the branch on condition codes with prediction instruction provides a displacement of ±1 Mbyte; the branch on register contents instruction provides a displacement of ±128 Kbytes, and the CALL instruction's 30-bit word displacement allows a control transfer to any address within ±2 gigabytes ($\pm 2^{31}$ bytes). **Note**: When 32-bit address masking is enabled (see 5.2.1.6, "PSTATE_address_mask (AM)"), the CALL instruction may transfer control to an arbitrary 32-bit address. The return from privileged trap instructions (DONE and RETRY) get their target address from the appropriate TPC or TNPC register.

3.2.4 State Register Access

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

Software can use read/write ancillary state register instructions to read/write the SPARC64-III-specific processor registers. See the subsection in 5.2.11, "Ancillary State Registers (ASRs)", for information about the implementation-dependent ASRs, including which of them are privileged.

3.2.5 Floating-point Operate

Floating-point operate (FPop) instructions perform all floating-point calculations; they are register-to-register instructions that operate on the floating-point registers. Like arithmetic/logical/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.

In addition the CPU has extended the SPARC-V9 architecture with floating-point multiply-add and multiply-subtract instructions. See A.23.1, "IMPDEP2 (Floating-point Multiply-Add/Subtract)", for more information.

3.2.6 Conditional Move

Conditional move instructions conditionally copy a value from a source register to a destination register, depending on an integer or floating-point condition code or upon the contents of an integer register. These instructions increase performance by reducing the number of branches.

3.2.7 Register Window Management

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

3.3 Traps

A **trap** is a vectored transfer of control to privileged software through a trap table that may contain the first eight instructions (thirty-two for fill/spill traps) of each trap handler. The base address of the table is established by software in a state register (the Trap Base Address register, TBA). The displacement within the table is encoded in the type number of each trap and the level of the trap. One half of the table is reserved for hardware traps; one quarter is reserved for software traps generated by trap (Tcc) instructions; the final quarter is reserved for future expansion of the architecture.

A trap causes the current PC and nPC to be saved in the TPC and TNPC registers. It also causes the CCR, ASI, PSTATE, and CWP registers to be saved in TSTATE. TPC, TNPC, and TSTATE are entries in a hardware trap stack, where the number of entries in the trap stack is equal to the number of trap levels supported (which is 4 in the CPU). A trap also sets bits in the PSTATE register, one of which can enable an alternate set of global registers for use by the trap handler. Normally, the CWP is not changed by a trap; on a window spill or fill trap, however, the CWP is changed to point to the register window to be saved or restored.

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

See Chapter 7, "Traps" for a complete description of traps.

3.4   SPARC64-III Processor Architecture

This section describes the internal architecture of the CPU. Figure 1 contains the SPARC64-III Block Diagram.
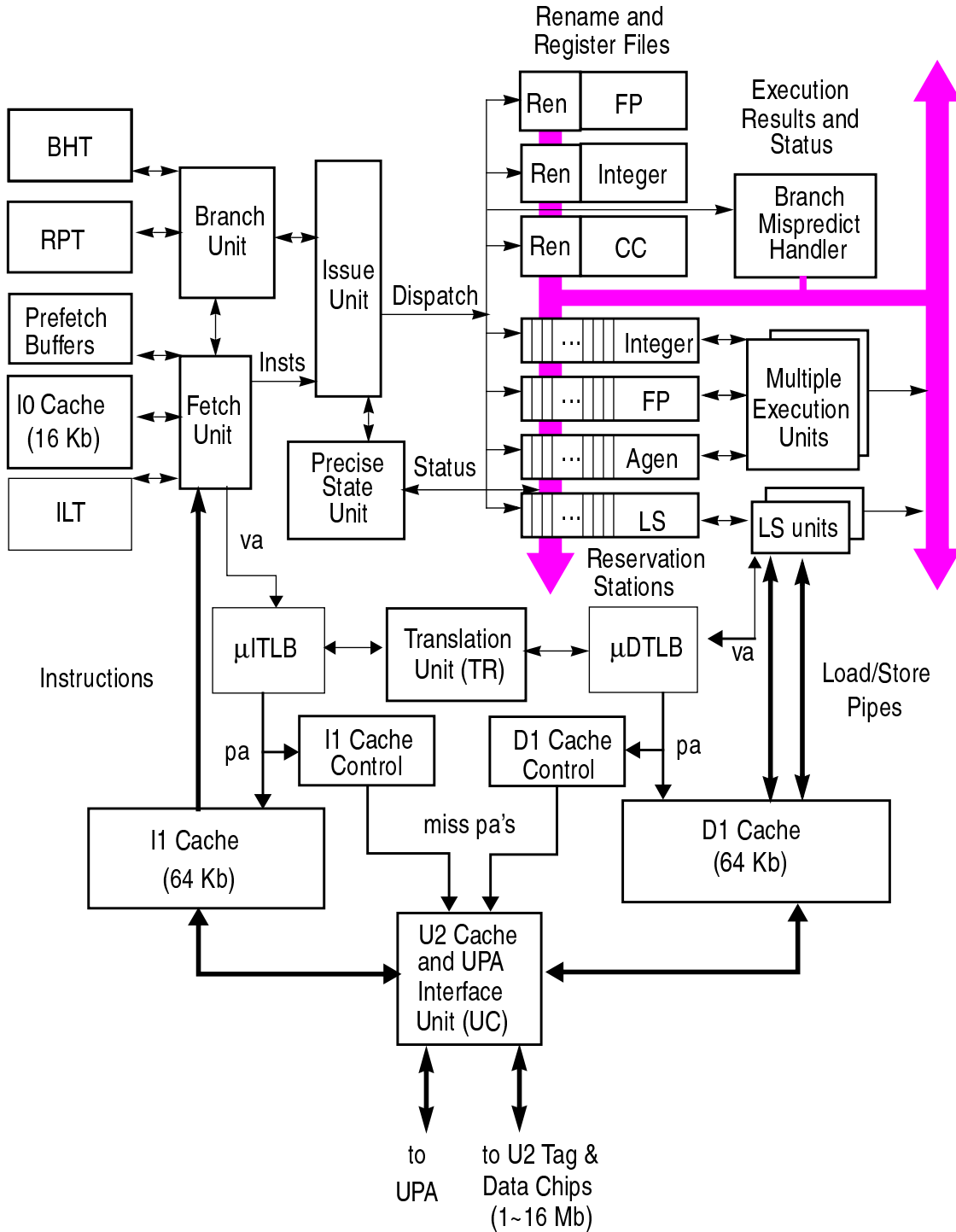
Figure 1: SPARC64-III CPU Block Diagram

3.4.1 Life Cycle of an Instruction

All instructions pass through the following states within the CPU:

Fetched:
>   Instructions are *fetched* from the external instruction cache, the internal I0 instruction cache, or the instruction prefetch buffers (described in 3.4.3, "Branch Unit (BRU)"); they are then sent to the Issue Unit.

Issued:
>   A serial number is assigned to each instruction when it is *issued*. After an instruction is *issued*, it is immediately *dispatched*.

Dispatched:
>   Instructions are *dispatched* when they are sent to a functional unit queue. For example, an add instruction is considered dispatched when it is sent to the queue for one of the integer adders in the Fixed-point Integer Functional Unit (described in 3.4.6.1.3).

Initiated:
>   An instruction is *initiated* when it has all of the resources it needs and it has been selected for execution by an execution unit.

Executed:
>   An instruction is *executed* by an execution unit in the Data Flow Unit; for example, an integer multiply is executed by a the integer multiplier (IMUL) in the Fixed-point Functional Unit (described in 3.4.6.1.3). An instruction is *in execution* as long as it is still being processed by an execution unit.

Finished:
>   An instruction is *finished* when it has completed execution in a functional unit and has written its results into a register file. When an instruction finishes, the execution unit informs the Issue Unit (ISU) and reports its status.

Completed:
>   An instruction is *completed* when it has *finished* and has sent a **nonerror** status to the ISU. **Note**: Although the state of the machine is altered temporarily when an instruction is *completed*, the state change is not yet permanent; the old machine state can be recovered until the instruction has been *committed*.

Committed:
>   An instruction is *committed* when it has completed **without error** and **all** prior instructions (in program order) have completed **without error.** When an instruction is committed, the state of the machine is permanently changed to reflect the result of the instruction.

Reclaimed:
>   All instruction-related resources are usually *reclaimed* a few cycles after the instruction is *committed*. After the resources are *reclaimed*, they are again available for subsequent instructions.

The following sequence clarifies the ordering within these states. Instructions are:

1. *Fetched* in order.

2. *Issued* in order.

3. *Dispatched* in order.

4. *Initiated* out of order.

5. *Executed* out of order.

6. *Completed* out of order.

7. *Committed* in order.

During an instruction's lifetime within the CPU it undergoes a series of transformations that allow it to be processed more efficiently. These transformations include:

Recoding:
>The instruction opcode is converted into a more efficient internal format.

Register Renaming:
>The source and destination registers encoded in the instruction are renamed by mapping them onto a much larger internal register set. A more complete description of this process is given in 3.4.1.1.

Numbering:
>A serial number is assigned to an instruction when it is *issued*. This number is unique in the system as long as the instruction is active; however, the numbers are reclaimed and reused after the instruction is *committed*.

Packetizing:
>An instruction packet (IP) is created. It contains the recoded opcode, the serial number, the renamed register numbers, and some information that allows the effects of the instruction to be undone if it was executed in error (for example, after a mispredicted branch). The functional units that actually execute the instructions deal only with IPs.

3.4.1.1 Register Renaming

Sometimes increasing the number of available registers can allow for more parallelism within a CPU. For example, consider the following code fragment:

```
1.   ld   [%r5+%r6], %r2        ! Load something into %r2
2.   add %r2,%r8,%r30           ! Use the loaded data
3.   beq %xcc,d12               ! Branch if equal
4.   add %r9,%r10,%r2           ! Not equal-store a result into %r2
```

Clearly, instruction #1 must complete before instruction #2 begins, because `%r2` is an output of #1 and an input to #2. Nothing that can be done in hardware will allow these instructions to execute in parallel. The only reason that instruction #4 cannot execute in parallel with #1 or #2, however, is that it needs to **reuse** `%r2`.

If the destination in instruction #4 is changed to `%r9`, the instruction could run in parallel with #1 or #2. But the hardware can't always use another architected register, because there isn't always one available.

SPARC64-III solves this problem by:

1. Providing more physical registers than architected registers, and

2. Providing a mapping between architected and physical registers.

This strategy is called register renaming; it changes the example as follows:

```
1.   ld  [%pr15+%pr22],%pr7    ! Load into physical register %pr7
2.   add %pr7,%pr8,%pr45       ! Use the loaded data
3.   beq %xcc,dog              ! Branch if equal
4.   add %pr9,%pr21,%pr37      ! Store a result into physical %pr37
```

Now #4 can be executed in parallel with #1 or #2, since it is no longer dependent on one of their registers. **Note**: #1 and #2 are still dependent (as they should be) and cannot be executed in parallel.

For a more complete discussion of data dependencies, see .

3.4.2 SPARC64-III Conceptual Architecture

Figure 2 shows a high-level view of the internal architecture of the CPU.
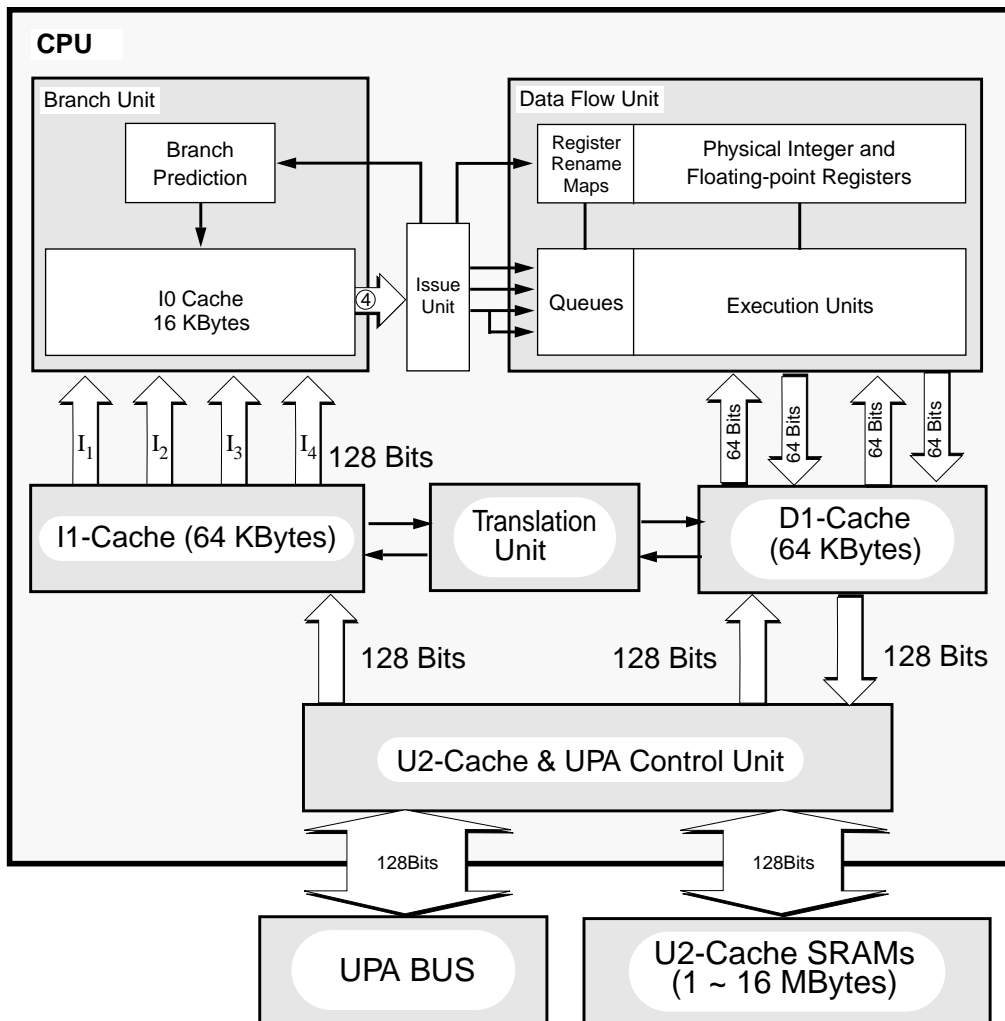
Figure 2: CPU High-Level Internal Architecture

Conceptually, the CPU contains seven sections:

Branch Unit (BRU):
Fetches instructions, controls program counter (PC) sequencing, and attempts to provide four instructions per clock to the Issue Unit.

Issue Unit (ISU):
Issues and dispatches up to four instructions per clock, and keeps track of all currently active instructions.

Data Flow Unit (DFU):

> Executes instructions using the integer and floating-point hardware registers, the register rename maps, the fixed-integer and floating-point execution units, and the load / store unit.

Level-1 Instruction Cache (I1-Cache):

> Receives one instruction fetch request (16 bytes per request) per cycle from BRU, accesses the level-1 instruction cache (4-way set-associative, 64 Kbytes), and returns 4 instructions in a cycle. When a cache miss happens, sends a request to UC to get the line data (64 bytes).

Level-1 Data Cache (D1-Cache):

> Receives up to two load or store requests (each request is up to 8 byte access) per cycle from LSU, accesses the level-1 data cache (4-way set-associative, 64 Kbytes), and returns up to 2 load data in a cycle. When a cache miss happens, sends a request to UC to get the cache line data (64 bytes).

Translation Unit (TR):

> Has the Main TLB which has 256 entries and is fully-associative, accesses it when the translation request comes from I1-Cache or D1-Cache, and returns the page table entry to I1-Cache or D1-Cache.

U2-Cache & UPA Control Unit (UC):

> Controls the external unified level-2 cache (U2-Cache, direct-map, 1~16 MBytes) and UPA bus interface. Receives level-1 cache miss requests from I1-Cache and D1-Cache, accesses U2-Cache, and returns the cache line data (64 Bytes) to I1-Cache and D1-Cache. When a cache miss happens, sends a request to UPA bus to get the cache line data.

3.4.3 Branch Unit (BRU)

The Branch Unit is responsible for sending up to four instructions per clock to the Issue Unit. The BRU gets these instructions from one of these locations:

■ The on-chip Level-0 Instruction (I0) Cache

■ The on-chip Prefetch Buffers

■ The external Level-1 Instruction Caches (through the Instruction Recode Unit)

The BRU contains the following components:

Instruction Recode Unit:

> Recodes instructions into a more efficient internal format. Instructions remain in recoded form throughout their lifetime in the CPU.

Instruction Prefetch Buffers:

> Prefetches instructions from the Level-1 Instruction Caches through the Instruction Recode Unit.

Instruction Level-0 Cache (I0 Cache):

> Holds up to 4,096 prefetched and recoded instructions.

Branch Prediction:

> Attempts to determine the address of the next four instructions to fetch, using feedback from the Issue Unit.

Fetch Unit:

> Supplies the addresses of the instructions to be fetched. This unit uses Branch Prediction to attempt to determine the correct fetch address. It then transmits this address to the I0 Cache, the Prefetch Buffers, and the external I1 Cache.

> If the initial fetch address proves to be incorrect, the Fetch Unit discards the incorrectly fetched instructions, recalculates the correct fetch address, and retransmits the correct address to the I0 Cache, the Prefetch Buffers, and the external I1 Cache.

> The Fetch Unit also handles mispredicted branches and processes traps.

3.4.4 Issue Unit (ISU)

The Issue Unit issues up to four instructions per clock and keeps track of all currently active instructions. It contains the following components:

Precise State Unit (PSU):

> Tracks the state of all instructions from the time they are *issued* until they are *reclaimed*. It:

● Assigns serial numbers to newly *issued* instructions

● Waits for the DFU to report *finished* status on *issued* instructions

● *Completes* instructions that are *finished* and *commits* them in order

● *Reclaims* resources used by *committed* instructions

> The PSU can back up the CPU to any previous uncommitted state in the event of an error or mispredicted branch. The PSU also handles exceptions, errors, and interrupts.

Register Rename / Freelist Unit:

> Manages the list of free physical registers that are available to rename the registers in the instructions that will be issued in this clock. For performance reasons, the physical registers and the maps of physical to architected register numbers are kept in the DFU, but they are logically part of the ISU.

I-Matrix:

> Determines how many instructions (from 0 to 4) can be *issued* in each clock. It gathers resource usage information (for example, how many queue slots are available) from many parts of the CPU.

Dispatch:

> Sends instruction packets to be enqueued at the DFU's functional units, described in 3.4.6 on page .

3.4.5 Instruction Fetch, Issue, and Dispatch

Figure 3 shows a detailed view of the instruction fetch and issue process.
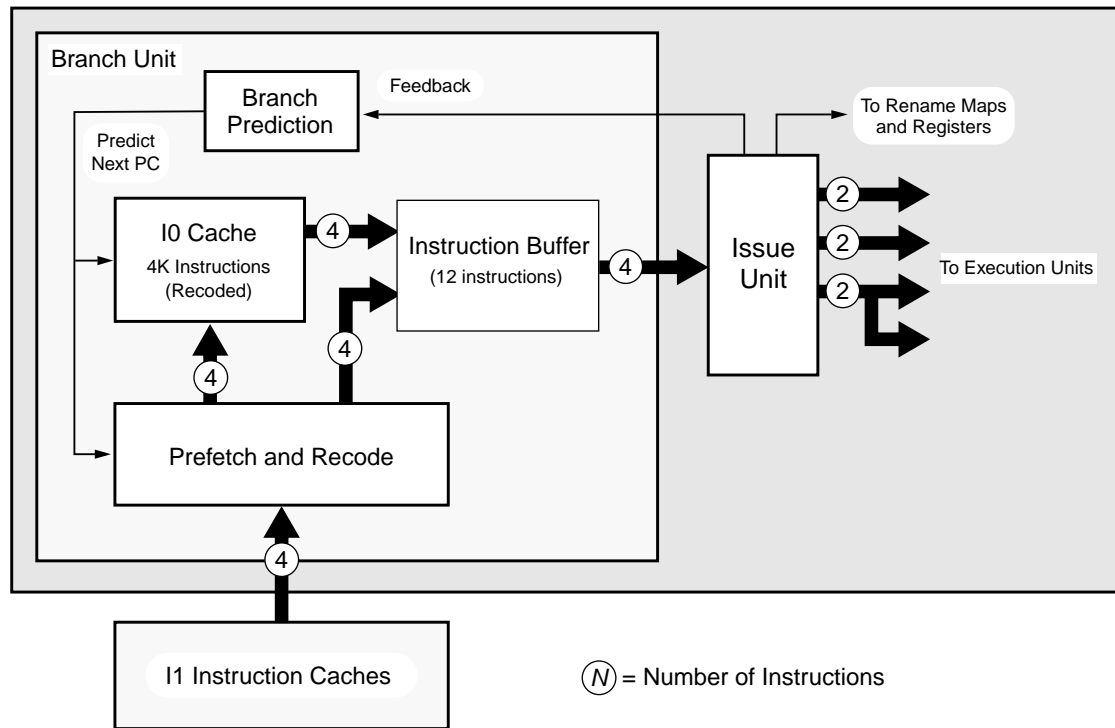
Figure 3: Instruction Fetch, Issue, and Dispatch

Using feedback from the Issue Unit, Branch Prediction attempts to predict the next PC value. The predicted address is called the Fetch PC (FPC). Next, the Branch Unit sends the four instructions at the FPC address to the Issue Unit. These instructions will come from:

■ The I0 cache, if present there

■ The Prefetch Buffers, if present there

■ The I1 cache, if the instruction is not already present in the CPU

The Prefetch Buffers are constantly prefetching instructions through the Instruction Recode Unit, which recodes them into an internal format. These recoded instructions are then cached in the Prefetch Buffers; eventually they may be sent on to the I0 cache or directly to the Issue Unit.

Using information gathered by the Precise State Unit, the Issue Unit determines whether the instructions it has received are from the correct address. If they are, the IU dispatches up to four of the instructions to the DFU. If the PC was mis-predicted, the Issue Unit feeds this information back to the BRU and discards the instructions.

3.4.6 Data Flow Unit (DFU)

The Data Flow Unit (shown in Figure 4) executes instructions using the integer and float-ing-point hardware registers and register rename maps, and the fixed-point integer, address generation, floating-point, and load / store functional units.
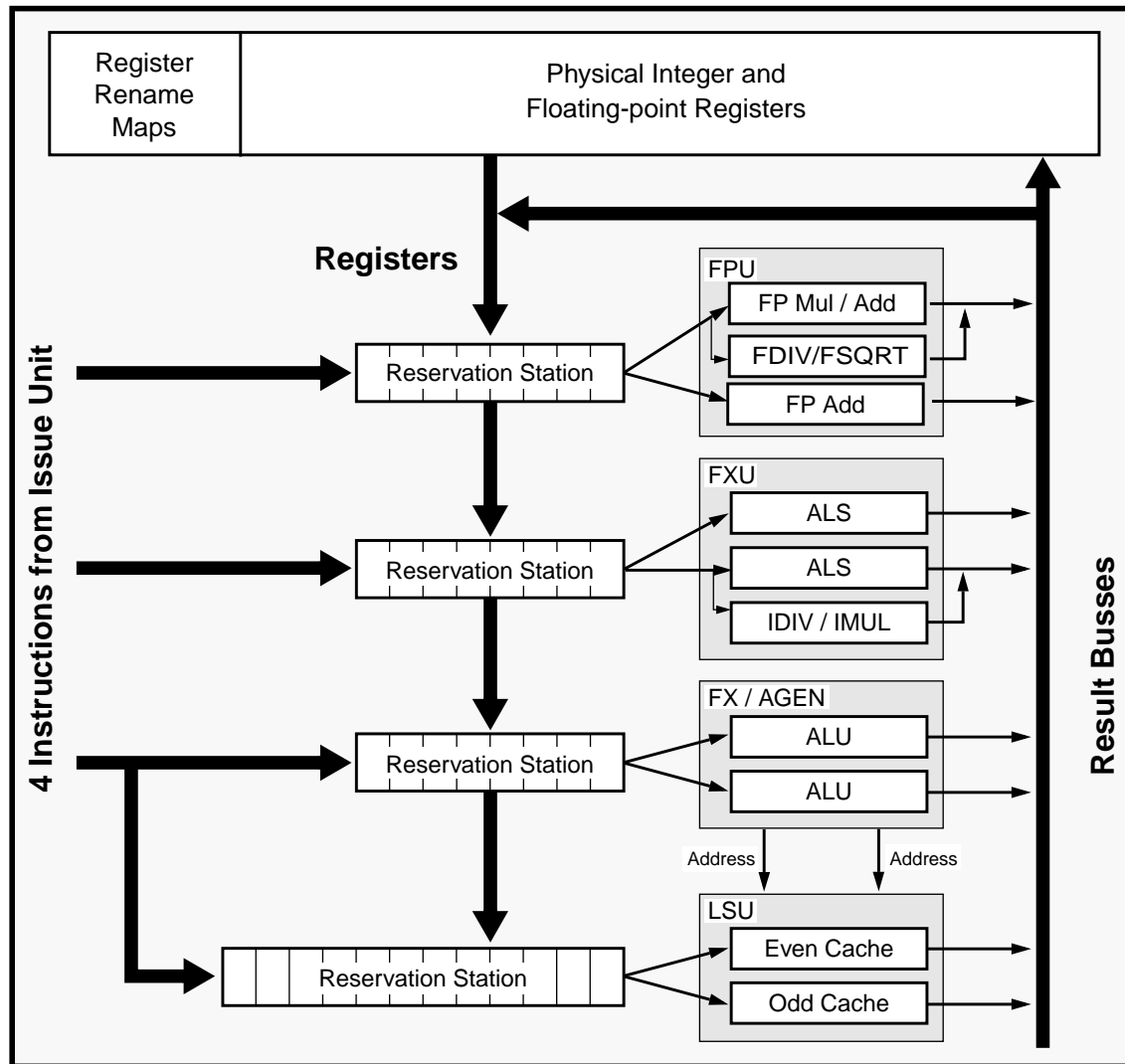
Figure 4: Data Flow Unit (DFU)

The functional units are described in 3.4.6.1, "DFU Functional Units." In addition to the functional units the DFU also contains:

Physical Integer Register File:
Physical registers to which the architected registers within each integer instruction are remapped.

Integer Register Rename Map:
Associations needed to remap each integer register reference within an instruction onto the appropriate physical integer register.

Physical Floating-point Register File:

> Physical registers to which the architected registers within each floating-point instruction are remapped.

Floating-point Register Rename Map:

> Associations needed to remap each floating-point register reference within an instruction onto the appropriate physical floating-point register.

3.4.6.1 DFU Functional Units

The DFU contains the four functional units that are responsible for executing CPU instructions. They are:

Floating-point Functional Unit (FPU):

> Performs floating-point arithmetic, comparisons, multiplies, divides, and square-roots (including floating-point multiply-add and multiply-subtract).

Fixed-point Integer Functional Unit (FXU):

> Performs fixed-point arithmetic, logical operations, shifts, multiplies, and divides.

Fixed-point Integer / Address Generation Functional Unit (FX/AGEN):

> Performs fixed-point arithmetic and logical operations and calculates load/store addresses.

Load / Store Functional Unit (LSU):

> Processes all load and store instructions.

Each functional unit contains:

Reservation Station:

> Stores the instruction packet (IP) and source register(s) for each instruction that has been *issued* and is waiting to be *initiated* in the functional unit.

Execution Units:

> Perform the processing necessary to execute each instruction. **Note**: The execution units for the LSU are actually the external data caches, which execute the loads and stores.

Subsection 3.4.6.1.1 describes the reservation stations. Subsections 3.4.6.1.3 through 3.4.6.1.5 describe the functional units.

3.4.6.1.1 Reservation Stations

Each reservation station contains an Instruction Packet Queue (IPQ), which holds the instruction packets (IPs) waiting to begin execution in the functional unit. Associated with each queue entry is one or more Register Caches, which hold the source operand(s) for the instruction. When a source operand of an enqueued instruction is generated by a previous

instruction, that queue entry must wait until its source operands are available. Figure 5 shows the general form of a reservation station.
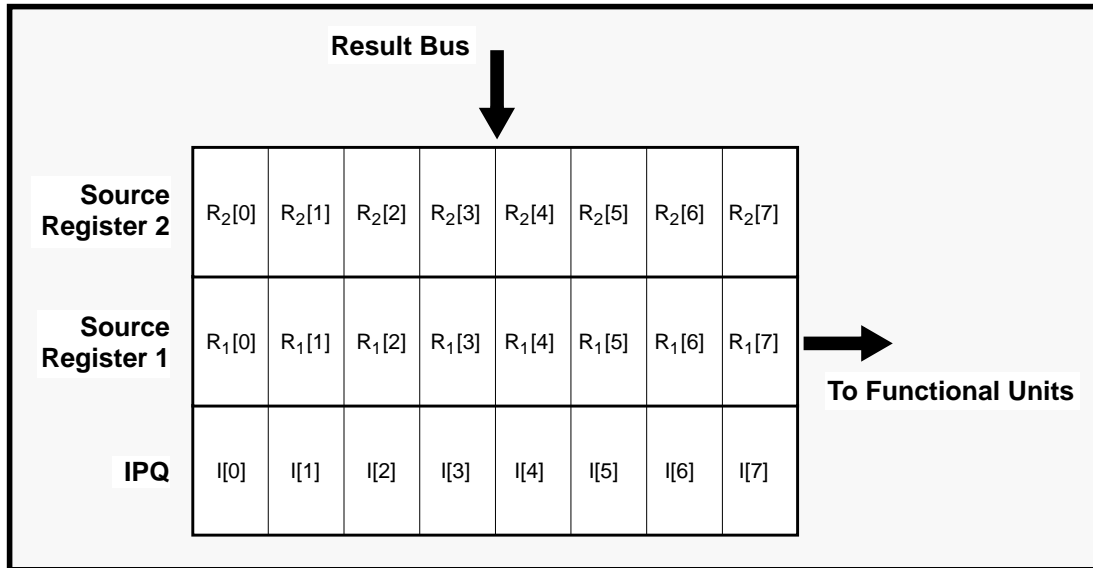
Figure 5: Reservation Station Detail

Each reservation station is an $N \times (S + 1)$ matrix, where N is the number of entries, S is the number of source registers, and 1 is the IPQ entry. Table 2 gives the values for N and S for each functional unit in the DFU.

Table 2—Reservation Station Sizes

Functional Unit	Number of Entries (N)	Number of Source Registers (S)
FXU	8	2
FX / AGEN	8	2
FPU	8	3
LSU	12	1

3.4.6.1.2 Floating-point Functional Unit (FPU)

The Floating-point Functional Unit (see Figure 4 on page 46) executes floating-point instructions using one floating-point multiply-adder (FMA), one floating-point divider (FDIV/FSQRT), and one floating-point adder (FA).

The FPU can initiate one floating-point add and either one floating-point multiply-add or one floating-point divide operation per clock, and can generate one floating-point add and one multiply-addition or one division result per clock. Once the operations are initiated, however, the FMA and FDIV/FSQRT units execute in parallel.

The FMA is pipelined and has a latency of 4 cycles. The FMA also performs floating-point moves, which take one clock. The FA is pipelined and has a latency of 3 cycles.

The FDIV/FSQRT has a latency of approximately 12 cycles (single) and 22-23 cycles (double). The FDIV/FSQRT unit blocks the FMA for one clock in order to start the division and for another clock when it places the result on the result bus. While the divide is in progress, however, the FMA and FDIV/FSQRT execute in parallel.

3.4.6.1.3 Fixed-point Integer Functional Unit (FXU)

The Fixed-point Integer Functional Unit (see Figure 4 on page 46) executes integer instructions using two independent Arithmetic/Logical/Shift Units (ALSs), one of which contains an integer multiplier/divider.

One of the ALSs shares its operand and result busses with the integer multiplier/divider. This combined unit can initiate either one MULDIV or one ALS operation per clock, and can generate one MULDIV or one ALS result per clock. Once the operations are initiated, however, the ALS and multiplier/divider units execute in parallel.

Each ALS contains:

■ An integer adder

■ A logic unit (computes AND, OR, and so forth)

■ A 64-bit barrel shifter

ALS operations take only one clock each, so there is no need to pipeline them.

An integer multiply operation takes between 4 (32-bits) and 6 (64-bits) clocks. An integer divide takes between 2 and 37 clocks, depending on the arguments; the average is about 13 clocks. The multiplier/divider "steals" one ALS cycle to place its results on the result bus.

3.4.6.1.4 Fixed-point/Address Generation Functional Unit (FX / AGEN)

The Fixed-point / Address Generation Functional Unit (see Figure 4 on page 46) executes integer instructions and calculates load/store addresses using two independent Arithmetic/Logical Units. **Note**: Unlike the FXU, the FX/AGEN does not contain a shifter or an integer multiplier/divider.

The FX/AGEN is used to calculate:

■ Effective addresses for loads and stores

■ Integer arithmetic operations that do not require a shift, multiply, or divide

Calculations are required for some CPU addressing modes; for example, in the instruction

```
ld [%o6 + 64], %i5
```

the FX/AGEN adds 64 to the contents of %o6 to obtain the effective address for the load. The FX/AGEN sends the calculated address and the instruction serial number to the Load Store Unit.

The FX/AGEN can perform up to two calculations per clock, in the following combinations:

■ Two effective address calculations

■ One address calculation and one integer calculation

■ Two integer calculations

3.4.6.1.5 Load / Store Functional Unit (LSU)

The Load / Store Functional Unit (see Figure 4 on page 46) executes load and store instructions accessing the level-1 even and odd data caches.

The LSU is responsible for guaranteeing that loads and stores execute correctly. Some out-of-order execution is allowed, but some operations must execute in order.

The LSU can perform two simultaneous loads or stores to the level-1 data caches per clock, one to the even cache and one to the odd cache. It can accept up to two new load or store requests per clock. The level-1 caches have a three-clock latency, but they are pipe-lined, so two new accesses can be started and two old accesses completed during each clock.

4 Data Formats

The CPU architecture recognizes these fundamental data types:

■ Signed Integer: 8, 16, 32, and 64 bits.

■ Unsigned Integer: 8, 16, 32, and 64 bits.

■ Floating Point: 32 and 64 bits. Operations on 128-bit floating-point data are emulated by system software.

The widths of the data types are:

■ Byte: 8 bits

■ Halfword: 16 bits

■ Word: 32 bits

■ Extended Word: 64 bits

■ Tagged Word: 32 bits (30-bit value plus 2-bit tag)

■ Doubleword: 64 bits

■ Quadword: 128 bits (emulated)

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

Subsections 4.1 through 4.11 illustrate the signed integer, unsigned integer, and tagged formats. Subsections 4.12 through 4.14 illustrate the floating-point formats. In 4.4, 4.9, 4.13, and 4.14, the individual subwords of the multiword data formats are assigned names. The arrangement of the subformats in memory and processor registers based on these names is shown in <u>Table 3</u> on page <u>56</u>. Tables 4 through 7 on pages <u>57</u> through <u>58</u> define the integer and floating-point formats.

4.1   Signed Integer Byte

Figure 6 illustrates the signed integer byte data format.

```
┌─┬──────────┐
│S│          │
└─┴──────────┘
 7 6        0
```
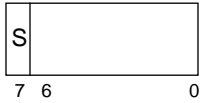
Figure 6: Signed Integer Byte Data Format

4.2   Signed Integer Halfword

Figure 7 illustrates the signed integer halfword data format.

```
┌─┬────────────────┐
│S│                │
└─┴────────────────┘
 15 14             0
```

Figure 7: Signed Integer Halfword Data Format

4.3   Signed Integer Word

Figure 8 illustrates the signed integer word data format.

```
┌─┬──────────────────────────┐
│S│                          │
└─┴──────────────────────────┘
 31 30                       0
```
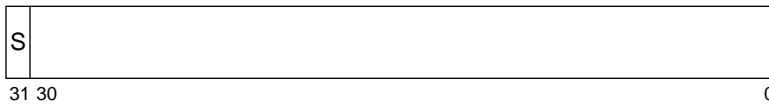
Figure 8: Signed Integer Word Data Format

4.4   Signed Integer Double

Figure 9 illustrates both components (SD-0 and SD-1) of the signed integer double data format.

SD–0

```
┌─┬──────────────────────────┐
│S│   signed_dbl_integer[62:32]  │
└─┴──────────────────────────┘
 31 30                       0
```

SD–1

```
┌────────────────────────────┐
│   signed_dbl_integer[31:0]     │
└────────────────────────────┘
 31                          0
```
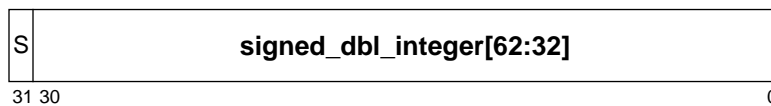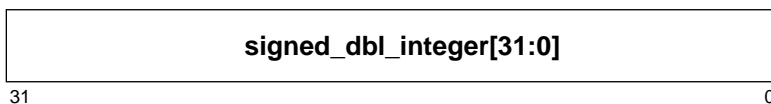
Figure 9: Signed Integer Double Data Format

4.5 Signed Extended Integer

Figure 10 illustrates the signed extended integer (SX) data format.

SX

S	signed_ext_integer

63 62   0
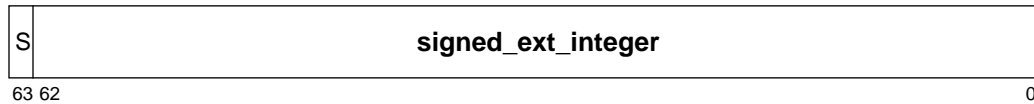
Figure 10: Signed Extended Integer Data Format

4.6 Unsigned Integer Byte

Figure 11 illustrates the unsigned integer byte data format.

7   0

Figure 11: Unsigned Integer Byte Data Format

4.7 Unsigned Integer Halfword

Figure 12 illustrates the unsigned integer halfword data format.

15   0

Figure 12: Unsigned Integer Halfword Data Format

4.8 Unsigned Integer Word

Figure 13 illustrates the unsigned integer word data format.

31   0
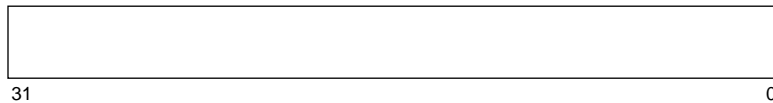
Figure 13: Unsigned Integer Word Data Format

4.9 Unsigned Integer Double

Figure 14 illustrates both components (UD-0 and UD-1) of the unsigned integer double data format.

UD–0

unsigned_dbl_integer[63:32]

31   0

UD–1

unsigned_dbl_integer[31:0]

31   0
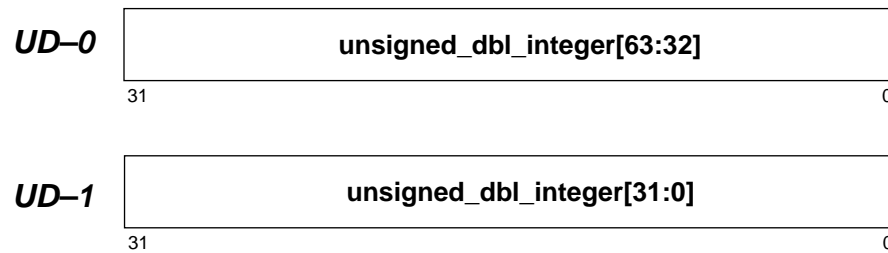
Figure 14: Unsigned Integer Double Data Format

4.10 Unsigned Extended Integer

Figure 15 illustrates the unsigned extended integer (UX) data format.

UX

unsigned_ext_integer

63   0

Figure 15: Unsigned Extended Integer Data Format

4.11 Tagged Word

Figure 16 illustrates the tagged word data format.

	tag

31   2 1 0
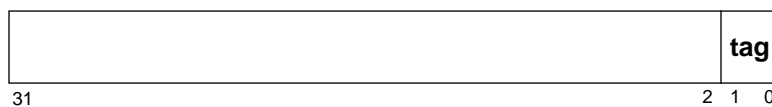
Figure 16: Tagged Word Data Format

4.12 Floating-point Single Precision

Figure 17 illustrates the floating-point single-precision data format.

S	exp[7:0]	fraction[22:0]

31 30   23 22   0
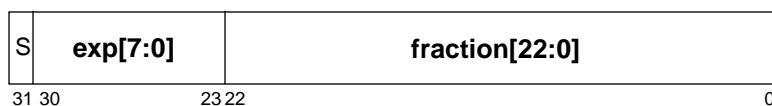
Figure 17: Floating-point Single-precision Data Format

4.13 Floating-point Double Precision

Figure 18 illustrates both components (FD-0 and FD-1) of the floating-point double-precision data format.

FD–0

S	exp[10:0]	fraction[51:32]

31 30   20 19   0

FD–1

fraction[31:0]

31   0
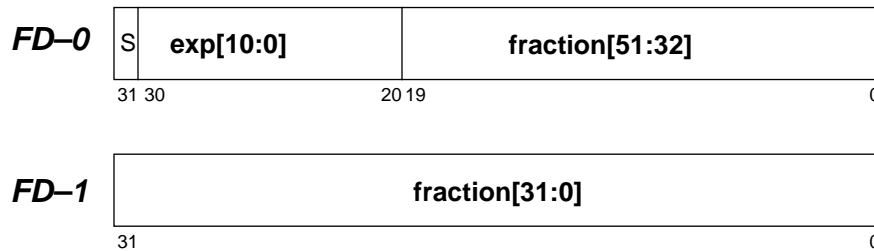
Figure 18: **Floating-point Double-precision Data Format**

4.14 Floating-point Quad-precision

The CPU does not implement any quad-precision floating-point operations in hardware. These operations cause an *fp_exception_other* trap with FSR.*ftt = unimplemented_FPop*). See 5.1.7, "Floating-point State Register (FSR)", for more information. The OS kernel then emulates the quad operation and stores the result into a quad-aligned set of floating-point registers, which are defined in Table 3, "Double- and Quadwords in Memory and Registers (V9=1)."

Implementation Note:
> The OS kernel does not contain emulation routines for the quad-precision multiply-add or multiply-subtract instructions.

The LDQF, LDQFA, STQF, and STQFA instructions cause an *illegal_instruction* exception; they are emulated by system software.

Figure 19 illustrates all four components (FQ-0 through FQ-3) of the floating-point quad-precision data format.

FQ–0

S	exp[14:0]	fraction[111:96]

31 30   16 15   0

FQ–1

fraction[95:64]

31   0

FQ–2

fraction[63:32]

31   0

FQ–3

fraction[31:0]

31   0
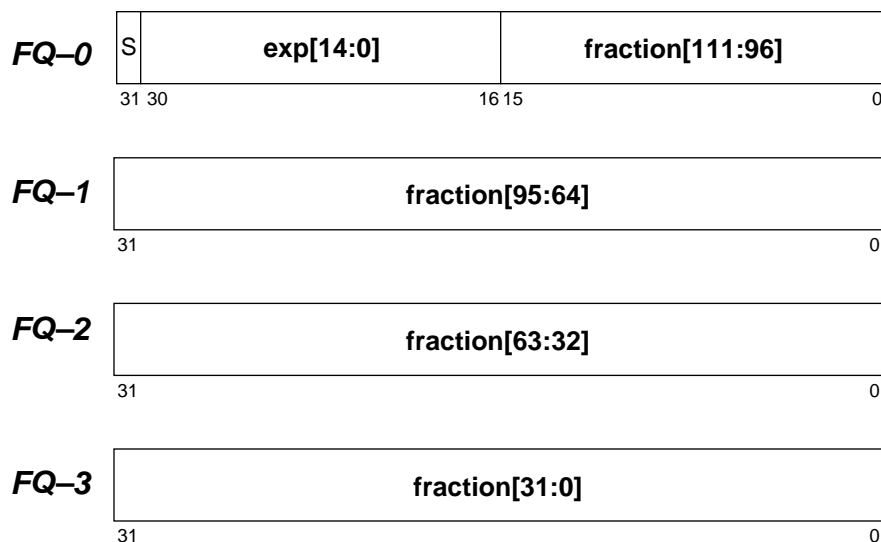
Figure 19: Floating-point Quad-precision Data Format

Table 3 describes the memory and register alignment for double- and quadword.

Table 3: Double- and Quadwords in Memory and Registers (V9=1)

Subformat Name	Subformat Field	Required Address Alignment	Memory Address	Register Number Alignment	Register Number
SD-0	signed_dbl_integer[63:32]	0 **mod** 8	n	0 **mod** 2	r
SD-1	signed_dbl_integer[31:0]	4 **mod** 8	$n + 4$	1 **mod** 2	$r + 1$
SX	signed_ext_integer[63:0]	0 **mod** 8	n	—	r
UD-0	unsigned_dbl_integer[63:32]	0 **mod** 8	n	0 **mod** 2	r
UD-1	unsigned_dbl_integer[31:0]	4 **mod** 8	$n + 4$	1 **mod** 2	$r + 1$
UX	unsigned_ext_integer[63:0]	0 **mod** 8	n	—	r
FD-0	s:exp[10:0]:fraction[51:32]	0 **mod** 4 [†]	n	0 **mod** 2	f
FD-1	fraction[31:0]	0 **mod** 4 [†]	$n + 4$	1 **mod** 2	$f + 1$
FQ-0	s:exp[14:0]:fraction[111:96]	0 **mod** 4 [‡]	n	0 **mod** 4	f
FQ-1	fraction[95:64]	0 **mod** 4 [‡]	$n + 4$	1 **mod** 4	$f + 1$
FQ-2	fraction[63:32]	0 **mod** 4 [‡]	$n + 8$	2 **mod** 4	$f + 2$
FQ-3	fraction[31:0]	0 **mod** 4 [‡]	$n + 12$	3 **mod** 4	$f + 3$

[†] Although a floating-point doubleword is required only to be word-aligned in memory, it is rec-
ommended that it be doubleword-aligned (that is, the address of its FD-0 word should be 0 **mod**
8).

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

Implementation Note:
  Floating-point quad is not implemented in the CPU. Quad-precision operations, except floating-
point multiply-add and multiply-subtract, are emulated in the OS kernel.

Table 4 describes the width and ranges of the signed, unsigned, and tagged integer data formats.

Table 4: Signed Integer, Unsigned Integer, and Tagged Format Ranges (V9=2)

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

Table 5 describes the floating-point single-precision data formats.

Table 5: Floating-point Single-precision Format Definition (V9=3)

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); $f = .0uu\text{-}uu$ (At least one bit of the fraction must be nonzero)
Quiet NaN	$s = u$; $e = 255$ (max); $f = .1uu\text{-}uu$
$-\infty$ (negative infinity)	$s = 1$; $e = 255$ (max); $f = .000\text{-}00$
$+\infty$ (positive infinity)	$s = 0$; $e = 255$ (max); $f = .000\text{-}00$

Table 6 describes the floating-point double-precision data formats.

Table 6: Floating-point Double-precision Format Definition (V9=4)

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

Table 7 describes the floating-point quad-precision data formats.

Table 7: Floating-point Quad-precision Format Definition (V9=5)

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 Registers

The CPU processor includes two types of registers: general-purpose, or working data registers, control/status registers, and ASI registers.

Working data registers include:

■ Integer working registers (*r* registers)

■ Floating-point working registers (*f* registers)

Control/status registers include:

■ Program Counter register (PC)

■ Next Program Counter register (nPC)

■ Processor State register (PSTATE)

■ Trap Base Address register (TBA)

■ Y register (Y)

■ Processor Interrupt Level register (PIL)

■ Current Window Pointer register (CWP)

■ Trap Type register (TT)

■ Condition Codes Register (CCR)

■ Address Space Identifier register (ASI)

■ Trap Level register (TL)

■ Trap Program Counter register (TPC)

■ Trap Next Program Counter register (TNPC)

■ Trap State register (TSTATE)

■ Hardware clock-tick counter register (TICK)

■ Savable windows register (CANSAVE)

■ Restorable windows register (CANRESTORE)

■ Other windows register (OTHERWIN)

■ Clean windows register (CLEANWIN)

■ Window State register (WSTATE)

■ Version register (VER)

■ Implementation-dependent Ancillary State Registers (ASRs)

■ Floating-point State Register (FSR)

■ Floating-point Registers State register (FPRS)

The ASI registers are defined in Appendix L, "ASI Assignments".

The SPARC-V9 architecture also defines two implementation-dependent registers: the IU Deferred-trap Queue and the Floating-point Deferred-trap Queue (FQ); the CPU does not need or contain either queue. All CPU traps are precise, except for the *data_breakpoint* trap. See *V9* for more information about these registers.

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 read as zeroes and, when written by software, should always be written as zeroes.

5.1 Nonprivileged Registers

The registers described in this subsection are visible to nonprivileged (application, or "user-mode") software.

5.1.1 General Purpose *r* Registers

At any moment, general-purpose registers appear to nonprivileged software as shown in Figure 20 on page 61.

The CPU contains 96 general-purpose 64-bit *r* registers. They are partitioned into eight *global* registers, eight *alternate global* registers, plus five 16-register sets. A register window consists of the current eight *in* registers, eight *local* registers, and eight *out* registers. See Table 8 on page 63.

5.1.1.1 Global *r* Registers

Registers $r[0]..r[7]$ refer to a set of eight registers called the global registers ($g0..g7$). At any time, one of two sets of eight registers is enabled and can be accessed as the global registers. The currently enabled set of global registers is selected by the Alternate Global (AG) field in the PSTATE register. See 5.2.1, "Processor State Register (PSTATE)," for a description of the AG field.

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

Compatibility Note:
> Since the PSTATE register is writable only by privileged software, existing nonprivileged SPARC-V8 software operates correctly on a SPARC64-III implementation if supervisor software ensures that nonprivileged software sees a consistent set of global registers.

i7	*r*[31]
i6	*r*[30]
i5	*r*[29]
i4	*r*[28]
i3	*r*[27]
i2	*r*[26]
i1	*r*[25]
i0	*r*[24]
l7	*r*[23]
l6	*r*[22]
l5	*r*[21]
l4	*r*[20]
l3	*r*[19]
l2	*r*[18]
l1	*r*[17]
l0	*r*[16]
o7	*r*[15]
o6	*r*[14]
o5	*r*[13]
o4	*r*[12]
o3	*r*[11]
o2	*r*[10]
o1	*r*[9]
o0	*r*[8]
g7	*r*[7]
g6	*r*[6]
g5	*r*[5]
g4	*r*[4]
g3	*r*[3]
g2	*r*[2]
g1	*r*[1]
g0	*r*[0]

Figure 20: **General-purpose Registers (Nonprivileged View) (<u>V9=1</u>)**

Programming Note:

> The alternate global registers are present to give trap handlers a set of scratch registers that are independent of nonprivileged software's registers. The AG bit in PSTATE allows supervisor software to access the normal global registers if required (for example, during instruction emulation).

5.1.1.2 Windowed *r* Registers

At any time, an instruction can access the eight *global registers* and a 24-register **window** into the *r* registers. A register window comprises the eight *in* and eight *local* registers of a particular register set, together with the eight *in* registers of an adjacent register set, which

are addressable from the current window as *out* registers. See Figure 21 on page 62 and Table 8 on page 63.

Window (CWP – 1)

| r[31] |
| :   ins |
| r[24] |
| r[23] |
| :   *local*s |
| r[16] |
| r[15] |
| :   *out*s |
| r[8] |

Window (CWP)

| r[31] |
| :   ins |
| r[24] |
| r[23] |
| :   *local*s |
| r[16] |
| r[15] |
| :   *out*s |
| r[8] |

Window (CWP + 1)

| r[31] |
| :   ins |
| r[24] |
| r[23] |
| :   *local*s |
| r[16] |
| r[15] |
| :   *out*s |
| r[8] |

| r[7] |
| :   *global*s |
| r[1] |
| r[0]   0 |

63   0
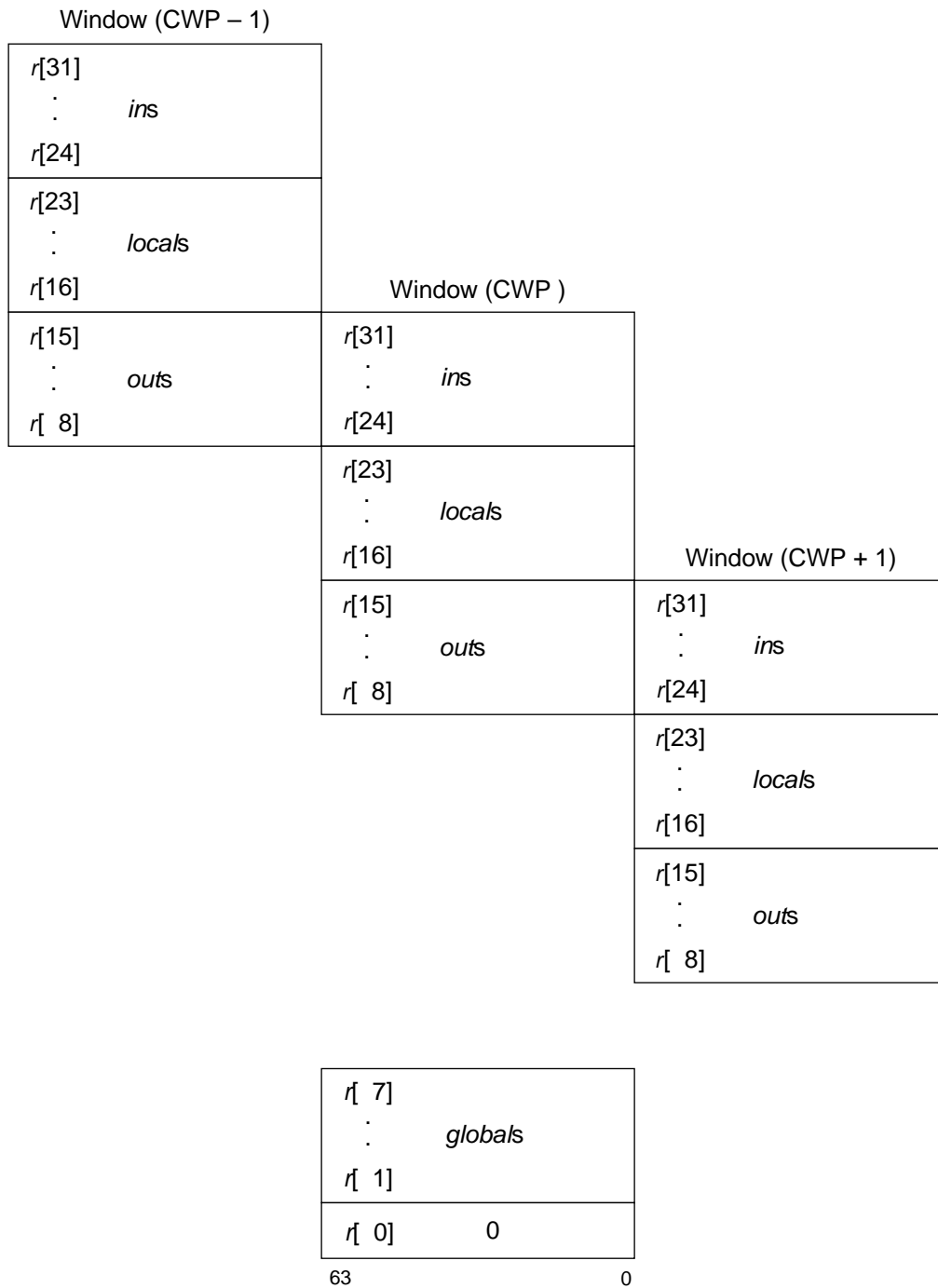
Figure 21: **Three Overlapping Windows and the Eight Global Registers (V9=2)**

The number of windows or register sets, NWINDOWS, is five for the CPU. The total number of *r* registers in a given implementation is eight (for the *global registers*), plus eight (for the alternate *global registers*), plus the number of sets times 16 registers/set. Thus, the

total number of *r* registers is 96 (five sets of 16 plus the 8 *global registers* and 8 alternate *global registers*).

Table 8: **Window Addressing (V9=6)**

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

The current window into the *r* registers is given by the current window pointer (CWP) register. The CWP is decremented by the RESTORE instruction and incremented by the SAVE instruction. Window overflow is detected via the CANSAVE register, and window underflow is detected via the CANRESTORE register, both of which are controlled by privileged software. A window overflow (underflow) condition causes a window spill (fill) trap.

5.1.1.3 Overlapping Windows

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

An *out*s 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 1 (modulo NWINDOWS). Likewise, a$_n$, *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 1 (modulo NWINDOWS). See Figure 21 on page 62 and Figure 22 on page 64.

Since CWP arithmetic is performed modulo NWINDOWS, the highest numbered implemented window (window 4 in the CPU) overlaps with window 0. The *out*s of window NWINDOWS−1 are the *in*s of window 0. Implemented windows are numbered contiguously from 0 through NWINDOWS−1 (4 in the CPU).

Programming Note:

$\triangledown$
V9

> Since the procedure call instructions (CALL and JMPL) do not change the CWP, a procedure can be called without changing the window. See H.1.2, "Leaf-Procedure Optimization," in *V9*.

> Because the windows overlap, the number of windows available to software is one less than the number of implemented windows; that is, NWINDOWS – 1 or 4 in SPARC64-III. When the register file is full, the *out*s of the newest window are the *in*s of the oldest window, which still contains valid data.

> 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, the resulting window's *local*s and *out*s may not be valid after the SAVE, since a trap may have occurred between the RESTORE and the SAVE. However, if the *clean_window* protocol is being used, system software must guarantee that registers

in the current window after a SAVE always contains only zeroes or valid data from that context. See 5.2.10.6, "Clean Windows (CLEANWIN) Register".

Section 6.3.6, "Register Window Management Instructions", describes how the windowed integer registers are managed.
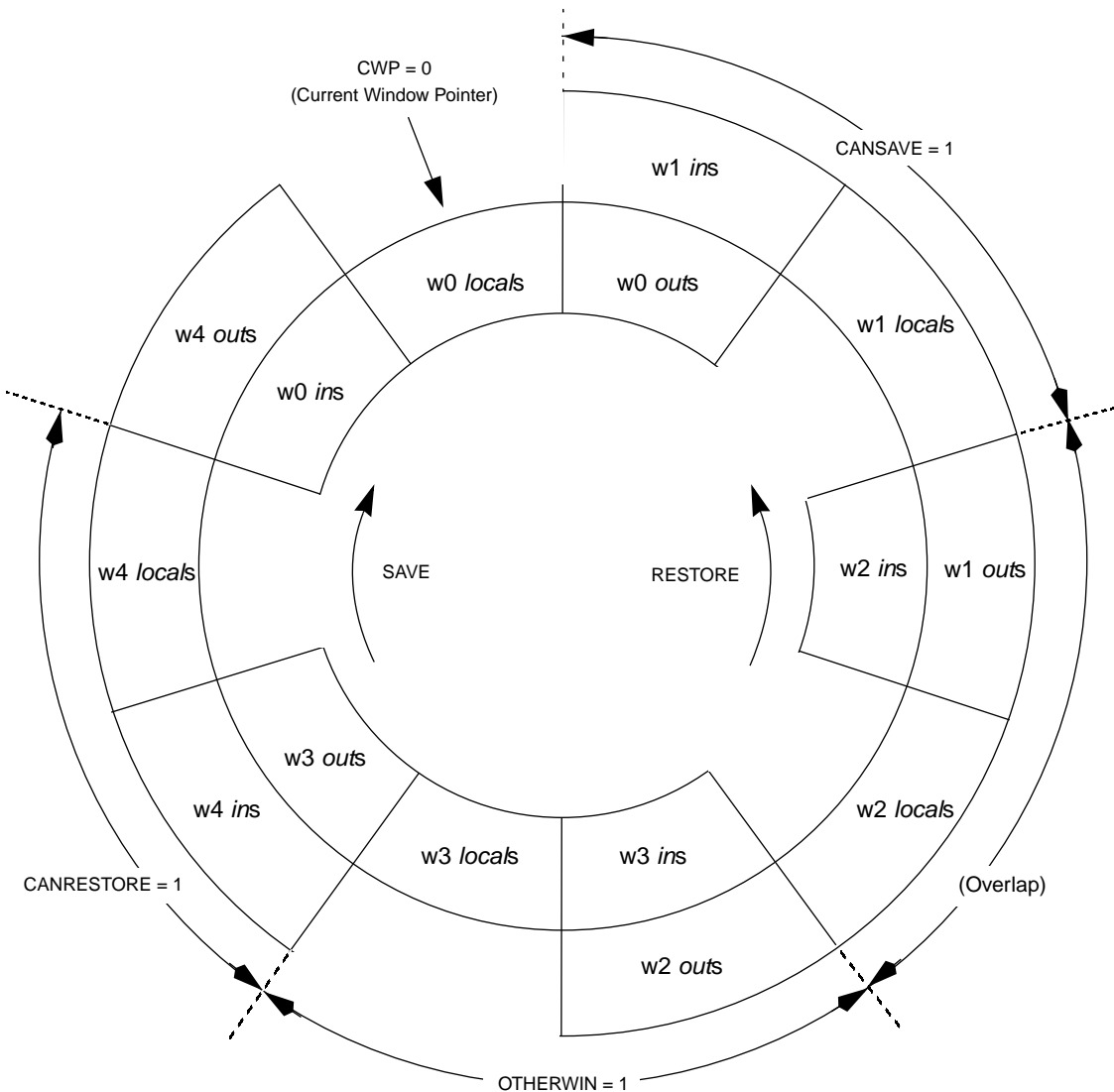
$$CANSAVE + CANRESTORE + OTHERWIN = NWINDOWS - 2$$

The current window (window 0) and the overlap window (window 2) account for the two windows in the right side of the equation. The "overlap window" is the window that must remain unused because its *ins* and *outs* overlap two other valid windows.

Figure 22: Windowed *r* Registers for NWINDOWS = 5 (V9=3)

5.1.2 Special *r* Registers

The usage of two of the *r* registers is fixed, in whole or in part, by the architecture:

■ The value of *r*[0] is always zero; writes to it have no program-visible effect.

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

5.1.2.1 Register-Pair Operands

LDD, LDDA, STD, and STDA instructions access a pair of words in adjacent *r* registers and require even-odd register alignment. The least-significant bit of an *r* register number in these instructions is reserved and should be supplied as zero by software.

When the *r*[0] – *r*[1] register pair is used as a destination in LDD or LDDA, only *r*[1] is modified. When the *r*[0] – *r*[1] register pair is used as a source in STD or STDA, a zero is written to the 32-bit word at the lowest address, and the least significant 32 bits of *r*[1] are written to the 32-bit word at the highest address (in big-endian mode).

An attempt to execute an LDD, LDDA, STD, or STDA instruction that refers to a mis-aligned (odd) destination register number causes an *illegal_instruction* trap.

5.1.2.2 Register Usage

V9 See H.1.1, "Registers" in *V9* for information about the conventional usage of the *r* registers.

In Figure 22 on page 64, NWINDOWS = 5. The eight *global register*s are not illustrated. CWP = 0, CANSAVE = 1, OTHERWIN = 1, and CANRESTORE = 1. If the procedure using window *w0* executes a RESTORE, window *w4* becomes the current window. If the procedure using window *w0* executes a SAVE, window *w1* becomes the current window.

5.1.3 IU Control/Status Registers

The nonprivileged IU control/status registers include the program counters (PC and nPC), the 32-bit multiply/divide (Y) register, and seven implementation-dependent Ancillary State Registers (ASRs), which are defined in 5.2.11, "Ancillary State Registers (ASRs)."

5.1.3.1 Program Counters (PC, nPC)

The PC contains the address of the instruction currently being executed. The 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, while the PC points to the delay instruction. See Chapter 6, "Instructions".

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 an RDPC instruction.

5.1.3.2 32-bit Multiply/Divide Register (Y)

> The Y register is deprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC-V9 software. It is recommended that all instructions that reference the Y register (that is, SMUL, SMULcc, UMUL, UMULcc, MULScc, SDIV, SDIVcc, UDIV, UDIVcc, RDY, and WRY) be avoided. See the appropriate pages in Appendix A, "Instruction Definitions", for suitable substitute instructions.

—	product<63:32> or dividend<63:32>

63   32 31   0

Figure 23: Y Register (V9=4)

The low-order 32 bits of the Y register, illustrated in Figure 23, contain the more significant word of the 64-bit product of an integer multiplication, as a result of either a 32-bit integer multiply (SMUL, SMULcc, UMUL, UMULcc) instruction or an integer multiply step (MULScc) instruction. The Y register also holds the more significant word of the 64-bit dividend for a 32-bit integer divide (SDIV, SDIVcc, UDIV, UDIVcc) instruction.

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

The Y register is read and written with the RDY and WRY instructions, respectively.

5.1.3.3 Ancillary State Registers (ASRs)

SPARC-V9 provides for optional ancillary state registers (ASRs). Access to a particular ASR may be privileged or nonprivileged; see 5.2.11, "Ancillary State Registers (ASRs)," for a more complete description of ASRs, including SPARC64-III's implementation-dependent ASRs.

5.1.4 Floating-point Registers

The FPU contains:

■ 32 single-precision (32-bit) floating-point registers, numbered $f[0], f[1], ... f[31]$.

■ 32 double-precision (64-bit) floating-point registers, numbered $f[0], f[2], ... f[62]$.

■ 16 quad-precision (128-bit) floating-point registers, numbered $f[0], f[4], ... f[60]$.

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 figures 24, 25, and 26. 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, and by load/store single/double/quad floating-point instructions.

Figure 24: Single-precision Floating-point Registers, with Aliasing (<u>V9=5</u>)

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

Figure 25: Double-precision Floating-point Registers, with Aliasing (<u>V9=6</u>)

Operand Register ID	Operand Field	From Register
f62	<63:0>	f62<63:0>
f60	<63:0>	f60<63:0>
f58	<63:0>	f58<63:0>
f56	<63:0>	f56<63:0>
f54	<63:0>	f54<63:0>
f52	<63:0>	f52<63:0>
f50	<63:0>	f50<63:0>
f48	<63:0>	f48<63:0>
f46	<63:0>	f46<63:0>
f44	<63:0>	f44<63:0>
f42	<63:0>	f42<63:0>
f40	<63:0>	f40<63:0>
f38	<63:0>	f38<63:0>
f36	<63:0>	f36<63:0>
f34	<63:0>	f34<63:0>
f32	<63:0>	f32<63:0>
f30	<31:0>	f31<31:0>
	<63:32>	f30<31:0>
f28	<31:0>	f29<31:0>
	<63:32>	f28<31:0>
f26	<31:0>	f27<31:0>
	<63:32>	f26<31:0>
f24	<31:0>	f25<31:0>
	<63:32>	f24<31:0>
f22	<31:0>	f23<31:0>
	<63:32>	f22<31:0>
f20	<31:0>	f21<31:0>
	<63:32>	f20<31:0>
f18	<31:0>	f19<31:0>
	<63:32>	f18<31:0>
f16	<31:0>	f17<31:0>
	<63:32>	f16<31:0>
f14	<31:0>	f15<31:0>
	<63:32>	f14<31:0>
f12	<31:0>	f13<31:0>
	<63:32>	f12<31:0>
f10	<31:0>	f11<31:0>
	<63:32>	f10<31:0>
f8	<31:0>	f9<31:0>
	<63:32>	f8<31:0>
f6	<31:0>	f7<31:0>
	<63:32>	f6<31:0>

Figure 25: Double-precision Floating-point Registers, with Aliasing (V9=6)

Operand Register ID	Operand Field	From Register
f4	<31:0>	f5<31:0>
	<63:32>	f4<31:0>
f2	<31:0>	f3<31:0>
	<63:32>	f2<31:0>
f0	<31:0>	f1<31:0>
	<63:32>	f0<31:0>

Figure 26: Quad-precision Floating-point Registers, with Aliasing (V9=7)

Operand Register ID	Operand Field	From Register
f60	<63:0>	f62<63:0>
	<127:64>	f60<63:0>
f56	<63:0>	f58<63:0>
	<127:64>	f56<63:0>
f52	<63:0>	f54<63:0>
	<127:64>	f52<63:0>
f48	<63:0>	f50<63:0>
	<127:64>	f48<63:0>
f44	<63:0>	f46<63:0>
	<127:64>	f44<63:0>
f40	<63:0>	f42<63:0>
	<127:64>	f40<63:0>
f36	<63:0>	f38<63:0>
	<127:64>	f36<63:0>
f32	<63:0>	f34<63:0>
	<127:64>	f32<63:0>
f28	<31:0>	f31<31:0>
	<63:32>	f30<31:0>
	<95:64>	f29<31:0>
	<127:96>	f28<31:0>
f24	<31:0>	f27<31:0>
	<63:32>	f26<31:0>
	<95:64>	f25<31:0>
	<127:96>	f24<31:0>
f20	<31:0>	f23<31:0>
	<63:32>	f22<31:0>
	<95:64>	f21<31:0>
	<127:96>	f20<31:0>
f16	<31:0>	f19<31:0>
	<63:32>	f18<31:0>
	<95:64>	f17<31:0>
	<127:96>	f16<31:0>
f12	<31:0>	f15<31:0>
	<63:32>	f14<31:0>
	<95:64>	f13<31:0>
	<127:96>	f12<31:0>
f8	<31:0>	f11<31:0>
	<63:32>	f10<31:0>
	<95:64>	f9<31:0>
	<127:96>	f8<31:0>

Figure 26: Quad-precision Floating-point Registers, with Aliasing (V9=7)

Operand Register ID	Operand Field	From Register
f4	<31:0>	f7<31:0>
	<63:32>	f6<31:0>
	<95:64>	f5<31:0>
	<127:96>	f4<31:0>
f0	<31:0>	f3<31:0>
	<63:32>	f2<31:0>
	<95:64>	f1<31:0>
	<127:96>	f0<31:0>

5.1.4.1 Floating-point Register Number Encoding

Register numbers for single, double, and quad registers are encoded differently in the 5-bit register number field of a floating-point instruction. If the bits in a register number field are labeled b<4>..b<0> (where b<4> is the most-significant bit of the register number), the encoding of floating-point register numbers into 5-bit instruction fields is as given in Table 9.

Table 9: Floating-point Register Number Encoding (V9=7)

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

Compatibility Note:

> In SPARC-V8, bit 0 of double and quad register numbers encoded in instruction fields was required to be zero. Therefore, all SPARC-V8 floating-point instructions can run unchanged on the SPARC64-III using the encoding in Table 9.

5.1.4.2 Double and Quad Floating-point Operands

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

Programming Note:

> 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) using single-precision

LDF instructions. If desired, it can then be copied into the upper 16 double registers (8 quad registers).

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

Programming Note:

Given the encoding in <u>Table 9</u>, it is impossible to specify a misaligned double-precision register.

The CPU does not handle quad-precision operands in hardware. All SPARC-V9 FP operations trap to the OS kernel and are emulated. Quad-precision multiply-add and multiply-subtract are not emulated.

5.1.5 Condition Codes Register (CCR)
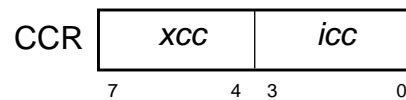
Figure 27: Condition Codes Register (<u>V9=8</u>)

The Condition Codes Register (CCR), shown in <u>Figure 27</u>, holds the integer condition codes.

5.1.5.1 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 1), but the 64-bit result is nonnegative (*xcc*.N is set to 0).

Each of the 4-bit condition-code fields is composed of four 1-bit subfields, as shown in <u>Figure 28</u>.
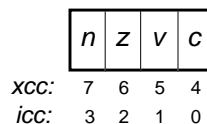
Figure 28: Integer Condition Codes (CCR_icc and CCR_xcc) (<u>V9=9</u>)

The *n* bits indicate whether the 2's-complement ALU result was negative for the last instruction that modified the integer condition codes. 1 = negative, 0 = not negative.

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

The *v* bits indicate whether the ALU result was within the range of (was representable in) 64-bit (*xcc*) or 32-bit (*icc*) 2's complement notation for the last instruction that modified the integer condition codes. 1 = overflow, 0 = no overflow.

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

5.1.5.1.1 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 considered to be 64 bits long. 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 based on the state of these bits.

5.1.5.1.2 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 considered to be 32 bits. 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, Bicc, 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 based on the state of these bits.

5.1.6 Floating-point Registers State (FPRS) Register

FPRS | FEF | DU | DL |
         2   1   0

Figure 29: **Floating-point Registers State Register (V9=10)**

The Floating-point Registers State (FPRS) register, shown in Figure 29, holds control information for the floating-point register file; this information is readable and writable by nonprivileged software.

5.1.6.1 FPRS_enable_fp (FEF)

Bit 2, FEF, determines whether the FPU is enabled. If it is disabled, executing a floating-point instruction causes an *fp_disabled* trap. If this bit is set but the PSTATE.PEF bit is not

set, then executing a floating-point instruction causes an *fp_disabled* trap; that is, both FPRS.FEF and PSTATE.PEF must be set to enable floating-point operations.

5.1.6.2 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. Its setting in the CPU is pessimistic; it is set whenever a floating-point instruction is issued, but if that instruction never completes, no output register is modified. The DU bit is cleared only by software.

5.1.6.3 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. Its setting in the CPU is pessimistic; it is set whenever a floating-point instruction is issued, but if that instruction never completes, no output register is modified. The DL bit is cleared only by software.

Implementation Note:
> SPARC64-III sets FPRS.DL and FPRS.DU pessimistically. Specifically, they are set whenever an instruction that might change one of the floating-point registers is *issued*. In some cases, the issued instruction is never committed, and the destination register is not actually changed.

5.1.7 Floating-point State Register (FSR)

The FSR register fields, illustrated in Figure 30, contain FPU mode and status information. The lower 32 bits of the FSR are read and written by the STFSR and LDFSR instructions; all 64 bits of the FSR are read and written by the STXFSR and LDXFSR instructions, respectively. The *ver*, *ftt*, and *reserved* fields are not modified by LDFSR or LDXFSR.

—														fcc3	fcc2	fcc1
63													38	37 36	35 34	33 32

RD	—	TEM	NS	—	ver	ftt	qne	—	fcc0	aexc	cexc
31 30	29 28	27 23	22	21 20	19 17	16 14	13	12	11 10	9 5	4 0
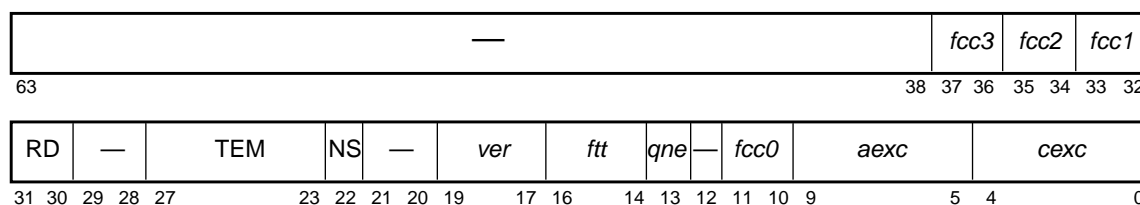
Figure 30: FSR Fields (V9=11)

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 STFSR.

Subsections 5.1.7.1 through 5.1.7.10.5 describe the remaining fields in the FSR.

5.1.7.1 FSR_fp_condition_codes (*fcc0, fcc1, fcc2, fcc3*)

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

Compatibility Note:

SPARC-V9's *fcc0* is the same as SPARC-V8'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 *fccn* fields are read and written by STXFSR and LDXFSR instructions, respectively. The *fcc0* field may 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 10, f_{rs1} and f_{rs2} correspond to the single, double, or quad values in the floating-point registers specified by a floating-point compare instruction's *rs1* and *rs2* fields. The question mark ('?') indicates an unordered relation, which is true if either f_{rs1} or f_{rs2} is a signalling NaN or quiet NaN. If FCMP or FCMPE generates an *fp_exception_ieee_754* exception, then *fccn* is unchanged.

Table 10: **Floating-point Condition Codes (*fccn*) Fields of FSR (V9=8)**

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*)

5.1.7.2 FSR_rounding_direction (RD)

Bits 31 and 30 select the rounding direction for floating-point results according to IEEE Std 754-1985. Table 11 shows the encodings.

Table 11: **Rounding Direction (RD) Field of FSR (V9=9)**

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

5.1.7.3 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 31 on page 80. If a floating-point operate instruction generates one or more exceptions and the TEM bit corresponding to any of the exceptions is 1, an *fp_exception_ieee_754* trap is caused. A TEM bit value of 0 prevents the corresponding exception type from generating a trap.

5.1.7.4 FSR_nonstandard_fp (NS)

SPARC-V9 defines the FSR.NS bit which, when set to 1, causes the FPU to produce implementation-defined results that may not correspond to IEEE Std 754-1985. SPARC64-III does not need and therefore does not implement any nonstandard floating-point functionality. Writes to FSR.NS are ignored; reads from FSR.NS always return zero. See *V9* for more information about FSR.NS and nonstandard floating-point operation in SPARC-V9.

Implementation Note:
> The multiply-add and multiply-subtract instructions are considered to be extensions to the SPARC-V9 architecture and not nonstandard floating-point behavior.

5.1.7.5 FSR_version (*ver*)

For each SPARC-V9 IU implementation (as identified by its VER.*impl* field), there may be one or more FPU implementations, or none. This field identifies the particular FPU implementation present. For the first SPARC64-III, FSR.*ver* = 0; however, future versions of the architecture may set FSR.*ver* to other values. Consult the SPARC64-III Data Sheet, which is described in the Bibliography, for the setting of FSR.*ver* for your chipset.

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

5.1.7.6 FSR_floating-point_trap_type (*ftt*)

Several conditions can cause a floating-point exception trap. 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." After a floating-point exception occurs, the *ftt* field encodes the type of the floating-point exception until an STFSR or an FPop is executed.

The *ftt* field can be read by the STFSR and STXFSR instructions. The LDFSR and LDXFSR instructions do not affect *ftt*.

Privileged software that handles floating-point traps must execute an STFSR (or STXFSR) to determine the floating-point trap type. STFSR and STXFSR shall zero *ftt* 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 this purpose, since both leave *ftt* unchanged. However, executing a nontrapping FPop such as "fmovs %f0,%f0" prior to returning to nonprivileged 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 12. **Note**: The value "7" is reserved for future expansion.

Table 12: Floating-point Trap Type (*ftt*) Field of FSR (V9=10)

ftt	Trap Type	Trap Vector	SPARC64-III Action
0	None	No trap taken	No trap taken
1	*IEEE_754_exception*	*fp_exception_ieee_754*	As described in *V9*
2	*unfinished_FPop*	*fp_exception_other*	See 5.1.7.6.2
3	*unimplemented_FPop*	*fp_exception_other*	See 5.1.7.6.3
4	*sequence_error*	—	Does not occur in SPARC64-III
5	*hardware_error*	—	Does not occur in SPARC64-III
6	*invalid_fp_register*	—	Does not occur in SPARC64-III
7	*reseved*	—	Does not occur in SPARC64-III

The *sequence_error, hardware_error,* and *invalid_fp_register* never occur in SPARC64-III. In contrast, *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.

2. The value of *cexc* is unchanged, except that for an *IEEE_754_exception* 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*.

3. The source registers are unchanged (usually implemented by leaving the destination registers unchanged).

4. The value of *fccn* is unchanged.

The foregoing describes the result seen by a user trap handler if an IEEE exception is signalled, either immediately from an *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 *unfinished_FPop* and *unimplemented_FPop* exceptions that do not subsequently generate IEEE traps, the recovery software should define *cexc*, *aexc*, and the destination registers or *fcc*s, as appropriate.

5.1.7.6.1   *ftt* = IEEE_754_exception

The *IEEE_754_exception* floating-point trap type indicates that a floating-point exception conforming to IEEE Std 754-1985 has occurred. The exception type is encoded in the *cexc* field. **Note**: The *aexc* and *fcc*s fields and the destination *f* register are not affected by an *IEEE_754_exception* trap.

5.1.7.6.2   *ftt* = unfinished_FPop

The *unfinished_FPop* floating-point trap type indicates that the CPU 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 following paragraphs discuss when the *unfinished_FPop* trap type can occur.

DIVIDE:

■ Either or both source operands are denormalized numbers in certain cases,

■ The result would have been a denormalized number

In these cases, the result register(s) will not be written by hardware. Also the FSR.cexc field will not be updated by hardware. The kernel trap routine will calculate the divide result and store it in the destination register and correctly set the FSR.cexc bits. If either operand has a special value (zero, infinity, NaN) or the result can be calculated based only on the exponent value, the CPU will return the correct special result.

SQUARE ROOT:

■ The source operand is a positive denormalized number.

In this case, the result register(s) will not be written by hardware. Also the FSR.cexc field will not be updated by hardware. The kernel trap routine will calculate the square root result and store it in the destination register and correctly set the FSR.cexc bits.

5.1.7.6.3   *ftt* = unimplemented_FPop

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

All quad FPops variations set *ftt* = *unimplemented_FPop*.

5.1.7.6.4   *ftt* = sequence_error

The *sequence_error* floating-point trap type can never occur on the CPU processor. See *V9* for more information about this trap.

5.1.7.6.5   *ftt* = hardware_error

The *hardware_error* floating-point trap type can never occur on the CPU processor. See *V9* for more information about this trap.

5.1.7.6.6   *ftt* = invalid_fp_register

Since the CPU does not implement any quad-precision operations in hardware, this error can never occur. Quad operations set *ftt* = *unimplemented_FPop*. System software checks for a valid quad register specifier during its emulation of the instruction.

5.1.7.7 FSR_FQ_not_empty (*qne*)

All floating-point traps on the CPU are precise; therefore, the CPU does not need or contain a Floating-point Deferred-trap Queue. Thus, writes to *qne* are ignored and reads from it always return zero.

5.1.7.8 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 32 on page 80. After an FPop is reclaimed with *ftt* = 0, the TEM and *cexc* fields are logically ANDed together. If the result is nonzero, *aexc* is left unchanged and an *fp_exception_ieee_754* trap is generated; otherwise, the new *cexc* field is ORed into the *aexc* field and no trap is generated. Thus, while (and only while) traps are masked, exceptions are accumulated in the *aexc* field.

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

5.1.7.9 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 33 on page 80.

On the CPU the *cexc* bits are set according the following pseudo-code:

```
if (<LDFSR or LDXFSR commits>)
    <update using data from LDFSR or LDXFSR>;
else if (<FPop commits with ftt == 0>)
    <update using value from FPU>
else if (<FPop commits with IEEE_754_exception>)
    <set one bit in the CEXC field as supplied by FPU>;
else if (<FPop commits with unfinished_FPop error>)
    <see 5.1.7.6.2 for details>;
else if (<FPop commits with unimplemented_FPop error>)
    <see 5.1.7.6.3 for details>;
else
    <no change>;
```

Note: If the FPop traps and software emulates or finishes the instruction, the system software in the trap handler is responsible for creating a correct FSR.*cexc* value before returning to a nonprivileged program.

If the execution of an FPop causes a trap other than *fp_exception_ieee_754* due to an IEEE Std 754-1985 exception, FSR.*cexc* is left unchanged.

5.1.7.10 Floating-point Exception Fields

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

NVM	OFM	UFM	DZM	NXM
27	26	25	24	23
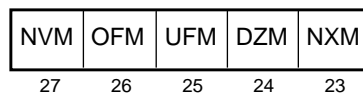
Figure 31: Trap Enable Mask (TEM) Fields of FSR (V9=12)

nva	ofa	ufa	dza	nxa
9	8	7	6	5

Figure 32: Accrued Exception Bits (*aexc*) Fields of FSR (V9=13)

nvc	ofc	ufc	dzc	nxc
4	3	2	1	0
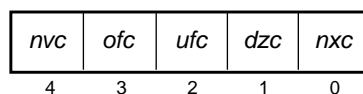
Figure 33: Current Exception Bits (*cexc*) Fields of FSR (V9=14)

5.1.7.10.1 FSR_invalid (*nvc, nva*)

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

5.1.7.10.2 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.

5.1.7.10.3 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 tininess to be detected either before or after rounding. In all cases and regardless of the setting of UFM, the CPU detects tininess before rounding.

5.1.7.10.4 FSR_division-by-zero (*dzc*, *dza*)

X ÷ 0.0, where X is subnormal or normalized. **Note**: 0.0 ÷ 0.0 **does not** set the *dzc* or *dza* bits. 1 = division by zero, 0 = no division by zero.

5.1.7.10.5 FSR_inexact (*nxc*, *nxa*)

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

5.1.7.11 FSR Conformance

SPARC-V9 allows the TEM, *cexc*, and *aexc* fields to be implemented in hardware in either of two ways (both of which comply with IEEE Std 754-1985). The CPU follows case (1); that is, it implements all three fields in conformance with IEEE Std 754-1985. See 5.1.7.11 in *V9* for more information about other implementation methods.

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.

5.1.8 Address Space Identifier (ASI) Register

ASI

```
7                    0
```
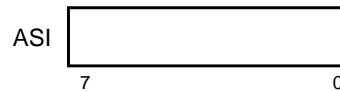
Figure 34: ASI Register (V9=15)

The ASI register specifies the address space identifier to be used for load and store alternate instructions that use the "*rs1 + simm13*" addressing form. Nonprivileged (user-mode) software may write any value into the ASI register; however, values with bit 7 = 0 indicate restricted ASIs. When a nonprivileged instruction makes an access that uses an ASI with bit 7 = 0, a *privileged_action* exception is generated. See 6.3.1.3, "Address Space Identifiers (ASIs)",for details.

5.1.9 Tick (TICK) Register

TICK NPT   *counter*

```
      63   62                                                                    0
```
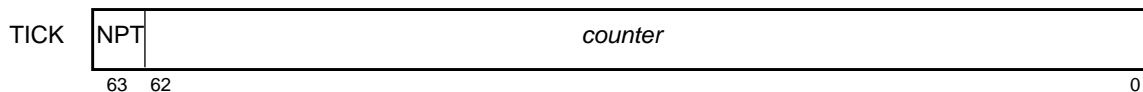
Figure 35: TICK Register (V9=16)

The *counter* field of the TICK register is a 63-bit counter that counts CPU clock cycles. Bit 63 of the TICK register is the Nonprivileged Trap (NPT) bit, which controls access to the TICK register by nonprivileged software. Privileged software can always read the

TICK register with either the RDPR or RDTICK instruction. Privileged software can always write the TICK register with the WRPR instruction; there is no WRTICK instruction.

Nonprivileged software can read the TICK register using the RDTICK instruction when TICK.NPT is 0. When TICK.NPT = 1, an attempt by nonprivileged software to read the TICK register causes a *privileged_action* exception. Nonprivileged software cannot write the TICK register.

TICK.NPT is set to 1 by a power-on reset trap. The value of TICK.*counter* is undefined 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 do 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.

Implementation Note:

> On the SPARC64-III the value returned when the TICK register is read is the value of TICK.*counter* when the RDTICK instruction is *issued*. The difference between the values read from the TICK register on two reads reflects the number of processor cycles executed between the *issues* of the RDTICK instructions, not their *commits*. In longer code sequences, the difference between this value and the value obtained when the instructions are committed will be small.

Programming Note:

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

5.2 Privileged Registers

The registers described in this subsection are visible only to software running in privileged mode; that is, when PSTATE.PRIV = 1. Privileged registers are written using the WRPR instruction and read using the RDPR instruction.

5.2.1 Processor State Register (PSTATE)

PSTATE	CLE	TLE	MM		RED	PEF	AM	PRIV	IE	AG
	9	8	7	6	5	4	3	2	1	0
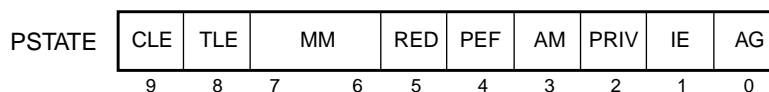
Figure 36: PSTATE Fields (V9=17)

The PSTATE register holds the current state of the processor. There is only one instance of the PSTATE register. See Chapter 7, "Traps", for more details.

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 PSTATE, respectively.

Subsections 5.2.1.1 through 5.2.1.10 describe the fields contained in the PSTATE register.

5.2.1.1 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.

5.2.1.2 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 allows system software to have a different implicit byte ordering than the current process. Thus, if PSTATE.TLE is set to 1, data accesses using an implicit ASI in the trap handler are little-endian. The original state of PSTATE.CLE is restored when the original PSTATE register is restored from the trap stack.

5.2.1.3 PSTATE_mem_model (MM)

This 2-bit field determines the memory model in use by the processor. The SPARC64-III values and the corresponding values defined in SPARC-V9 are shown in Table 13:

<p align="center">Table 13: MM Encodings</p>

MM Value	SPARC64-III Memory Model	SPARC-V9
00	Load/Store Order (LSO)	TSO
01	Total Store Order (TSO)	PSO
10	Store Order (STO)	RMO
11	*reserved*	*reserved*

The current memory model is determined by the value of PSTATE.MM. For more information about how the SPARC64-III determines the memory model, see Chapter 8, "Memory Models". Software should always refrain from using the combination '11', since it is *reserved* for future SPARC-V9 extensions.

Load/Store Order (LSO):

> Loads and stores are ordered with loads and stores. Thus, loads and stores   cannot bypass earlier loads and stores. Load Store Order in SPARC64-III is   stronger than Total Store Order in SPARC-V9.

Total Store Order (TSO):

> Loads are ordered with loads. Stores are ordered with loads and stores. Thus, loads can bypass earlier stores, but cannot bypass earlier loads; Stores   cannot

bypass earlier loads and stores. Total Store Order in SPARC64-III is   stronger than Total Store Order and Partial Store Order in SPARC-V9.

Store Order (STO):

Stores are ordered with loads and stores. Thus, loads can bypass earlier   loads and stores; Stores cannot bypass earlier loads and stores. Store Order   in SPARC64-III is stronger than Relaxed Memory Order in SPARC-V9.

5.2.1.4 PSTATE_RED_state (RED)

PSTATE.RED (**R**eset, **E**rror, and **D**ebug state) is set whenever the SPARC64-III processor takes a RED state disruptive or nondisruptive trap. See 7.2.1, "RED_state". It can also be set by software using a WRPR %PSTATE or WRPR %TSTATE, followed by a DONE or RETRY. Software can exit RED_state by one of two methods:

1. Execute a DONE or RETRY instruction, which restores the stacked copy of PSTATE and clears PSTATE.RED if it was 0 in the stacked copy.

2. Write a 0 to PSTATE.RED with a WRPR instruction.

Programming Note:

Changing PSTATE.RED may cause a change in address mapping on some systems. It is recommended that writes of PSTATE.RED be placed in the delay slot of a JMPL; the target of this JMPL should be in the new address mapping. The JMPL sets the nPC, which becomes the PC for the instruction that follows the WPR in its delay slot. The effect of the WPR instruction is immediate.

When the CPU enters RED_state, the I0 cache and prefetch buffers are invalidated. See Chapter 7, "Traps", for more details.

5.2.1.5 PSTATE_enable_floating-point (PEF)

When set to 1, this bit enables the floating-point unit, 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 (including the SPARC64-III-specific multiply-add and multiply-subtract instructions) that tries to reference the FPU causes an *fp_disabled* trap.

5.2.1.6 PSTATE_address_mask (AM)

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

■ Before data addresses are sent out of the CPU

■ For instruction accesses to internal or external caches

■ Before being used to detect a data breakpoint (ASR #26—see 5.2.11.9)

The following cases disregard the setting of PSTATE.AM:

■ When an exception occurs, all 64 bits are stored in the Data Fault Address Register (ASR #28). See 5.2.11.10.1, "Data Access Fault Address Register (ASR28)".

■ CALL, JMPL, and RDPC always transmit the full 64-bit address to the specified destination registers.

■ Traps always transmit all 64 bits of the PC and NPC to TPC[*n*] and TNPC[*n*] respectively.

Thirty-two-bit application software must run with this bit set.

5.2.1.7 PSTATE_privileged_mode (PRIV)

When PSTATE.PRIV = 1, the processor is in privileged mode.

5.2.1.8 PSTATE_interrupt_enable (IE)

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

5.2.1.9 PSTATE_alternate_globals (AG)

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

5.2.1.10 PSTATE_interrupt_globals_enable, PSTATE_MMU_globals_enable

These bits, which are defined for Sun Microsystems' UltraSPARC processor, are not part of the SPARC-V9 standard and are not defined for the SPARC64-III.

5.2.2 Trap Level Register (TL)

TL

TL

2   0
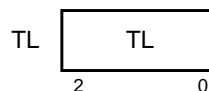
Figure 37: Trap Level Register (V9=18)

The trap level register 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," which is always equal to the number of supported trap levels beyond level 0. See Chapter 7, "Traps", for more details about the TL register. The SPARC64-III CPU supports four trap levels beyond level 0; that is, MAXTL = 4.

Programming Notes:
Writing the TL register with a value of 5, 6, or 7 causes the value 4 to be written.

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

SPARC64-III renames trap levels, allowing speculative entrance into common exceptions such as window spill/fill traps. Four extra trap levels are available for renaming, making a total of 8 trap levels. Due to pipelining effects and in order to reduce complexity, the 8th trap level is only usable at certain times.

When only 2 trap levels are available for renaming and back-to-back traps are encountered, the second trap will stall until the processor pipeline is able to update resource information.

5.2.3 Processor Interrupt Level (PIL) Register

Figure 38: Processor Interrupt Level Register (V9=19)

The processor interrupt level (PIL) 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. See Table 29 on page 149 for a list of exception and interrupt priorities.

Compatibility Note:

> On SPARC-V8 processors, the level 15 interrupt is considered to be nonmaskable, so it has different semantics from other interrupt levels. SPARC-V9 processors do not treat level 15 interrupts differently from other interrupt levels. See 7.6.2.4, "Externally Initiated Reset (XIR) Traps", for a facility in SPARC-V9 that is similar to a nonmaskable interrupt.

The CPU exhibits some special timing associated with the PIL register, which is related to the fact that the PIL register has only one level of register renaming. If a WRPR to the PIL is issued using a register other than %g0, a pending write to the PIL is generated. When such a pending write exists, the following instructions stall the machine:

■ rdpr %pil

■ wrpr %pil

In addition, while a write to the PIL is pending, all interrupts (INTR requests) are ignored. This pending write exists until the CPU calculates a new value for the PIL. The actual time that interrupts are ignored in practice is only a few cycles at most.

The following instruction writes the *immed_val* directly into the PIL and **does not** cause a pending write to the PIL; thus, RDPRs and WRPRs that occur later in the code do not stall the machine.

■ wrpr %g0, *immed_val*, %pil

5.2.4 Trap Program Counter (TPC) Register

TPC$_1$	PC from trap while TL = 0	00
TPC$_2$	PC from trap while TL = 1	00
TPC$_3$	PC from trap while TL = 2	00
TPC$_4$	PC from trap while TL = 3	00

63   2 1 0

Figure 39: Trap Program Counter Register (V9=20)

The TPC register contains the program counter (PC) from the previous trap level. There are MAXTL instances (four in SPARC64-III) of the TPC, but only one is accessible at any time. The current value in the TL register determines which instance of the TPC register is accessible. An attempt to read or write the TPC register when TL = 0 causes an *illegal_ instruction* exception.

On SPARC64-III all 64 bits of the PC are written into TPC on a trap, regardless of the setting of PSTATE.AM.

After a power-on reset the contents of TPC[1..4] are undefined. During normal operation the value of TPC[n], when n is greater than the current trap level ($n > $ TL), is undefined.

5.2.5 Trap Next Program Counter (TNPC) Register

TNPC$_1$	nPC from trap while TL = 0	00
TNPC$_2$	nPC from trap while TL = 1	00
TNPC$_3$	nPC from trap while TL = 2	00
TNPC$_4$	nPC from trap while TL = 3	00

63   2 1 0

Figure 40: Trap Next Program Counter Register (V9=21)

The TNPC register, shown in Figure 40, is the next program counter (nPC) from the previous trap level. There are MAXTL instances (four in SPARC64-III) of the TNPC, but only one is accessible at any time. The current value in the TL register determines which instance of the TNPC register is accessible. An attempt to read or write the TNPC register when TL = 0 causes an *illegal_instruction* exception.

On SPARC64-III all 64 bits of the NPC are written into TNPC on a trap, regardless of the setting of PSTATE.AM.

After a power-on reset the contents of TNPC[1..4] are undefined. During normal operation the value of TNPC[n], when n is greater than the current trap level ($n > $ TL), is undefined.

5.2.6 Trap State (TSTATE) Register

	CCR from TL = 0	ASI from TL = 0	—	PSTATE from TL = 0	—	CWP from TL = 0
TSTATE$_1$	CCR from TL = 0	ASI from TL = 0	—	PSTATE from TL = 0	—	CWP from TL = 0
TSTATE$_2$	CCR from TL = 1	ASI from TL = 1	—	PSTATE from TL = 1	—	CWP from TL = 1
TSTATE$_3$	CCR from TL = 2	ASI from TL = 2	—	PSTATE from TL = 2	—	CWP from TL = 2
TSTATE$_4$	CCR from TL = 3	ASI from TL = 3	—	PSTATE from TL = 3	—	CWP from TL = 3

```
39        32 31        24 23       18 17              8 7  5 4          0
```

Figure 41: Trap State Register (V9=22)

The Trap State (TSTATE) Register, shown in Figure 41, contains the state from the previous trap level, comprising the contents of the CCR, ASI, CWP, and PSTATE registers from the previous trap level. There are MAXTL instances (four in SPARC64-III) of the TSTATE register, but only one is accessible at a time. The current value in the TL register determines which instance of TSTATE is accessible. An attempt to read or write the TSTATE register when TL = 0 causes an *illegal_instruction* exception.

Since the CCR is renamed on the CPU, a write to TSTATE due to a trap may need to wait for a few cycles for the appropriate copy of CCR to be available. Because it is not desirable to serialize the machine in this case, the write to TSTATE "completes" but the CCR value is allowed to arrive late. If the issue unit encounters a DONE, RETRY, or rdpr %tstate instruction before the CCR value has been stored in TSTATE.CCR, the machine stalls until the value arrives. Stalling is not expected to be a problem in normal code, since the CCR normally arrives within a few cycles.

The CPU implements only bits 2..0 of the TSTATE.CWP field. Writes to bits 4 and 3 are ignored and reads of these bits always return zeroes.

Note:

> Since a certain class of exceptions (issue traps) may be entered and completed speculatively, modification of certain fields in the TSTATE register by privileged software will result in speculative restoration of control registers. For example, if the ASI field in TSTATE contained ASI_PO and it were changed to ASI_PRIMARY, upon a DONE instruction, all succeeding LD %asi will not be in program order. This may be dangerous if these load instructions are to I/O space.

Note:

> Spurious setting of the PSTATE.RED bit by privileged software should not be performed, since it will take the SPARC64-III into RED mode without the required sequencing.

After a power-on reset the contents of TSTATE[1..4] are undefined. During normal operation the value of TSTATE[*n*], when *n* is greater than the current trap level (*n* > TL), is undefined.

5.2.7 Trap Type (TT) Register

TT_1 | Trap type from trap while TL = 0
TT_2 | Trap type from trap while TL = 1
TT_3 | Trap type from trap while TL = 2
TT_4 | Trap type from trap while TL = 3

8   0

Figure 42: Trap Type Register (V9=23)

The TT register normally contains the trap type of the trap that caused entry to the current trap level. On a reset trap the TT field contains the trap type of the reset (see 7.2.1.1, "RED_state Trap Table").

There are MAXTL instances of the TT register (four in SPARC64-III), but only one is accessible at a time. The current value in the TL register determines which instance of the TT register is accessible. An attempt to read or write the TT register when TL = 0 causes an *illegal_instruction* exception.

After a power-on reset the contents of TT[1..3] are undefined; TT[4] = 1. During normal operation the value of TT[n], when n is greater than the current trap level (n > TL) is undefined.

5.2.8 Trap Base Address (TBA) Register

Trap Base Address	000000000000000

63   15 14   0

Figure 43: Trap Base Address Register (V9=24)

The TBA register, shown in Figure 43, provides the upper 49 bits of the address used to select the trap vector for a trap. The lower 15 bits of the TBA always read as zero, and writes to them are ignored.

The full address for a trap vector is specified by the TBA, TL, TT[TL], and five zeroes:

TBA<63:15>	TL>0	TT_{TL}	00000

63   15 14 13   5 4   0

Figure 44: Trap Vector Address Register (V9=25)

Note: The "(TL>0)" bit is 0 if TL = 0 when the trap was taken, and 1 if TL > 0 when the trap was taken. This implies that there are two trap tables: one for traps from TL = 0 and one for traps from TL > 0. See Chapter 7, "Traps", for more details on trap vectors.

5.2.9 Version (VER) Register

manuf	impl	mask	—	maxtl	—	maxwin

63   48 47   32 31   24 23   16 15   8 7 5 4   0

Figure 45: **Version Register (V9=26)**

The version register, shown in Figure 45, specifies the fixed parameters pertaining to a par-
ticular CPU implementation and mask set. The VER register is read-only. Table 14 shows
the values for the VER register for SPARC64-III:

Table 14: VER Register Encodings

Field	Value
manuf	0004_{16}
impl	3
mask	n [a]
maxtl	4
maxwin	4

[a] The value of *n* depends on
   the processor chip version.

The *manuf* field contains Fujitsu's 8-bit JEDEC code in the lower 8 bits and zeroes in the
upper 8 bits. The *manuf, impl*, and *mask* fields are implemented so that they may change in
future CPU versions. The *mask* field is incremented by one any time a programmer-visible
revision is made to the CPU. See the SPARC64-III Data Sheet, described in the Bibliogra-
phy, to determine the current setting of the *mask* field.

The CPU also contains an 8-bit revision register which is visible via scan during bringup.
The revision register will be integrated into the command register of each chip's Test
Access Port (TAP) unit.

5.2.10 Register-Window State Registers

The state of the register windows is determined by a set of privileged registers. They can
be read/written by privileged software using the RDPR/WRPR instructions. In addition,
these registers are modified by instructions related to register windows and are used to
generate traps that allow supervisor software to spill, fill, and clean register windows.

The details of how the window-management registers are used by hardware are presented
in 6.3.6, "Register Window Management Instructions".

Implementation Note:

> Because NWINDOWS = 5 in SPARC64-III, only the lower 3 bits are implemented in the CWP,
> CANSAVE, CANRESTORE, and OTHERWIN registers described in the following subsections.
> When any of these registers is moved into a 64-bit integer register with an RDPR instruction, the
> upper 61 bits are set to zero.

Programming Note:

> CANSAVE, CANRESTORE, and OTHERWIN must never be set to 4. Setting any of these to 4 violates the register window state definition in 6.4.1, "Register Window State Definition". Notice that hardware does not enforce this restriction; it is up to system software to keep the window state consistent.

5.2.10.1 Current Window Pointer (CWP) Register
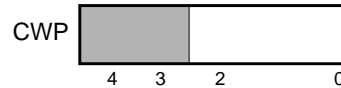
Figure 46: Current Window Pointer Register (V9=27)

The CWP register, shown in Figure 46, is a counter that identifies the current window into the set of integer registers. See 6.3.6, "Register Window Management Instructions", and Chapter 7, "Traps", for information on how hardware manipulates the CWP register.

Compatibility Note:

> The following differences between SPARC-V8 and SPARC-V9 are visible only to privileged software; they are invisible to nonprivileged software:

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

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

3) In SPARC-V8, writing a value into PSR.CWP that is greater than or equal to the number of implemented windows causes an *illegal_instruction* exception. In SPARC-V9, the effect of writing an out-of-range value to CWP is undefined.

5.2.10.2 Savable Windows (CANSAVE) Register

Figure 47: CANSAVE Register (V9=28)

The CANSAVE register, shown in Figure 47, contains the number of register windows following CWP that are not in use and are, hence, available to be allocated by a SAVE instruction without generating a window spill exception

5.2.10.3 Restorable Windows (CANRESTORE) Register

Figure 48: CANRESTORE Register (V9=29)

The CANRESTORE, shown in Figure 48, register contains the number of register windows preceding CWP that are in use by the current program and can be restored (via the RESTORE instruction) without generating a window fill exception.

5.2.10.4 Other Windows (OTHERWIN) Register

OTHERWIN

```
4        3 2        0
```

Figure 49: OTHERWIN Register (V9=30)

The OTHERWIN register, shown in Figure 49, contains the count of register windows that will be spilled/filled using a separate set of trap vectors based on the contents of WSTATE_OTHER. If OTHERWIN is zero, register windows are spilled/filled using trap vectors based on the contents of WSTATE_NORMAL.

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

5.2.10.5 Window State (WSTATE) Register

WSTATE

| OTHER | NORMAL |
```
5        3 2        0
```
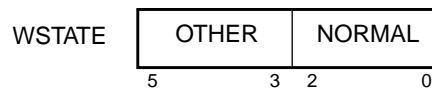
Figure 50: WSTATE Register (V9=31)

The WSTATE register, shown in Figure 50, specifies bits that are inserted into $TT_{TL}<4:2>$ on traps caused by window spill and fill exceptions. These bits are used to select one of eight different window spill and fill handlers. If OTHERWIN = 0 at the time a trap is taken due to a window spill or window fill exception, then the WSTATE.NORMAL bits are inserted into TT[TL]. Otherwise, the WSTATE.OTHER bits are inserted into TT[TL]. See 6.4, "Register Window Management", for details of the semantics of OTHERWIN.

5.2.10.6 Clean Windows (CLEANWIN) Register

CLEANWIN

```
4        3        2        0
```
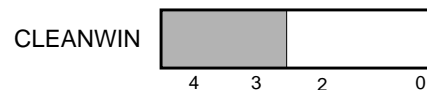
Figure 51: CLEANWIN Register (V9=32)

The CLEANWIN register, shown in Figure 51, contains the number of windows that can be used by the SAVE instruction without causing a *clean_window* exception.

The CLEANWIN register counts the number of register windows that are "clean" with respect to the current program; that is, register windows that contain only zeros, valid addresses, or valid data from that program. Registers in these windows need not be cleaned before they can be used. The count includes the register windows that can be

restored (the value in the CANRESTORE register) and the register windows following CWP that can be used without cleaning. When a clean window is requested (via a dSAVE instruction) and none is available, a *clean_window* exception occurs to cause the next window to be cleaned.

Implementation Note:
> Only the lower three bits are implemented in the CLEANWIN register. When this register is moved into a 64-bit integer register with an RDPR instruction, the upper 61 bits are set to zero.

Programming Note:
> CLEANWIN must never be set to 6 or 7. Setting CLEANWIN > 5 would violate the register window state definition in 6.4.1, "Register Window State Definition". **Note**: Hardware does not enforce this restriction; it is up to system software to keep the window state consistent.

5.2.11 Ancillary State Registers (ASRs)

The SPARC-V9 architecture provides for up to 25 ancillary state registers (ASRs), numbered from 7 through 31. SPARC64-III implements 13 ASRs.

ASRs numbered 7..15 are reserved for future use by the architecture and should not be referenced by software.

The SPARC-V9 architecture leaves ASRs numbered 16..31 available for implementation-dependent uses. The SPARC64-III implements ASRs #18 through #31; they are defined in the subsections that follow.

An ASR is read and written with the RDASR and WRASR instructions, respectively. An RDASR or WRASR instruction is privileged if the accessed register is privileged.

5.2.11.1 Hardware Mode Register (ASR18)

	DPE	BRM	—	RMO	PSO	TSO
0						
63	11 10 9	8 7	6 5	4 3	2 1	0
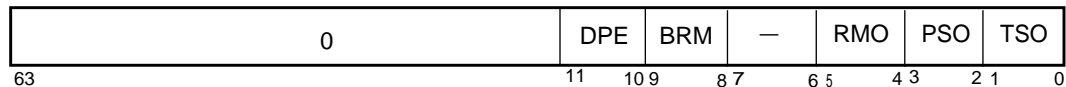
Figure 52: Hardware Mode Register (ASR18)

DPE: Data Prefetch Enable
> When set, two consecutive D1 cache lines from within a 4Kb boundary will be fetched from the UC in case of a D1 cache miss.

BRM: Branch Prediction Mode

See 9.3, "Branches and Branch Prediction" for details.

Table 15: ASR18 Branch Prediction Mode Field Encodings

Encoding	Meaning
00	2-level adaptive
01	Conventional 2-bit scheme
10	*reserved*

11	Conventional 2-bit scheme for Bicc/FBfcc:
	Use instruction *p* bit for BRcc, BPr, and FBPfcc:
	    If $p = 0$, then predict not taken
	    If $p = 1$, then predict taken

Resv: Reserved

Reads as zero, writes are ignored.

RMO: Relaxed Memory Order Memory Model

Selects the hardware memory model to use when PSTATE.MM = 2.

PSO: Partial Store Order Memory Model

Selects the hardware memory model to use when PSTATE.MM = 1.

TSO: Total Store Order Memory Model

Selects the hardware memory model to use when PSTATE.MM = 0.

Table 16: ASR18 RMO, PSO, and TSO Field Encodings

Encoding	Hardware MM
00	HLSO
01	HTSO
10	HSTO
11	*reserved*

See 8.1.1, "SPARC64-III Hardware Memory Models" for details of the SPARC64-III hardware memory models.

5.2.11.2   Graphic Status Register (GSR) (ASR19)

0	SF	AO

63   7 6 5 4 3 2 1 0
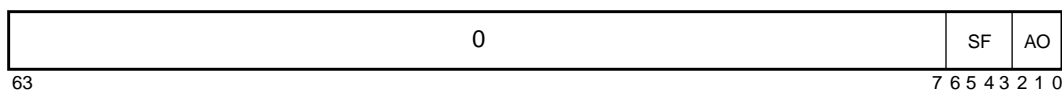
Figure 53: Graphic Status Register (GSR) (ASR19)

Non-privileged read/write register used for VIS (Sun Microsystems' Visual Instruction Set) emulation.

Access to this register causes an *fp_disabled* exception if either PSTATE.PEF or FPRS.FEF is 0.

SF: scale_factor

> See *UltraSPARC Programmer's Reference Manual* for details.

AO: alignaddr_offset

> See *UltraSPARC Programmer's Reference Manual* for details.

5.2.11.3 Set SCHED_INT Register (ASR20)

0	ASR22 Bits to be set	—
63	16 15	1 0
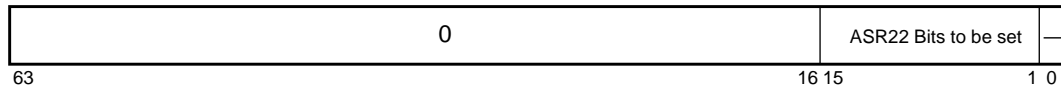
Figure 54: Set SCHED_INT Register (ASR20)

A Write State Register instruction (WR) to ASR20 sets the corresponding bits in the SCHED_INT Register (ASR22); that is, when set, bits <15:1> in ASR20 set the corresponding bits in ASR22. Other bits in ASR20 are ignored.

ASR20 is a privileged, write-only register.

5.2.11.4 Clear SCHED_INT Register (ASR21)

0	ASR22 Bits to be cleared	—
63	16 15	1 0
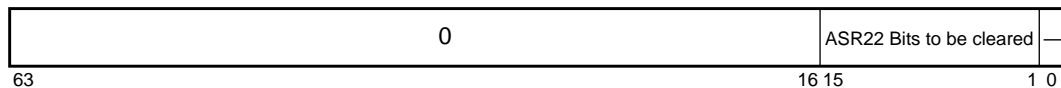
Figure 55: Clear SCHED_INT Register (ASR21)

A Write State Register instruction (WR) to ASR21 clears the corresponding bits in the SCHED_INT Register (ASR22); that is, when set, bits <15:1> in ASR21 clear the corresponding bits in ASR22. Other bits in ASR21 are ignored.

ASR21 is a privileged, write-only register.

5.2.11.5 Schedule Interrupt (SCHED_INT) Register (ASR22)

0	INT_LEVEL	T M
63	16 15	1 0
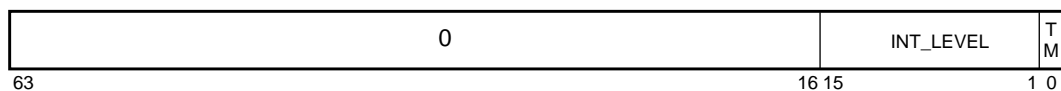
Figure 56: SCHED_INT Register (ASR22)

The OS kernel uses this privileged, read/write register to schedule interrupts.

INT_LEVEL:

> When a bit is set within this field (bits <15:1>), it causes an interrupt at the corresponding interrupt level.

TM = TICK_MATCH:

> When the TICK_MATCH (ASR23) Register's MATCH_DIS (match disable) field is zero (that is, tick matching is *enabled*) and its MATCH_VALUE field matches the value in the TICK register, then the TICK_MATCH field in ASR22 is set and a level 14 interrupt is generated. See 5.2.11.6, "TICK Match Register (ASR23)" for details.

5.2.11.6 TICK Match Register (ASR23)
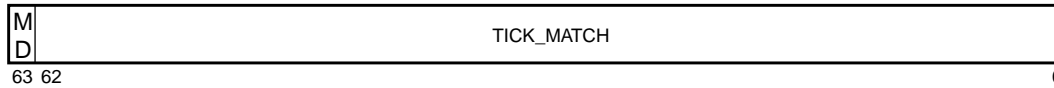
M D	TICK_MATCH
63 62	0

Figure 57: TICK Match Register (ASR23)

ASR23 is a privileged, read/write register.

MD = MATCH_DISABLE

> If this field is 0 (that is, tick matching is *enabled*) a level 14 interrupt will be generated and the TICK_MATCH field of ASR22 will be set whenever the TICK_MATCH condition is met, as described in the next field.

TICK_MATCH

> When the TICK_MATCH field matches the current value in the TICK register (bits <62:0>), the TICK_MATCH condition is true.

5.2.11.7 Instruction Access Fault Type Register (ASR24)

0	TYPE
63	4 3 0
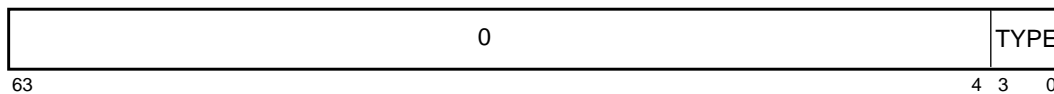
Figure 58: Instruction Access Fault Type Register (ASR24)

This privileged, read-only register is written by the hardware on *instruction_access_error* traps. Reading ASR24 clears the TYPE field to zero.

Programming Note:

> If multiple errors occur on the same machine cycle, the TYPE field will contain the highest priority error, as determined by Table 17.

TYPE

> The instruction access error, as encoded in the following table:

Table 17: Instruction Access Fault Type Encoding

<3:0>	TYPE	Priority
0_{16}	No Error	—
1_{16}	uITLB Multiple Hit	01
2_{16}	MTLB Parity Error	02

Table 17: Instruction Access Fault Type Encoding

<3:0>	TYPE	Priority
3_{16}	MTLB Multiple Hit	03
4_{16}	I1 Cache Tag Parity Error	04
5_{16}	I1 Cache Tag Multiple Hit	05
6_{16}	I1 Cache Data ECC Single Error	07
7_{16}	I1 Cache Data ECC Multiple Error	06
8_{16}	UPA Bus Error	08
9_{16}	UPA Time Out	09
A_{16}	(*unused*)	—
B_{16}	(*unused*)	—
C_{16}	(*unused*)	—
D_{16}	(*unused*)	—
E_{16}	I0 Cache Tag Parity Error	10
F_{16}	I0 Cache Data Parity Error.	11

The Instruction Access Fault Type Register facilitates the handling of traps that involve an instruction access. The register is privileged and read-only. System software must take care to read this register on entry to the fault handler before any other instruction faults can occur that would overwrite it.

System software should use the TPC value in the Trap Stack to identify the instruction that caused an *instruction_access_error* trap. The TPC value on *instruction_access_error* trap is guaranteed to point to the cache line address (64 byte block address) in which the error condition exists, but is not guaranteed to point to the exact address of the instruction that caused the error.

See 7.7, "Exception and Interrupt Descriptions" for detailed information on the *instruction_ access_error* trap.

5.2.11.8 Software Scratch Registers 0 through 3 (ASR25)

Bits <9:8> of the opcode specifies one of 4 Software Scratch Registers. These registers are privileged, read/write. Writing the scratch registers causes a machine sync; reading them does **not** cause a machine sync.

Note:

> The four registers can be used in conjunction with the eight ASI_MMU_SCRATCH registers. See Appendix L, "ASI Assignments" for details of the ASI_MMU_SCRATCH registers

5.2.11.9 Data Breakpoint Registers (ASR26A and ASR26B)

These privileged read/write registers are used to trap any data accesses to a double word aligned breakpoint address. Selection of ASR26a or ASR26b is made by bit 12 of the opcode.

5.2.11.9.1 Data Breakpoint Address Register (ASR26ᴀ)

Breakpoint Address	P	R	W
63	3	2 1	0
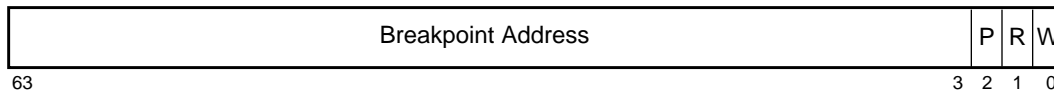
Figure 59: **Data Breakpoint Address Register (ASR26ᴀ)**

This privileged read/write register specifies the double-word aligned virtual address of the data breakpoint.

Breakpoint Address:
> A doubleword-aligned breakpoint address

P:
> Privileged / Nonprivileged access mode (Privileged = 1)

R:
> Read breakpoint enable

W:
> Write breakpoint enable

For each read/write, the virtual address is masked as specified by the Data Breakpoint Mask Register (see below) and then compared with the double-word aligned breakpoint address field in ASR26ᴀ. If the PSTATE.AM bit is set, the upper 32-bits of the 61-bit virtual address are masked to zero before the compare. If the addresses match and the breakpoint is enabled for the type of access, the CPU will generate an implementation specific *data_breakpoint* trap (TT = 0x61, priority = 14).

If the P bit is set, the access mode used for the access must be privileged in order for a match to occur. Conversely, if the P bit is cleared, the access must be non-privileged in order for a match to occur.

The access mode used for the comparison to the P bit is the access mode as seen by the Caches and MMU. For example, if the access is done in privileged mode but with ASI_AS_IF_USER, then the access mode is non-privileged.

If the R bit is set and W is not set, then the access must be a read and the access mode must match the P bit for the trap to occur. If the W bit is set and R is not set, then the access must be a write and the access mode must match the P bit for the trap to occur. If both R and W are set then the breakpoint is enabled for both reads and writes that have the prescribed access mode.

If bits R and W are both set to zero then the data breakpoint feature is disabled.

The *data_breakpoint* trap is a deferred trap; it is generated after the completion of the corresponding read/write instruction. The TPC points to the instruction that caused the *data_breakpoint*.

A data breakpoint can be installed by privileged software using the WRASR instruction. ASR 26ᴀ is readable in the SPARC64-III.

5.2.11.9.2 Data Breakpoint Mask Register (ASR26B)

—	HI	LO	—

63   16 15 13 12   3 2 0
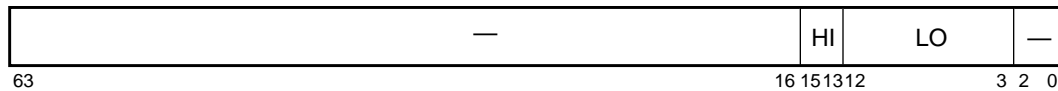
Figure 60: **Data Breakpoint Mask Register (ASR26B)**

This privileged read/write register specifies a mask to be used when comparing the current virtual address with the Data Breakpoint Address in ASR26A.

HI: Mask High

Bits set to 1 in Mask High specify which bits of VA<63:61> should be *ignored* in the comparison.

LO: Mask Low

Bits set to 1 in Mask Low specify which bits of VA<12:3> should be *ignored* in the comparison.

VA<60:13> are always *used* in the comparison; VA<2:0> are always *ignored* in the comparison.

5.2.11.10 Data Access Fault Address and Data Access Fault Type Registers (ASR28 and ASR29)

The Data Access Fault Address and Data Access Fault Type registers facilitate the handling of traps that involve a data memory access. The registers are privileged and read-only. System software must read these registers on entry to the fault handler before any other fault can occur that would overwrite them.

The following trap types cause ASR 28 and ASR 29 to be set:

- *data_access_exception*
- *data_access_error*
- *mem_address_not_aligned*
- *LDDF_mem_address_not_aligned*
- *STDF_mem_address_not_aligned*
- *data_breakpoint*
- *privileged_action* (memory access only)
- *32i_data_access_MMU_miss*
- *32i_data_access_protection*

5.2.11.10.1 Data Access Fault Address Register (ASR28)

The Data Access Fault Address Register contains the virtual address (VA) of the last data memory access that has caused the CPU to trap. The register contains all 64 bits of the address, even if PSTATE.AM is set.

5.2.11.10.2 Data Access Fault Type Register (ASR29)

This register records information about the last data memory access that caused the CPU to trap. The Data Access Fault Type Register has the following format:

—	CONTEXT	—	FTYPE	ASI	–	P	R	W

63   60 59   48 47   16 15   12 11   4 3 2 1 0
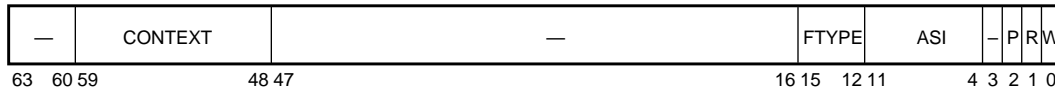
Figure 61: Data Access Fault Type Register (ASR29)

CONTEXT:

> The identifier of the context that caused the fault. This field is interpreted according to the value stored in the ASI field.

ASI:

> The 8-bit ASI that was used for the access. For a normal load/store access this field should be set to ASI_PRIMARY (80_{16}) or ASI_PRIMARY_LITTLE (88_{16}), the default value. **Note**: ASI_PRIMARY_LITTLE is set if PSTATE.CLE = 1.

P: PRIV

> The access mode as seen by the Caches and MMU. PRIV is set to one if the access was privileged and to zero if the access was non-privileged. Note that the access mode may be different from the execution mode of the CPU. For example, if the access was done in privileged mode but with ASI_AS_IF_USER, then PRIV would be zero.

R and W:

> These bits are set if the access involved a read or a write to memory, respectively. Both bits are set for atomic load-store instructions (CAS, LDSTUB, SWAP).

FTYPE:

> This field defines the type of error or fault that caused the trap. The encoding of this field is given below. The priority of each type is also given below.

Table 18: Data Access Fault Type Encoding

FTYPE	Description	Priority	Trap Generated
0_{16}	No error	—	—
1_{16}	µDTLB Multiple Hit	02	*data_access_error*
2_{16}	MTLB Parity Error	03	*data_access_error*
3_{16}	MTLB Multiple Hit	04	*data_access_error*
4_{16}	D1 Cache Tag Parity Error	10	*data_access_error*
5_{16}	D1 Cache Tag Multiple Hit	11	*data_access_error*
6_{16}	D1 Cache Data ECC Single Bit Error	13	*data_access_error*
7_{16}	D1 Cache Data ECC Multiple Bit Error	12	*data_access_error*
8_{16}	UPA Bus Error	14	*data_access_error*
9_{16}	UPA Time Out	15	*data_access_error*
A_{16}	(*unused*)	—	—
B_{16}	(*unused*)	—	—

Table 18: Data Access Fault Type Encoding

FTYPE	Description	Priority	Trap Generated
C_{16}	Illegal Access to Strongly Ordered (SO) page	07	*data_access_exception*
D_{16}	Illegal Access to Non Faulting Only (NFO) page	08	*data_access_exception*
E_{16}	Illegal Access to Noncacheable Page	06	*data_access_exception*
F_{16}	Invalid ASI	01	*data_access_exception*

Note that *data_access_error*, *data_access_exception*, *32i_data_access_MMU_miss*, and *32i_data_access_protection* trap priorities are all assigned to the same rank in 7.5.2, "Trap Type (TT)". The priorities of *32i_data_access_protection* and *32i_data_access_MMU_miss* trap are defined below in relation to *data_access_execption* and *data_access_error*.

■ *32i_data_access_protection*: priority=09

■ *32i_data_access_MMU_miss*: priority=05

See 7.7, "Exception and Interrupt Descriptions" for detailed descriptions of *data_access_error*, *data_access_exception*, *32i_data_access_MMU_miss*, and *32i_data_access_protection* traps.

5.2.11.11 Performance Monitor Registers (ASR30)

This privileged read/write register specifies 35 event and latency monitors which are used to track processor performance. The performance monitors can be divided into four groups:

1. Issue and commit counters that measure instruction execution rate.

2. Stall counters that indicate the mixture and number of various issue stall conditions.

3. Memory access latency monitors that measure the latency of certain types of memory accesses.

4. Memory access event monitors that measure the frequency of certain types of memory accesses.

Details of the performance monitor registers are described in Appendix Q, "Performance Monitoring". Performance monitor registers can be enabled for User, Supervisor, or User & Supervisor mode via the PMEM_SEL bits in ASR31 (bits 15:14).

5.2.11.12 State Control Register (ASR 31)

The privileged State Control Register (SCR) is a 64-bit implementation-specific register containing flags that control the state of the CPU. The register can be read/written via the RDASR/WRASR instruction.

Table 19 describes the State Control Register fields and their meanings:

Table 19: State Control Register (ASR31) Field Definitions

Bits	Abbrev	Name	Meaning
63:53	—	(*reserved*)	
52	D_ASEET	DISABLE_ASYNC_SECC_ERR_TRAP	When set, any Asynchronous Single ECC Error (from U2 Cache or UPA) won't cause a trap.
51	D_AET	DISABLE_ASYNC_ERROR_TRAP	When set, any Asynchronous Error won't cause trap.
50	D_U2E	DISABLE_U2_ECC_CHECK	When set, ECC error checking of the U2 cache data is disabled.
49	D_D1E	DISABLE_D1_ECC_CHECK	When set, ECC error checking of the D1 cache data is disabled.
48	D_I1E	DISABLE_I1_ECC_CHECK	When set, ECC error checking of the I1 cache data is disabled.
47	D_UPA	DISABLE_UPA_ECC_CHECK	When set, ECC error checking of the UPA is disabled.
46	D_MTP	DISABLE_MTLB_PARITY_CHECK	When set, parity error checking of the MTLB is disabled.
45	D_UDTP	DISABLE uDTLB_MULTIPLE_HIT_CHECK	When set, multiple hit error checking of the μDTLB is disabled.
44	D_UITM	DISABLE_uITLB_MULTIPLE_HIT_CHECK	When set, multiple hit error checking of the μITLB is disabled.
43	D_U2P	DISABLE_U2_PARITY_CHECK	When set, parity error checking of the U2 cache tag is disabled.
42	D_D1P	DISABLE_D1_PARITY_CHECK	When set, parity error checking of the D1 cache tag is disabled.
41	D_I1P	DISABLE_I1_PARITY_CHECK	When set, the parity checking of the I1 Cache is disabled
40	D_I0P	DISABLE_I0_PARITY_CHECK	When set, the parity checking of the I0 Cache is disabled
39	D_DCW3	DISABLE_D1_WAY3	When set, Way 3 of the D1 Cache is disabled
38	D_DCW2	DISABLE_D1_WAY2	When set, Way 2 of the D1 Cache is disabled
37	D_DCW1	DISABLE_D1_WAY1	When set, Way 1 of the D1 Cache is disabled
36	D_DCW0	DISABLE_D1_WAY0	When set, Way 0 of the D1 Cache is disabled
35	D_ICW3	DISABLE_I1_WAY3	When set, Way 3 of the I1 Cache is disabled
34	D_ICW2	DISABLE_I1_WAY2	When set, Way 2 of the I1 Cache is disabled
33	D_ICW1	DISABLE_I1_WAY1	When set, Way 1 of the I1 Cache is disabled
32	D_ICW0	DISABLE_I1_WAY0	When set, Way 0 of the I1 Cache is disabled
31	D_UAE	DISABLE_UPA_ADDR_ERR	When set, UPA Address parity checking is disabled
30	D_MDTLB	DISABLE_MDTLB	When set, the Main TLB is disabled for Data
29	D_MITLB	DISABLE_MITLB	When set, the Main TLB is disabled for Instructions
28	D_UDTLB	DISABLE_UDTLB	When set, the μDTLB is disabled
27	D_UITLB	DISABLE_UITLB	When set, the μITLB is disabled

Table 19: **State Control Register (ASR31) Field Definitions** (*Continued*)

Bits	Abbrev	Name	Meaning
26:21	—	*unused*	—
20	E_CSE	ENABLE_ CONSERVATIVE_ STORE_EXECUTION	When set, store won't start its execution until all previous instructions except load/store have been completed.
19:18	DB_ CSEL	DEBUG_BUS_CPU_ SELECTION[1:0]	Selects one of the four CPU views for debug signals. 00 Instruction Tracking Bus I 01 Instruction Tracking Bus II 10 Load Store Bus Control 11 Load Store Bus VA and Control
17:16	DB_ GSEL	DEBUG_BUS_MMU_ SELECTION[1:0]	Selects one of the four MMU views for debug signals. 00 MMU Global 01 IC Intensive + DC Misc. 10 DC Intensive 11 UC Intensive
15:14	PMEN_ SEL	PERFORMANCE_ ENABLE_SELEC- TION[1:0]	Select when performance counters get updated. 00 User & Supervisor mode 01 User mode only 10 Supervisor mode only
13	PM_US	PERFORMANCE_ MONITOR_USER_ ACCESS	When not set (PM_US=0), an attempt by nonprivileged software to read or write the performance monitor registers causes a *privileged_opcode* exception. When set (PM_US=1), nonprivileged and privileged software can read or write the performance monitor registers.
12:11	W_EN W_RED	WDT_EN WDT_RED	00 Watchdog trap never occurs. 01 Undefined. Should not be specified 10 Watchdog trap is processed in Exec_state in case the watchdog timer counts 2^n, where n is dependent on W_SEL 11 Watchdog trap is processed in RED_state in case the watchdog timer counts 2^n, where *n* is dependent on W_SEL

Table 19: **State Control Register (ASR31) Field Definitions** (*Continued*)

Bits	Abbrev	Name	Meaning
10:8	W_SEL	WDT_SELECT[2:0]	Selects the watchdog timer value that will cause a trap. If *watchdog_trap*s are disabled (WDT_ENABLE = 0), the value of WDT_SELECT has no meaning. The encodings are: $\quad\quad 000=2^{12}$ $\quad\quad 001=2^{16}$ $\quad\quad 010=2^{18}$ $\quad\quad 011=2^{20}$ $\quad\quad 100=2^{22}$ $\quad\quad 101=2^{24}$ $\quad\quad 110=2^{28}$ $\quad\quad 111=2^{30}$ If the watchdog timer counts to 2^{31}, a RED_state trap is taken regardless of the values of WDT_RED. This lets software attempt to recover before entering error_state. If the watchdog timer counts to 2^{32}, the CPU enters error_state unconditionally and asserts CPU_HALTED, preserving as much state as possible. The CPU must be rescanned with the reset state before execution can resume. See 7.2.2, "Error_state" for more information.
7	TR	SOFTWARE_TRIGGER	When this bit is set, the software_trigger bit going out on the debug bus is asserted. This allows software to determine when to trigger equipment attached to the debug bus.
6:3	—	*unused*	—
2	PM	PIPELINE_MODE	When set, the processor will execute all instructions sequentially, one-at-a-time, in pipeline mode. Specifically, the CPU attempts to issue one instruction every cycle. The CPU will still speculate and predict branches. Instruction issue constraints and processor operation is identical to superscalar operation, with the additional constraint that a maximum of one instruction is issued per cycle.
1	II0	INVALIDATE_I0	A write-only bit; always reads as zero. When set, causes the level-0 instruction cache (I0) and all prefetch buffers to be invalidated.
0	SM	SEQUENTIAL_MODE	When set, the processor will execute all instructions sequentially, one-at-a-time. Specifically, before an instruction is issued, the previous instruction must have been retired and have completed all modifications to machine state. The SM bit also disables speculative instruction prefetching. Instruction accesses occur only when an instruction can be issued. Note that block prefetch around the instruction is still possible. This mode of operation severely cripples CPU performance, so use it sparingly. When the SM bit is reset, normal superscalar execution is taking place (unless the PM bit is set). The SM bit (SEQUENTIAL_MODE) takes precedence over the PM bit (PIPELINE_MODE).

When the CPU enters or exits from RED_state, the I0 cache and prefetch buffers are invalidated.

When the CPU enters RED_state due to a trap or reset, bit 0 (SM), bit 27 (D_UITLB), bit 28 (D_UDTLB), bit 29 (D_MITLB), and bit 30 (D_MDTLB) are set by the hardware. It is the software's responsibility to reset these bits when required (for example, when the CPU exits from RED_state).

When the CPU enters RED_state *not* due to a trap or reset (that is, when software sets the PSTATE.RED bit using WRPR), all of the SCR register bits are unchanged unlike the case above.

When the CPU is in RED_state, it behaves as if bit 0 (SM), bit 27 (D_UITLB), and bit 29 (D_MITLB) are set, regardless of their actual values in the SCR register.

5.2.12 Floating-point Deferred-trap Queue (FQ)

The CPU does not contain a Floating-Point Deferred-trap Queue. An attempt to read FQ with an RDPR instruction causes an *illegal_instruction* exception.

5.2.13 IU Deferred-trap Queue

The CPU does not have or need an IU Deferred-trap Queue.

5.2.14 RSTV Register

The CPU implements RSTV, the RED_State Trap Vector as a constant address:

■ VA = FFFF FFFF F000 0000$_{16}$

■ PA = 0000 01FF F000 0000$_{16}$

5.2.15 Emulation Trap Registers

The CPU provides a mechanism for the hardware to trap or sync certain instruction encodings. This mechanism was designed to provide a way around hardware errors that may be found in silicon during bringup. For example, if an instruction is failing on a particular mask set, it can be trapped and emulated in software.

The CPU implements the following registers to support emulation traps:

■ Four Emulation Trap Register Values (ETRVs)

■ Four Emulation Trap Register Masks (ETRMs)

■ Four sets of 2 control bits

All of these registers are "scan-only." They can be scanned in by the debug monitor during bringup. The ETRV and ETRM registers are each 27 bits wide. The purpose of these registers is to identify a pattern of instructions.

A pair of registers, ETRV and ETRM, form a "CAM" like pattern matcher. The associated Control bits determine what action will be taken on a match. There are 4 pairs of ETRV and ETRM registers and 4 pairs of control bits. Each pair of registers applies to instruc-

tions in all the issue slots.[1] This requires 16 simultaneous compares for each set of instructions fetched from the I0 Cache.

The ETRM and ETRV register bits correspond to the upper 27 bits in an instruction[2]. When a bit in an ETRM is set that bit in an instruction being issued will be ignored (don't care). If it is reset, that bit of the ETRV specifies a value to match. For example:

```
ETRM:    0011 1111 1111 0000 0011 1111 111- ----
ETRV:    0000 0000 0000 1010 1000 0000 000- ----
matches: 00xx xxxx xxxx 1010 10xx xxxx xxxx xxxx
```

When an instruction matches, the value in the Control bits is checked to see what action to take.

Table 20:

Control Bits	Action
00	No action. (ETR is disabled.)
01	Make the instruction a syncing instruction.
10	Generate a non-syncing issue trap.
11	Generate a syncing issue trap.

If more than one ETR pair matches in the same cycle, the earliest instruction that matches (in program order) will cause the Action in the above table to be initiated.

Also, if more than one ETR pair matches the same instruction the control bits will be or'ed together and the result will be used to determine the action. Syncing issue traps will wait for all previous instructions to commit and retire before the trap is taken. A non-syncing issue trap will vector to the appropriate trap target speculatively. It may later be undone by a backup. If a trap is taken it is the *programmed_emulation_trap* (tt = 62_{16}, priority = 6).

---

1. This is different from the SPARC64-III where each ETRV:ETRM pair only applied to one particular issue slot.
2. The *rs2* field was omitted from the comparison because of its low utility.

6 Instructions

Instructions are accessed by the processor from memory and are executed, annulled, or trapped. Instructions are encoded in five major formats and partitioned into eleven general categories.

6.1 Instruction Execution

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

An instruction may generate an exception if it encounters some condition that makes it impossible to complete normal execution. Such an exception may in turn generate a precise trap. Other events may also cause traps: an exception caused by a previous instruction (a deferred trap), an interrupt or asynchronous error (a disrupting trap), or a reset request (a reset trap). If a trap occurs, control is vectored into a trap table. See Chapter 7, "Traps", for a detailed description of exception and trap processing.

If a trap does not occur and the instruction is not a control transfer, the next program counter is copied into the PC, and the nPC is incremented by 4 (ignoring overflow, if any). If the instruction is a control-transfer instruction, the next program counter 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, or ASI, to the 64-bit memory address. Load/store alternate instructions (see 6.3.1.3, "Address Space Identifiers (ASIs)") can provide an arbitrary ASI with their data addresses, or use the ASI value currently contained in the ASI register.

The CPU is a superscalar implementation of SPARC-V9. Several instructions may be issued and executed in parallel. Although the CPU provides serial program execution semantics, some of the implementation characteristics described below are part of the architecture visible to software for correctness and efficiency considerations. The affected software includes optimizing compilers and supervisor code.

6.1.1 Speculative Execution

The CPU does speculative (out of program order) execution of instructions; the effect of these instructions can be undone if the speculation proves to be incorrect. In general, this mechanism is transparent to software, since the CPU maintains serial program execution semantics for all architecturally visible state. The exception to this rule is the load instruction, which produces two types of side effects visible to software.

1. Speculative loads may be issued to any address including a memory-mapped I/O register that causes side effects in an I/O device. The supervisor software must ensure that all memory-mapped pages containing registers and memory that could be affected by speculative loads must have the "strongly ordered" bit set in the page table entry. If the strongly ordered (SO) bit is not set for such pages, incorrect behavior may occur.

2. Speculative loads often are useful to prefetch data. Sometimes, however, they may cause the cache, MMU, and memory subsystem to make spurious accesses; that is, some of these loads may replace "useful" lines in the cache. This simply degrades system throughput; it does not cause incorrect program behavior.

The CPU provides two mechanisms to avoid speculative execution of a load:

1. Avoid speculation by disallowing speculative accesses to certain memory pages or I/O spaces. This can be done by setting the SO (Strongly Ordered) bit in the PTE for all memory pages that should *not* allow speculation. All accesses made to memory pages that have the SO bit set in their PTE will be delayed until they are no longer speculative or until they are cancelled. See Appendix F, "MMU Architecture" for details.

2. Alternate space load instructions that force program order such as ASI_PO_P will not be speculatively executed. See Appendix L, "ASI Assignments" for details of the CPU ASIs.

Method one is preferred for the SPARC64-III. Method two is supported for backward compatibility with SPARC64-I and SPARC64-II systems.

6.1.2 Instruction Prefetch

The CPU prefetches instructions in order to minimize cases where the CPU must wait for instruction fetch. In combination with branch prediction, prefetching may cause the CPU to access instructions that are not subsequently executed. In some cases the speculative instruction accesses will reference data pages. In order to avoid speculative instruction accesses to the area which has side-effects by the references, the software should not set any executable permission bits (SX and UX) to the page. Then the data in those pages are never referenced or cached in the I0- or I1-Cache. The CPU does not generate a trap for any exception that is caused by an instruction fetch until all of the instructions before it (in program order) have been committed.[1]

---

1. Hardware errors and other asynchronous errors may generate a trap even if the instruction that caused the trap is never committed.

Instructions prefetched by the CPU are cached in an internal cache, called the I0 cache. The prefetch action is independent of the data access path and is not directly affected by instructions that serialize the execution of instructions or the data path. The I0 cache can be invalidated but not disabled. See the description of the SCR register in 5.2.11.12, "State Control Register (ASR 31)", and the description of the WRASR instruction in A.63, "Write State Register", for details.

Programming Note:
> The I0 cache also caches instructions that cause faults, because it cannot determine that they will fault when executed later. For example, an instruction with only one executable permission bit (SX or UX) set may be cached in I0. If the kernel software changes the page's PTE to set both executable permission bits, it must also invalidate the I0 cache to avoid getting stale information.

Instruction cache lines are also prefetched into the level 1 instruction cache (I1) from the Unified Cache. This prefetching should improve performance of database applications. Prefetching into I1 is invisible to the application programmer.

6.1.3 Serializing Instructions

Serializing instructions are issued one at a time and only when the CPU has no other uncommitted instruction pending. After an instruction in this class is issued, the CPU waits until the instruction is committed before issuing the next instruction. In other words, the following sequence is observed:

1. All outstanding instructions must commit before the serializing instruction is issued (this is equivalent to machine sync)

2. The serializing instruction is issued by itself

3. The serializing instruction is executed by itself

4. The serializing instruction is committed by itself

5. Instructions following the serializing instruction begin to be issued again

Table 21 lists the serializing RDASR and WRASR instructions:

Table 21: SPARC64-III Serializing RDASR and WRASR Instructions

ASR Number	Name	Priv?	Serialize on Reading?	Serialize on Writing
0	Y Register	No	No	Yes
2	Condition Code Register	No	No	No
3	ASI Register	No	No	Yes[a]
4	Tick Register	Yes/No[b]	No	—
5	Program Counter	No	No	—
6	Floating-Point Register Status	No	No	Yes[a]
15	Store Barrier, Membar, SIR[c]	Yes/No[d]	Yes/No[e]	Yes
18	Hardware Mode Register	Yes	No	Yes
19	Graphic Status Register	No	No	Yes
20	Set Bits in Sched_Int Register	Yes	—	Yes
21	Clear Bits in Sched_Int Register	Yes	—	Yes
22	Sched_Int Register	Yes	No	Yes
23	Tick Match Register	Yes	No	Yes
24	Instruction Fault Type	Yes	Yes	—
25	Scratch Registers	Yes	No	Yes
26	Data Breakpoint Registers	Yes	Yes	Yes
28	Data Fault Address	Yes	Yes	—
29	Data Fault Type	Yes	Yes	—
30	Performance Monitor Registers	Yes	Yes	Yes
31	System Control Register	Yes	No	Yes

a. Does not sync if address mode is %g0 + *imm*.

b. Privileged if TICK.NPT = 1, otherwise it is not privileged.

c. These are not really considered RD/WRASR instructions by SPARC-V9, but they share the RD/WRASR opcode encoding so they are listed here for clarity.

d. SIR is treated as NOP if executed while PSTATE.PRIV = 0.

e. Sync only for MEMBAR#Sync or MEMBAR#MemIssue.

6.1.3.1 Other Serializing Instructions

The following instructions also serialize the CPU:

■ WRPR for CWP, CANSAVE, CANRESTORE, CLEANWIN, WSTATE and OTHER-WIN when the addressing mode is not "%g0 + *simm13*".

■ WRPR for TPC, TNPC, TSTATE, TT, TICK, TBA, PSTATE and TL.

■ TCC[1] (except for TA %g0 + *simm13*).

■ LD(X)FSR and ST(X)FSR.

■ CAS(X)A

---

1. All Tcc instructions except TA (with %g0 or immediate addressing) can be used to cause the CPU to sync. TN is suggested for forcing a sync.

■ MULScc, UMUL, UMULcc, SMUL, SMULcc (32-bit multiplies)

■ FLUSH

■ WRY

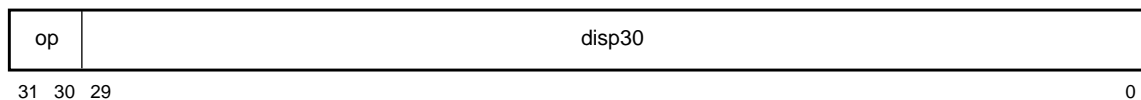### 6.1.4   Issue Stalling Instructions

The following instructions prevent new instructions from issuing until they have completed:

■ RDPR %PIL and WRPR %PIL (if there is a pending write to the PIL register). There is a pending write to the PIL register if there was a previous WRPR %PIL instruction with a source register (other than %g0) that has not completed.

■ DONE, RETRY and RDPR TSTATE stall until the TSTATE register is updated with the correct value of the condition codes. When a trap occurs, the TSTATE register may not be updated with the condition codes if there are pending instructions that modify the condition codes. See 5.2.6, "Trap State (TSTATE) Register" for further details.

6.2   Instruction Formats

Instructions are encoded in five major 32-bit formats and several minor formats, as shown in Figure 62, Figure 63 on page 112, and Figure 64 on page 113.

Format 1 (*op* = 1): CALL

op	disp30

31 30 29   0

Format 2 (*op* = 0): SETHI and Branches (Bicc, BPcc, BPr, FBfcc, FBPfcc)

op	rd	op2	imm22

op	a	cond	op2	disp22

op	a	cond	op2	cc1	cc0	p	disp19

op	a	0	rcond	op2	d16hi	p	rs1	d16lo

31 30 29 28   25 24   22 21 20 19 18   14 13   0
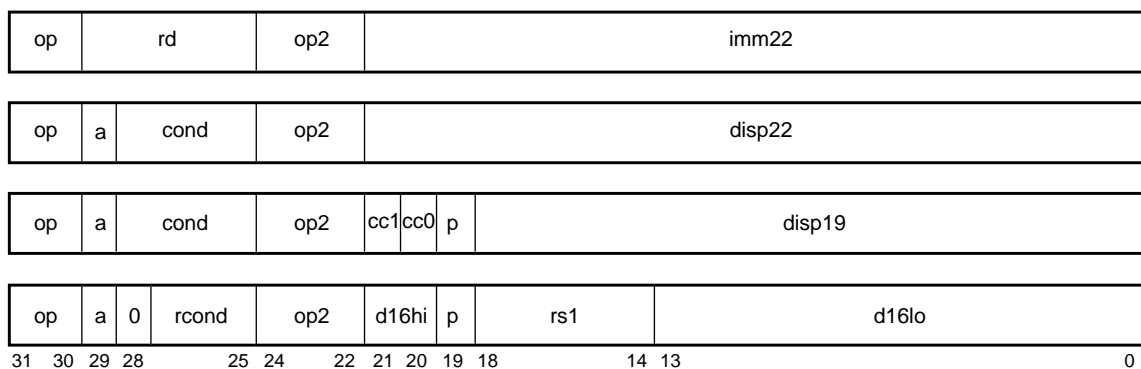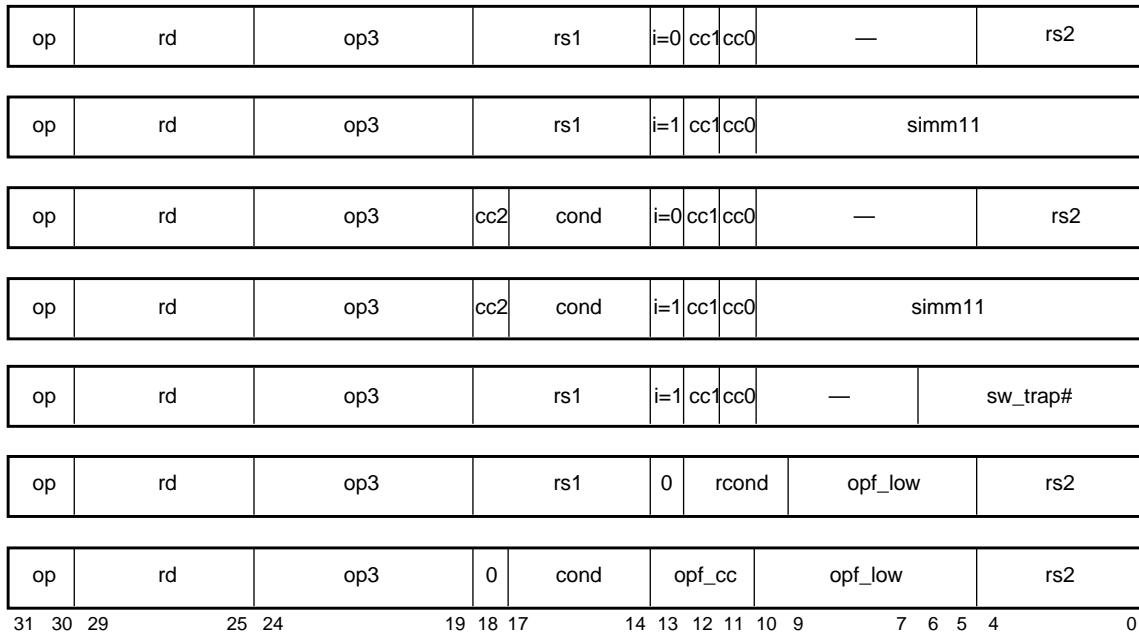
Figure 62: Summary of Instruction Formats: Formats 1 and 2 (V9=33)

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=0	—	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	000 cc1 cc0	op3	rs1		opf	rs2
op	rd	op3	rs1		opf	rs2
op	rd	op3	rs1		—	
op	fcn	op3	—			
op	rd	op3	—			

31   30 29   25 24   19 18   14 13 12 11 10 9 8 7 6 5 4 3   0

Figure 63: Summary of Instruction Formats: Format 3 ((V9=33) and (V9=34))

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

op	rd	op3	rs1	i=0	cc1	cc0	—	rs2

op	rd	op3	rs1	i=1	cc1	cc0	simm11

op	rd	op3	cc2	cond	i=0	cc1	cc0	—	rs2

op	rd	op3	cc2	cond	i=1	cc1	cc0	simm11

op	rd	op3	rs1	i=1	cc1	cc0	—	sw_trap#

op	rd	op3	rs1	0	rcond	opf_low	rs2

op	rd	op3	0	cond	opf_cc	opf_low	rs2

```
31  30  29        25 24        19 18 17      14 13 12 11 10 9      7  6  5  4           0
```

Format 5 (op = 2, op3 = 0x37): FMADD and FMSUB (in place of IMPDEP2)

op	rd	op3	rs1	rs3	var	size	rs2

```
31  30  29        25 24        19 18 17      14 13 12 11 10 9  8  7  6  5  4           0
```
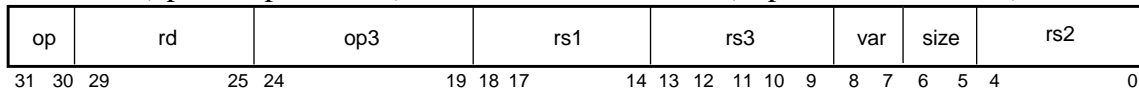
Figure 64: **Summary of Instruction Formats: Formats 4 and 5 (<u>V9=34</u>)**

6.2.1 Instruction Fields

The instruction fields are interpreted as follows:

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, and cc0:

> *cc2*:*cc1*:*cc0* specify the condition codes (*icc*, *xcc*, *fcc0*, *fcc1*, *fcc2*, *fcc3*) to be used in the instruction. Individual bits of the same logical field are present in several other instructions: Branch on Floating-point Condition Codes with Prediction Instructions (FBPfcc), Branch on Integer Condition Codes with Prediction (BPcc), Floating-point Compare Instructions, Move Integer Register If Condition Is Satisfied (MOVcc), Move Floating-point Register If Condition Is Satisfied (FMOVcc), and 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. See <u>Table 77 on page 365</u> for a description of these fields' values.

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. See Appendix E, "Opcode Maps", for descriptions of its values.

d16hi and d16lo:

These 2-bit and 14-bit fields together comprise a word-aligned, sign-extended, PC-relative displacement for a branch-on-register-contents with prediction (BPr) instruction.

disp19:

This 19-bit field is a word-aligned, sign-extended, PC-relative displacement for an integer branch-with-prediction (BPcc) instruction or a floating-point branch-with-prediction (FBPfcc) instruction.

disp22 and 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 and RETRY instructions.

i:

The i bit selects the second operand for integer arithmetic and load/store instructions. If $i = 0$, the operand is r[rs2]. If $i = 1$, 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 address space identifier in instructions that access alternate space.

impl-dep:

The meaning of these fields is completely implementation-dependent for IMPDEP1 and IMPDEP2 instructions.

mmask:

This 4-bit field imposes order constraints on memory references appearing before and after a MEMBAR instruction.

op and op2:

These 2- and 3-bit fields encode the three major formats and the Format 2 instructions. See Appendix E, "Opcode Maps" for descriptions of their values.

op3:

This 6-bit field (together with one bit from *op*) encodes the Format 3 instructions. See Appendix E, "Opcode Maps" for descriptions of its values.

opf:

This 9-bit field encodes the operation for a floating-point operate (FPop) instruction. See Appendix E, "Opcode Maps" for possible values and their meanings.

opf_cc:

> Specifies the condition codes to be used in FMOVcc instructions. See **cc0, cc1, and cc2** above 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, as show in Table 22:

<div align="center">

Table 22: Branch Prediction Bit (*p*) Encodings

p	**Branch Prediction**
0	Predict that branch will not be taken
1	Predict that branch will be taken

</div>

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. See Appendix E, "Opcode Maps" for descriptions of its values.

***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 $i = 0$.

***rs3*:**

> This 5-bit field is the address of the third *f* register source operand for the floating-point multiply-add and multiply-subtract instruction.

shcnt32:

> This 5-bit field provides the shift count for 32-bit shift instructions.

shcnt64:

> This 6-bit field provides the shift count for 64-bit shift instructions.

simm10:

> This 10-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for a MOVr instruction when $i = 1$.

simm11:

> This 11-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for a MOVcc instruction when $i = 1$.

simm13:

> This 13-bit field is an immediate value that is sign-extended to 64 bits and used as the second ALU operand for an integer arithmetic instruction or for a load/store instruction when $i = 1$.

size:

> Specifies the size of the operands for the floating-point multiply-add and multiply-subtract instructions.

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.

var:

> Specifies which specific operation (variation) to perform for the floating-point multiply-add and multiply-subtract instructions.

x:

> The x bit selects whether a 32- or 64-bit shift will be performed.

6.3   Instruction Categories

SPARC-V9 instructions can be grouped into the following categories:

- Memory access

- Memory synchronization

- Integer arithmetic

- Control transfer (CTI)

- Conditional moves

- Register window management

- State register access

- Privileged register access

- Floating-point operate

- Implementation-dependent

- Reserved

Each of these categories is further described in the following subsections.

6.3.1   Memory Access Instructions

Load, Store, Prefetch, Load Store Unsigned Byte, Swap, and Compare and Swap are the only instructions that access memory. All of the instructions except Compare and Swap use either two *r* registers or an *r* register and *simm13* to calculate a 64-bit byte memory

address. Compare and Swap uses a single *r* register to specify a 64-bit byte memory address. To this 64-bit address, the IU appends an ASI that encodes address space information.

The destination field of a memory reference instruction specifies the *r* or *f* register(s) that supply the data for a store or receive the data from a load or LDSTUB. For SWAP, the destination register identifies the *r* register to be exchanged atomically with the calculated memory location. For Compare and Swap, an *r* register is specified whose value is compared with the value in memory at the computed address. If the values are equal, the destination field specifies the *r* register that is to be exchanged atomically with the addressed memory location. If the values are unequal, the destination field specifies the *r* register that is to receive the value at the addressed memory location; in this case, the addressed memory location remains unchanged.

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

Integer load and store instructions support byte (8-bit), halfword (16-bit), word (32-bit), and doubleword (64-bit) accesses. Floating-point load and store instructions support word, doubleword, and quadword memory accesses. LDSTUB accesses bytes, SWAP accesses words, and CAS accesses words or doublewords. PREFETCH accesses at least 64 bytes.

Programming Note:

> By setting *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of an address space can be accessed without using a register to hold part of the address.

6.3.1.1 Memory Alignment Restrictions

Halfword accesses shall be **aligned** on 2-byte boundaries, word accesses (which include instruction fetches) shall be aligned on 4-byte boundaries, extended word and doubleword accesses shall be aligned on 8-byte boundaries, and quadword accesses shall be aligned on 16-byte boundaries.

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

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

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

■ An LDQF, LDQFA, STQF, or STQFA instruction causes an *illegal_instruction* exception; all of these instructions are emulated in software.

6.3.1.2 Addressing Conventions

The CPU uses big-endian byte order for all instruction accesses and, by default, for data accesses. It is possible to access data in little-endian format by using selected ASIs. It is

also possible to change the default byte order for implicit data accesses. See 5.2.1, "Processor State Register (PSTATE)", for more information.[1]

6.3.1.2.1 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 65 and defined as follows:

byte:

> A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.

halfword:

> For a load/store halfword instruction, two bytes are accessed. The most significant byte (bits 15..8) is accessed at the address specified in the instruction; the least significant byte (bits 7..0) is accessed at the address + 1.

word:

> For a load/store word instruction, four bytes are accessed. The most significant byte (bits 31..24) is accessed at the address specified in the instruction; the least significant byte (bits 7..0) is accessed at the address + 3.

doubleword or extended word:

> For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The most significant byte (bits 63..56) is accessed at the address specified in the instruction; the least significant byte (bits 7..0) is accessed at the address + 7.
>
> For the deprecated integer load/store 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.

quadword:

> For a load/store quadword instruction, sixteen bytes are accessed. The most significant byte (bits 127..120) is accessed at the address specified in the instruction; the least significant byte (bits 7..0) is accessed at the address + 15.

---

1.   See Cohen, D., "On Holy Wars and a Plea for Peace," *Computer* 14:10 (October 1981), pp. 48-54.
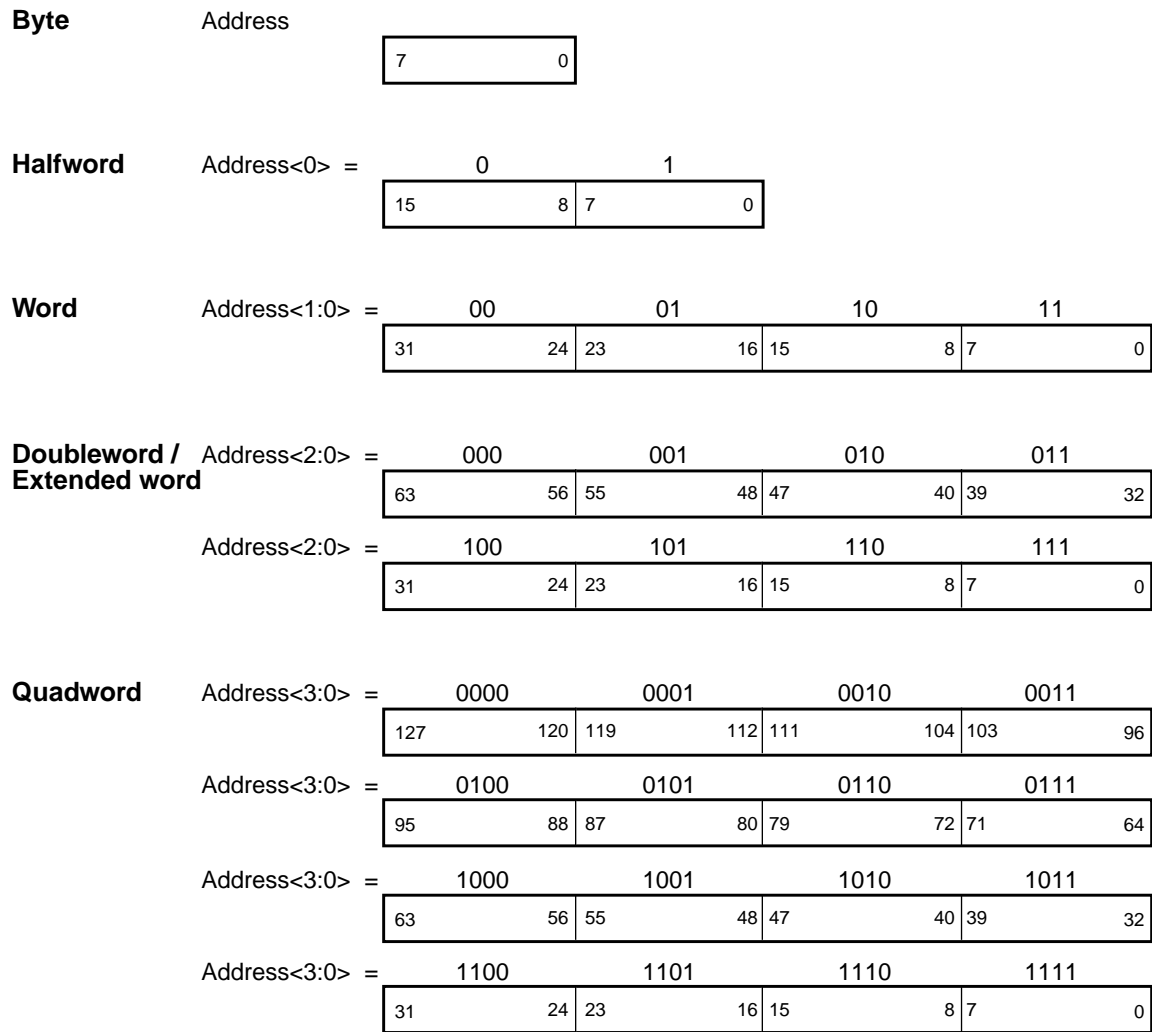
Byte Address

| 7 | 0 |

Halfword Address<0> = 0 1

| 15 | 8 | 7 | 0 |

Word Address<1:0> = 00 01 10 11

| 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |

Doubleword / Address<2:0> = 000 001 010 011
Extended word

| 63 | 56 | 55 | 48 | 47 | 40 | 39 | 32 |

Address<2:0> = 100 101 110 111

| 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |

Quadword Address<3:0> = 0000 0001 0010 0011

| 127 | 120 | 119 | 112 | 111 | 104 | 103 | 96 |

Address<3:0> = 0100 0101 0110 0111

| 95 | 88 | 87 | 80 | 79 | 72 | 71 | 64 |

Address<3:0> = 1000 1001 1010 1011

| 63 | 56 | 55 | 48 | 47 | 40 | 39 | 32 |

Address<3:0> = 1100 1101 1110 1111

| 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |

Figure 65: **Big-endian Addressing Conventions (V9=35)**

6.3.1.2.2 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 66 and defined as follows:

byte:

A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.

halfword:

For a load/store halfword instruction, two bytes are accessed. The least significant byte (bits 7..0) is accessed at the address specified in the instruction; the most significant byte (bits 15..8) is accessed at the address + 1.
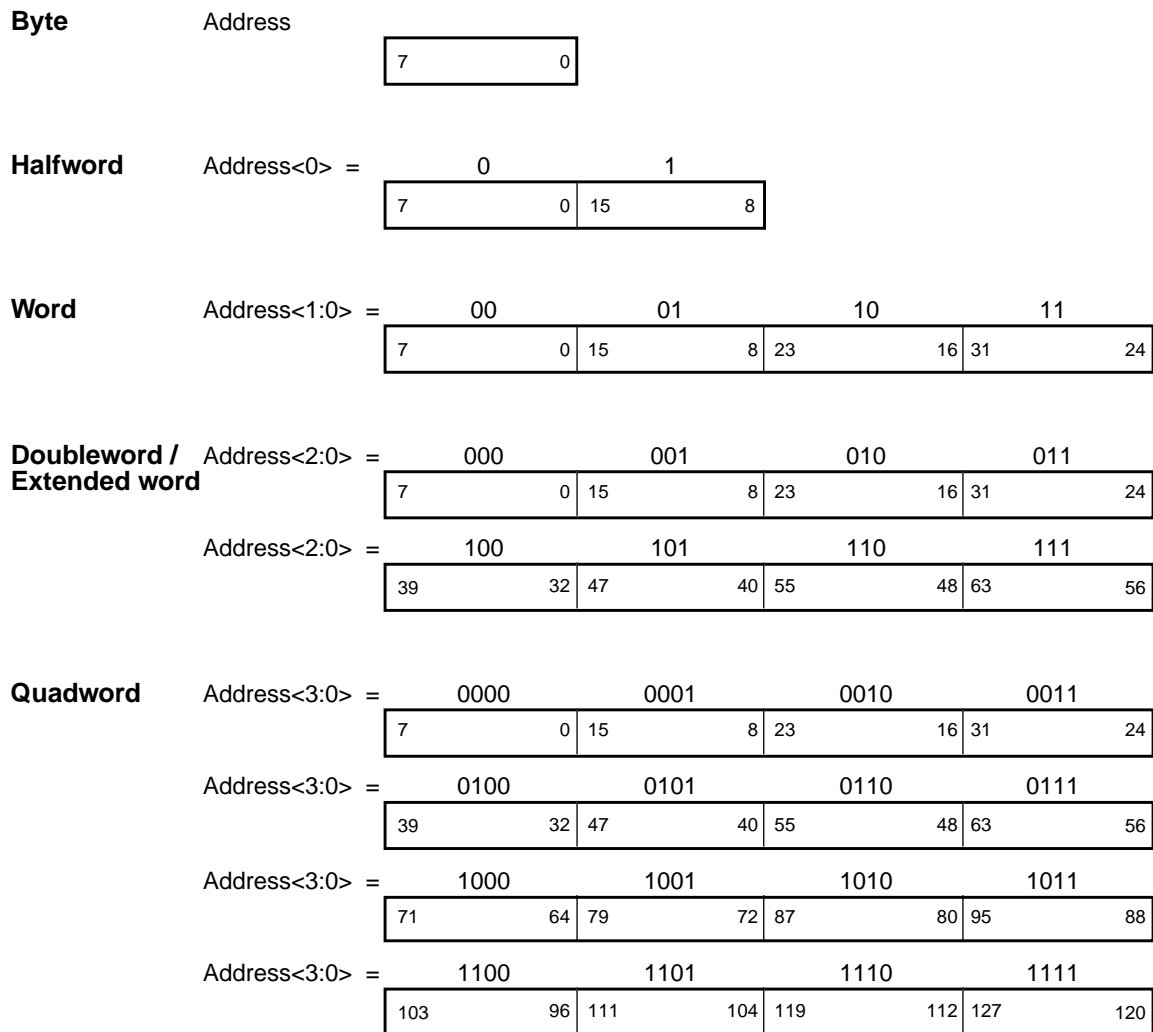
word:

For a load/store word instruction, four bytes are accessed. The least significant byte (bits 7..0) is accessed at the address specified in the instruction; the most significant byte (bits 31..24) is accessed at the address + 3.

doubleword or extended word:

For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The least significant byte (bits 7..0) is accessed at the address specified in the instruction; the most significant byte (bits 63..56) is accessed at the address + 7.

For the deprecated integer load/store double instructions (LDD/STD), two little-endian words are accessed. The word at the address specified in the instruction + 4 corresponds to the even register in the instruction; the word at the address specified in the instruction corresponds to the following odd-numbered register.

quadword:

For a load/store quadword instruction, sixteen 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.

Byte Address

```
┌─────────────────┐
│ 7             0 │
└─────────────────┘
```

Halfword Address<0> =   0   1

```
┌────────────────┬────────────────┐
│ 7           0  │ 15           8 │
└────────────────┴────────────────┘
```

Word Address<1:0> =   00   01   10   11

```
┌──────────┬──────────┬──────────┬──────────┐
│ 7      0 │ 15     8 │ 23    16 │ 31    24 │
└──────────┴──────────┴──────────┴──────────┘
```

Doubleword / Address<2:0> =   000   001   010   011
Extended word

```
┌──────────┬──────────┬──────────┬──────────┐
│ 7      0 │ 15     8 │ 23    16 │ 31    24 │
└──────────┴──────────┴──────────┴──────────┘
```

Address<2:0> =   100   101   110   111

```
┌──────────┬──────────┬──────────┬──────────┐
│ 39    32 │ 47    40 │ 55    48 │ 63    56 │
└──────────┴──────────┴──────────┴──────────┘
```

Quadword Address<3:0> =   0000   0001   0010   0011

```
┌──────────┬──────────┬──────────┬──────────┐
│ 7      0 │ 15     8 │ 23    16 │ 31    24 │
└──────────┴──────────┴──────────┴──────────┘
```

Address<3:0> =   0100   0101   0110   0111

```
┌──────────┬──────────┬──────────┬──────────┐
│ 39    32 │ 47    40 │ 55    48 │ 63    56 │
└──────────┴──────────┴──────────┴──────────┘
```

Address<3:0> =   1000   1001   1010   1011

```
┌──────────┬──────────┬──────────┬──────────┐
│ 71    64 │ 79    72 │ 87    80 │ 95    88 │
└──────────┴──────────┴──────────┴──────────┘
```

Address<3:0> =   1100   1101   1110   1111

```
┌──────────┬──────────┬──────────┬───────────┐
│ 103   96 │ 111  104 │ 119  112 │ 127   120 │
└──────────┴──────────┴──────────┴───────────┘
```

Figure 66: Little-endian Addressing Conventions (V9=36)

6.3.1.3 Address Space Identifiers (ASIs)

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 $i = 1$.

ASIs 00_{16} through $7F_{16}$ are restricted; only privileged software is allowed to access them. An attempt to access a restricted ASI by nonprivileged 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 nonprivileged mode. This is illustrated in Table 23.

<center>**Table 23: Allowed Accesses to ASIs (V9=11)**</center>

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

The SPARC-V9 architecture defines the following ASI assignments as implementation-dependent: restricted ASIs $00_{16}..03_{16}$, $05_{16}..0B_{16}$, $0D_{16}..0F_{16}$, $12_{16}..17_{16}$, and $1A_{16}..7F_{16}$; and unrestricted ASIs $C0_{16}..FF_{16}$. SPARC64-III's implementation of ASIs completely conforms to the SPARC-V9 architecture specification. See 6.3.1.3, "Address Space Identifiers (ASIs)" in *V9* for more information about the implementation-dependent aspects of ASIs. See Appendix L, "ASI Assignments" for the complete list of ASI assignments for SPARC64-III.

6.3.1.4 Separate Instruction Memory

The CPU uses separate level-1 instruction and data caches referred to as the I1 and D1 caches. These caches can be bypassed; see 5.2.11.12, "State Control Register (ASR 31)".

In addition to the I1 and D1 caches, the CPU contains an internal level 0 instruction cache, called the I0 cache. The I0 cache can be invalidated but it cannot be disabled or bypassed. See 5.2.11.12, "State Control Register (ASR 31)" for details.

A program containing self-modifying code must have the FLUSH instructions before the program executes its self-modifying code portions to ensure the consistency of the program execution.

6.3.2 Memory Synchronization Instructions

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

The CPU syncs and also waits for all pending cacheable and non-cacheable stores to reach the global visibility before issuing MEMBAR#MemIssue and MEMBAR#Sync.

6.3.3 Integer Arithmetic Instructions

The integer arithmetic instructions are generally triadic-register-address instructions that compute a result that is 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, the operand is *r*[*rs2*]; if *i* = 1, the operand is the constant *simm10*, *simm11*, or *simm13* sign-extended to 64 bits.

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

6.3.3.1 Setting Condition Codes

Most integer arithmetic instructions have two versions: one sets the integer condition codes (*icc* and *xcc*) as a side effect; the other does not affect the condition codes. A special comparison instruction for integer values is not needed, since it is easily synthesized using the "subtract and set condition codes" (SUBcc) instruction. See G.3, "Synthetic Instructions", for details.

6.3.3.2 Shift Instructions

Shift instructions shift an *r* register left or right by a constant or variable amount. None of the shift instructions changes the condition codes.

6.3.3.3 Set High 22 Bits of Low Word

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

6.3.3.4 Integer Multiply/Divide

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

6.3.3.5 Tagged Add/Subtract

The tagged add/subtract instructions assume tagged-format data, in which the tag is the two low-order bits of each operand. If either of the two operands has a nonzero tag, or if 32-bit arithmetic overflow occurs, tag overflow is detected. TADDcc and TSUBcc set the CCR.*icc*.V bit if tag overflow occurs; they set the CCR.*xcc*.V bit if 64-bit arithmetic overflow occurs. The trapping versions (TADDccTV, TSUBccTV) of these instructions cause a *tag_overflow* trap if tag overflow occurs. If 64-bit arithmetic overflow occurs but tag overflow does not, TADDccTV and TSUBccTV set the CCR.*xcc*.V bit but do not trap.

6.3.4 Control-transfer Instructions (CTIs)

These are the basic control-transfer instruction types:

■ Conditional branch (Bicc, BPcc, BPr, FBfcc, FBPfcc)

■ Unconditional branch

■ Call and link (CALL)

■ Jump and link (JMPL, RETURN)

■ Return from trap (DONE, RETRY)

■ Trap (Tcc)

A control-transfer instruction functions by changing the value of the next program counter (nPC) or by changing the value of both the program counter (PC) and the next program counter (nPC). When only the next program counter, nPC, is changed, the effect of the transfer of control is delayed by one instruction. Most control transfers in SPARC-V9 are of the delayed variety. The instruction following a delayed control transfer instruction is said to be in the **delay slot** of the control transfer instruction. Some control transfer instructions (branches) can optionally annul, that is, not execute, the instruction in the delay slot, depending upon whether the transfer is taken or not-taken. Annulled instructions have no effect upon the program-visible state nor can they cause a trap.

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.

Table 24 below defines the value of the program counter and the value of the next program counter after execution of each instruction. Conditional branches have two forms: branches that test a condition, represented in the table by "Bcc," 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 by fetching and annulling the instruction.

The effective address, EA in Table 24, specifies the target of the control transfer instruction. The effective address is computed in different ways, depending on the particular instruction.

PC-relative Effective Address:

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

Register-indirect Effective Address:

> A register-indirect effective address computes its target address as either r[*rs1*]+r[*rs2*] if $i = 0$, or *r*[*rs1*]+*sign_ext(simm13)* if $i = 1$.

Trap Vector Effective Address:

> A trap vector effective address 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 effective address is generated by concatenating the contents of the TBA register, the "TL > 0" bit, and

the contents of TT[TL]. See 5.2.8, "Trap Base Address (TBA) Register", for details.

Trap State Effective Address:

A trap state effective address is not computed but is taken directly from either TPC[TL] or TNPC[TL].

Compatibility Note:

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

Compatibility Note:

SPARC-V8 left as 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 backwards compatibility is an issue.

Table 24: Control Transfer Characteristics (V9=13)

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-ind.	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

6.3.4.1 Conditional Branches

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

See 9.3, "Branches and Branch Prediction" for detailed information about branch prediction in SPARC64-III.

6.3.4.2 Unconditional Branches

An unconditional branch transfers control unconditionally if its specified condition is "always"; it never transfers control if its specified condition is "never." If the annul bit is 0, the instruction in the delay slot is always executed. If the annul bit is 1, the instruction in the delay slot is **never** executed. **Note**: The annul behavior of an unconditional branch is different from that of a taken conditional branch.

6.3.4.3 CALL and JMPL Instructions

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

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

The CPU always writes all 64 bits of the PC into the destination register. The upper 32 bits of *r*[15] (CALL) or to *r*[*rd*] (JMPL) are *not* cleared when PSTATE.AM = 1.

The CPU implements special JMPL and CALL prediction hardware which is 4-entry, 64-bit FILO to make function returns faster. This hardware is called the Return Prediction Stack (RPS). When a CALL or JMPL that writes to %o7 (*r*[*15*]) occurs, the CPU saves the return address (PC+8) into the RPS entry indexed by some hardware pointer. When the synthetic instructions *retl* (JMPL [%o7+8]) and *ret* (JMPL [%i7+8]) are executed, the return address is predicted to be the address stored in the RPS[CWP].

If the prediction in the RPS is incorrect, the CPU backs up and starts issuing instructions from the correct target address. This backup takes a few extra cycles but does not affect correctness.

Programming Note:

> For maximum performance, software and compilers must take into account how the RPS works. For example, tricks that do nonstandard returns in hopes of boosting performance may require more cycles if they cause the wrong RPS value to be used for predicting the address of the return. Heavily nested calls will also cause earlier entries in the RPS to be overwritten by newer entries, since the RPS has only four entries. Eventually, some return addresses will be mispredicted because of the overflow of the RPS.

6.3.4.4 RETURN Instruction

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

6.3.4.5 DONE and RETRY Instructions

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

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

6.3.4.6 Trap Instruction (Tcc)

The Tcc instruction initiates a trap if the condition specified by its *cond* field matches the current state of the condition code 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. See 5.2.8, "Trap Base Address (TBA) Register".

A Tcc instruction can specify one of 128 software trap types. When a Tcc is taken, 256 plus the 7 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 7, "Traps", for more information.

Programming Note:
> Tcc can be used to implement breakpointing, tracing, and calls to supervisor software. Tcc can also be used for run-time checks, such as out-of-range array index checks or integer overflow checks.

6.3.5 Conditional Move Instructions

6.3.5.1 MOVcc and FMOVcc Instructions

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

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

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

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

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

can be coded as:

```
     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.

6.3.5.2 MOVr and FMOVr Instructions

The MOVr and FMOVr instructions allow the contents of any integer or floating-point register to be moved to a destination integer or floating-point register if a condition specified by the instruction is satisfied. The conditions to test are enumerated in Table 25:

Table 25: MOVr and FMOVr Test Conditions

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

Any of the integer registers 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 non-zero value.

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

6.3.6 Register Window Management Instructions

This subsection describes the instructions used to manage register windows in SPARC64-III. The privileged registers affected by these instructions are described in 5.2.10, "Register-Window State Registers".

6.3.6.1 SAVE Instruction

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

If CANSAVE = 0, execution of a SAVE instruction causes a *window_spill* exception.

If CANSAVE ≠ 0, but the number of clean windows is zero, that is:

$$(\text{CLEANWIN} - \text{CANRESTORE}) = 0$$

then SAVE causes a *clean_window* exception.

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

6.3.6.2 RESTORE Instruction

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

If CANRESTORE = 0, execution of a RESTORE instruction causes a *window_fill* exception.

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

Programming Note:

> This note describes a common convention for use of register windows, SAVE, RESTORE, CALL, and JMPL instructions.
>
> A procedure is invoked by executing a CALL (or a JMPL) instruction. If the procedure requires a register window, it executes a SAVE instruction. A routine that does not allocate a register window of its own (possibly a leaf procedure) should not modify any windowed registers except *out* registers 0 through 6. See H.1.2, "Leaf-Procedure Optimization" in *V9*.
>
> A procedure that uses a register window returns by executing both a RESTORE and a JMPL instruction. A procedure that has not allocated a register window returns by executing a JMPL only. The target address for the JMPL instruction is normally eight plus the address saved by the calling instruction, that is, the instruction after the instruction in the delay slot of the calling instruction.
>
> The SAVE and RESTORE instructions can be used to atomically establish a new memory stack pointer in an *r* register and switch to a new or previous register window. See H.1.4, "Register Allocation within a Window" in *V9*.

6.3.6.3 SAVED Instruction

The SAVED instruction should be used by a spill trap handler to indicate that a window spill has completed successfully. It increments CANSAVE:

$$\text{CANSAVE} \leftarrow (\text{CANSAVE} + 1)$$

If the saved window belongs to a different address space (OTHERWIN ≠ 0), it decrements OTHERWIN:

$$\text{OTHERWIN} \leftarrow (\text{OTHERWIN} - 1)$$

Otherwise, the saved window belongs to the current address space (OTHERWIN = 0), so SAVED decrements CANRESTORE:

$$\text{CANRESTORE} \leftarrow (\text{CANRESTORE} - 1)$$

6.3.6.4 RESTORED Instruction

The RESTORED instruction should be used by a fill trap handler to indicate that a window has been filled successfully. It increments CANRESTORE:

$$\text{CANRESTORE} \leftarrow (\text{CANRESTORE} + 1)$$

If the restored window replaces a window that belongs to a different address space (OTHERWIN ≠ 0), it decrements OTHERWIN:

$$\text{OTHERWIN} \leftarrow (\text{OTHERWIN} - 1)$$

Otherwise, the restored window belongs to the current address space (OTHERWIN = 0), so RESTORED decrements CANSAVE:

$$\text{CANSAVE} \leftarrow (\text{CANSAVE} - 1)$$

If CLEANWIN is less than NWINDOWS – 1, the RESTORED instruction increments CLEANWIN:

if (CLEANWIN < (NWINDOWS – 1)) **then** CLEANWIN ← (CLEANWIN + 1)

6.3.6.5 Flush Windows Instruction

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

$$\text{NWINDOWS} - 2 - \text{CANSAVE}$$

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

6.3.7 State Register Access

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

The supported RDASR and WRASR instructions are described in Table 26; for more information see 5.2.11, "Ancillary State Registers (ASRs)".

Table 26: Supported RDASR and WRASR Instructions

ASR Number	Name	Priv?
0	Y Register	No
2	Condition Code Register	No

Table 26: Supported RDASR and WRASR Instructions

ASR Number	Name	Priv?
3	ASI Register	No
4	Tick Register	Yes/No[a]
5	Program Counter	No
6	Floating-Point Register Status	No
15	Store Barrier, Membar, SIR[b]	Yes/No[c]
18	Hardware Mode Register	Yes
19	Graphic Status Register	No
20	Set Bits in Sched_Int Register	Yes
21	Clear Bits in Sched_Int Register	Yes
22	Sched_Int Register	Yes
23	Tick Match Register	Yes
24	Instruction Fault Type	Yes
25	Scratch Registers	Yes
26	Data Breakpoint Registers	Yes
28	Data Fault Address	Yes
29	Data Fault Type	Yes
30	Performance Monitor Registers	Yes
31	System Control Register	Yes

[a.] Privileged if TICK.NPT = 1, otherwise it is not privileged.

[b.] These are not really considered RD/WRASR instructions by SPARC-V9, but they share the RD/WRASR opcode encoding so they are listed here for clarity.

[c.] SIR is treated as NOP if executed while PSTATE.PRIV = 0.

6.3.8 Privileged Register Access

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

6.3.9 Floating-point Operate (FPop) Instructions

Floating-point operate instructions (FPops) 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. The exceptions are:

■ 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 *fccn* 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 and FBPfcc) or the load/store floating-point instructions.

The FMOVcc instructions function for the floating-point registers as the MOVcc instructions do for the integer registers. See 6.3.5.1, "MOVcc and FMOVcc Instructions."

The FMOVr instructions function for the floating-point registers as the MOVr instructions do for the integer registers. See 6.3.5.2, "MOVr and FMOVr Instructions."

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

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

See 5.1.7.6.2, "ftt = unfinished_FPop" to see which instructions can produce an *unfinished_FPop* exception. See 5.1.7.6.3, "ftt = unimplemented_FPop" to see which instructions can produce an *unimplemented_FPop* exception.

The CPU-specific FMADD and FMSUB instructions (described in 6.3.10, "Implementation-dependent Instructions") are also floating-point operations. They require the floating-point unit to be enabled; otherwise, an *fp_disabled* trap is generated. They also affect the FSR like FPop instructions. However, these instructions are not included in the FPop category and, hence, reserved encodings in these opcodes result in an *illegal_instruction* trap, as defined in section 6.3.11, "Reserved Opcodes and Instruction Fields".

6.3.10 Implementation-dependent Instructions

SPARC-V9 provides two instructions that are entirely implementation-dependent: IMPDEP1 and IMPDEP2.

Compatibility Note:
            The IMPDEP*n* instructions replace the CPop*n* instructions in SPARC-V8.

SPARC64-III has used the IMPDEP2 instruction to implement the Floating-point Multiply-Add/Subtract and Negative Multiply-Add/Subtract instructions; these have an *op3* field = 37_{16} (IMPDEP2). See A.23.1, "IMPDEP2 (Floating-point Multiply-Add/Subtract)", for more full definitions of these instructions. Opcode space is reserved in IMPDEP2 for the quad-precision forms of these instructions. However, SPARC64-III does not currently implement the quad-precision forms, and the CPU takes an *illegal_instruction* exception if a quad precision form is specified. Since these instructions are not part of the required SPARC-V9 architecture, the OS does not supply software emulation routines for the quad versions of these instructions.

6.3.11 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 causes an *fp_exception_other* trap (with FSR.*ftt* = *unimplemented_FPop*); attempting to execute any other reserved opcode shall cause an *illegal_instruction* trap. See Appendix E, "Opcode Maps", for a complete enumeration of the reserved opcodes.

6.3.12 Summary of Unimplemented Instructions

Certain SPARC-V9 instructions are not implemented in hardware in the CPU. Executing any of these instructions results in implementation-dependent behavior, described in Table 27.

Table 27: SPARC64-III Actions on Unimplemented Instructions

Instructions	Trap Taken	SPARC64-III-specific Behavior
Quad FPops: (including FdMULq)	*fp_exception_other*	*FSR.ftt = unimplemented_FPop*
POPC	*illegal_instruction*	None
RDPR FPQ	*illegal_instruction*[a]	There is no FPQ
STQF(A) LDQF(A)	*illegal_instruction*	No hardware quad support

    [a.] SPARC64-III causes an *illegal_instruction* exception rather than the optional *fp_exception_other* (with FSR.*ftt* set to *sequence_error*).

Programming Note:

> The OS emulates all of these instructions except RDPR FQ (SPARC64-III does not have or need an FQ).

6.4 Register Window Management

The state of the register windows is determined by the contents of the set of privileged registers described in 5.2.10, "Register-Window State Registers". Those registers are affected by the instructions described in 6.3.6, "Register Window Management Instructions". Privileged software can read/write these state registers directly by using RDPR/WRPR instructions.

6.4.1 Register Window State Definition

In order for the state of the register windows to be consistent, the following must always be true:

$$CANSAVE + CANRESTORE + OTHERWIN = NWINDOWS - 2$$

Figure 22 on page 64 shows how the register windows are partitioned to obtain the above equation. The partitions are as follows:

■ The current window and the window that overlaps two other valid windows and so must not be used (in Figure 22, windows 0 and 2, respectively) are always present and account for the 2 subtracted from NWINDOWS in the right side of the equation.

■ Windows that do not have valid contents and can be used (via a SAVE instruction) without causing a spill trap. These windows (window 1 in Figure 22) are counted in CANSAVE.

■ Windows that have valid contents for the current address space and can be used (via the RESTORE instruction) without causing a fill trap. These windows (window 4 in Figure 22) are counted in CANRESTORE.

■ Windows that have valid contents for an address space other than the current address space. An attempt to use these windows via a SAVE (RESTORE) instruction results in a spill (fill) trap to a separate set of trap vectors, as discussed in the following subsection. These windows (window 3 in Figure 22) are counted in OTHERWIN.

In addition,

$$CLEANWIN \geq CANRESTORE$$

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

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

Programming Note:
> System software is responsible for keeping the state of the register windows consistent at all times. Failure to do so will cause undefined behavior. For example, CANSAVE, CANRESTORE, and OTHERWIN must never be greater than or equal to 4 (NWINDOWS–1).

6.4.2 Register Window Traps

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

6.4.2.1 Window Spill and Fill Traps

A window overflow occurs when a SAVE instruction is executed and the next register window is occupied (CANSAVE = 0). An overflow causes a spill trap that allows privileged software to save the occupied register window in memory, thereby making it available for use.

A window underflow occurs when a RESTORE instruction is executed and the previous register window is not valid (CANRESTORE = 0). An underflow causes a fill trap that allows privileged software to load the registers from memory.

6.4.2.2 Clean-Window Trap

The CPU provides the *clean_window* trap so that software can create a secure environment in which it is guaranteed that register windows contain only data from the same address space.

A clean register window is one in which all of the registers, including uninitialized registers, contain either zero or data assigned by software executing in the address space to which the window belongs. A clean window cannot contain register values from another process, that is, software operating in a different address space.

Supervisor software specifies the number of windows that are clean with respect to the current address space in the CLEANWIN register. This number includes register windows that can be restored (the value in the CANRESTORE register) and the register windows following CWP that can be used without cleaning. Therefore, the number of clean windows that are available to be used by the SAVE instruction is

$$\text{CLEANWIN} - \text{CANRESTORE}$$

The SAVE instruction causes a *clean_window* trap if this value is zero. This allows supervisor software to clean a register window before it is accessed by a user.

6.4.2.3 Vectoring of Fill/Spill Traps

In order to make handling of fill and spill traps efficient, SPARC-V9 provides multiple trap vectors for the fill and spill traps. These trap vectors are determined as follows:

■ Supervisor software can mark a set of contiguous register windows as belonging to an address space different from the current one. The count of these register windows is kept in the OTHERWIN register. A separate set of trap vectors (*fill_n_other* and *spill_n_other*) is provided for spill and fill traps for these register windows (as opposed to register windows that belong to the current address space).

■ Supervisor software can specify the trap vectors for fill and spill traps by presetting the fields in the WSTATE register. This register contains two subfields, each three bits wide. The WSTATE.NORMAL field is used to determine one of eight spill (fill) vectors to be used when the register window to be spilled (filled) belongs to the current address space (OTHERWIN = 0). If the OTHERWIN register is nonzero, the WSTATE.OTHER field selects one of eight *fill_n_other* (*spill_n_other*) trap vectors.

See Chapter 7, "Traps", for more details on how the trap address is determined.

6.4.2.4 CWP on Window Traps

On a window trap the CWP is set to point to the window that must be accessed by the trap handler, as follows (**Note**: All arithmetic on CWP is done **modulo** NWINDOWS).

■ If the spill trap occurs due to a SAVE instruction (when CANSAVE = 0), there is an overlap window between the CWP and the next register window to be spilled:

$$\text{CWP} \leftarrow (\text{CWP} + 2) \bmod \text{NWINDOWS}$$

If the spill trap occurs due to a FLUSHW instruction, there can be unused windows (CANSAVE) in addition to the overlap window, between the CWP and the window to be spilled:

$$CWP \leftarrow (CWP + CANSAVE + 2) \textbf{ mod } NWINDOWS$$

Implementation Note:
    All spill traps can use

$$CWP \leftarrow (CWP + CANSAVE + 2) \textbf{ mod } NWINDOWS$$

    since CANSAVE is zero whenever a trap occurs due to a SAVE instruction.

■ On a fill trap, the window preceding CWP must be filled:

$$CWP \leftarrow (CWP - 1) \textbf{ mod } NWINDOWS$$

■ On a clean_window trap, the window following CWP must be cleaned. Then

$$CWP \leftarrow (CWP + 1) \textbf{ mod } NWINDOWS$$

6.4.2.5 Window Trap Handlers

The trap handlers for fill, spill, and *clean_window* traps must handle the trap appropriately and return using the RETRY instruction, to reexecute the trapped instruction. The state of the register windows must be updated by the trap handler, and the relationship among CLEANWIN, CANSAVE, CANRESTORE, and OTHERWIN must remain consistent. The following recommendations should be followed:

■ A spill trap handler should execute the SAVED instruction for each window that it spills.

■ A fill trap handler should execute the RESTORED instruction for each window that it fills.

■ A *clean_window* trap handler should increment CLEANWIN for each window that it cleans:

$$CLEANWIN \leftarrow (CLEANWIN + 1)$$

Window trap handlers in SPARC64-III can be very efficient. See H.2.2, "Example Code for Spill Handler" in *V9* for details and sample code.

7 Traps

7.1 Overview

A trap is a vectored transfer of control to supervisor software through a trap table that contains the first eight (thirty-two for clean window, spill, fill, 32i_instruction_access_MMU_ miss, 32i_data_access_MMU_miss, and 32i_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

1. Save certain processor 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.

7.2 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- 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 67 shows the processor state diagram.
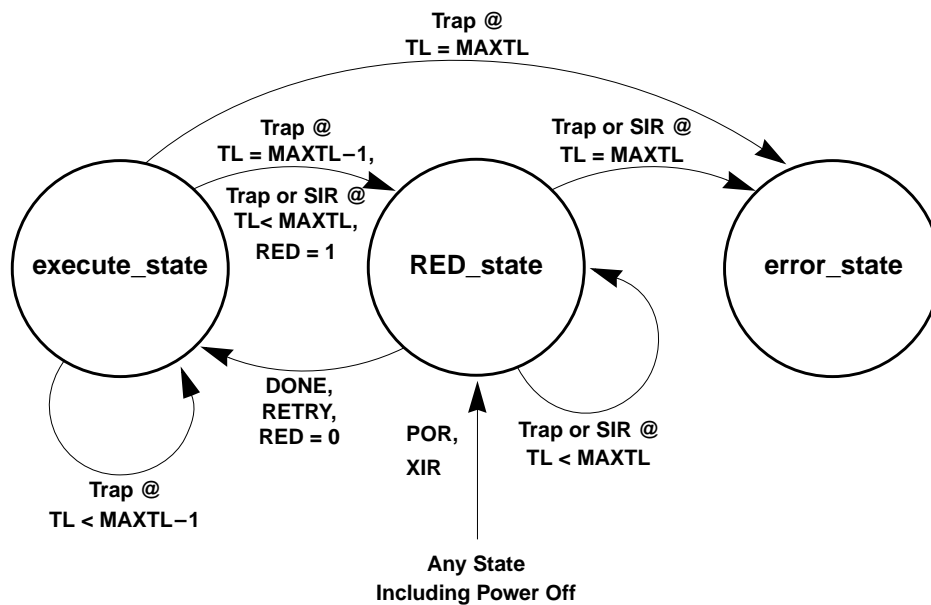
Figure 67: Processor State Diagram (V9=37)

7.2.1 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.

■ Any of the four reset requests occurs (POR, XIR, SIR).

■ System software sets PSTATE.RED = 1.

RED_state serves two mutually exclusive purposes:

■ During trap processing, it indicates that there are no more available trap levels; 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 PC can be changed in concert with the state change.

Programming Note:
> 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 via a WRPR instruction causes the CPU to execute in RED_state. This results in the execution environment, as defined in 7.2.1.2, "RED_state Execution Environment". However, it is different from a RED_state trap in the sense that there are no trap related changes in the machine state (e.g., TL does not change).

7.2.1.1 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. Figure 68 illustrates the RED_state trap vector layout.

Offset	TT	Reason
00_{16}	0	*Reserved* (SPARC-V8 reset)
20_{16}	1	Power-on reset (POR)
60_{16}	$3^{\ddagger}$	Externally initiated reset (XIR)
80_{16}	4	Software-initiated reset (SIR)
$A0_{16}$	*	All other exceptions in RED_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 29 on page 149.

Figure 68: RED_state Trap Vector Layout (V9=38)

The RED_state trap vector is located at an implementation-dependent address referred to as RSTVaddr. In SPARC64-III RSTVaddr is implemented as a 49-bit wide register in the CPU. The value of RSTVaddr is a constant which, for SPARC64-III is FFFF FFFF F000 0000_{16}. This translates to physical address 0000 01FF F000 0000_{16} in RED_state.

7.2.1.2 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.

Programming Note:
> The values are overridden, not set, allowing them to be switched atomically.

SPARC64-III has the following implementation-dependent behavior in RED_state.

■ The CPU executes in sequential mode. This overrides the setting in SCR.*sm*.

■ On entry to and exit from RED_state, the CPU invalidates the I0 cache, prefetch buffers, and the instruction buffer (IB).

■ On entry to RED_state, Main Instruction TLB, Micro Instruction TLB, Main Data TLB, Micro Data TLB are disabled and the corresponding ASR31 bits <30:27> are set. However, control functions like write to Main TLB are still available.

■ While Main TLBs and Micro TLBs are disabled, all accesses are assumed to be non-cacheable and strongly ordered for data access. See F.7.2, "Translation Off Mode" for details.

■ The software can reset ASR31 bits <30:27> during the RED_state. By doing this, Main Data TLB and Micro Data TLB can be re-enabled in RED_state. But Main Instruction TLB and Micro Instruction TLB cannot be re-enabled although the software reset their disable bits in RED_state. RED_state overrides ASR31 bits <27, 29> and assumes they are always one. The bits become effective right after exiting RED_state.

■ XIR Errors are not masked and can cause a trap.

Programming Note:
>   When RED_state is entered due to 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.

7.2.1.3 RED_state Entry Traps

The following traps are processed in RED_state in all cases.

■ **POR** (Power-on reset): Implemented by scan in SPARC64-III; not really a trap

■ **WDR** (Watchdog reset): Not implemented in SPARC64-III

In addition, the following traps are processed in RED_state if TL < MAXTL when the trap is taken. Otherwise it is processed in error_state.

■ **SIR** (Software-initiated Reset)

■ **XIR** (Externally initiated reset)

The following SPARC64-III implementation-dependent traps cause entry into RED_state.

■ Some watchdog timer overflows cause entry to RED_state with the trap type set to *watchdog* (TT = $7F_{16}$). See 5.2.11.12, "State Control Register (ASR 31)", for details.

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 nonreset 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 + $A0_{16}$.

In systems in which it is desired that traps not enter RED_state, the RED_state handler may transfer to the normal trap vector by executing the following code:

```
! Assumptions:
!    -- In RED_state handler, therefore we know that
!       PSTATE.RED = 1, so a WRPR can directly toggle it to 0
!       and, we don't have to worry about intervening traps.
!    -- Registers %g1 and %g2 are available as scratch registers.
...
#define PSTATE_RED     0x0020          ! PSTATE.RED is bit 5
...
rdpr      %tt,%g1                      ! Get the normal trap vector
rdpr      %tba,%g2                     ! address in %g2.
add       %g1,%g2,%g2
rdpr      %pstate,%g1                  ! Read PSTATE into %g1.
jmpl      %g2                          ! Jump to normal trap vector,
wrpr      %g1,PSTATE_RED,%pstate       ! toggling PSTATE.RED to 0.
```

7.2.1.4 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 minimum value for MAXTL is 4; typical usage of the trap levels is shown in Table 28:

Table 28: 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	RED_state handler

Programming Note:
> In order 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.

7.2.2 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 following implementation-dependent condition causes SPARC64-III to enter error_state:

■ An internal CPU watchdog time-out occurs after no instruction has been committed for 2^{32} cycles. This is approximately 17 seconds with a 4 nsec clock.

On entry into error_state, the CPU asserts the output signal P_FERR to the UPA Bus.

Note:

> Entry into error_state due to watchdog time-outs (see <u>5.2.11.12, "State Control Register (ASR</u> <u>31)"</u>) can be disabled by resetting the WDT_EN and WDT_RED bits in ASR31 to 0. This feature should only be used during system bringup in order to allow single-stepping in one processor, while the other processors in an MP system continue to operate on a free-running clock.

7.3 Trap Categories

An exception or interrupt request can cause any of the following trap types:

■ A precise trap

■ A deferred trap

■ A disrupting trap

■ A reset trap

7.3.1 Precise Traps

SPARC64-III will generate a **precise trap for all traps** induced by instruction execution, except for *data_breakpoint* traps.

SPARC64-III implements precise traps using two different approaches; these approaches affect the performance of the CPU but not the semantics of the trap.

Precise Issue Traps (Itraps):

> Itraps (issue traps) are detected when the instruction is issued. The CPU treats the trap as a branch to the trap handler and starts issuing instructions from the corresponding trap handler. Itrap examples include *window_fill*, *window_spill*, and *illegal_ instruction* exceptions, the trap always (TA) instruction, and so on.

Precise Execution Traps (Etraps):

> Etraps (execution traps) are detected when the instruction is being executed. The CPU cancels all instructions following the trap-inducing instruction and waits for all instructions prior to the trap-inducing instruction to complete. If the trap is still pending (there are no prior instruction traps), the CPU takes the trap and starts issuing instructions from the corresponding trap handler. Etrap examples include *fp_exception_other* with *ftt = unimplemented_FPop*, *data_access_exception*, and *division_by_zero* exceptions, and so on.

7.3.2 Deferred Traps

A **deferred trap** is also induced by a particular instruction, but unlike a precise trap, a deferred trap may occur after program-visible state has been changed. Such state may have been changed by the execution of the trap-inducing instruction.

Associated with a particular deferred-trap implementation, there 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:
> Among the actions that software can take after a deferred trap are:

     ●   Emulate the instructions that caused the exception and use RETRY to return control to the instruction at which the deferred trap was invoked, or

     ●   Terminate the program or process associated with the trap.

SPARC64-III implements a deferred trap for the following trap type.

data_breakpoint (SPARC64-III-specific)**:**
> The CPU completes all program visible changes for the trap-inducing instruction. Privileged software can skip the trap-inducing instruction to continue execution in the context of the trap. The priority of the *data_breakpoint* is lower than all other traps that can be induced via a load/store operation. See 5.2.11.9, "Data Breakpoint Registers (ASR26a and ASR26b)", for details.

For deferred traps, the TPC points to the trap inducing instruction, which may have made program visible state changes, as described above. All instructions prior to TPC have completed and all instruction subsequent to TPC remain unexecuted. **Note:** SPARC64-III does not need a deferred-trap queue as described in SPARC-V9, since deferred traps are deferred only within the scope of the trap-inducing instruction.

7.3.3 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; this distinguishes it from precise and deferred traps. When a disrupting trap has been serviced, program execution resumes where it left off. This differentiates disrupting traps from reset traps, which resume execution at the unique reset address.

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 lower than that specified in PIL.

A disrupting trap may be due either to an interrupt request not directly related to a previously executed instruction or to 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. Examples of this are 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 which caused the instruction to induce the trap may lead to unrecoverable errors, since the imple-

mentation may not preserve the necessary state. An example of this 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.

Programming Note:

> Among the actions that trap-handler software might take after a disrupting trap are:

● Use RETRY to return to the instruction at which the trap was invoked

> (PC ← old PC, nPC ← old nPC), or

● Terminate the program or process associated with the trap.

SPARC64-III implements disrupting traps in response to asynchronous events that are detected via input signals as well as for asynchronous errors detected within the CPU. The SPARC64-III CPU recognizes two categories of disrupting traps.

Normal Disrupting Trap:

> A normal disrupting trap is caused by interrupt signals from the UPA bus, setting SCHED_INT register (ASR22), or a watchdog timer overflow when WDT_EN = 1 and WDT_RED = 0 in the SCR register, or by an asynchronous error. See 5.2.11.12, "State Control Register (ASR 31)", for details.

● Interrupts: When the CPU is ready to accept an interrupt signal (based on PSTATE.IE and the PIL), it stops issuing instructions and waits for the CPU to quiesce. It then issues instructions from the corresponding trap handler if the interrupt condition is still valid. The TPC points to the instruction that would have executed in the absence of the disrupting trap. All instructions prior to TPC have completed and all instructions including and subsequent to TPC remain unexecuted.

● Watchdog: The TPC points to the earliest uncommitted instruction.

● Asynchronous error: The TPC points to the instruction that would have executed in the absence of the disrupting trap. All instruction including and subsequent to TPC remain unexecuted.

RED_state Disrupting Trap:

> A RED-state disrupting trap is caused by an external signal (XIR). Watchdog timer overflows can also cause a RED_state disrupting trap if WDT_RED = 1 in the SCR register. See 5.2.11.12, "State Control Register (ASR 31)" for details. When the CPU detects such a signal, it cancels all instructions that have been issued and waits for the CPU to quiesce. The CPU may not be able to cancel a store instruction that has been issued to the D1-Cache. **Note:** The same applies to loads (to I/O registers) that have side effects. The TPC points to the instruction that has not executed (has not made any program-visible state change). The exception to this rule: TPC points to a load/store operation, in which case the load/store operation may have caused program visible state changes. All instructions prior to TPC have completed, and all instructions subsequent to TPC remain unexecuted.

7.3.4 Reset Traps

A **reset trap** occurs when supervisor software or the implementation's hardware deter-mines that the machine must be reset to a known state. Reset traps differ from disrupting traps, since they do not resume execution of the program that was running when the reset trap occurred.

Power-on Reset (POR) is implemented by scanning in the reset state on SPARC64-III.

The following reset traps are defined for SPARC64-III:

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.

Externally initiated reset (XIR):
> Initiated in response to an external signal. This reset trap is normally used for criti-cal system events, such as power failure.

7.3.5 Uses of the Trap Categories

The SPARC-V9 **trap model** stipulates that:

1. Reset traps, except *software_initiated_reset* traps, occur asynchronously to program exe-cution.

2. When recovery from an exception can affect the interpretation of subsequent instruc-tions, such exceptions shall be precise. These exceptions are:
- *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. All exceptions that occur as the result of program execution are precise in SPARC64-III, except for *data_breakpoint*.

4. An exception caused after the initial access of a multiple-access load or store instruction (load-store doubleword, LDSTUB, CASA, CASXA, or SWAP) that causes a catastrophic exception is precise.

5. Exceptions caused by external events unrelated to the instruction stream, such as interrupts, are disrupting.

A deferred trap may occur one instruction after the trap-inducing instruction is dispatched.

The only deferred trap in SPARC64-III is *data_breakpoint*. Even on a deferred trap the TPC points to the instruction that caused the trap and following instructions have not yet executed. Thus SPARC64-III does not need an integer or floating-point deferred-trap queue.

7.4 Trap Control

Several registers control how any given trap is processed:

■ The interrupt enable (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 using the big- or little-endian byte order.

7.4.1 PIL Control

SPARC64-III receives external interrupts from the UPA interconnect. They cause an *interrupt_vector_trap* (TT$=60_{16}$). The interrupt vector trap handler reads the interrupt information and then schedules SPARC-V9 compatible interrupts by writing bits in the SCHED_INT register. See 5.2.11.5, "Schedule Interrupt (SCHED_INT) Register (ASR22)" for details.

If the PIL register is modified (with a WRPR instruction) using a source register (other than *%g0 + imm*), the CPU does not respond to interrupt requests until the WRPR instruction completes (that is, until the new value of PIL is available).

During handling of SPARC-V9 compatible interrupts by the CPU, the PIL register is checked twice. The first check causes the CPU to quiesce to a point where no instructions are active. At this point the value of the PIL register is compared against the current interrupt level to determine if all criteria are still met for the interrupt. If so, the interrupt will be taken.

Between the execution of instructions, the IU prioritizes the outstanding exceptions and interrupt requests according to Table 30 on page 150. At any given time, only the highest priority exception or interrupt request is taken as a trap.[1] When there are multiple out-

standing exceptions or interrupt requests, SPARC-V9 assumes that lower-priority interrupt requests will persist and lower-priority exceptions will recur if an exception-causing instruction is reexecuted.

For interrupt requests, the IU compares the interrupt request level against the processor interrupt level (PIL) register. If the interrupt request level is greater than PIL, the processor takes the interrupt request trap, assuming there are no higher-priority exceptions outstanding.

SPARC64-III takes a normal disrupting trap for external interrupt signals.

7.4.2 TEM Control

The occurrence of floating-point traps of type *IEEE_754_exception* can be controlled with the user-accessible trap enable mask (TEM) field of the FSR. If a particular bit of TEM is 1, the associated *IEEE_754_exception* can cause an *fp_exception_ieee_754* trap.

If a particular bit of TEM is 0, the associated *IEEE_754_exception* does not cause an *fp_exception_ieee_754* trap. Instead, the occurrence of the exception is recorded in the FSR's accrued exception field (*aexc*).

If an *IEEE_754_exception* results in an *fp_exception_ieee_754* trap, then the destination *f* register, *fccn*, and *aexc* fields remain unchanged. However, if an *IEEE_754_exception* does not result in a trap, then the *f* register, *fccn*, and *aexc* fields are updated to their new values.

7.5 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 0 if TL = 0 and to 1 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 0 (hence each trap-table entry is at least 2^5 or 32 bytes long). Figure 69 illustrates this.

TBA<63:15>	TL>0	TT$_{TL}$	00000
63	15 14	13 5	4 0
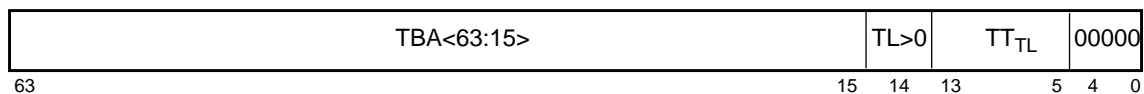
Figure 69: Trap Vector Address (V9=39)

---

1. The highest priority exception or interrupt is the one with the lowest priority value in Table 30. For example, a priority 2 exception is processed before a priority 3 exception.

7.5.1 Trap Table Organization

The trap table layout is as illustrated in Figure 70.

	Trap Table Contents	Trap Type
Value of TL Before the Trap **TL = 0**	Hardware traps	$000_{16}..07F_{16}$
	Spill/fill traps	$080_{16}..0FF_{16}$
	Software traps	$100_{16}..17F_{16}$
	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 70: Trap Table Layout (V9=40)

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 32K bytes. However, if privileged software does not use software traps (Tcc instructions) at TL > 0, the table can be made 24K bytes long.

7.5.2 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] (see 5.2.8, "Trap Base Address (TBA) Register"). 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 Note:

> The spill/fill and *clean_window* trap types are spaced such that their trap-table entries are 128 bytes (32 instructions) long. This allows the complete code for one spill/fill or *clean_window* routine to reside in one trap-table entry.

When a special trap occurs, the TT register is set as described in 7.2.1, "RED_state", Control is then transferred into the RED_state trap table to an address formed by the RST-Vaddr 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. The assignment of TT values to traps is shown in Table 29; Table 30 lists the traps in priority order. Traps marked with '†' are the SPARC64-III-specific traps.

Programming Note:

> These two tables correspond to tables 14 and 15 in *V9*, except that unimplemented traps in SPARC64-III are omitted from tables 29 and 30.

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. Window spill/

fill traps are described in 7.5.2.1. Three subsequent trap vectors are reserved for each spill/
fill vector, to allow for an inline (branchless) trap handle

Table 29: Exception and Interrupt Requests, by TT Value (<u>V9=14</u>)

Exception or Interrupt Request	TT	Priority	Comments
power_on_reset	001_{16}	0	
externally_initiated_reset	003_{16}	1	
software_initiated_reset	004_{16}	1	
RED_state_exception	005_{16}	1	
instruction_access_exception	008_{16}	5	MMU misses–protection exceptions only
instruction_access_error	$00A_{16}$	3	
illegal_instruction	010_{16}	7	
privileged_opcode	011_{16}	6	
fp_disabled	020_{16}	8	
fp_exception_ieee_754	021_{16}	11	
fp_exception_other	022_{16}	11	
tag_overflow	023_{16}	14	
clean_window	$024_{16}..027_{16}$	10	
division_by_zero	028_{16}	15	
data_access_exception	030_{16}	12	
data_access_error	032_{16}	12	
mem_address_not_aligned	034_{16}	10	
LDDF_mem_address_not_aligned	035_{16}	10	
STDF_mem_address_not_aligned	036_{16}	10	
privileged_action	037_{16}	11	
interrupt_level_n $(n = 1..15)$	$041_{16}..04F_{16}$	$32-n$	
interrupt_vector_trap †	060_{16}	16	
data_breakpoint †	061_{16}	14	
programmed_emulation_trap †	062_{16}	6	
async_error †	063_{16}	2	
32i_instruction_access_MMU_miss †	$064_{16}..067_{16}$	2	32 instr. trap for I-TLB miss
32i_data_access_MMU_miss †	$068_{16}..06B_{16}$	12	32 instr. trap for D-TLB miss
32i_data_access_protection †	$06C_{16}..06F_{16}$	12	32 instr. trap for data protection, including write to clean page.
Watchdog †	$07F_{16}$	1	
spill_n_normal $(n = 0..7)$	$080_{16}..09F_{16}$	9	
spill_n_other $(n = 0..7)$	$0A0_{16}..0BF_{16}$	9	
fill_n_normal $(n = 0..7)$	$0C0_{16}..0DF_{16}$	9	
fill_n_other $(n = 0..7)$	$0E0_{16}..0FF_{16}$	9	
trap_instruction	$100_{16}..17F_{16}$	16	

Table 30: Exception and Interrupt Requests, by Priority (0 = High; 31 = Low) (<u>V9=15</u>)

Exception or Interrupt Request	TT	Priority
power_on_reset	001_{16}	0
externally_initiated_reset	003_{16}	1
software_initiated_reset	004_{16}	1
RED_state_exception	005_{16}	1
Watchdog †	$07F_{16}$	1
asnc_error †	063_{16}	2
32i_instruction_access_MMU_miss †	$064_{16}..067_{16}$	2
instruction_access_error	$00A_{16}$	3
instruction_access_exception	008_{16}	5
privileged_opcode	011_{16}	6
programmed_emulation_trap †	062_{16}	6
illegal_instruction	010_{16}	7
fp_disabled	020_{16}	8
spill_n_normal ($n = 0..7$)	$080_{16}..09F_{16}$	9
spill_n_other ($n = 0..7$)	$0A0_{16}..0BF_{16}$	9
fill_n_normal ($n = 0..7$)	$0C0_{16}..0DF_{16}$	9
fill_n_other ($n = 0..7$)	$0E0_{16}..0FF_{16}$	9
clean_window	$024_{16}..027_{16}$	10
mem_address_not_aligned	034_{16}	10
LDDF_mem_address_not_aligned	035_{16}	10
STDF_mem_address_not_aligned	036_{16}	10
fp_exception_ieee_754	021_{16}	11
fp_exception_other	022_{16}	11
privileged_action	037_{16}	11
data_access_exception	030_{16}	12
data_access_error	032_{16}	12
32i_data_access_MMU_miss †	$064_{16}..06B_{16}$	12
32i_data_access_protection †	$06C_{16}..06F_{16}$	12
tag_overflow	023_{16}	14
data_breakpoint †	061_{16}	14
division_by_zero	028_{16}	15
interrupt_vector_trap †	060_{16}	16
trap_instruction	$100_{16}..17F_{16}$	16
interrupt_level_n ($n = 1..15$)	$041_{16}..04F_{16}$	$32-n$

7.5.2.1 Trap Type for Spill/Fill Traps

The trap type for window spill/fill traps is determined based on the contents of the OTH-ERWIN and WSTATE registers as shown in Figure 71:

Trap Type	SPILL_OR_FILL	OTHER	WTYPE	0	0
	8 6	5 4	2	1	0
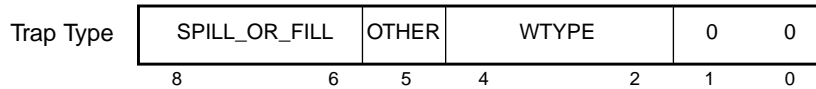
Figure 71: Trap Type Encoding For Spill / Fill Traps

The fields have the following values:

SPILL_OR_FILL:
010_2 for spill traps; 011_2 for fill traps

OTHER:
(OTHERWIN$\neq$0)

WTYPE:
If (OTHER) then WSTATE.OTHER else WSTATE.NORMAL

7.5.3 Details of Supported Traps

7.5.3.1 New 32 Instruction Length Traps

SPARC64-III supports three 32 instruction traps for handling the most performance sensitive MMU traps:

1. *32i_instruction_access_MMU_miss*

2. *32i_data_access_MMU_miss*

3. *32i_data_access_protection*

The first two traps are taken when the main 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 would be when a write request is made to a page marked as clean in the TLB.

Each of these trap vectors takes up 4 slots in the trap table; this means that each trap handlers can contain up to 32 instructions before a branch is needed.

7.5.3.2 Other SPARC64-III Implementation-Specific Traps

SPARC64-III supports the following implementation-specific trap types:

1. *interrupt_vector_trap*

2. *data_breakpoint*

3. *async_error*

4. *watchdog*

5. *programmed_emulation_trap*

7.5.3.3 Unimplemented Traps in **SPARC64-III**

1. Watchdog reset

2. instruction_access_MMU_miss

3. unimplemented_LDD

4. unimplemented_STD

5. internal_processor_error

6. data_access_MMU_miss

7. data_access_protection

8. LDQF_mem_address_not_aligned

9. STQF_mem_address_not_aligned

10. async_data_error

7.5.4 Trap Priorities

Table 29 on page 149 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 29 must remain the same for every implementation.

The trap priorities given above always need to be considered in light of how the CPU 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 occurs because the CPU 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 CPU. 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 on an instruction. Once the early trap is taken, any errors that might have occurred later in the instruction's life will not be seen.

7.5.4.1 Priority Ordering of Priority 1 and 2 Traps

Since multiple priority 1 and 2 traps can occur simultaneously, SPARC64-III enforces the following priority within the priority 1 and 2 traps:

1. XIR

2. SIR

3. async_error

For example, if the SPARC64-III CPU detects an *async_error* and *SIR* simultaneously, it takes the *SIR*.

7.6 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. 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. In order 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. RED_state vectoring and the setting of the TT value for RED_state traps are described in 7.2.1, "RED_state."

Traps that occur with TL = MAXTL – 1 are processed in RED_state. In addition, reset traps are also processed in RED_state. Reset trap processing is described in 7.6.2, "Special Trap Processing." 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. Tables 31, 32, and 33 describe the processor mode and trap-level transitions involved in handling traps:

Table 31: Trap Received While in execute_state (V9=16)

Original State	New State, after receiving trap type			
	Normal Trap or Interrupt	POR	XIR, Impl. Dep.	SIR
execute_state TL < MAXTL – 1	execute_state TL + 1	RED_state MAXTL	RED_state TL + 1	RED_state TL + 1
execute_state TL = MAXTL – 1	RED_state MAXTL	RED_state MAXTL	RED_state MAXTL	RED_state MAXTL
execute_state[†] TL = MAXTL	error_state MAXTL	RED_state MAXTL	error_state MAXTL	error_state 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 32: Trap Received While in RED_state (V9=17)

Original State	New State, after receiving trap type			
	Normal Trap or Interrupt	POR	XIR, Impl. Dep.	SIR
RED_state TL < MAXTL – 1	RED_state TL + 1	RED_state MAXTL	RED_state TL + 1	RED_state TL + 1
RED_state TL = MAXTL – 1	RED_state MAXTL	RED_state MAXTL	RED_state MAXTL	RED_state MAXTL
RED_state TL = MAXTL	error_state MAXTL	RED_state MAXTL	error_state MAXTL	error_state MAXTL

Table 33: Reset Received While in error_state (V9=18)

Original State	New State, after receiving trap type			
	Normal Trap or Interrupt	POR	XIR, Impl. Dep.	SIR
error_state TL < MAXTL – 1	—	RED_state MAXTL	RED_state TL + 1	—
error_state TL = MAXTL – 1	—	RED_state MAXTL	RED_state MAXTL	—
error_state TL = MAXTL	—	RED_state MAXTL	error_state MAXTL	—

Implementation Note:

The processor does not recognize interrupts while it is in error_state.

7.6.1 Normal Trap Processing

A normal trap causes the following state changes to occur:

■ If the processor is already in RED_state, the new trap is processed in RED_state unless TL = MAXTL. See 7.6.2.6, "Normal Traps When the Processor Is in RED_state."

■ 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. See 7.2.1, "RED_state" and 7.6.2.1, "Normal Traps with TL = MAXTL – 1."

■ 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, the processor enters error_state. See 7.2.2, "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	$\leftarrow$ CCR
TSTATE[TL].ASI	$\leftarrow$ ASI
TSTATE[TL].PSTATE	$\leftarrow$ PSTATE
TSTATE[TL].CWP	$\leftarrow$ CWP
TPC[TL]	$\leftarrow$ PC
TNPC[TL]	$\leftarrow$ nPC

■ The trap type is preserved.

| TT[TL] | $\leftarrow$ the trap type |

■ The PSTATE register is updated to a predefined state

PSTATE.MM	is unchanged
PSTATE.RED	$\leftarrow$ 0
PSTATE.PEF	$\leftarrow$ 1 if FPU is present, 0 otherwise
PSTATE.AM	$\leftarrow$ 0 (address masking is turned off)
PSTATE.PRIV	$\leftarrow$ 1 (the processor enters privileged mode)
PSTATE.IE	$\leftarrow$ 0 (interrupts are disabled)
PSTATE.AG	$\leftarrow$ 1 (global regs are replaced with alternate globals)
PSTATE.CLE	$\leftarrow$ 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 $CWP \leftarrow CWP + 1$.

● If $(080_{16} \leq TT[TL] \leq 0BF_{16})$ (window spill trap), then $CWP \leftarrow CWP + CANSAVE + 2$.

● If $(0C0_{16} \leq TT[TL] \leq 0FF_{16})$ (window fill trap), then $CWP \leftarrow CWP–1$.

For nonregister-window traps, CWP is not changed.

■ Control is transferred into the trap table:

$$PC \;\; \leftarrow TBA\langle 63:15\rangle \; \Box \; (TL{>}0) \; \Box \; TT[TL] \; \Box \; 0\ 0000$$
$$nPC \;\; \leftarrow TBA\langle 63:15\rangle \; \Box \; (TL{>}0) \; \Box \; TT[TL] \; \Box \; 0\ 0100$$

where "(TL>0)" is 0 if TL = 0, and 1 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*.

7.6.2 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

■ Implementation-dependent traps

7.6.2.1 Normal Traps with TL = MAXTL − 1

Normal traps that occur when TL = MAXTL − 1 are processed in RED_state. The following state changes occur:

■ 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 if FPU is present, 0 otherwise

PSTATE.AM	$\leftarrow$ 0 (address masking is turned off)
PSTATE.PRIV	$\leftarrow$ 1 (the processor enters privileged mode)
PSTATE.IE	$\leftarrow$ 0 (interrupts are disabled)
PSTATE.AG	$\leftarrow$ 1 (global regs are replaced with alternate globals)
PSTATE.CLE	$\leftarrow$ 0 (big-endian mode for traps)
PSTATE.TLE	$\leftarrow$ 0 (big-endian mode for traps)

■ 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 $\leftarrow$ CWP + 1.

● If (080_{16} ≤ TT[TL] ≤ $0BF_{16}$) (window spill trap), then CWP $\leftarrow$ CWP + CANSAVE + 2.

● If ($0C0_{16}$ ≤ TT[TL] ≤ $0FF_{16}$)(window fill trap), then CWP $\leftarrow$ CWP–1.

For nonregister-window traps, CWP is not changed.

■ Implementation-specific state changes; for example, disabling an MMU

■ Control is transferred into the RED_state trap table

$$PC \leftarrow RSTVaddr\langle 63:8\rangle \parallel 1010\ 0000_2$$
$$nPC \leftarrow RSTVaddr\langle 63:8\rangle \parallel 1010\ 0100_2$$

7.6.2.2 Power-On Reset (POR) Traps

POR traps occur when power is applied to the processor. If the processor is in error_state, a power-on reset (POR) brings the processor out of error_state and places it in RED_state. Processor state is undefined after POR, except for the following:

■ The trap level is set.

TL $\leftarrow$ MAXTL

■ The trap type is set.

TT[TL] $\leftarrow 001_{16}$

■ The PSTATE register is set as follows:

PSTATE.MM	$\leftarrow$ 00_2 (TSO)
PSTATE.RED	$\leftarrow$ 1 (enter RED_state)
PSTATE.PEF	$\leftarrow$ 1 if FPU is present, 0 otherwise
PSTATE.AM	$\leftarrow$ 0 (address masking is turned off)
PSTATE.PRIV	$\leftarrow$ 1 (the processor enters privileged mode)
PSTATE.IE	$\leftarrow$ 0 (interrupts are disabled)

PSTATE.AG	← 1 (global regs are replaced with alternate globals)
PSTATE.TLE	← 0 (big-endian mode for traps)
PSTATE.CLE	← 0 (big-endian mode for non-traps)

■ The TICK register is protected.

TICK.NPT	← 1 (TICK unreadable by nonprivileged software)

■ Implementation-specific state changes; for example, disabling an MMU

■ Control is transferred into the RED_state trap table

$$PC \leftarrow RSTVaddr<63:8> \square \; 0010 \; 0000_2$$
$$nPC \leftarrow RSTVaddr<63:8> \square \; 0010 \; 0100_2$$

For any reset when TL = MAXTL, for all n<MAXTL, the values in TPC[n], TNPC[n], and TSTATE[n] are undefined.

In SPARC64-III POR state is scanned into the CPU from the Scan PROM. After the scan the CPU clock is started and the CPU begins executing the POR trap handler.

See O.1.1, "Power-on Reset (POR)" for more information.

7.6.2.3 Watchdog Reset (WDR) Traps

SPARC64-III does not support Watchdog Reset (WDR) reset traps.

7.6.2.4 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 system-wide reset in a multiprocessor.

The following state changes occur:

■ If TL = MAXTL, the CPU enters *error_state*. Otherwise, it does the following:

■ The trap level is set.

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

■ TT[TL] is set as described below.

■ The PSTATE register is set as follows:

PSTATE.MM $\leftarrow 00_2$ (TSO)

PSTATE.RED $\leftarrow$ 1 (enter RED_state)

PSTATE.PEF $\leftarrow$ 1 if FPU is present, 0 otherwise

PSTATE.AM $\leftarrow$ 0 (address masking is turned off)

PSTATE.PRIV $\leftarrow$ 1 (the processor enters privileged mode)

PSTATE.IE $\leftarrow$ 0 (interrupts are disabled)

PSTATE.AG $\leftarrow$ 1 (global regs are replaced with alternate globals)

PSTATE.CLE $\leftarrow$ 0 (big endian mode for traps)

PSTATE.TLE $\leftarrow$ 0 (big endian mode for traps)

■ Implementation-specific state changes; for example, disabling an MMU.

■ Control is transferred into the RED_state trap table.

PC $\leftarrow$ RSTVaddr<63:8> $\parallel$ 0110 0000$_2$

nPC $\leftarrow$ RSTVaddr<63:8> $\parallel$ 0110 0100$_2$

TT is set in the same manner as for watchdog reset. If the processor is in execute_state when the externally initiated reset (XIR) occurs, TT = 3.

For any reset when TL = MAXTL, for all n<MAXTL, the values in TPC[n], TNPC[n], and TSTATE[n] are undefined.

See O.1.3, "Externally Initiated Reset (XIR)" for more information.

7.6.2.5 Software-initiated Reset (SIR) Traps

Normally in SPARC-V9 CPUs, SIR traps are initiated by executing an SIR instruction. In SPARC64-III, however, SIR is initiated by a WRASR #27 (WRSIR) instruction. Supervisor software uses the SIR trap as a panic operation, or a meta-supervisor trap.

The following state changes occur:

■ If TL = MAXTL, then enter error_state. Otherwise, do the following:

■ The trap level is set.

TL $\leftarrow$ TL + 1

■ Existing state is preserved

TSTATE[TL].CCR $\leftarrow$ CCR

TSTATE[TL].ASI $\leftarrow$ ASI

TSTATE[TL].PSTATE $\leftarrow$ PSTATE

TSTATE[TL].CWP   ← CWP

TPC[TL]   ← PC

TNPC[TL]   ← nPC

■ 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 if FPU is present, 0 otherwise

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 regs are replaced with alternate globals)

PSTATE.CLE   ← 0 (big endian mode for traps)

PSTATE.TLE   ← 0 (big endian mode for traps)

■ 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.

See O.1.4, "Software Initiated Reset (SIR)" for more information.

7.6.2.6 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.

The processor state shall be set as follows:

■ The trap level is set.

TL ← TL + 1

■ Existing state is preserved.

TSTATE[TL].CCR   ← CCR

TSTATE[TL].ASI   ← ASI

TSTATE[TL].PSTATE $\leftarrow$ PSTATE

TSTATE[TL].CWP $\leftarrow$ CWP

TPC[TL] $\leftarrow$ PC

TNPC[TL] $\leftarrow$ nPC

■ The trap type is preserved.

TT[TL] $\leftarrow$ trap type

■ The PSTATE register is set as follows:

PSTATE.MM $\leftarrow$ 00_2 (TSO)

PSTATE.RED $\leftarrow$ 1 (enter RED_state)

PSTATE.PEF $\leftarrow$ 1 if FPU is present, 0 otherwise

PSTATE.AM $\leftarrow$ 0 (address masking is turned off)

PSTATE.PRIV $\leftarrow$ 1 (the processor enters privileged mode)

PSTATE.IE $\leftarrow$ 0 (interrupts are disabled)

PSTATE.AG $\leftarrow$ 1 (global regs are replaced with alternate globals)

PSTATE.CLE $\leftarrow$ 0 (big endian mode for traps)

PSTATE.TLE $\leftarrow$ 0 (big endian mode for traps)

■ 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 $\leftarrow$ CWP + 1.

● If ($080_{16} \leq$ TT[TL] $\leq 0BF_{16}$) (window spill trap), then
CWP $\leftarrow$ CWP + CANSAVE + 2.

● If ($0C0_{16} \leq$ TT[TL] $\leq 0FF_{16}$) (window fill trap), then CWP $\leftarrow$ CWP – 1.

For nonregister-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 $\leftarrow$ RSTVaddr<63:8> ⦂ 1010 0000$_2$

nPC $\leftarrow$ RSTVaddr<63:8> ⦂ 1010 0100$_2$

7.7 Exception and Interrupt Descriptions

The following paragraphs describe the various exceptions and interrupt requests and the conditions that cause them. Each exception and interrupt request describes the corresponding trap type as defined by the trap model. Traps marked with a closed bullet '●' are

defined by SPARC-V9 and implemented in SPARC64-III. Traps marked with a dagger '†' are implementation-dependent and defined only in SPARC64-III. **Note:** This encoding differs from that shown in *V9*. Each trap is marked as precise, deferred, disrupting, or reset. Example exception conditions are included for each exception type. Appendix A, "Instruction Definitions", enumerates which traps can be generated by each instruction.

† **32i_data_access_MMU_miss** [$tt = 068_{16}$ through $06B_{16}$] (Precise ETrap)

This trap occurs when the MMU detects a Main TLB miss while making an data access. This trap takes 4 trap vectors. This allows the main TLB miss handler to execute up to 32 instructions before needing to branch out of the trap handler.

† **32i_data_access_protection** [$tt = 06C_{16}$ through $06F_{16}$] (Precise ETrap)

This trap occurs when the MMU detects a Main TLB protection violation while making a data access. This trap takes 4 trap vectors. This allows the main TLB miss handler to execute up to 32 instructions before needing to branch out of the trap handler.

† **32i_instruction_access_MMU_miss** [$tt = 064_{16}$ through 067_{16}] (Precise ITrap)

This trap occurs when the MMU detects a Main TLB miss while making an instruction access. This trap takes 4 trap vectors. This allows the main TLB miss handler to execute up to 32 instructions before needing to branch out of the trap handler.

† **async_error** [$tt = 063_{16}$] (Disrupting)

The CPU detects an asynchronous error. See P.7.1, "Read/Write TDU Error Log Register", P.7.2, "Read/Write ICU Error Log Register", and P.7.3, "Read/Write DC Error Log Register" for details of the types of hardware errors that can cause this trap.

● **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)

An error occurred on a data access. The detailed information of the data access error is logged into the FTYPE field of Data Access Fault Type Register (ASR29). Below is the list of errors and their descriptions which cause an data_access_error trap.

μDTLB Multiple Hit:

A mulitple hit in uDTLB occurs on a data access.

MTLB Parity Error:

A parity error in MTLB occurs on a data access.

MTLB Multiple Hit:

A mulitple hit in MTLB occurs on a data access.

D1 Cache Tag Parity Error:

A parity error in D1 Cache Tag occurs on a data access.

D1 Cache Tag Multiple Hit:

A multiple hit in D1 Cache Tag occurs on a data access.

D1 Cache Data ECC Single Bit Error:

An ECC single bit error in D1 Cache Data occurs on a data access.

D1 Cache Data ECC Multiple Bit Error:

An ECC multiple bit error in D1 Cache Data occurs on a data access.

UPA Bus Error:

A S_ERR reply is received from the system controller for a data access.

UPA Time Out:

A S_RTO reply is received from the system controller for a data access.

See 5.2.11.10.2, "Data Access Fault Type Register (ASR29)" for theTYPE encoding and the error priority.

● **data_access_exception** [$tt = 030_{16}$] (Precise)

An exception occurred on a data access. The detailed information of the data access error is logged into FTYPE field of Data Access Fault Type Register (ASR29). Below is the list of exceptions and their descriptions which cause an data_access_exception trap.

Invalid ASI:

An attempt to do load or store with undefined or reserved ASI.

Illegal Access to Strongly Ordered Page:

An attempt to access a strongly ordered page by any type of load instruction with non_faulting ASI.
An attempt to access a strongly ordered page 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 Noncacheble Page:

An attempt to access a noncacheable page by atomic instructions (CASA, CASXA, SWAP, SWAPA, LDSTUB, LDSTUBA).
An attempt to access a noncacheable page by atomic quad load instructions (LDDA with ASI=24, 2c).
An attempt to access a noncacheable page by FLUSH instruction.

See 5.2.11.10.2, "Data Access Fault Type Register (ASR29)" for the TYPE encoding and the exception prioprity.

† **data_breakpoint** [$tt = 061_{16}$] (Deferred)

The virtual address and access privilege of a committed load or store match the address and access privilege in the Data Breakpoint Register.

● **division_by_zero** [$tt = 028_{16}$] (Precise)

    An integer divide instruction attempted to divide by zero**.**

● **externally_initiated_reset** [$tt = 003_{16}$] (Reset)

    An external signal was asserted. This trap is used for catastrophic events such as power failure, reset button pressed, and system-wide reset in multiprocessor systems.

● **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 an 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.

● **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 1. The floating-point exception type, *IEEE_754_exception*, is encoded in the FSR.*ftt*, and specific *IEEE_754_exception* information is encoded in FSR.*cexc*.

● **fp_exception_other** [$tt = 022_{16}$] (Precise)

    An FPop instruction generated an exception other than an *IEEE_754_exception*. For example, 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.

● **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, or an instruction that would result in illegal processor state. **Note**: Unimplemented FPop instructions generate *fp_exception_other* traps.

● **instruction_access_error** [$tt = 00A_{16}$] (Precise)

    An error occurred on an instruction access. The detailed information of the instruction access error is logged into the FTYPE field of Instruction Access Fault Type Register (ASR24). Below is the list of errors and their descriptions which cause an instruction_access_error trap.

    I0 Cache Tag Parity Error:

        A parity error in I0 Cache Tag occurs on an instruction access.

    I0 Cache Data Parity Error:

        A parity error in I0 Cache Data occurs on an instruction access.

μ**ITLB Multiple Hit:**

A mulitple hit in uITLB occurs on an instruction access.

MTLB Parity Error:

A parity error in MTLB occurs on an instruction access.

MTLB Multiple Hit:

A mulitple hit in MTLB occurs on an instruction access.

I1 Cache Tag Parity Error:

A parity error in I1 Cache Tag occurs on an instruction access.

I1 Cache Tag Multiple Hit:

A multiple hit in I1 Cache Tag occurs on an instruction access.

I1 Cache Data ECC Single Bit Error:

An ECC single bit error in I1 Cache Data occurs on an instruction access.

I1 Cache Data ECC Multiple Bit Error:

An ECC multiple bit error in I1 Cache Data occurs on an instruction access.

UPA Bus Error:

A S_ERR reply is received from the system controller for an instruction access.

UPA Time Out:

A S_RTO reply is received from the system controller for an instruction access.

See 5.2.11.7, "Instruction Access Fault Type Register (ASR24)" for the TYPE encoding and the error prioprity.

● **instruction_access_exception** [$tt = 008_{16}$] (Precise)

A protection exception occurred on an instruction access. That is, an MMU indicated that the page was not executable.

● **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).

† **interrupt_vector_trap** [$tt = 060_{16}$] (Disrupting)

PSTATE.IE is set and an interrupt transaction (P_INT_REQ) and the interrupt data are received from the UPA bus.

● **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. See A.25, "Load Floating-point".

● **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** [$tt = 001_{16}$] (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. Examples: a data access by nonprivileged software using an ASI value with its most significant bit = 0 (a restricted ASI), or an attempt to read the TICK register by nonprivileged software when TICK.NPT = 1.

● **privileged_opcode** [$tt = 011_{16}$] (Precise)

   An attempt was made to execute a privileged instruction while PSTATE.PRIV = 0.

Compatibility Note:

   This trap type is identical to the SPARC-V8 *privileged_instruction* trap. The name was changed to distinguish it from the new *privileged_action* trap type.

†   **programmed_emulation_trap** (tt = 0x062, priority = 6, Precise).

   Allow emulation of certain instructions that have not been implemented correctly in a particular revision of the SPARC64-III CPU. The encoding of instructions that will cause this trap can be "scanned" into internal registers (not visible to supervisor software) via the debugging console. See 5.2.15, "Emulation Trap Registers" for details.

● **software_initiated_reset** [$tt = 004_{16}$] (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.

● **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. See A.52, "Store Floating-point".

● **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.

† **watchdog** [$tt = 07F_{16}$] (Disrupting)

This trap occurs when the watchdog timer (which will be increased every cycle and reset on any instruction committed) overflows a value specified in the SCR register. Whether this trap is handled in RED_state or normal state is determined by the W_RED bit in the SCR. See 5.2.11.12, "State Control Register (ASR 31)" for details on how the watchdog timer is enabled and controlled.

All other trap types are reserved.

8 Memory Models

Although this chapter contains a great deal of theoretical information, we have included it so that the discussion of the SPARC64-III's memory models in 8.1.1 has sufficient background.

8.1 Introduction

The SPARC-V9 **memory models** define the semantics of memory operations. The instruction set semantics require that loads and stores *seem* to be performed in the order in which they appear in the dynamic control flow of the program. The *actual* order in which they are processed by the memory may be different. The purpose of the memory models is to specify what constraints, if any, are placed on the order of memory operations.

The memory models apply both to uniprocessor and to shared-memory multiprocessors. Formal memory models are necessary in order to precisely define the interactions between multiple processors and input/output devices in a shared-memory configuration. Programming shared-memory multiprocessors requires a detailed understanding of the operative memory model and the ability to specify memory operations at a low level in order to build programs that can safely and reliably coordinate their activities. See Appendix J, "Programming With the Memory Models" in *V9* for additional information on the use of the models in programming real systems.

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 here and defined in Appendix D, "Formal Specification of the Memory Models" in *V9*.

The SPARC-V9 architecture defines three different memory models: **Total Store Order** (**TSO**), **Partial Store Order** (**PSO**), and **Relaxed Memory Order** (**RMO**). All SPARC-V9 processors must provide Total Store Order (or a more strongly ordered model, for example, Sequential Consistency) to ensure SPARC-V8 compatibility.

Whether the PSO or RMO models are supported by SPARC-V9 systems is implementation-dependent; they are not supported as defined by SPARC64-III. See 8.1.1, "SPARC64-III Hardware Memory Models" for details.

Figure 72 shows the relationship of the various SPARC-V9 memory models, from the least restrictive to the most restrictive. Programs written assuming one model will function correctly on any included model.
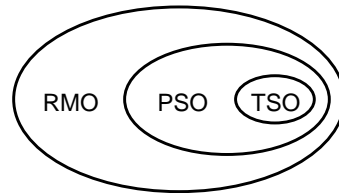
Figure 72: **Memory Models: Least Restrictive (RMO) to Most Restrictive (TSO)**
(V9=41)

SPARC-V9 provides multiple memory models so that:

■   Implementations can schedule memory operations for high performance.

■   Programmers can create synchronization primitives using shared memory.

These models are described informally in this subsection and formally in Appendix D, "Formal Specification of the Memory Models" in *V9*. If there is a conflict in interpretation between the informal description provided here and the formal models, the formal models supersede the informal description.

There is no preferred memory model for SPARC-V9. Programs written for Relaxed Memory Order will work in both Partial Store Order and Total Store Order. Programs written for Partial Store Order will work in Total Store Order. Programs written for a weak model, such as RMO, may execute more quickly, since the model exposes more scheduling opportunities, but may also require extra instructions to ensure synchronization. Multiprocessor programs written for a stronger model will behave unpredictably if run in a weaker model.

Machines that implement **sequential consistency** (also called strong ordering or strong consistency) automatically support programs written for TSO, PSO, and RMO. Sequential consistency is not a SPARC-V9 memory model. In sequential consistency, the loads, stores, and atomic load-stores of all processors are performed by memory in a serial order that conforms to the order in which these instructions are issued by individual processors. A machine that implements sequential consistency may deliver lower performance than an equivalent machine that implements a weaker model. Although particular SPARC-V9 implementations may support sequential consistency, portable software must not rely on having this model available.

Notes About the SPARC64-III Memory Models

From the programmers point of view SPARC64-III completely supports the memory models specified in SPARC-V9.

However, SPARC64-III makes a distinction between the memory model chosen by the programmer for running his code and the underlying memory models supported by the hardware. When a programmer writes a piece of code he assumes that the code will be run

in one of the SPARC-V9 memory models. His code will be written with the proper memory barriers and synchronization for the model he has selected.

SPARC-V9 does not specify exactly how the hardware must support a particular SPARC-V9 memory model, except that the hardware support for the V9 memory model must guarantee that a correct program written for that memory model will run correctly on the hardware. For example, a slightly stronger (more restrictive) hardware memory model might be used than what is required by the SPARC-V9 memory model.

One problem with permanently binding a particular hardware memory model to a SPARC-V9 model is that the optimal hardware memory model may vary based on the system running the program. For example, the optimal binding for a uniprocessor might be different from the optimal binding for a multiprocessor.

Note:

> For the remainder of this chapter, the words "*V9 memory model*" will be used to denote the memory model selected by the programmer in PSTATE.MM, while the words "*hardware memory model*" will be used to denote the underlying hardware memory models.

8.1.1 SPARC64-III Hardware Memory Models

The SPARC64-III supports these hardware memory models:

Load/Store Order (HLSO)

> The CPU orders all loads and stores. This is superset of TSO, PSO and RMO, the SPARC-V9 memory models. Therefore, programs written for TSO, PSO or RMO will always work on SPARC64-III if run under Load/Store Order.

Total Store Order (HTSO)

> All loads are ordered with respect to loads, and all stores are ordered with respect to loads and stores. This is superset of PSO and RMO, the SPARC-V9 memory models. Therefore, programs written for PSO or RMO will always work on SPARC64-III if run under Total Store Order.

Store Order (HSTO)

> All stores are ordered with respect to each other, but loads are not ordered with respect to stores or to other loads. This is a superset of RMO (Relaxed Memory Order). Programs written for RMO will always work on SPARC64-III if run under Store Order.

In uniprocessor systems Store Order can also be used to run application programs written for TSO or PSO, since "program consistency" guarantees that a uniprocessor will not be able to detect the difference between LSO and STO.

Figure 73 illustrates the general relationship among the five memory models; see the note following the picture, however. As in Figure 72, the models are listed in order from least to most restrictive.
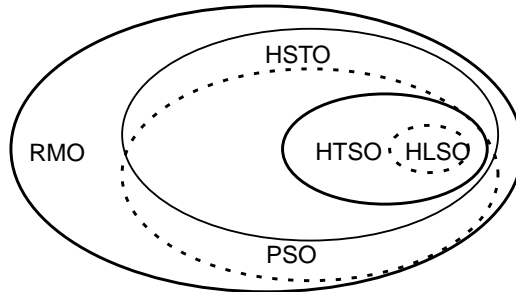
Figure 73: Hardware Memory Models from Least Restrictive to Most Restrictive

8.1.2   Mapping SPARC-V9 Memory Models to Hardware Memory Models

SPARC64-III contains a register that maps the SPARC-V9 memory models to the hardware memory models. This register contains three fields that map the corresponding SPARC-V9 memory model as specified in PSTATE.MM into the Hardware Memory Model. For details of the register containing the mapping fields see 5.2.11.1, "Hardware Mode Register (ASR18)".

Below is the definition of the PSTATE.MM bits:

PSTATE.MM	Memory Model
00	TSO
01	PSO
10	RMO
11	*reserved*

The table below has the encoding of hardware memory models for SPARC64-III:

Hardware Memory Model Encoding	Hardware Memory Model
00	HLSO
01	HTSO
10	HSTO
11	*reserved*

The diagram below show the mapping fields in ASR18. During boot[1] the operating system will write these bits with the mappings that are most appropriate for the current system.

---
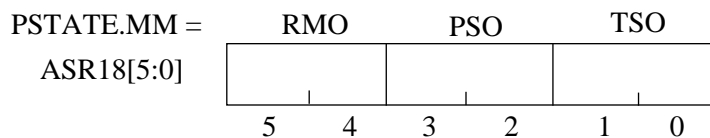
1.   They can be changed at any time, but most likely they will only be written at boot time.

For example, the mapping for a uniprocessor workstation might be:

PSTATE.MM =	RMO	PSO	TSO
ASR18[5:0]	HSTO	HSTO	HSTO
	5 4	3 2	1 0

This mapping works for most uniprocessors since the weaker HSTO can be used for PSO and TSO without causing programs written for PSO or TSO to fail.

The following could be used in a coherent multiprocessor system.

PSTATE.MM =	RMO	PSO	TSO
ASR18[5:0]	HSTO	HTSO	HTSO
	5 4	3 2	1 0

In the MP case TSO must be mapped to the hardware HTSO mode. PSO must also be mapped to the stronger HTSO.

8.2 Memory, Real Memory, and I/O Locations

Memory is the collection of locations accessed by the load and store instructions (described in Appendix A, "Instruction Definitions"). Each location is identified by an address consisting of two elements: an **address space identifier** (ASI), which identifies an address space, and a 64-bit **address,** which is a byte offset into that address space. Memory addresses may be interpreted by the memory subsystem to be either physical addresses or virtual addresses; addresses may be remapped and values cached, provided that memory properties are preserved transparently and coherency is maintained.

When two or more data addresses refer to the same datum, the address is said to be **aliased**. In this case, the processor and memory system must cooperate to maintain consistency; that is, a store to an aliased address must change all values aliased to that address.

Memory addresses identify either real memory or I/O locations.

Real memory stores information without side effects. A load operation returns the value most recently stored. Operations are side-effect-free in the sense that a load, store, or atomic load-store to a location in real memory has no program-observable effect, except upon that location.

I/O locations may not behave like memory and may have side effects. Load, store, and atomic load-store operations performed on I/O locations may have observable side effects, and loads may not return the value most recently stored. The value semantics of operations on I/O locations are **not** defined by the memory models, but the constraints on the order in which operations are performed is the same as it would be if the I/O locations were real memory. The storage properties, contents, semantics, ASI assignments, and addresses of I/O registers are implementation-dependent.

Programming Note:

    It is recommended that all memory pages that contain I/O registers having side effects or requiring strong ordering be mapped with the MMU Strongly Ordered (SO) bit set in the TLB. All accesses

to pages with the SO bit set will be forced to occur in the order they were written by the program-
mer and will not be executed speculatively.

Compatibility Note:

> Operations to I/O locations are **not** guaranteed to be sequentially consistent among themselves, as
> they are in SPARC-V8.
>
> SPARC-V9 does not distinguish real memory from I/O locations in terms of ordering. All refer-
> ences, both to I/O locations and real memory, conform to the memory model's order constraints.
> References to I/O locations may need to be interspersed with MEMBAR instructions to guarantee
> the desired ordering.
>
> Systems supporting SPARC-V8 applications that use memory mapped I/O locations must ensure
> that SPARC-V8 sequential consistency of I/O locations can be maintained when those locations are
> referenced by a SPARC-V8 application. The MMU either must enforce such consistency or cooper-
> ate with system software and/or the processor to provide it.

8.3 Addressing and Alternate Address Spaces

An address in SPARC-V9 is a tuple consisting of an 8-bit address space identifier (ASI)
and a 64-bit byte-address offset in the specified address space. Memory is byte-addressed,
with halfword accesses aligned on 2-byte boundaries, word accesses (which include
instruction fetches) aligned on 4-byte boundaries, extended-word and doubleword
accesses aligned on 8-byte boundaries, and quadword quantities aligned on 16-byte
boundaries. With the possible exception of the cases described in 6.3.1.1, "Memory Align-
ment Restrictions", an improperly aligned address in a load, store, or load-store instruc-
tion always causes a trap to occur. The largest datum that is guaranteed to be atomically
read or written is an aligned doubleword. Also, memory references to different bytes, half-
words, and words in a given doubleword are treated for ordering purposes as references to
the same location. Thus, the unit of ordering for memory is a doubleword.

Programming Note:

> While the doubleword is the coherency unit for update, programmers should not assume that dou-
> bleword floating-point values are updated as a unit unless they are doubleword-aligned and always
> updated using double-precision loads and stores. Some programs use pairs of single-precision oper-
> ations to load and store double-precision floating-point values when the compiler cannot determine
> that they are doubleword-aligned. Also, while quad-precision operations are defined in the
> SPARC-V9 architecture, the granularity of loads and stores for quad-precision floating-point values
> may be word or doubleword.

The processor provides an address space identifier with every address. This ASI may serve
several purposes:

■ To identify which of several distinguished address spaces the 64-bit address offset is to
  be interpreted as addressing

■ To provide additional access control and attribute information, for example, the pro-
  cessing which is to be taken if an access fault occurs or to specify the endian-ness of
  the reference

■ To specify the address of an internal control register in the processor, cache, or mem-
  ory management hardware

The memory management hardware can associate an independent 2^{64}-byte memory address space with each ASI. If this is done, it becomes possible to allow system software easy access to the address space of the faulting program when processing exceptions, or to implement access to a client program's memory space by a server program.

The architecturally specified ASIs are listed in Appendix L, "ASI Assignments".

When TL = 0, normal accesses by the processor to memory when fetching instructions and performing loads and stores implicitly specify ASI_PRIMARY or ASI_PRIMARY_ LITTLE, depending on the setting of the PSTATE.CLE bit.

When TL > 0 the implicit ASI for instruction and data fetches is ASI_NUCLEUS. Loads and stores will use ASI_NUCLEUS if PSTATE.CLE = 0 or ASI_NUCLEUS_LITTLE if PSTATE.CLE = 1. (Impl. Dep. #124)

SPARC64-III supports the PRIMARY{_LITTLE}, SECONDARY{_LITTLE}, and NUCLEUS{_LITTLE} address spaces.

Accesses to other address spaces use the load/store alternate instructions. For these accesses, the ASI is either contained in the instruction (for the register-register addressing mode) or taken from the ASI register (for register-immediate addressing).

ASIs are either nonrestricted or restricted. A nonrestricted ASI is one that may be used independent of the privilege level (PSTATE.PRIV) at which the processor is running. Restricted ASIs require that the processor be in privileged mode for a legal access to occur. Restricted ASIs have their high-order bit equal to zero. The relationship between processor state and ASI restriction is shown in Table 23 on page 122.

Several restricted ASIs must be provided: ASI_AS_IF_USER_PRIMARY{_LITTLE} and ASI_AS_IF_USER_SECONDARY{_LITTLE}. The intent of these ASIs is to give system software efficient access to the memory space of a program.

The normal address space is **primary address space**, which is accessed by the unrestricted ASI_PRIMARY{_LITTLE}. The **secondary address space**, which is accessed by the unrestricted ASI_SECONDARY{_LITTLE}, is provided to allow a server program to access a client program's address space.

ASI_PRIMARY_NOFAULT{_LITTLE} and ASI_SECONDARY_NOFAULT{_LIT-TLE} support **nonfaulting loads.** These ASIs are aliased to ASI_PRIMARY{_LITTLE} and ASI_SECONDARY{_LITTLE}, respectively, and have exactly the same action. They may be used to color (that is, distinguish into classes) loads in the instruction stream so that, in combination with a judicious mapping of low memory and a specialized trap handler, an optimizing compiler can move loads outside of conditional control structures.

Programming Note:
> Nonfaulting loads allow optimizations that move loads ahead of conditional control structures that guard their use; thus, they can minimize the effects of load latency by improving instruction scheduling. The semantics of nonfaulting loads are the same as for any other load, except when non-recoverable catastrophic faults occur (for example, address-out-of-range errors). When such a fault occurs, it is ignored and the hardware and system software cooperate to make the load appear to complete normally, returning a zero result. The compiler's optimizer generates load-alternate instructions with the ASI field or register set to ASI_PRIMARY_NOFAULT{_LITTLE} or ASI_SECONDARY_NOFAULT{_LITTLE} for those loads it determines should be nonfaulting. To

minimize unnecessary processing if a fault does occur, it is desirable to map low addresses (especially address zero) to a page of all zeros, so that references through a NULL pointer do not cause unnecessary traps.

8.4 SPARC-V9 Memory Model

The SPARC-V9 processor architecture specifies the organization and structure of a SPARC-V9 central processing unit but does not specify a memory system architecture. Appendix F, "MMU Architecture", summarizes the MMU support required by a SPARC-V9 central processing unit.

The memory models specify the possible order relationships between memory-reference instructions issued by a processor and the order and visibility of those instructions as seen by other processors. The memory model is intimately intertwined with the program execution model for instructions.

8.4.1 SPARC-V9 Program Execution Model

The SPARC-V9 processor model consists of three units: an issue unit, a reorder unit, and an execute unit, as shown in Figure 74.

The issue unit reads instructions over the instruction path from memory and issues them in **program order**. Program order is precisely the order determined by the control flow of the program and the instruction semantics, under the assumption that each instruction is performed independently and sequentially.

Issued instructions are collected, reordered, and then dispatched to the execute unit. Instruction reordering allows an implementation to perform some operations in parallel and to better allocate resources. The reordering of instructions is constrained to ensure that the results of program execution are the same as they would be if the instructions were performed in program order. This property is called **processor self-consistency**.
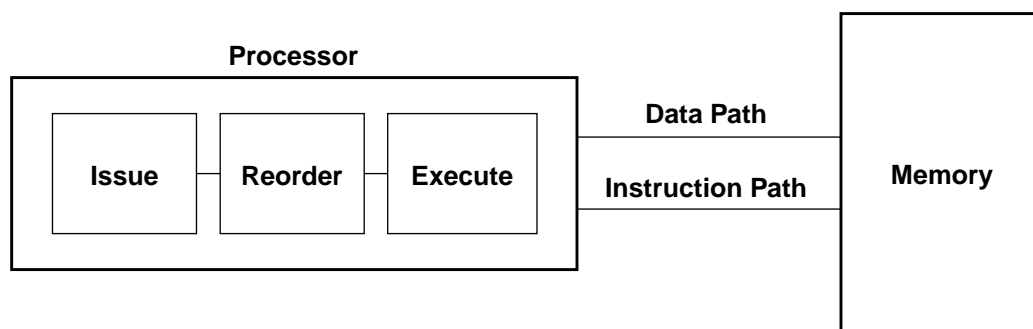
Figure 74: Processor Model: Uniprocessor System (V9=42)

Processor self-consistency requires that the result of execution, in the absence of any shared memory interaction with another processor, be identical to the result that would be observed if the instructions were performed in program order. In the model in Figure 74, instructions are issued in program order and placed in the reorder buffer. The processor is

allowed to reorder instructions, provided it does not violate any of the data-flow constraints for registers or for memory.

The data-flow order constraints for register reference instructions are:

1. An instruction cannot be performed until all earlier instructions that set a register it uses have been performed (read-after-write hazard; write-after-write hazard).

2. An instruction cannot be performed until all earlier instructions that use a register it sets have been performed (write-after-read hazard).

An implementation can avoid blocking instruction execution in case 2 by using a renaming mechanism that provides the old value of the register to earlier instructions and the new value to later uses.

The data-flow order constraints for memory-reference instructions are those for register reference instructions, plus the following additional constraints:

1. A memory-reference instruction that sets (stores to) a location cannot be performed until all previous instructions that use (load from) the location have been performed (write-after-read hazard).

2. A memory-reference instruction that uses (loads) the value at a location cannot be performed until all earlier memory-reference instructions that set (store to) the location have been performed (read-after-write hazard).

As with the case for registers, implementations can avoid blocking instructions in case (2) by providing an additional mechanism, in this case, a write buffer which guarantees that the value returned by a load is that which would be returned by the most recent store, even though the store has not completed. As a result, the value associated with an address may appear to be different when observed from a processor that has written the location and is holding the value in its write buffer than it would be when observed from a processor that references memory (or its own write buffer). Moreover, the load that was satisfied by the write buffer never appears at the memory.

Memory-barrier instructions (MEMBAR and STBAR) and the active memory model specified by PSTATE.MM also constrain the issue of memory-reference instructions. See 8.4.3, "MEMBAR Instruction," and 8.4.4, "Memory Models," for a detailed description.

The constraints on instruction execution assert a partial ordering on the instructions in the reorder buffer. Every one of the several possible orderings is a legal execution ordering for the program. See Appendix D, "Formal Specification of the Memory Models" in *V9* for more information.

8.4.2 Processor / Memory Interface Model

Each processor in a multiprocessor system is modeled as shown in Figure 75; that is, having two independent paths to memory: one for instructions and one for data. Caches and mappings are considered to be part of the memory. Data caches are maintained by hardware to be consistent (coherent). Instruction caches need not be kept consistent with data caches and, therefore, require explicit program action to ensure consistency when a pro-

gram modifies an executing instruction stream. Memory is shared in terms of address space, but it may be inhomogeneous and distributed in an implementation. Mapping and caches are ignored in the model, since their functions are transparent to the memory model.[1]

In real systems addresses may have attributes that the processor must respect. The processor executes loads, stores, and atomic load-stores in whatever order it chooses, as constrained by program order and the current memory model. The ASI address-couples it generates are translated by a memory management unit (MMU), which associates attributes with the address and may, in some instances, abort the memory transaction and signal an exception to the CPU. For example, a region of memory may be marked as non-prefetchable, noncacheable, read-only, or restricted. It is the MMU's responsibility, working in conjunction with system software, to ensure that memory attribute constraints are not violated. See Appendix F, "MMU Architecture", for more information.

Instructions are performed in an order constrained by local dependencies. Using this dependency ordering, an execution unit submits one or more pending memory transactions to the memory. The memory performs transactions in **memory order**. The memory unit may perform transactions submitted to it out of order; hence, the execution unit must not submit two or more transactions concurrently that are required to be ordered.
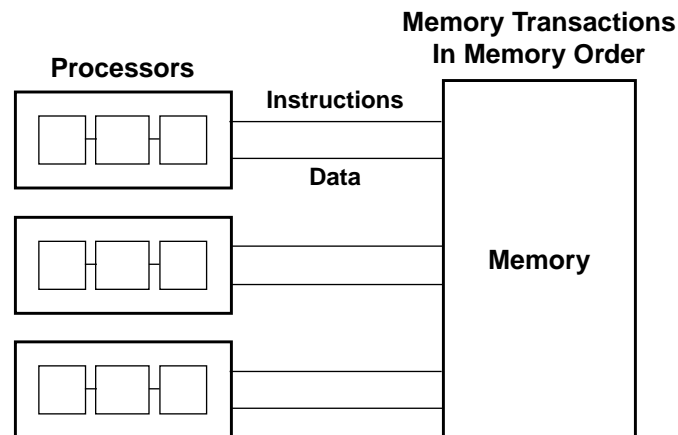
Figure 75: **Data Memory Paths: Multiprocessor System (V9=43)**

The memory accepts transactions, performs them, and then acknowledges their completion. Multiple memory operations may be in progress at any time and may be initiated in a nondeterministic fashion in any order, provided that all transactions to a location preserve the per-processor partial orders. Memory transactions may complete in any order. Once initiated, all memory operations are performed atomically: loads from one location all see the same value, and the result of stores are visible to all potential requestors at the same instant.

---

1. The model described here is only a model; implementations of SPARC-V9 systems are unconstrained, as long as their observable behaviors match those of the model.

The order of memory operations observed at a single location is a **total order** that preserves the partial orderings of each processor's transactions to this address. There may be many legal total orders for a given program's execution.

8.4.3   MEMBAR Instruction

MEMBAR serves two distinct functions in SPARC-V9. One variant of the MEMBAR, the ordering MEMBAR, provides a way for the programmer to control the order of loads and stores issued by a processor. The other variant of MEMBAR, the sequencing MEMBAR, allows the programmer to explicitly control order and completion for memory operations. Sequencing MEMBARs are needed only when a program requires that the effect of an operation become globally visible, rather than simply being scheduled.[1] As both forms are bit-encoded into the instruction, a single MEMBAR can function both as an ordering MEMBAR and as a sequencing MEMBAR.

MEMBAR#Lookaside, MEMBAR#StoreStore, and MEMBAR#LoadStore are treated as NOPs in SPARC64-III, since the hardware memory models always enforce the semantics of these MEMBARs for all memory accesses. MEMBAR#StoreLoad and MEMBAR#LoadLoad enforce the ordering specified by the instruction in the Load/Store Unit in SPARC64-III. MEMBAR#Sync and MEMBAR#MemIssue cause the processor to sync and cause the effects of all cacheable and noncacheable memory accesses made before the MEMBAR to be visible from the other processors in the system.

8.4.3.1   Ordering MEMBAR Instructions

Ordering MEMBAR instructions induce an ordering in the instruction stream of a single processor. Sets of loads and stores that appear before the MEMBAR in program order are ordered with respect to sets of loads and stores that follow the MEMBAR in program order. Atomic operations (LDSTUB(A), SWAP(A), CASA, and CASXA) are ordered by MEMBAR as if they were both a load and a store, since they share the semantics of both. An STBAR instruction, with semantics that are a subset of MEMBAR, is provided for SPARC-V8 compatibility. MEMBAR and STBAR operate on all pending memory operations in the reorder buffer, independent of their address or ASI, ordering them with respect to all future memory operations. This ordering applies only to memory-reference instructions issued by the processor issuing the MEMBAR. Memory-reference instructions issued by other processors are unaffected.

The ordering relationships are bit-encoded as shown in Table 34. For example, MEMBAR 01_{16}, written as "`membar   #LoadLoad`" in assembly language, requires that all load operations appearing before the MEMBAR in program order complete before any of the load operations following the MEMBAR in program order complete. Store operations are unconstrained in this case. MEMBAR 08_{16} (`#StoreStore`) is equivalent to the STBAR instruction; it requires that the values stored by store instructions appearing in program

---

1. Sequencing MEMBARs are needed for some input/output operations, forcing stores into specialized stable storage, context switching, and occasional other systems functions. Using a Sequencing MEMBAR when one is not needed may cause a degradation of performance. See Appendix J, "Programming With the Memory Models" in *V9* for examples of their use.

order prior to the STBAR instruction be visible to other processors prior to issuing any store operations that appear in program order following the STBAR.

In Table 34 these ordering relationships are specified by the '$<m$' symbol, which signifies memory order. See Appendix D, "Formal Specification of the Memory Models" in *V9* for a formal description of the $<m$ relationship.

Table 34: Ordering Relationships Selected by Mask (V9=19)

Ordering Relation, Earlier < Later	Suggested Assembler Tag	Mask Value	*nmask* Bit #
Load $<m$ Load	`#LoadLoad`	01_{16}	0
Store $<m$ Load	`#StoreLoad`	02_{16}	1
Load $<m$ Store	`#LoadStore`	04_{16}	2
Store $<m$ Store	`#StoreStore`	08_{16}	3

Selections may be combined to form more powerful barriers. For example, a MEMBAR instruction with a mask of 09_{16} (`#LoadLoad | #StoreStore`) orders loads with respect to loads and stores with respect to stores, but it does not order loads with respect to stores or vice versa.

Programming Note:

> Future versions of HAL machines will support several MEMBAR variations. Thus, programs should use the correct MEMBAR encoding for upward compatibility.

8.4.3.2 Sequencing MEMBAR Instructions

A sequencing MEMBAR exerts explicit control over the completion of operations. There are three sequencing MEMBAR options, each with a different degree of control and a different application.

Lookaside Barrier:

> Ensures that loads following this MEMBAR are from memory and not from a lookaside into a write buffer. **Lookaside Barrier** requires that pending stores issued prior to the MEMBAR be completed before any load from that address following the MEMBAR may be issued. A **Lookaside Barrier** MEMBAR may be needed to provide lock fairness and to support some plausible I/O location semantics. See the example in J.14.1, "I/O Registers With Side Effects" in *V9*.
> SPARC64-III ensures this sequencing all the time. Therefore this sequencing MEMBAR is treated as a NOP in the CPU.

Memory Issue Barrier:

> Ensures that all memory operations appearing in program order before the sequencing MEMBAR complete before any new memory operation may be initiated. See the example in J.14.2, "The Control and Status Register (CSR)" in *V9*.
> This sequencing MEMBAR behaves in the same way as the Synchronization Barrier MEMBAR in SPARC64-III.

Synchronization Barrier:

> Ensures that all instructions (memory reference and others) preceding the MEMBAR complete and the effects of any fault or error have become visible before any

instruction following the MEMBAR in program order is initiated. A **Synchroniza-tion Barrier** MEMBAR fully synchronizes the processor that issues it.

Table 35 shows the encoding of these functions in the MEMBAR instruction.

Table 35: Sequencing Barrier Selected by Mask (<u>V9=20</u>)

Sequencing Function	Assembler Tag	Mask Value	*cmask* Bit #
Lookaside Barrier	`#Lookaside`	10_{16}	0
Memory Issue Barrier	`#MemIssue`	20_{16}	1
Synchronization Barrier	`#Sync`	40_{16}	2

8.4.4 Memory Models

The SPARC-V9 memory models are defined below in terms of order constraints placed upon memory-reference instruction execution, in addition to the minimal set required for self-consistency. These order constraints take the form of MEMBAR operations implicitly performed following some memory-reference instructions.

8.4.4.1 Relaxed Memory Order (RMO)

Relaxed Memory Order places no ordering constraints on memory references beyond those required for processor self-consistency. When ordering is required, it must be provided explicitly in the programs using MEMBAR instructions.

8.4.4.2 Partial Store Order (PSO)

Partial Store Order may be provided for compatibility with existing SPARC-V8 programs. Programs that execute correctly in the RMO memory model will execute correctly in the PSO model.

The rules for PSO are:

■ Loads are blocking and ordered with respect to earlier loads.

■ Atomic load-stores are ordered with respect to loads.

Thus, PSO ensures that:

■ Each load and atomic load-store instruction behaves as if it were followed by a MEM-BAR with a mask value of 05_{16}.

■ Explicit MEMBAR instructions are required to order store and atomic load-store instructions with respect to each other.

8.4.4.3 Total Store Order (TSO)

Total Store Order must be provided for compatibility with existing SPARC-V8 programs. Programs that execute correctly in either RMO or PSO will execute correctly in the TSO model.

The rules for TSO are:

■ Loads are blocking and ordered with respect to earlier loads.

■ Stores are ordered with respect to stores.

■ Atomic load-stores are ordered with respect to loads and stores.

Thus, TSO ensures that:

■ Each load instruction behaves as if it were followed by a MEMBAR with a mask value of 05_{16}.

■ Each store instruction behaves as if it were followed by a MEMBAR with a mask of 08_{16}.

■ Each atomic load-store behaves as if it were followed by a MEMBAR with a mask of $0D_{16}$.

8.4.5 Mode Control

The memory model is specified by the two-bit state in PSTATE.MM, described in 5.2.1.3, "PSTATE_mem_model (MM)".

Writing a new value into PSTATE.MM causes subsequent memory reference instructions to be performed with the order constraints of the specified memory model.

SPARC-V9 processors need not provide all three memory models; undefined values of PSTATE.MM have implementation-dependent effects.

Except when a trap enters RED_state, PSTATE.MM is left unchanged when a trap is entered and the old value is stacked. When entering RED_state, the value of PSTATE.MM is set to TSO.

8.4.6 Hardware Primitives for Mutual Exclusion

In addition to providing memory-ordering primitives that allow programmers to construct mutual-exclusion mechanisms in software, SPARC-V9 provides three hardware primitives for mutual exclusion:

■ Compare and Swap (CASA, CASXA)

■ Load Store Unsigned Byte (LDSTUB, LDSTUBA)

■ Swap (SWAP, SWAPA)

Each of these instructions has the semantics of both a load and a store in all three memory models. They are all **atomic,** in the sense that no other store can be performed between the load and store elements of the instruction. All of the hardware mutual exclusion operations conform to the memory models and may require barrier instructions to ensure proper data visibility.

When the hardware mutual-exclusion primitives address I/O locations, the results are implementation-dependent. In addition, the atomicity of hardware mutual-exclusion primitives is guaranteed only for processor memory references and not when the memory location is simultaneously being addressed by an I/O device such as a channel or DMA.

The Compare and Swap instructions (CASA and CASXA) serialize the SPARC64-III CPU (see section 6.1.3, "Serializing Instructions" for details).

The Load Store Unsigned Byte (LDSTUB and LDSTUBA) and the Swap (SWAP and SWAPA) instructions have implementation-specific memory ordering behavior in the SPARC64-III CPU. If the CPU is running with the SPARC64-III HLSO model, the behavior is as expected. Since these instructions have both load and store semantics, they are executed in order.

8.4.6.1 Compare and Swap (CASA, CASXA)

Compare-and-swap is an atomic operation that compares a value in a processor register to a value in memory, and, if and only if they are equal, swaps the value in memory with the value in a second processor register. Both 32-bit (CASA) and 64-bit (CASXA) operations are provided. The compare-and-swap operation is atomic in the sense that once begun, no other processor can access the memory location specified until the compare has completed and the swap (if any) has also completed and is potentially visible to all other processors in the system.

Compare-and-swap is substantially more powerful than the other hardware synchronization primitives. It has an infinite consensus number; that is, it can resolve, in a wait-free fashion, an infinite number of contending processes. Because of this property, compare-and-swap can be used to construct wait-free algorithms that do not require the use of locks. See Appendix J, "Programming With the Memory Models" in *V9* for examples.

8.4.6.2 Swap (SWAP)

SWAP atomically exchanges the lower 32 bits in a processor register with a word in memory. Swap has a consensus number of two; that is, it cannot resolve more than two contending processes in a wait-free fashion.

8.4.6.3 Load Store Unsigned Byte (LDSTUB)

LDSTUB loads a byte value from memory to a register and writes the value FF_{16} into the addressed byte atomically. LDSTUB is the classic test-and-set instruction. Like SWAP, it has a consensus number of two and so cannot resolve more than two contending processes in a wait-free fashion.

8.4.7 Synchronizing Instruction and Data Memory

The SPARC-V9 memory models do not require that instruction and data memory images be consistent at all times. The instruction and data memory images may become inconsistent if a program writes into the instruction stream. As a result, whenever instructions are modified by a program in a context where the data (that is, the instructions) in the memory

and the data cache hierarchy may be inconsistent with instructions in the instruction cache hierarchy, some special programmatic action must be taken.

The FLUSH instruction will ensure consistency between the instruction stream and the data references across any local caches for a particular doubleword value in the processor executing the FLUSH. It will ensure eventual consistency across all caches in a multiprocessor system. The programmer must be careful to ensure that the modification sequence is robust under multiple updates and concurrent execution. Since, in the general case, loads and stores may be performed out of order, appropriate MEMBAR and FLUSH instructions must be interspersed as needed to control the order in which the instruction data is mutated.

The FLUSH instruction ensures that subsequent instruction fetches from the doubleword target of the FLUSH by the processor executing the FLUSH appear to execute after any loads, stores, and atomic load-stores issued by the processor to that address prior to the FLUSH. FLUSH acts as a barrier for instruction fetches in the processor that executes it and has the properties of a store with respect to MEMBAR operations.

FLUSH has no latency on the issuing processor; the modified instruction stream is immediately available.[1]

If all caches in a system (uniprocessor or multiprocessor) have a unified cache consistency protocol, FLUSH need do nothing for correctness.

Use of FLUSH in a multiprocessor environment may cause unexpected performance degradation in some systems, because every processor that may have a copy of the modified data in its instruction cache must invalidate that data. In the worst case naive system, **all** processors must invalidate the data.

Programming Note:
> Because FLUSH is designed to act on a doubleword, and because, on some implementations, FLUSH may trap to system software, it is recommended that system software provide a user-callable service routine for flushing arbitrarily sized regions of memory. On some implementations, this routine would issue a series of FLUSH instructions; on others, it might issue a single trap to system software that would then flush the entire region.

---

1. SPARC-V8 specified a five-instruction latency. Invalidation of instructions in execution in the instruction cache is likely to force an instruction-cache fault.

9 Guidelines for Instruction Scheduling

9.1 Introduction

SPARC64-III is a Superscalar RISC Processor that implements the SPARC-V9 architecture. It can issue at most four instructions per clock; that is, it is a four-way superscalar machine. It has a data flow back-end that can commit up to eight instructions per clock. Up to 63 instructions can be in progress in the machine at any time, where "in progress" refers to instructions that have been *issued* but not yet *reclaimed*. (Terms that describe instruction states are defined in 9.1.1, "Life Cycle of an Instruction".) The average number of instructions committed per clock is called the IPC (Instructions Per Cycle). All current superscalar processors have constraints that cause them to issue and commit fewer than the maximum number of instructions per clock on average; SPARC64-III is no exception.

This chapter outlines strategies that the compiler writer can use to more optimally generate code for SPARC64-III. The text describes some constraint, limitation, or feature of the CPU and then suggests strategies to avoid the constraint or to utilize the feature. Before proceeding you should familiarize yourself with the terminology and concepts introduced in 3.4, "SPARC64-III Processor Architecture".

Note:

> In order to determine the version of a SPARC64 CPU, code can examine the Version (VER) register, in particular, bits <47:32>, VER.*impl*; its value is '3' for a SPARC64-III CPU. The entire VER register is reproduced below:
>
> VER: 0004 0003 XX00 0404$_{16}$

9.1.1 Life Cycle of an Instruction

The following terms describe the states that an instruction goes through in its lifetime. They are commonly used by the CPU designers when discussing SPARC64-III internal states.

Fetched:

> Instructions are *fetched* from memory, the external U2 cache, the internal I1 cache, or the internal I0 cache; they are then sent to the Issue Unit.

Issued:

> An instruction is *issued* when it is assigned a serial number.

Dispatched:

An instruction is *dispatched* when it sent to a functional unit queue. For example, an ADD instruction is considered *dispatched* when it is sent to the queue for one of the adders.

Initiated:

An instruction is *initiated* when it has all of the resources it needs (for example, its source operands) and it has been selected for execution (that is, it enters an execution unit).

Executed:

An instruction is *executed* by an execution unit such as a Floating-point Multiply Adder (FMA). An instruction is *in execution* as long as it is still being processed by an execution unit.

Finished:

An instruction is *finished* when it has completed execution in an execution unit and has written its results onto a result bus. Results on the result busses go to register files and to waiting instructions in the in the instruction queues.

Completed:

An instruction is *completed* when it has *finished* and has sent a non-error status to the Issue Unit (ISU).

Note:

Although the state of the machine has been temporarily altered when an instruction is *completed*, the state has not yet been permanently changed and the old state can be recovered up until the time that the instruction is *committed*.

Committed:

An instruction can be *committed* only when it has *completed* **without error** and **all** prior instructions have *completed* **without error.** When an instruction is *committed* the state of the machine is permanently changed to reflect the result of the instruction.

Reclaimed:

All resources relating to the instruction that were held until it was *committed* have been released and are available for use by subsequent instructions. Instruction resources usually are reclaimed a few cycles after the instruction is *committed* (for example, the serial number can be reused).

Figure 76 illustrates the SPARC64-III pipeline as it relates to the instruction states described above.
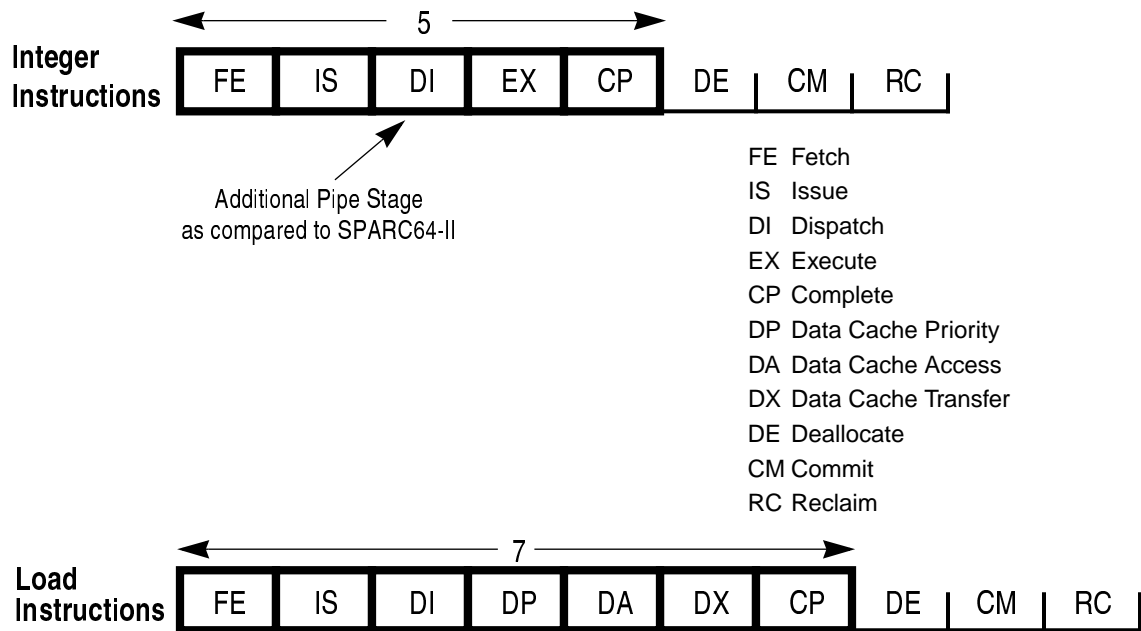
Figure 76: SPARC64-III Pipeline Diagram with Instruction States

9.2 Instruction Fetch

During the fetch phase the Fetch Unit fetches instructions from one of these locations:

■ The internal Level-0 Instruction Cache (I0)

■ The Prefetch Buffers

■ The internal Level-1 Instruction Cache (I1)

■ The external Level-2 Unified Cache (U2)

■ Memory

It then presents the Issue Unit (ISU) with four candidate instructions. The Fetch Unit is also responsible for predicting branches and fetching and issuing instructions from the predicted branch path.

Note:
> Only instructions that are *committed* are counted in the IPC; instructions that are *completed* but are later found to be from mispredicted branch paths do not affect the IPC.

9.2.1 Internal Level-0 Instruction Cache (I0)

The I0 Cache is direct mapped and contains 4,096 recoded instructions (occupying approximately 16K bytes). The I0 cache line size is 16 instructions (64-bytes before recoding). The logic that sends instructions from the I0 Cache to the Issue Unit (ISU)

attempts to fetch four instructions at the current predicted fetch PC. The I0 Cache allows the set of four instructions to be on any arbitrary alignment in the I0 Cache; that is, the four instructions are not required to be aligned modulo 4 instructions. There is one exception to this, however; if the set of four instructions would cross the boundary into a different I0 Cache line, only the instructions remaining in the current I0 Cache line can be supplied in one clock.

Table 36: I0 Accesses Need Not be Aligned on Four-instruction Boundaries

	inst 0	inst 1	inst 2	inst 3
I0 Line 1:	inst 4	inst 5	inst 6 – YES	inst 7 – YES
	inst 8 – YES	inst 9 – YES	inst 10	inst 11
	inst 12	inst 13	inst 14	inst 15

In the example shown in Table 36 instructions 6 through 9 are sent to the ISU in one clock.

Table 37: I0 Accesses Cannot Span Two I0 Cache Lines

	inst 0	inst 1	inst 2	inst 3
I0 Line 1:	inst 4	inst 5	inst 6	inst 7
	inst 8	inst 9	inst 10	inst 11
	inst 12	inst 13	inst 14 – YES	inst 15 – YES

	inst 0 – NO!	inst 1 – NO!	inst 2	inst 3
I0 Line 2:	inst 4	inst 5	inst 6	inst 7
	inst 8	inst 9	inst 10	inst 11
	inst 12	inst 13	inst 14	inst 15

In the example in Table 37 only two instructions can be sent to the ISU on the first cycle. Instructions 0 and 1 in Line 2 cannot be sent in cycle 1, because they are in a different cache line. In the second cycle instructions 0 through 3 in line 2 could be sent to the ISU.

This is called the "I0 Line Break" constraint.

9.2.2 I0 Cache Strategies

The following subsections contain strategies for handling constraints and features of the I0 instruction cache.

9.2.2.1 Avoid I0 Cache Thrashing

Since the I0 Cache is direct mapped, the compiler and linker should avoid placing major code blocks 16K apart (that is, at addresses that are equal modulo 16K) or thrashing may occur in the I0 Cache. For example, if the main loop of a program and a procedure called from within that loop are mapped modulo 16K bytes, the two blocks will constantly overwrite each other's lines in the I0 cache.

9.2.2.2 Align Short Loops to Avoid 64-byte Boundaries.

Many short loops fit within 16 instructions. If these loops are all in one cache line, they incur only one cache miss at the start of the loop. Also the loop will not suffer from the "I0 Line Break" constraint, as described in 9.2.1. **Note**: The line break penalty is paid for each iteration of the loop; thus, it is advisable to align short loops so that the entire loop is contained within one 64-byte I0 Cache line. The compiler may need to generate some NOPs to make the loop fall in one cache line, or it could put the loop into one cache line and then branch to that cache line.

The above can be extended to loops with greater than 16 instructions. For example, a loop with 32 instructions or less would only cause two cache misses and one line break if it is aligned within two cache lines. If unaligned, it might cause three cache misses and two line breaks. But as the loops get bigger, the percentage of the loop time that is saved by aligning the loops gets smaller. For large loops 64-byte alignment provides very little performance gain.

9.2.2.3 Align the Start of Code Sections on 64-byte Boundaries

Aligning the start of all program code sections on 64-byte boundaries may avoid extra cache misses. Align all procedures and library routines to start modulo 64-bytes (but not modulo 16K bytes).

It may appear that these alignment adjustments do not help very much. However, performance measurements and code analysis have shown that the optimizations described in this subsection provide significant gains in throughput. Keeping the path from the I0 Cache to the ISU as full as possible provides significant performance gains.

9.2.3 Internal Level 1 Instruction Cache (I1)

Instructions are fetched from the I1-Cache whenever the I0-Cache does not contain the desired cache lines. The I1-Cache size is 64K bytes; the line size is 64 bytes (16 instructions). The I1-Cache has a 3-cycle latency but it is pipelined, so it can send new instructions to the CPU during each clock cycle.

There is little that the compiler or assembly language programmer must do for the I1-Cache. Since the compiler and linkers should already be trying to align major code segments on 64-byte boundaries (as described in 9.2.1 above) this will also be an adequate alignment for the I1-Cache.

The I1-Cache is four-way set associative, so it is not necessary for the linker to be too concerned about placement of routines to avoid thrashing in the I1-Cache.

Note:

>   I0 strategies to avoid thrashing, which are described in 9.2.2, also prevent thrashing in the I1-Cache.

9.2.4 External Unified Cache (U2)

Instructions are fetched from the U2-Cache whenever the I1-Cache does not contain the desired cache lines. The U2-Cache size is between 1 Mbyte and 16 Mbytes; it is direct mapped, and the line size is 64 bytes (16 instructions). The U2-Cache has an 8 cycle latency, and is shared for both instructions and data.

There is little that the compiler or assembly language programmer must do for the U2-Cache. Since the compiler and linker should already by trying to align major code segments on 64 byte boundaries (as described above), this will also be an adequate alignment for the U2-Cache.

9.3 Branches and Branch Prediction

Branches are always expensive for any superscalar processor; they cause several problems:

■ The CPU branch prediction logic may incorrectly predict the branch and start speculatively issuing instructions from the wrong path. Once the correct path is determined the speculative instructions must be cancelled and the machine must reset itself to the state that existed before the branch. This causes from one to several clocks of delay. Also, the instructions that were discarded cannot be counted in the committed IPC.

■ Speculative instructions that are later cancelled may have side affects. For example, a speculative load that causes a cache miss creates needless work for the memory system. Thus, future **non**speculative cache misses may need to wait for the memory to become free. Speculative misses can also pollute the caches by replacing good data/instructions with data/instructions that will not be used. However, in many cases the speculative data/instructions act as prefetches for future activity.

SPARC64-III implements two different branch prediction schemes, 2-bit conventional and 2-level adaptive, one of which is selected through ASR18<9:8>.

Note:

> Changing ASR18<9:8> causes the current Branch History Table (BHT) to be discarded, because both branch prediction mechanisms use the same physical BHT RAM. Therefore, it is not advisable to change these bits frequently.

9.3.1 Two-Bit Conventional Branch Prediction

This scheme is selected when ASR18<9:8> are set to 01_2 or 11_2. The branch prediction is done through a 2-bit saturating up-down counter kept in an $8K \times 2$ entry BHT RAM, indexed by Fetch Program Counter (FPC) bits <14:2>.

If ASR18<9:8> is 11_2 and the branch is with prediction, the instruction prediction bit *always* has higher priority in the prediction than the hardware counter. Figure 77 illustrates 2-bit Conventional Branch Prediction

Branch History Table (BHT)
(2bits × 8K-entries = 2KByte)

Fetch PC<14:2>

Index

BPR<1:0>
2-bit saturating up-down counter

00 Strongly not taken
01 Weakly not taken
10 Weakly taken
11 Strongly taken
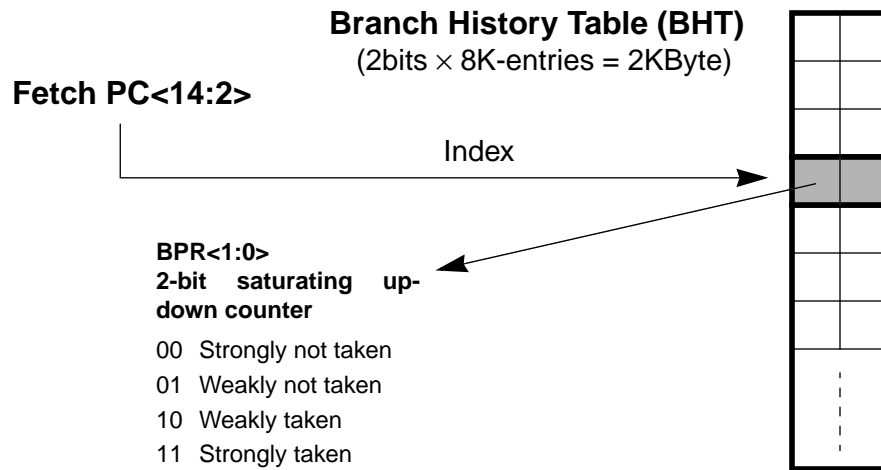
Figure 77: 2-Bit Conventional Branch Prediction

9.3.2 2-Level Adaptive Branch Prediction

This scheme is selected when ASR18<9:8> = 00_2.

The adaptive branch prediction algorithm shows better prediction accuracy than conventional 2-bit prediction. In 1991 Yale N. Patt *et al* at Michigan University introduced the new 2-level adaptive branch prediction scheme. This scheme boasts the highest prediction accuracy of all schemes that have been proposed so far.

There are some variations in the 2-level adaptive scheme; SPARC64-III uses one called "global-branch-history-register and global-pattern-history-table with branch address hashing." This is easier to implement and costs less than other variations, and yet still provides good prediction accuracy. SPARC64-III contains a 2K Byte Branch History Table (BHT).

The Branch-History-Register (BHR) accumulates the recent taken/not-taken information of predicted branches. The BHT is made of RAM which is indexed by the concatenated result of BHR and FPC<9:4>. Each entry of BHT consists of a conventional 2-bit saturating up-down counter. Figure 78 illustrates the SPARC64-III 2-Level Adaptive Branch Prediction scheme.
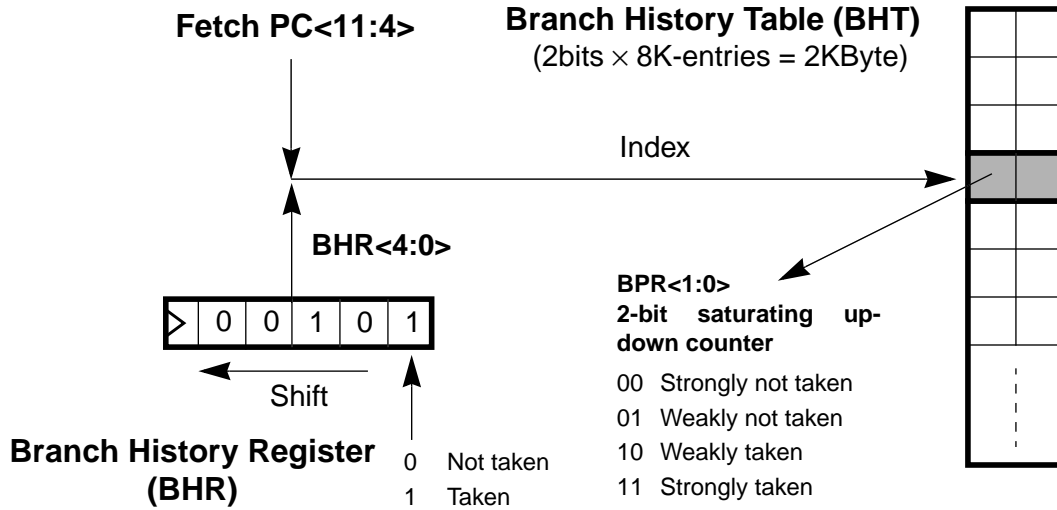
Figure 78: 2-Level Adaptive Branch Prediction

9.3.3 Computed Branches

Computed branches using JMPLs cause issue to stall until the JMPL target is calculated. Special hardware is provided to handle the case when the JMPL functions as a subroutine return.

When a CALL or JMPL with a destination register of %o7 is issued (JMPL call) the return address is calculated and pushed onto an internal hardware 4-level stack called the Return Prediction Stack (RPS). When a JMPL with the address specified as [%o7+8] or [%i7+8](which usually are subroutine returns) is encountered, the return address is predicted to be the value stored in the RPS.

Thus, if a subroutine returns through [%o7+8] or [%i7+8], the next PC will be predicted much like a conditional branch. Also like a conditional branch, if the actual target address is eventually determined to be different from the predicted address, the CPU must discard the incorrect instructions and begin fetching the correct ones.

9.3.4 Branch and Branch Prediction Strategies

The following subsections describe strategies for handling and predicting branches.

9.3.4.1 Eliminate Branches and Make Larger Basic Blocks

Without question, the **most important** thing a SPARC64-III compiler or assembly language programmer can do is to eliminate as many branches as possible.

There are several techniques that can be used to remove branches and create larger basic blocks:

1. Replace branches with conditional moves. Sometimes simple basic blocks can be removed by doing one or several conditional moves. The CPU can perform several

conditional moves for the price of one branch. MOVcc, FMOVcc, MOVr, and FMOVr all take the same amount of time to execute.

2. *Loop Unrolling*: Even a modest amount of loop unrolling can be a big benefit. Unrolling by two might double the size of a basic block and greatly improve the efficiency of SPARC64-III.

3. *Procedure Inlining*: Most compilers implement some inlining. A lot of non-superscalar compliers inline mostly to prevent pipeline stalls, to cover cache latencies, to remove prologue and epilog code, and to allow better register allocation. On SPARC64-III all of these reasons apply. In addition, SPARC64-III benefits by the removal of the subroutine calls and returns and by reducing the change of a window spill or fill trap. **Note**: Call and return are considered as branches in the CPU.

4. Moving instructions from one basic block to another to make a bigger basic block. Recent literature contains examples of this kind of code movement. Some of these techniques require compensation code in the less likely paths in order to expand the size of the more likely paths.

5. All of this not withstanding, it is still a good idea for the compiler to use standard strength reduction techniques to remove redundant instructions. That is, do not use redundant or useless instructions simply to create larger basic blocks.

Many processors benefit from these techniques, but a superscalar processor like SPARC64-III benefits even more.

9.3.4.2 Arrange Code for the Fall-through Case

Since SPARC64-III does not issue past the delay slot of a predicted taken branch, there is an advantage if the code can be reorganized so that the most likely path through the code is in the branch not taken path (the fall through path). The compiler or assembly language programmer can do this if it is possible to statically predict the direction of a branch with reasonable accuracy. Using this static prediction the compiler or programmer can ensure that in most cases the predicted code is in the fall through path.

The compiler or assembly language programmer should avoid using anulled branches if these branches are usually not taken because SPARC64-III will create hardware "glitches" to annul instructions and thereby cause some performance loss.

9.3.4.3 Calculate Condition Codes Early

Attempt to place the comparisons that set condition codes as far as possible before the branches that use the condition codes. The hardware still predicts the branch using the branch prediction hardware instead of the condition code bits, but it looks at the condition code bits in the next cycle or as soon as they are available. If the condition code bits are not valid at the time of the branch, the CPU cannot tell if the branch was mispredicted until they are available.

Note:

Setting the condition code or register value early benefits the CPU, but it is not as great a benefit for SPARC64-III as it is for scalar or pipelined processors. This is because SPARC64-III performs

branch prediction and therefore does not stall waiting for the branch condition to be set. However, if the branch prediction is incorrect, the speculative instructions must be cancelled and the correct path fetched. The sooner the CPU discovers it mispredicted the branch, the fewer cycles it will waste executing the wrong instructions.

9.3.4.4 Subroutine Returns

Use a CALL instruction or a JMPL with a destination of `%o7` for all procedure calls. For subroutine returns, use a JMPL with `[%i7+8]` if the routine did a SAVE or a JMPL with `[%o7+8]` if it was a leaf routine. Avoid using JMPLs with `%o7` or `%i7` except for subroutine calls or returns, since they corrupt the RPS. For example, avoid using a JMPL `%o7` for a switch statement, because it corrupts the RPS and might cause a future return to be mispredicted.

9.3.4.5 Align Short Loops to Make "Delay Slot" the Last Instruction

It may be difficult or impossible to align short loops to start on an I0 cache line boundary. In these cases an alternate and equally efficient method is to align the loop so that the last instruction in a cache line is the delay slot of a branch. This improves throughput, because it puts the delay slot in a known I0 cache line break; that is, there was going to be a code break caused by the end of the cache line anyway.

9.4   Instruction Issue

The strategies outlined above should produce a regular stream of instructions to the Issue Unit (ISU). At this point the compiler should adjust the order and mix of the instructions to maximize performance. The CPU can issue at most four instructions per clock. However, various issue constraints may make this impossible in some cases. The following sections describe these issue constraints and present some strategies to reduce their effects.

9.4.1 Issue Strategies

If the compiler or assembly language programmer could always determine exactly which instructions were going to be issued in one clock, it would be easier to determine the optimum instruction mix to generate for the next clock. But this isn't always possible, because of SPARC64-III's data flow nature. For example, the compiler might generate two integer instructions and two loads and expect that they would issue in one clock, since there are enough ports to the queues for these instructions. However, if the Load/Store Unit (LSU) queue has 11 of its queue slots occupied when the CPU attempts to issue these instructions, only one of the loads could actually be issued. If the compiler assumes that all four were issued, it is "out of sync" with the hardware.

Sometimes, however, the compiler or assembly language programmer has a fairly good idea which instructions will be issued next. For example, after a branch to an I0 aligned (64-byte aligned) location, the compiler can be fairly certain that the CPU will attempt to issue the next four instructions if there are no static or dynamic issue constraints.

It is not possible to schedule instructions perfectly. However, there are some ground rules for scheduling a basic block. First, break the instructions into the following basic classes:

INT only:

> Shifts, normal integer instructions, integer multiplies and divides, and instructions with a condition code (*icc* or *xcc*) source.

INT / AGEN:

> Normal integer instructions, MOVr, atomics, prefetches, and loads and stores.

FP:

> All floating-point instructions.

LSU:

> Loads, stores, prefetches, and atomic instructions.

See Table 39 on page 200 for a complete list of the instructions that can be executed in each execution unit.

Now, assume that the compiler must schedule an instruction of one class. For example, assume that the first instruction in the dependency graph is a shift. We know that the shift will be sent to the INT queue, since that is the only place where it can be executed. For the next instruction the compiler should check the dependency graph to see if it can find an instruction that can be issued into one of the other classes; for example, an FMUL. The compiler repeats this process, always looking for an instruction that is **not** in the same class. This algorithm attempts to ensure that no two adjacent instructions are in the same class; it guarantees that no more than two instructions are issued to any class in one clock, regardless of where the CPU starts issuing the instructions.

It may not always be possible to find an instruction for a different class that can be inserted into the code stream. At this point the compiler must do some experimentation to find an algorithm that gives the best performance. Larger basic blocks always help, since they give the algorithm more instructions to choose from for scheduling.

9.5 Instruction Dispatch, and the DFM Queue

Figure 79 on page 196 shows a block diagram of the SPARC64-III Data Flow Unit.

Up to 4 instructions are issued and dispatched in each cycle from the Issue Unit and arrive at the Data Flow Unit. These instructions are sent to the Queues where they await all their source operands (if they are not already available from previous instructions). Once all of the source operands are available the instructions are eligible to be sent to one of the execution units. (If all the source operands are available when an instruction is issued and dispatched, and if the appropriate queue is empty, then the instruction can go directly from the Issue Unit to the appropriate execution unit.)

When the results are generated by the execution units, they are sent on the Result Busses to the Register File and also sent to the Queues where they are captured by any waiting instruction that needs the result as an input operand. When more instructions are eligible

for execution than there are execution units served by the instruction's queue, then the oldest eligible instructions are selected for execution.
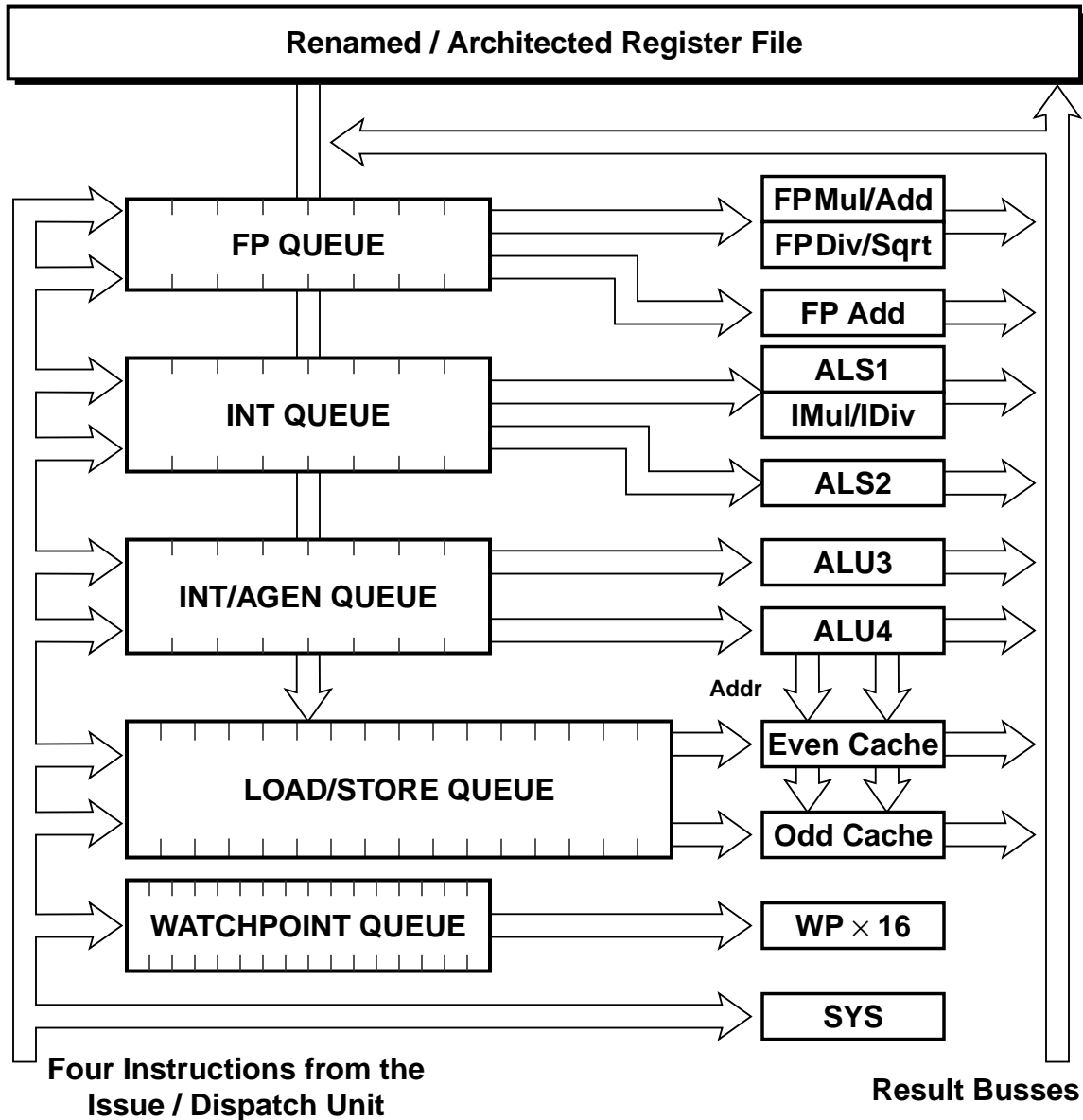
Figure 79: SPARC64-III Data Flow Unit

This collection of queues and execution units is called a "data flow" unit, because the order of execution of the instructions is determined by the availability of the data needed to start execution (the flow of the data) and not by the original order of the instructions in the program.

The SPARC64-III CPU contains five instruction queues. Each queue may feed one or more execution units. Table 38 on page 197 describes the queues, their ports, the number of queue entries, and the number and type of functional units they feed with instructions, and the latency of the functional units.

Table 38: DFM Queue Structures

Queue Name	WPorts/ Entries/ RPorts	Execution Units Supplied from the Queue	Execution Latency
Floating Point (FP)	2 / 8 / 2	1 Floating Point Multiply Add Unit (FMA) 1 Floating Point Divide/Sqrt Unit 1 Floating Point Add Unit (FA)	4 (FMA)[a] 1 (FMOV) 12(FDIVs) 22(FDIVd) 12(FSQRTs) 22(FSQRTd) 3 (FADD)[a]
Integer (INT)	2 / 8 / 2	2 ALS (Arithmetic/Logical/Shift) Units 1 Integer Multiply/Divide Unit	1 4 (32bits multiply) 6 (64bits multiply) 2-37(13 avg divide)
Integer and Address Generation (INT/AGEN)	2 / 8 / 2	2 ALU (Arithmetic/Logical Units) Note: ALUs used for Address Generation	1
Load/Store (LSU)	2[b] / 12 / 2	Even and Odd Data Caches	3 (hit)[a]
Watchpoint (WP)	1 / 16 / 16	Watchpoint x 16 (branch condition calculation)	1

a. Pipelined

b. 1 for STDF, STDFA, STF, STFA, and STFSR

The "WPorts" column defines the maximum number of instructions that can be sent from the ISU to that queue on each cycle. This is one of the issue constraints. The number of queue ports is a static constraint; that is, the CPU can never issue more instructions to any queue in any one clock than there are queue ports. The "RPorts" column defines the maximum number of instructions that can be sent from the queue to the execution units on each cycle.

The "Entries" column defines the maximum number of instructions that can be resident in each queue. If the queue is full, the ISU cannot issue more instructions to this queue until at least one instruction finishes its execution. Instructions leave their queues and are sent to the execution units when all of their source operands are available. The number of free queue entries is a dynamic constraint; it is difficult for the compiler to know before an instruction is issued whether or not the required queue is full.

The "Execution Latency" column defines the amount of time it takes to execute the operation once it has been dispatched, assuming that it need not wait in a queue for its operands. The FP divide/sqrt and the integer multiply/divide instructions can run in parallel with other FP or integer operations. However, they share a result bus and must steal one clock from the Floating Point Multiply/Add unit or one of the integer ALS's when they generate a result.

The Floating point Multiply-Add Unit and Floating point Divide/Sqrt Unit share the source/result data bus. Thus, they can neither start nor finish their execution at the same time. But the Floating point Multiply-Add Unit can start or finish a new multiply-add while the Floating point Divide/Sqrt Unit is in the middle of execution.

The Floating point Multiply-Add Unit, Floating point Add unit, and Data Cache Unit are pipelined. They can start a new multiply/add, load/store each clock and complete one each clock. Floating point moves bypass the pipeline and can complete in 1 cycle, if there is no Floating point Multiply-Add instruction in the pipeline which will produce a result in the next cycle.

The Floating point Divide/Sqrt Unit, the Integer Multiply/Add Unit are **not** pipelined. They **cannot** start new operations while they are busy.

Anything issued to the Load Store Unit (LSU) queue must also be issued to the INT/ AGEN queue to have its address generated. The INT/AGEN units can do address genera- tions or most integer instructions. However, they cannot do shifts, multiplies, divides, or instructions that have a condition code register as a source operand; for example, MOVcc, ADDC, or SUBC. The INT units cannot do MOVr instructions. For a complete listing of what integer instruction can be issued to INT or INT/AGEN see 9.6.3, "Where Instruc- tions Are Executed".

9.6 Data Flow Unit

9.6.1 Data Dependencies

There are three types of data dependency: true dependency, output dependency, and anti- dependency. The following code fragment contains an example of each type; they are described further in the subsections that follow:

1. `add  r1, r2 ➜r3`   True Dependency

2. `sub  r3, r4 ➜r5`   Output Dependency

3. `or   r7, r8 ➜r3`   Anti-Dependency

All three of these instructions can be issued into the queues in one clock.

9.6.1.1 True Dependencies

Instruction 1 writes a result to r3 and instruction 2 uses that result as a source operand. This is called a "true dependency." It is impossible to execute instruction 2 before instruc- tion 1 and obtain the correct answer. No computer can remove true dependencies.

9.6.1.2 Output Dependencies

Instruction 1 writes a result to r3. Instruction 3 also writes a result to r3. It appears that instruction a can not execute before or in parallel with instruction 1, or else instruction 2 might get the wrong source value in r3. However, register renaming as found on SPARC64-III can remove this restriction. Register renaming works by "renaming" all des- tination registers from their architectural register name to an internal "physical" register name. The renaming is done at dispatch time. To make renaming work efficiently the CPU needs more physical registers than architected registers. For instance, assume that the

example above were renamed in the following manner. (Here we use the notation $\mathbf{p}_n$ to mean physical register "*n*".)

1. add p_1, p_2 ➤ $\mathbf{p}_{78}$

2. sub $\mathbf{p}_{78}$, p_4 ➤ p_{42}

3. or p_7, p_8 ➤ $\mathbf{p}_{34}$

First the destination of 1 is renamed to p_{78}. This also causes the source register in 2 to be renamed to p_{78}. After renaming, the program is logically equivalent to the original. However, now there is no reason that instruction 3 could not be executed before or simultaneously with 1, because they now write different destination registers. Thus, register renaming removed the output dependency between instructions 1 and 3.

9.6.1.3 Anti-dependencies

In the original example instruction 2 uses r3 as a source and instruction 3 writes its result into r3. It appears that instruction 3 can not execute before instruction 2, because instruction 3 would overwrite the source register (r3) needed in instruction 2. This is called an anti-dependency.

The renamed register version in 9.6.1.2 above illustrates how SPARC64-III handles this problem. Since the destination of 3 was renamed to p_{34} and instruction 2 needs p_{78}, instruction 3 can now be executed before or with instruction 2b. Thus register renaming also removes anti-dependencies.

9.6.2 True Dependency Strategies

Since register renaming removes *output* and *anti* dependencies, the compiler or assembly language programmer need only be concerned about *true* dependencies. These cannot be removed by register renaming. In fact, there is no way to remove true dependencies and have a correctly operating program.

For short sequences of code or for code that is not in a loop, it is probably not worthwhile to worry about true dependencies. SPARC64-III can have up to 64 instructions in its instruction queues at any time. This allows enough buffering so that most true dependencies are resolved before they cause any machine stalls. This is especially true for integer operations where the instruction latencies are usually only 1 clock.

Note:
         Load and Store will be discussed in more detail later.

In certain loops, especially in floating-point loops, the scheduling of true dependencies may make a big difference in performance. If one iteration of a loop generates multiple floating-point results that are used in the next iteration of the loop, the FP queue can become filled. That is, the ISU can issue two instructions per clock to the FP Queue and each operation has at least three clocks of latency. In these cases it is possible for true dependencies to stall the machine because the FP queue is full. If possible, the compiler should schedule these loops to prevent the FP queue from filling.

9.6.3 Where Instructions Are Executed

Table 39 lists the queues to which instructions can be issued. A check mark (✔) in a column indicates that the instruction(s) can be executed in the associated unit; an 'X' indicates that it uses multiple execution units. 'L' indicates that it uses Even/Odd Cache in addition to ALU1/2 attached to the INT/AGEN queue.

Table 39: Where Instructions are Executed

Queue	INT			INT/AGEN		FP			WP	SYS
Result bus	FX1		FX2	FX3	FX4	FP1		FP2		
Execution Unit	ALS1	Imul/Div	ALS2	ALU3	ALU4	FMA	Fdiv/Sqrt	FA	WP	SYS
ADD, ADDcc	✓		✓	✓	✓					
ADDC, ADDCcc	✓		✓							
AND, ANDcc, ANDN, ANDNcc	✓		✓	✓	✓					
BPcc, Bicc, BPr									✓	
CALL			X						X	
CASA, CASXA				L	L					
DONE	✓		✓						X	
FABS						✓				
FADD						✓		✓		
FBfcc, FBPfcc									✓	
FCMP, FCMPE, FiTO(s, d)						✓				
FDIV							✓			
FLUSH				L	L					
FLUSHW										✓
FMOV, FMOVcc, FMOVr, FMOVr, FMUL, FNEG, FsMULd, F(s, d)TOi, F(s, d)TO(s, d), F(s, d)TOx, FxTO(s, d)						✓				
FSQRT							✓			
FSUB						✓		✓		
ILLTRAP										✓
IMPDEP2(FMA)						✓				
JMPL			X						X	
LDD, LDDA				L	L				X	
LDDF, LDDF, LDDFA, LDF, LDFA, LDFSR				L	L					
LDSB, LDSBA, LDSH, LDSHA, LDSTUB, LDSTUBA, LDSW, LDSWA, LDUB, LDUBA, LDUH, LDUHA, LDUW, LDUWA, LDX, LDXA, LDXFSR				L	L					
MEMBAR				L	L					
MOVcc	✓		✓							
MOVr				✓	✓					
MULScc		✓								
MULX		✓								
NOP	✓		✓	✓	✓					
UMUL, SMUL, UMULcc, SMULcc		✓								

Table 39: Where Instructions are Executed (*Continued*)

Queue	INT			INT/AGEN		FP			WP	SYS
Result bus	FX1		FX2	FX3	FX4	FP1		FP2		
Execution Unit	ALS1	Imul/Div	ALS2	ALU3	ALU4	FMA	Fdiv/Sqrt	FA	WP	SYS
OR, ORcc, ORN, ORNcc	✓		✓	✓	✓					
PREFETCH, PREFETCHA				L	L					
RDASI, RDASR, RDCCR, RDFPRS, RDPC, RDPR, RDTICK, RDY			✕						✕	
RESTORE	✓		✓							
RESTORED									✕	✓
RETRY	✓		✓						✕	
RETURN			✕						✕	
SAVE	✓		✓						✕	
SAVED									✕	✓
SDIV, SDIVcc, SDIVX		✓								
SETHI	✓		✓	✓	✓					
SIR										✓
SLL, SLLX	✓		✓							
SDIV, SDIVXcc, SDIVX		✓								
SLL, SRL	✓		✓							
SMUL, SMULcc		✓								
SRA, SRAX, SRL, SRLX	✓		✓							
STB, STBA, STBAR				L	L					
STD, STDA				L	L				✕	
STDF, STDFA, STF, STFA, STFSR				L	L					
STH, STHA, STW, STWA, STX, STXA, STXFSR				L	L					
SUB, SUBcc	✓		✓	✓	✓					
SUBC, SUBCcc	✓		✓							
SWAP, SWAPA				L	L					
TADDcc, TADDccTV	✓		✓							
Tcc			✓						✕	
TSUBcc, TSUBccTV	✓		✓							
UDIV, UDIVcc, UDIVX		✓								
UMUL, UMULcc		✓								
WRASI, WRASR, WRCCR, WRFPRS, WRPR, WRY			✓						✕	
XOR, XORcc, XNOR, XNORcc	✓		✓	✓	✓					

9.6.4 Loads and Stores

SPARC64-III contains a 64K byte Data Cache, which is divided into two banks, even and odd, with 8 bytes boundary. The Data Cache is indexed by virtual address bit <13:6>. The data caches have a line size of 64 bytes and are four-way set associative. The associativity means that cache thrashing is minimized. Nevertheless, the compiler should not align large data arrays modulo 16K bytes apart.

Note:

> This is 16K byes instead of 64K, because of the four-way set associativity.

The caches are "non-blocking". That is, while the cache is waiting for data from memory because of a cache miss, it can still process further cache accesses, most of which will be hits. The cache only blocks further accesses when it encounters a miss that requires a fourth data cache line to be loaded from memory. (Misses to a data cache line that is in the process of being loaded from memory do not block other data accesses to the cache.) The non-blocking nature of the caches allows the CPU to perform a great deal of work while waiting for cache misses.

SPARC64-III has a large (256 entry, fully associative) Translation Lookaside Buffer (TLB). Nevertheless, it is important to avoid TLB misses as much as possible, since they are handled in software and take a great deal of time, typically 80-100 cycles or more.

The SPARC64-III Load/Store Unit (LSU) can start two loads or two stores in each clock, as long as one has an even address and the other has an odd address. These accesses can be totally independent; they can even be a mixture of loads and stores.

The data cache requires a load/store latency of three cycles, but it is pipelined and can accept two new load or store requests every clock and can complete two loads or stores every clock.

The SPARC64-III level 2 cache system has about a 13 clock latency for loads that miss in the data cache. Fortunately, the data flow nature of SPARC64-III allows the CPU to do other useful work while waiting for a cache miss. However, it is hard for the CPU alone to find enough work to do fill 13 clocks of latency. Subsequent load misses to the same cache line are queued in the data cache and will complete as soon as the data arrives (usually only a few cycle after the 13 cycles to get the data for the first miss). Too many data cache misses to different cache lines in a short period of time eventually cause the CPU to stall.

The next section describes strategies the compiler can use to avoid the cache miss latency penalty.

9.6.4.1 Load and Store Latencies

Table 40 shows the Data Cache latencies (in clocks) for various load and store events:

Table 40: Data Cache Latencies

Type	Latency[a] (load)	Latency[a] (store)	Comments
Hit at D1$	3	3	Pipelined
Miss at D1$, but Hit in U2$	3+10	3+10	Non-blocking
Miss at D1$ and U2$	3+10+46	3+10+53	Non-blocking
Miss at TLB	100+	100+	Handled by Software

[a]. All latencies are approximate; many factors determine the actual number of cycles of latency that will occur.

9.6.4.2 Load and Store Ordering Constraints

SPARC64-III does **not** allow stores to execute out of order, even when the CPU is using the Relaxed Memory Order (RMO) memory model. SPARC64-III **does** allow loads to execute out of order in RMO mode, however. It also allows loads to pass stores, if the load is not to the same page offset as any store that is also in the Load/Store Queue. Explicitly, in order for a load to pass a store in the Load/Store Queue, address bits 13..0 of the load and store must differ.

Speculative stores, which are stores that are encountered on a predicted, but not yet verified, branch path, are not allowed to execute until they are no longer speculative. Stores in a nonspeculative path are also not executed until it is known that all instructions before the store will complete without error. This prevents a store from erroneously modifying a cache or memory location.

Speculative loads, which are loads on a predicted branch path, are allowed to execute. If the CPU later determines that the branch was mispredicted, all speculative loads and speculative stores that are still in the LSU queue are cancelled. Loads that have already started execution (that is, they are in the process of loading data from the U2-Cache or memory) are not cancelled; instead, the data is loaded into the level-1 data cache, but the data is not sent to the DFMLSU.

9.6.5 Load and Store Strategies

The following sections discuss strategies for hiding data cache latencies.

9.6.5.1 Schedule Loads as Early as Possible

It is a good idea to schedule loads as early as possible; that is, try to schedule loads as far before any instructions that uses the result of the load as possible. This **may** cover the entire latency of a data cache hit, and it will definitely help if there is a data cache miss. Loop unrolling or software pipelining may allow a load to be started long before the data is needed. Larger basic blocks also make it easier to schedule loads earlier.

Several factors may make it difficult or undesirable to schedule loads too far before the data is used:

■   The load instructions tie up an architected register until the data is used. Too many "early" loads would tie up all of the architected registers and prevent later instructions from issuing.

■   Often loads cannot be moved from a later basic block to an earlier one, because the intervening branch might be checking to determine if it is valid to do the load. For example, while traversing a linked list, it would speed things up if the compiler could start to access the data in the next list node before checking to see if the node pointer is null. However, with normal loads using a null pointer causes a *data_access_exception*, so this load cannot be moved before the branch.

The next sections discuss two techniques available in SPARC64-III that can circumvent some of these restrictions.

9.6.5.2 Data Prefetches

SPARC-V9 provides a set of data prefetch instructions. These instructions are defined to "attempt" to prefetch data. The prefetch instructions specify an address like a normal load or store, but do not require a destination (source) register. The prefetch operation attempts to move the data as close as possible to the CPU. For SPARC64-III the prefetch loads the data into the level-1 data cache, if it is not already there.

If a *data_access_exception* occurs during the prefetch, the instruction is treated as a NOP; that is, the data is **not** loaded and **no exception** occurs. This allows the compiler to schedule prefetches before it is certain that the source address is valid.

On SPARC64-III the compiler needs to schedule only one prefetch per cache line (64 bytes), since the prefetch always loads an entire cache line into the cache.

SPARC-V9 defines five different prefetch types, but SPARC64-III supports only two types: "Prefetch for Several Reads" and "Prefetch for Several Writes." (See A.42, "Prefetch Data", for more information.) Both types cause a cache line to be loaded into the data cache if it is not already there. The "write" version also informs the coherence mechanism that the requestor must have exclusive ownership of the cache line. All prefetches are non-blocking; that is, the data cache can process other cache accesses while the prefetch is loading the cache line from the Unified Cache (U2) or memory.

A Prefetch that misses in the Data Cache causes a cache reload buffer to be busy when the data returns from the U2 cache or memory. If too many prefetches miss, then subsequent real loads (or stores) will be delayed because all available prefetch buffers will be in use. From 2 to 4 prefetches (depending on their address) can miss before a subsequent load miss will be stalled.

9.6.5.3 Non-Faulting Loads

Non-Faulting Loads are similar to prefetches except that:

■ They are only available for loads,

■ They load data into an architected register as well as the data cache.

If any data access protection violation occurs during the load, zero is returned to the destination register but no error trap is taken.

Note:
        Error traps are taken for hardware errors such as ECC.

The compiler must verify that the address used for a non-faulting load was valid before it attempts to use the loaded data. For example, in traversing a linked list the compiler could use non-faulting loads to access the data in the next node before checking for a null node pointer. However, the compiled code **must** check the pointer before using the data that was loaded, since no exception occurs if the pointer was invalid.

9.6.5.4 Non-Faulting Loads vs. Data Prefetches

Prefetch has the following benefits:

■   It does not require a register for the result.

■   It can be made before the prefetch address is validated.

■   Only one Prefetch is needed per data cache line.

■   It can be used for loads and stores.

Prefetch has the following disadvantages:

■   The data is loaded only into the data cache. The prefetch must be followed by a "real" load in order to get the data from the cache and to ensure that the address of the prefetch was valid.

■   Too many prefetch misses may make all of the data cache reload buffers busy. This may then block nonprefetched loads and stores that are more important for making forward progress in the program.

Non-Faulting Loads have the following advantage:

■   The data is loaded into a register, therefore, a follow-up load is not needed.

Non-Faulting Loads have the following disadvantages:

■   The address must be validated before the data can be used.

■   An architected register is tied up until the data is used.

■   They are available only for loads.

■   Too many non-faulting loads that miss can make the data cache reload buffers busy.

9.7 Some Implementation Specifics

In addition to the topics discussed above a few topics relate to the particular implementation of the SPARC64-III machine that the compiler writer or assembly language programmer should be aware of to generate efficient code. These are all covered in other sections of this manual. In addition, they are collected and summarized below.

9.7.1 Unimplemented Instructions

SPARC64-III does not implement some SPARC-V9 instructions in hardware. If the compiler issues these instructions, the machine traps and the kernel emulates the instruction. Avoid these instructions when possible, because kernel emulation is slow. Table 41 enumerates the non-privileged instructions that are unimplemented in SPARC64-III.

Table 41: Unimplemented instructions

Unimplemented Instructions	Notes
All Quad FPops LDQF/STQF	Kernel emulation
POPC	Kernel emulation

9.7.2   Overloaded Instructions

The following instructions are overloaded in SPARC64-III. That is, the instructions are mapped onto other valid instructions.

Table 42: SPARC64-III Unimplemented Non-privileged Instructions

Overloaded Instructions	Overload to:
PREFETCH (one read)	Mapped to PREFETCH (several reads)
PREFETCH (one write)	Mapped to PREFETCH (several writes)
PREFETCH (page)	Mapped to PREFETCH (several reads)

9.7.3   Register Windows

SPARC64-III implements five register windows (NWINDOWS = 5). This is fewer windows than most SPARC machines. However, the cost of additional registers in a register renamed, superscalar processor such as SPARC64-III is very high. (More register windows take more space on the chip and are slower.) However, the window spill and fill routines are much faster in SPARC-V9 than in SPARC-V8, because of the new SPARC-V9 instructions. Also SPARC64-III speculatively executes into the spill/fill routines, which allows the spills and fills to start early. Thus, the cost of a window spill or fill is much less for SPARC-V9 implementations like SPARC64-III than it is for SPARC-V8 machines.

To keep the spill/fill cost low the compiler should not generate unnecessary SAVE and RESTORE instructions, since these might cause extra window spills or fills. Inlining small functions helps, since this removes the SAVE and RETURN instructions. Similarly, The compiler should generate routines as leaf routines whenever possible.

9.7.4   Deprecated Instructions

lists the SPARC-V8 instructions that have been deprecated in SPARC64-III. The deprecated instructions may not be supported or may perform poorly in future SPARC-V9 hardware and may be dropped in SPARC-V10; new compilers should not generate deprecated instructions.

Table 43: Deprecated instructions

Deprecated Instructions	Comments
Bicc	No performance penalty. No branch prediction bit.
FBFcc	No performance penalty, No branch prediction bit.
LDD(A)	Single issue instruction
LDFSR	Syncs the CPU
MULScc	Syncs the CPU
SDIV, SDIVcc	No performance penalty
SMUL, SMULcc	Syncs the CPU
STBAR	No performance penalty
STD(A)	Single issue instruction
STFSR	Syncs the CPU
SWAP, SWAPA	No performance penalty
TADDccTV	No performance penalty. Useless in non-tagged programs
TSUBccTV	No performance penalty. Useless in non-tagged programs
UDIV,UDIVcc	No performance penalty
UMUL, UMULcc	Sync the CPU
WRY	Sync the CPU

9.8 Grouping Rules

A maximum of 4 instructions can be issued in a cycle, depending on restrictions described as follows.

Any data dependency (true, output, anti-dependency) between instructions does not affect the number of instruction being issued, because of SPARC64-III's renaming capability.

9.8.1 Fetch limitation

Instruction issue is limited to the number of instructions available in the 12-entry instruction buffer. The number of instructions that can be fetched (max 4) is dependent on the conditions described in the following subsections.

9.8.1.1 Instruction Lookaside Table (ILT) Miss

The next fetch address is always predicted through the 4k entry Instruction Lookaside Table (ILT). The mispredict penalty is 1 cycle.

9.8.1.2 I0 cache Miss

The I0 Cache is 16KB-direct; the I1 Cache is 64Kb, 4-way set associative. The I0 Cache miss penalty is 3 cycles, and between 10 and 56 (U2 Cache hit/miss) more cycles are required if the I1 Cache misses as well.

9.8.1.3 I0 Cache Line Break

Each I0 Cache line includes 16 instructions; instructions that cross a cache line boundary cannot be fetched in the same cycle. See 9.2.1, "Internal Level-0 Instruction Cache (I0)" for the details.

9.8.1.4 Control Transfer Instruction (CTI) Fetch

Instruction addresses have to be contiguous to be issued in the same cycle. A taken CTI and the target of that CTI cannot both be fetched in the same cycle. A non-taken DCTI with the annul bit on and the next instruction cannot be fetched in the same cycle. In addition, multiple CTIs cannot be fetched in the same cycle.

Every CTI except DCTI requires 2 cycle bubbles for the subsequent instructions to be issued.

9.8.2 Syncing Instructions

Several instructions cause SPARC64-III to sync; that is, they cause the machine to stop issuing instructions until all previously issued instructions commit. The CPU then executes the syncing instruction by itself and waits for it to commit before proceeding. Fortunately, only a few of these instructions might be generated by a compiler. Table 44 lists the instructions that cause the SPARC64-III to sync.

Table 44: SPARC64-III Syncing Instructions

Syncing Instructions	Suggestions
MEMBAR (#sync, #memissue)	
FLUSH	
SMUL/SMULcc	Use MULX
UMUL/UMULcc	Use MULX
MULScc	Use MULX
Tcc except 'ta %g0+imm'	Use 'ta %g0+imm' form for Unix system calls
RDASR %asr24, %asr26,%asr28,%asr29, %asr30	
WRASR %asr18, %asr19, %asr20, %asr21, %asr22, %asr23, %asr25, %asr26, %asr30, %asr31	
WRPR except %pil '%g0+imm %cwp/%cansave/%canrestore/%cleanwin/%otherwin/ %wstate'	
LDFSR/LDXFSR	
STFSR/STXFSR	
CASA/CASXA	
SIR	
WRY	
WRASI	
WRFPRS	

9.8.3 slot0_only and last_to_be_issued

Some instructions are marked as *slot0_only* or *last_to_be_issued*; that is, the instruction must be the first or last slot of the current issuing instruction window. An instruction marked as **both** *slot0_only* **and** *last_to_be_issued* is a single-issue instruction.

Table 45 lists the SPARC64-III *slot0_only* and *last_to_be_issued* instructions.

Table 45: *slot0_only* **and** *last_to_be_issued* **Instructions.**

Instructions	slot0_only	last_to_be_issued
FMOVrval	✓	
JMPL(except ret/retl)		✓
LDD, LDDA	✓	✓
RDASI, RDASR, RDFPRS, RDPC, RDPR, RDTICK	✓	
RESTORE	✓	✓
RESTORED	✓	
SAVE	✓	
SAVED	✓	
STD, STDA	✓	✓
Tcc	✓	✓
WRASI, WRASR, WRCCR, WRFPRS, WRPR, WRY	✓	
RETURN		✓

9.8.4 DFM Q Write Port

Each instruction has attributes indicating to which DFM queue the instruction can be issued and dispatched. Since each DFM queue can accept a limited number of instructions, the number of issuing instructions is dependent on the type of the instructions. Table 46 indicates each instruction's type. A check mark (✓) in the column indicates that the instruction(s) can be queued in the named queue; an '✗' indicates that it uses multiple queues.

Note:

> Unlike Serial number and rename register, the DFM queue entry becomes available when the instruction using the entry starts its execution, rather than when the instruction is retired except DFM LSU.

.

Table 46: Where Instructions are Queued

Instructions	INT Queue	INT/ AGEN Queue	Load/ Store Queue	FP Queue	WP Queue	SYS
ADD, ADDcc	✓	✓				
ADDC, ADDCcc	✓					
AND, ANDcc, ANDN, ANDNcc	✓	✓				
BPcc, Bicc, BPr					✓	
CALL	✗				✗	
CASA, CASXA		✓				
DONE	✗				✗	

Table 46: Where Instructions are Queued (*Continued*)

Instructions	INT Queue	INT/ AGEN Queue	Load/ Store Queue	FP Queue	WP Queue	SYS
FABS				✓		
FADD				✓		
FBfcc, FBPfcc					✓	
FCMP, FCMPE, FiTO(s,d)				✓		
FDIV				✓		
FLUSH		✗	✗			
FLUSHW						✓
FMOV, FMOVcc, FMOVr, FMOVr, FMUL, FNEG, FsMULd, F(s,d)TOi, F(s,d)TO(s,d), F(s,d)TOx, FxTO(s,d)				✓		
FSQRT				✓		
FSUB				✓		
ILLTRAP						✓
IMPDEP2(FMA)				✓		
JMPL		✗			✗	
LDD, LDDA		✗	✗		✗	
LDDF, LDDF, LDDFA, LDF, LDFA, LDFSR		✗	✗			
LDSB, LDSBA, LDSH, LDSHA, LDSTUB, LDSTUBA, LDSW, LDSWA, LDUB, LDUBA, LDUH, LDUHA, LDUW, LDUWA, LDX, LDXA, LDXFSR		✗	✗			
MEMBAR		✗	✗			
MOVcc	✓					
MOVr		✓				
MULScc	✓					
MULX	✓					
NOP	✓	✓				
UMUL,SMUL,UMULcc,SMULcc	✓					
OR, ORcc, ORN, ORNcc	✓	✓				
PREFETCH, PREFETCHA		✗	✗			
RDASI, RDASR, RDCCR, RDFPRS, RDPC, RDPR, RDTICK, RDY	✗				✗	
RESTORE	✓					
RESTORED					✗	✗
RETRY	✗				✗	
RETURN	✗				✗	
SAVE	✗				✗	
SAVED					✗	✗
SDIV, SDIVcc, SDIVX	✓					
SETHI	✓	✓				
SIR						✓
SLL, SLLX	✓					

Table 46: Where Instructions are Queued (*Continued*)

Instructions	INT Queue	INT/ AGEN Queue	Load/ Store Queue	FP Queue	WP Queue	SYS
SDIV, SDIVXcc, SDIVX	✓					
SLL, SRL	✓					
SMUL, SMULcc	✓					
SRA, SRAX, SRL, SRLX	✓					
STB, STBA, STBAR		✗	✗			
STD, STDA		✗	✗		✗	
STDF, STDFA, STF, STFA, STFSR		✗	✗			
STH, STHA, STW, STWA, STX, STXA, STXFSR		✗	✗			
SUB, SUBcc	✓	✓				
SUBC, SUBCcc	✓					
SWAP, SWAPA		✗	✗			
TADDcc, TADDccTV	✓					
Tcc	✗				✗	
TSUBcc, TSUBccTV	✓					
UDIV, UDIVcc, UDIVX	✓					
UMUL, UMULcc	✓					
WRASI, WRASR, WRCCR, WRFPRS, WRPR, WRY	✗				✗	
XOR,XORcc,XNOR,XNORcc	✓	✓				

9.8.5 Mixture of Normal Integer Instructions

When issuing a mixture of normal integer instructions (for example, ADD and SHIFT), the CPU exhibits a specific behavior. If an issue window contains 4 integer/shift instructions, they could be issued in 2 possible ways, based on the order of the operations. Table 47 and Table 48 illustrate the two possible issue orderings:

Table 47: Order = shift1, shift2, add1, add2

Execution Unit	Operation	Cycle #
ALS1	`shift 1`	1
ALS2	`shift 2`	1
ALU3	`add 1`	1
ALU4	`add 2`	1

Table 48: Order = add1, add2, shift1, shift2

Execution Unit	Operation	Cycle #
ALS1	`add 1`	1
ALS2	`add 2`	1
ALS1	`shift 1`	2
ALS2	`shift 2`	2

The second case takes two clocks. This anomaly occurs because the first 2 integer or shift instructions are always sent to the INT queue and the remaining integer instructions are checked to see if they can go to the INT/AGEN queue. Since shift cannot go to the INT/AGEN queue, the second case cannot execute the shifts in the same clock as the ADDs. This issue constraint is caused by a critical speed path, which does not allow enough time to switch the order of the instructions so that they could all issue in a single clock.

Note:

> Although the number of instructions being issued is calculated assuming the first 2 integer instructions are always sent to the INT queue, it is not guaranteed that the first two integer instructions will be dispatched to the INT queue, except shift and some other instructions (see Table 46 on page 209 for details). SPARC64-III tries to issue and dispatch instructions to the INT queue and INT/AGEN queue in turn to prevent ALU3/ALU4 from being idle.

9.8.6 Dynamic Resources

An instruction cannot be issued if corresponding dynamic resources are not available. The instruction will stall until previous instructions using the same resource are retired and the resource again becomes available.

9.8.6.1 Serial Number

Every issued instruction is tagged with a serial number. The number of available serial numbers is 63.

9.8.7 Rename Register

Any instruction which will write into an integer register (except %g0, %ag0), a condition code register, or floating point register requires an available rename register. The number of available rename registers is 34 for integer, 27 for cc, and 32 (even)+32 (odd) for floating point registers.

A Instruction Definitions

A.1 Overview

This appendix describes each SPARC64-III instruction. Related instructions are grouped into subsections. Each subsection consists of these 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 a conforming SPARC-V9 implementation encounters nonzero values in these fields, its behavior is undefined. See Appendix I, "Extending the SPARC-V9 Architecture" in *V9* for information about extending the SPARC-V9 instruction set.

3. A list of the suggested assembly language syntax; the syntax notation is described in Appendix G, "Assembly Language Syntax".

4. A description of the features, restrictions, and exception-causing conditions.

5. (5) 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,* ***32i_instruction_access_MMU_miss, async_error, watchdog,*** and interrupts are not listed since they can occur on any instruction. Also any instruction that is not implemented in hardware shall generate an *illegal_instruction* exception (or *fp_exception_other* exception with *ftt=unimplemented_FPop* for floating-point instructions) when it is executed. The ***data_breakpoint*** trap can occur on any data memory access instruction and the ***programmed_emulation_trap*** can occur during chip debug on any instruction that has been programmed into one of the CPU's Emulation Trap Registers (ETR). These traps are also not listed under each instruction.

The following traps **never** occur in SPARC64-III:

■ watchdog_reset

■ instruction_access_MMU_miss

■ internal_processor_error

■ data_access_MMU_miss

■ data_access_protection

■ unimplemented_LDD

■ unimplemented_STD

■ LDQF_mem_address_not_aligned

■ STQF_mem_address_not_aligned

■ async_data_error

■ fp_exception_other (ftt = invalid_fp_register)

The descriptions in this Appendix list the traps that will not occur for each instruction group. However, traps in the list above are omitted from the lists in the following pages since they can never occur in SPARC64-III.

This appendix does not include any timing information (in either cycles or clock time), since timing is implementation-dependent.

Table 50 summarizes the instruction set; the instruction definitions follow the table. Within Table 50, throughout this appendix, and in Appendix E, "Opcode Maps," certain opcodes are marked with mnemonic superscripts. The superscripts and their meanings are defined in Table 49:

Table 49: Opcode Superscripts (V9=21)

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 TICK.NPT = 1

Table 50: Instruction Set (V9=22)

Opcode	Name	Page
ADD (ADDcc)	Add (and modify condition codes)	218
ADDC (ADDCcc)	Add with carry (and modify condition codes)	218
AND (ANDcc)	And (and modify condition codes)	270
ANDN (ANDNcc)	And not (and modify condition codes)	270
BPcc	Branch on integer condition codes with prediction	229
Bicc^D	Branch on integer condition codes	227
BPr	Branch on contents of integer register with prediction	219
CALL	Call and link	232
CASA^{PASI}	Compare and swap word in alternate space	233
CASXA^{PASI}	Compare and swap doubleword in alternate space	233
DONE^P	Return from trap	238
FABS(s,d,q)	Floating-point absolute value	246

Table 50: Instruction Set (*Continued*)(**V9=22**)

Opcode	Name	Page
FADD(s,d,q)	Floating-point add	239
FBfcc[D]	Branch on floating-point condition codes	221
FBPfcc	Branch on floating-point condition codes with prediction	224
FCMP(s,d,q)	Floating-point compare	240
FCMPE(s,d,q)	Floating-point compare (exception if unordered)	240
FDIV(s,d,q)	Floating-point divide	248
FdMULq	Floating-point multiply double to quad	248
FiTO(s,d,q)	Convert integer to floating-point	245
FLUSH	Flush instruction memory	251
FLUSHW	Flush register windows	253
FMOV(s,d,q)	Floating-point move	246
FMOV(s,d,q)cc	Move floating-point register if condition is satisfied	275
FMOV(s,d,q)*r*	Move f-p reg. if integer reg. contents satisfy condition	279
FMUL(s,d,q)	Floating-point multiply	248
FNEG(s,d,q)	Floating-point negate	246
FsMULd	Floating-point multiply single to double	248
FSQRT(s,d,q)	Floating-point square root	250
F(s,d,q)TOi	Convert floating point to integer	242
F(s,d,q)TO(s,d,q)	Convert between floating-point formats	243
F(s,d,q)TOx	Convert floating point to 64-bit integer	242
FSUB(s,d,q)	Floating-point subtract	239
FxTO(s,d,q)	Convert 64-bit integer to floating-point	245
ILLTRAP	Illegal instruction	254
IMPDEP1	Implementation-dependent instruction	255
IMPDEP2	Implementation-dependent instruction	255
JMPL	Jump and link	258
LDD[D]	Load doubleword	263
LDDA[D, PASI]	Load doubleword from alternate space	265
LDDF	Load double floating-point	259
LDDFA[PASI]	Load double floating-point from alternate space	261
LDF	Load floating-point	259
LDFA[PASI]	Load floating-point from alternate space	261
LDFSR[D]	Load floating-point state register lower	259
LDQF	Load quad floating-point	259
LDQFA[PASI]	Load quad floating-point from alternate space	261
LDSB	Load signed byte	263
LDSBA[PASI]	Load signed byte from alternate space	265
LDSH	Load signed halfword	263
LDSHA[PASI]	Load signed halfword from alternate space	265
LDSTUB	Load-store unsigned byte	268
LDSTUBA[PASI]	Load-store unsigned byte in alternate space	269
LDSW	Load signed word	263
LDSWA[PASI]	Load signed word from alternate space	265
LDUB	Load unsigned byte	263
LDUBA[PASI]	Load unsigned byte from alternate space	265

Table 50: **Instruction Set** (*Continued*)(**V9=22**)

Opcode	Name	Page
LDUH	Load unsigned halfword	263
LDUHA[PASI]	Load unsigned halfword from alternate space	265
LDUW	Load unsigned word	263
LDUWA[PASI]	Load unsigned word from alternate space	265
LDX	Load extended	263
LDXA[PASI]	Load extended from alternate space	265
LDXFSR	Load floating-point state register	259
MEMBAR	Memory barrier	272
MOVcc	Move integer register if condition is satisfied	281
MOVr	Move integer register on contents of integer register	285
MULScc[D]	Multiply step (and modify condition codes)	290
MULX	Multiply 64-bit integers	287
NOP	No operation	292
OR (ORcc)	Inclusive-or (and modify condition codes)	270
ORN (ORNcc)	Inclusive-or not (and modify condition codes)	270
POPC	Population count	293
PREFETCH	Prefetch data	295
PREFETCHA[PASI]	Prefetch data from alternate space	295
RDASI	Read ASI register	303
RDASR[PASR]	Read ancillary state register	303
RDCCR	Read condition codes register	303
RDFPRS	Read floating-point registers state register	303
RDPC	Read program counter	303
RDPR[P]	Read privileged register	301
RDTICK[PNPT]	Read TICK register	303
RDY[D]	Read Y register	303
RESTORE	Restore caller's window	307
RESTORED[P]	Window has been restored	309
RETRY[P]	Return from trap and retry	238
RETURN	Return	306
SAVE	Save caller's window	307
SAVED[P]	Window has been saved	309
SDIV[D] (SDIVcc[D])	32-bit signed integer divide (and modify condition codes)	235
SDIVX	64-bit signed integer divide	287
SETHI	Set high 22 bits of low word of integer register	310
SIR	Software-initiated reset	313
SLL	Shift left logical	311
SLLX	Shift left logical, extended	311
SMUL[D] (SMULcc[D])	Signed integer multiply (and modify condition codes)	288
SRA	Shift right arithmetic	311
SRAX	Shift right arithmetic, extended	311
SRL	Shift right logical	311
SRLX	Shift right logical, extended	311
STB	Store byte	319
STBA[PASI]	Store byte into alternate space	321

Table 50: Instruction Set (*Continued*)(**V9=22**)

Opcode	Name	Page
STBAR^D	Store barrier	314
STD^D	Store doubleword	319
STDA$^{D, PASI}$	Store doubleword into alternate space	321
STDF	Store double floating-point	315
STDFA^{PASI}	Store double floating-point into alternate space	317
STF	Store floating-point	315
STFA^{PASI}	Store floating-point into alternate space	317
STFSR^D	Store floating-point state register	315
STH	Store halfword	319
STHA^{PASI}	Store halfword into alternate space	321
STQF	Store quad floating-point	315
STQFA^{PASI}	Store quad floating-point into alternate space	317
STW	Store word	319
STWA^{PASI}	Store word into alternate space	321
STX	Store extended	319
STXA^{PASI}	Store extended into alternate space	321
STXFSR	Store extended floating-point state register	315
SUB (SUBcc)	Subtract (and modify condition codes)	323
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)	323
SWAP^D	Swap integer register with memory	324
SWAPA$^{D, PASI}$	Swap integer register with memory in alternate space	325
TADDcc (TADDccTV^D)	Tagged add and modify condition codes (trap on overflow)	327
Tcc	Trap on integer condition codes	331
TSUBcc (TSUBccTV^D)	Tagged subtract and modify condition codes (trap on overflow)	329
UDIV^D (UDIVcc^D)	Unsigned integer divide (and modify condition codes)	235
UDIVX	64-bit unsigned integer divide	287
UMUL^D (UMULcc^D)	Unsigned integer multiply (and modify condition codes)	287
WRASI	Write ASI register	337
WRASR^{PASR}	Write ancillary state register	337
WRCCR	Write condition codes register	337
WRFPRS	Write floating-point registers state register	337
WRPR^P	Write privileged register	334
WRY^D	Write Y register	337
XNOR (XNORcc)	Exclusive-nor (and modify condition codes)	270
XOR (XORcc)	Exclusive-or (and modify condition codes)	270

A.2 Add

Opcode	Op3	Operation
ADD	00 0000	Add
ADDcc	01 0000	Add and modify cc's
ADDC	00 1000	Add with Carry
ADDCcc	01 1000	Add 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	
add	reg_{rs1}, reg_or_imm, reg_{rd}
addcc	reg_{rs1}, reg_or_imm, reg_{rd}
addc	reg_{rs1}, reg_or_imm, reg_{rd}
addccc	reg_{rs1}, reg_or_imm, reg_{rd}

Description:

ADD and ADDcc compute "$r[rs1] + r[rs2]$" if $i = 0$, or "$r[rs1]$ + sign_ext($simm13$)" if $i = 1$, and write the sum into $r[rd]$.

ADDC and ADDCcc ("ADD with carry") also add the CCR register's 32-bit carry ($icc.c$) bit; that is, they compute "$r[rs1] + r[rs2] + icc.c$" or "$r[rs1]$ + sign_ext($simm13$) + $icc.c$" and write the sum into $r[rd]$.

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

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

Compatibility Note:
> ADDC and ADDCcc were named ADDX and ADDXcc, respectively, in SPARC-V8.

Exceptions:
> (none)

A.3   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	$r[rs1] \leq 0$
BRLZ	011	Branch on Register Less Than Zero	$r[rs1] < 0$
—	100	*Reserved*	—
BRNZ	101	Branch on Register Not Zero	$r[rs1] \neq 0$
BRGZ	110	Branch on Register Greater Than Zero	$r[rs1] > 0$
BRGEZ	111	Branch on Register Greater Than or Equal to Zero	$r[rs1] \geq 0$

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*." The preceding table indicates that the ",a" is optional by enclosing it in braces. 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".

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 1, the delay instruction is annulled (not executed).

The predict bit (*p*) is used to give the hardware a hint about whether the branch is expected to be taken. A 1 in the *p* bit indicates that the branch is expected to be taken; a 0 indicates that the branch is expected not to be taken.

See 6.3.4.1, "Conditional Branches" for details on how SPARC64-III interprets the "p" (branch prediction) bit.

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

Implementation Note:

> If this instruction is implemented by tagging each register value with an N (negative) bit and Z (zero) bit, use the table below to determine if *rcond* is TRUE:

Branch	Test
BRNZ	**not** Z
BRZ	Z
BRGEZ	**not** N
BRLZ	N
BRLEZ	N **or** Z
BRGZ	**not** (N **or** Z)

Exceptions:

> *illegal_instruction* (if $rcond = 000_2$ or 100_2)

A.4   Branch on Floating-point Condition Codes (FBfcc)

> The FBfcc 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. It is recommended that the FBPfcc instructions be used in their place.

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*." The preceding table indicates that the "`,a`" is optional by enclosing it in braces.

Description:

Unconditional Branches (FBA, FBN):

> If its annul field is 0, an FBN (Branch Never) instruction acts like a NOP. If its annul field is 1, the following (delay) instruction is annulled (not executed) when the FBN is executed. In neither case does a transfer of control take place.
>
> FBA (Branch Always) causes a PC-relative, delayed control transfer to the address "PC + (4 × sign_ext(*disp22*))," regardless of the value of the floating-point condition code bits. If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

Fcc-Conditional Branches:

> Conditional FBfcc instructions (except FBA and FBN) evaluate floating-point condition code zero (*fcc0*) according to the *cond* field of the instruction. Such evaluation produces either a TRUE or FALSE result. If TRUE, the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 × sign_ext(*disp22*))." If FALSE, the branch is not taken.
>
> If a conditional branch is taken, the delay instruction is always executed, regardless of the value of the annul field. If a conditional branch is not taken and the *a* (annul) field is 1, the delay instruction is annulled (not executed). **Note**: The annul bit has a **different** effect on conditional branches than it does on unconditional branches.

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

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 an *fp_disabled* exception.

Exceptions:

fp_disabled

A.5 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	*fcc0*
01	*fcc1*
10	*fcc2*
11	*fcc3*

Assembly Language Syntax		
`fba{,a}{,pt|,pn}`	`%fccn, label`	
`fbn{,a}{,pt|,pn}`	`%fccn, label`	
`fbu{,a}{,pt|,pn}`	`%fccn, label`	
`fbg{,a}{,pt|,pn}`	`%fccn, label`	
`fbug{,a}{,pt|,pn}`	`%fccn, label`	
`fbl{,a}{,pt|,pn}`	`%fccn, label`	
`fbul{,a}{,pt|,pn}`	`%fccn, label`	
`fblg{,a}{,pt|,pn}`	`%fccn, label`	
`fbne{,a}{,pt|,pn}`	`%fccn, label`	(*synonym*: `fbnz`)
`fbe{,a}{,pt|,pn}`	`%fccn, label`	(*synonym*: `fbz`)
`fbue{,a}{,pt|,pn}`	`%fccn, label`	
`fbge{,a}{,pt|,pn}`	`%fccn, label`	
`fbuge{,a}{,pt|,pn}`	`%fccn, label`	
`fble{,a}{,pt|,pn}`	`%fccn, label`	
`fbule{,a}{,pt|,pn}`	`%fccn, label`	
`fbo{,a}{,pt|,pn}`	`%fccn, label`	

Programming Note:

To set the annul bit for FBPfcc instructions, append "`,a`" to the opcode mnemonic. For example, use "`fbl,a %fcc3,label`." The preceding table indicates that the "`,a`" is optional by enclosing it in braces. 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 (FBPA, FBPN):

If its annul field is 0, an FBPN (Floating-Point Branch Never with Prediction) instruction acts like a NOP. If the Branch Never's annul field is 0, the following (delay) instruction is executed; if the annul field is 1, the following instruction is annulled (not executed). In no case does an FBPN cause a transfer of control to take place.

FBPA (Floating-Point Branch Always with Prediction) causes an unconditional PC-relative, delayed control transfer to the address "PC + ($4 \times$ sign_ext(*disp19*))." If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

Fcc-Conditional Branches:

Conditional FBPfcc instructions (except FBPA and FBPN) evaluate one of the four floating-point condition codes (*fcc0, fcc1, fcc2, fcc3*) as selected by *cc0* and *cc1*, according to the *cond* field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + ($4 \times$ sign_ext(*disp19*))." If FALSE, the branch is not taken.

If a conditional branch is taken, the delay instruction is always executed, regard-less of the value of the annul field. If a conditional branch is not taken and the *a* (annul) field is 1, the delay instruction is annulled (not executed). **Note**: The annul bit has a **different** effect on conditional branches than it does on unconditional branches.

The predict bit (*p*) is used to give the hardware a hint about whether the branch is expected to be taken. A 1 in the *p* bit indicates that the branch is expected to be taken. A 0 indicates that the branch is expected not to be taken. See 5.2.11.1, "Hardware Mode Register (ASR18)" and 9.3, "Branches and Branch Prediction" for the details of the predict bit handling in SPARC64-III.

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

If FPRS.FEF = 0 or PSTATE.PEF = 0, or if an FPU is not present, an FBPfcc instruction is not executed and instead, generates an *fp_disabled* exception.

Compatibility Note:
Unlike SPARC-V8, SPARC-V9 does not require an instruction between a floating-point compare operation and a floating-point branch (FBfcc, FBPfcc).

Exceptions:
fp_disabled

A.6 Branch on Integer Condition Codes (Bicc)

> The Bicc 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. It is recommended that the BPcc instructions be used in their place.

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):

00	a	cond	010	disp22

31 30 29 28   25 24   22 21   0

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*." The preceding table indicates that the "`,a`" is optional by enclosing it in braces.

Description:

Unconditional Branches (BA, BN):

If its annul field is 0, a BN (Branch Never) instruction is treated as a NOP by SPARC64-III. If its annul field is 1, the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

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

Icc-Conditional Branches:

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

If a conditional branch is taken, the delay instruction is always executed regardless of the value of the annul field. If a conditional branch is not taken and the *a* (annul) field is 1, the delay instruction is annulled (not executed). **Note**: The annul bit has a **different** effect on conditional branches than it does on unconditional branches.

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

Exceptions:

(none)

A.7 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):

00	a	cond	001	cc1	cc0	p	disp19

31 30 29 28   25 24   22 21 20 19 18   0

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`." The preceding table indicates that the ",a" is optional by enclosing it in braces. 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 (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. **Note:** SPARC64-III treats this instruction as a NOP; it cannot be used as an instruction prefetch. If the Branch Never's annul field is 1, the following (delay) instruction is annulled (not executed). If the annul field is 0, 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 1, the delay instruction is annulled (not executed). If the annul field is 0, 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 *a* (annul) field is 1, 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 1 in the *p* bit indicates that the branch is expected to be taken; a 0 indicates that the branch is expected not to be taken. See 5.2.11.1, "Hardware Mode Register (ASR18)" and 9.3, "Branches and Branch Prediction" for the details of the predict bit handling in SPARC64-III.

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

Exceptions:
   illegal_instruction $(cc1 \;\square\; cc0 = 01_2$ or $11_2)$

A.8 Call and Link

Opcode	op	Operation
CALL	01	Call and Link

Format (1):

01	disp30

31 30 29   0
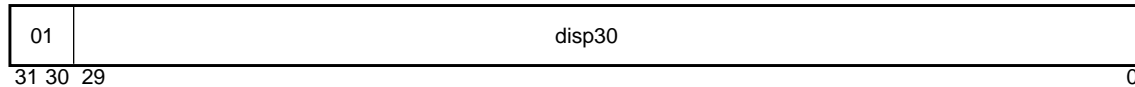
Assembly Language Syntax
call   *label*

Description:

The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address PC + ($4 \times$ sign_ext(*disp30*)). Since the word displacement (*disp30*) field is 30 bits wide, the target address lies within a range of -2^{31} to $+2^{31} - 4$ bytes. The PC-relative displacement is formed by sign-extending the 30-bit word displacement field to 62 bits and appending two low-order zeros 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). **Note:** SPARC64-III stores all 64 bits of the PC value in *r*[15], regardless of the setting of PSTATE.AM. The value written into *r*[15] is visible to the instruction in the delay slot.

Programming Note:
> In SPARC64-III the return address of the CALL (PC + 8) is stored in a hardware table. When a *ret* or *retl* is executed, the value in the table is used to predict the return address. See 6.3.4.3, "CALL and JMPL Instructions" for details of how this hardware table works.

Exceptions:
> (none)

A.9 Compare and Swap

Opcode	op3	Operation
CASA^{PASI}	11 1100	Compare and Swap Word from Alternate space
CASXA^{PASI}	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:

These instructions are used for synchronization and memory updates by concurrent processes. Uses of compare-and-swap include spin-lock operations, updates of shared counters, and updates of linked-list pointers. The latter 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, 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, a compare, and a 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. CASXA and CASA also cause the CPU to sync.

The coherence and atomicity of memory operations between processors and I/O DMA memory accesses are implementation-dependent (impl. dep #120).

Implementation Note:
> An implementation might cause an exception due to an error during the store memory access, even though there was no error during the load memory access.

Programming Note:
> Compare and Swap (CAS) and Compare and Swap Extended (CASX) synthetic instructions are available for "big endian" memory accesses. Compare and Swap Little (CASL) and Compare and Swap Extended Little (CASXL) synthetic instructions are available for "little endian" memory accesses. See G.3, "Synthetic Instructions", for these synthetic instructions' syntax.

The compare-and-swap instructions do not affect the condition codes.

Exceptions:
> *privileged_action*
> *mem_address_not_aligned*
> *data_access_exception*
> *data_access_error*
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.10 Divide (64-bit / 32-bit)

> The UDIV, UDIVcc, SDIV, and SDIVcc 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. It is recommended that the UDIVX and SDIVX instructions be used in their place.

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):

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	
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 $\square$ *lower 32 bits of r[rs1]*) ÷ *lower 32 bits of r[rs2]*." Otherwise (that is, if $i = 1$), the divide instructions compute "(Y $\square$ *lower 32 bits of r[rs1]*) ÷ *lower 32 bits of* sign_ext(*simm13*)." In either case, if overflow does not occur, the less significant 32 bits of the integer quotient are sign-or zero-extended to 64 bits and are written into *r[rd]*.

The contents of the Y register are undefined after any 64-bit by 32-bit integer divide operation.

Unsigned Divide:

Unsigned divide (UDIV, UDIVcc) assumes an unsigned integer doubleword dividend (Y $\square$ *lower 32 bits of r[rs1]*) and an unsigned integer word divisor (*lower 32 bits of r[rs2]* or *lower 32 bits of* sign_ext(*simm13*)) and computes an unsigned integer word quotient (*r[rd]*). Immediate values in *simm13* are in the ranges $0..2^{12} - 1$ and $2^{32} - 2^{12}..2^{32} - 1$ for unsigned divide instructions.

Unsigned division rounds an inexact rational quotient toward zero.

Programming Note:

> The **rational quotient** is the infinitely precise result quotient. It includes both the integer part and the fractional part of the result. For example, the rational quotient of $11/4 = 2.75$ (Integer part $= 2$, fractional part $= .75$).

The result of an unsigned divide instruction can overflow the low-order 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 is specified in Table 51.

Table 51: UDIV / UDIVcc Overflow Detection and Value Returned (V9=23)

Condition under Which Overflow Occurs	Value Returned in *r[rd]*
Rational quotient $\geq 2^{32}$	$2^{32}-1$ (0000 0000 FFFF FFFF$_{16}$)

When no overflow occurs, the 32-bit result is zero-extended to 64 bits and written into register $r[rd]$.

UDIV does not affect the condition code bits. UDIVcc writes the integer condition code bits as shown in Table 52. **Note**: Negative (N) and zero (Z) are set according to the value of $r[rd]$ after it has been set to reflect overflow, if any.

Table 52: Integer Condition Code Bits for UDIVcc

Bit	UDIVcc
icc.N	Set if $r[rd]\langle 31\rangle = 1$
icc.Z	Set if $r[rd]\langle 31{:}0\rangle = 0$
icc.V	Set if overflow (*per Table 51*)
icc.C	Zero
xcc.N	Set if $r[rd]\langle 63\rangle = 1$
xcc.Z	Set if $r[rd]\langle 63{:}0\rangle = 0$
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, $-7 \div 4$ 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 53.

Table 53: SDIV / SDIVcc Overflow Detection and Value Returned (V9=24)

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 Table 54. **Note**: Negative (N) and zero (Z) are set according to the value of *r[rd]* after it has been set to reflect overflow, if any.

Table 54: Integer Condition Code Bits for SDIVcc

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 53*)
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.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:
> The DONE and RETRY instructions should be used to return from privileged trap handlers.

Exceptions:
> *privileged_opcode*
> *illegal_instruction* (if TL = 0 or *fcn* = 2..31)

A.12 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, and they 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, and write the difference into the floating-point register(s) specified by the *rd* field.

Rounding is performed as specified by the FSR.RD field.

Note: SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt = unimplemented_FPop*). Supervisor software then emulates these instructions.

Exceptions:

> *fp_disabled*
> *fp_exception_ieee_754* (OF, UF, NX, NV)
> *fp_exception_other* (*ftt = unimplemented_FPop* (FADDQ and FSUBQ only))

A.13 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*) according to Table 55:

Table 55: Floating-point Condition Code Values

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 above table indicates that the comparison is unordered. The unordered condition occurs when one or both of the operands to the compare is a signaling 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 signaling NaN.

Compatibility Note:
> Unlike SPARC-V8, SPARC-V9 does not require an instruction between a floating-point compare operation and a floating-point branch (FBfcc, FBPfcc).

Compatibility Note:
> 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 are used to 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 based on `fcc0`.

Note: SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt = unimplemented_FPop*). Supervisor software then emulates these instructions.

Exceptions:
> *fp_disabled*
> *fp_exception_ieee_754* (NV)
> *fp_exception_other* (*ftt = unimplemented_FPop* (FCMPq, FCMPEq only))

A.14   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, an invalid (NV) exception occurs. The value written into the floating-point register(s) specified by *rd* in these cases is defined in B.5, "Integer Overflow Definition".

Note:

> SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt = unimplemented_FPop*). Supervisor software then emulates these instructions.

Exceptions:

> *fp_disabled*
> *fp_exception_ieee_754* (NV, NX)
> *fp_exception_other* (*ftt = unimplemented_FPop* (FqTOi, FqTOx only))

A.15   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 signaling NaN.

B.2.1, "Untrapped Result in Different Format from Operands", defines the rules for converting NaNs from one floating-point format to another.

Note:

SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt* = *unimplemented_FPop*). Supervisor software then emulates these instructions.

Exceptions:

    fp_disabled

    fp_exception_ieee_754 (OF, UF, NV, NX)

    fp_exception_other (*ftt = unimplemented_FPop* (FsTOq, FdTOq, FqTOs, FqTOd
        only))

A.16 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 register(s) specified by *rs2* into a floating-point number in the destination format. The source register, floating-point register(s) specified by *rs2*, must be an even-numbered (that is, double-precision) floating-point register.

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: SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt = unimplemented_FPop*). Supervisor software then emulates these instructions.

Exceptions:

   fp_disabled
   fp_exception_ieee_754 (NX (FiTOs, FxTOs, FxTOd only))
   fp_exception_other (*ftt = unimplemented_FPop* (FiTOq, FxTOq only))

A.17   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 forms 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:

SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt* = *unimplemented_FPop*). Supervisor software then emulates these instructions.

Exceptions:
>*fp_disabled*
>*fp_exception_other* (*ftt = unimplemented_FPop* (FMOVq, FNEGq, FABSq only))

A.18   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, and they 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, and write the quotient into the floating-point register(s) specified by the *rd* field.

Rounding is performed as specified by the FSR.RD field.

Note:

SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt* = *unimplemented_FPop*). Supervisor software then emulates these instructions.

Note:

> For FDIVs and FDIVd, an *fp_exception_other* with *ftt* = *unfinished_FPop* can occur if the divide unit detects certain unusual conditions. See 5.1.7.6, "FSR_floating-point_trap_type (ftt)", for details.

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* (FDIVs and FDIVd only))

A.19 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. In SPARC-V9 rounding is performed as specified by the FSR.RD field.

Note:

> SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt = unimplemented_FPop*). 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 possitive denormalized. See 5.1.7.6, "FSR_floating-point_trap_type (ftt)" for additional details.

Exceptions:

> *fp_disabled*
> *fp_exception_ieee_754 (IEEE_754_exception (NV,NX))*
> *fp_exception_other* (*unimplemented_FPop*) *(Quad forms)*
> *fp_exception_other (unfinished_FPop)*

A.20 Flush Instruction Memory

Opcode	op3	Operation
FLUSH	11 1011	Flush Instruction Memory

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
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 A.32, "Memory Barrier".

The effective address operand for the FLUSH instruction is "$r[rs1] + r[rs2]$" if $i = 0$, or "$r[rs1] + \text{sign\_ext}(simm13)$" if $i = 1$. The least significant two address bits of the effective address are unused and should be supplied as zeros by software. Bit 2 of the address is ignored, because FLUSH operates on at least a doubleword.

On SPARC64-III FLUSH operates on the full cache line (64 bytes) containing the addressed location.

Programming Notes:

1. Typically, FLUSH is used in self-modifying code. See H.1.6, "Self-Modifying Code" in *V9* for information about use of the FLUSH instruction in portable self-modifying code. The use of self-modifying code is discouraged.

2. The order in which memory is modified can be controlled by using 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, care must be taken to 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 implementation-dependent. No mechanism is provided to ensure or test completion.

6. Because FLUSH is designed to act on a doubleword, and because, on some implementations, FLUSH may trap to system software, it is recommended that system software provide a user-callable service routine for flushing arbitrarily sized regions of memory. On some implementations, this routine would issue a series of FLUSH instructions; on others, it might issue a single trap to system software that would then flush the entire region.

Implementation Notes:

The effect of a FLUSH instruction as observed from P_{FLUSH} is immediate. Other processors in a multiprocessor system eventually will see the effect of the FLUSH, but the latency is implementation-dependent.

Exceptions:

Since the FLUSH instruction accesses the TLB's in SPARC64-III, the traps listed below can occur on a FLUSH. However, it is expected that the kernel trap handlers will make the FLUSH appear to have not trapped to a non privileged programmer. Therefore, to the non privileged programmer the FLUSH appears to conform with the SPARC V9 definition of not trapping.

data_access_exception
data_access_error
32i_data_access_MMU_miss
32i_data_access_protection

A.21 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
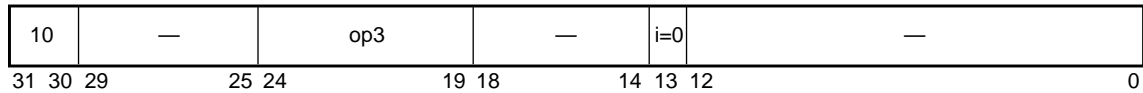
Assembly Language Syntax
flushw

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 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). See 6.3.6, "Register Window Management Instructions".

Programming Note:
> Typically, the spill handler saves a window on a memory stack and returns to reexecute the FLUSHW instruction. Thus, FLUSHW traps and reexecutes until all active windows other than the current window have been spilled.

Exceptions:
> *spill_n_normal*
> *spill_n_other*

A.22   Illegal Instruction Trap

Opcode	op	op2	Operation
ILLTRAP	00	000	*illegal_instruction* trap

Format (2):

00	—	000	const22

31 30 29   25 24   22 21   0
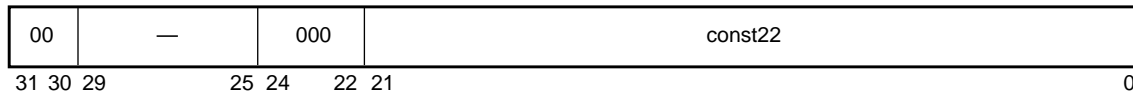
Assembly Language Syntax
illtrap   *const22*

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.23 Implementation-dependent Instructions

Opcode	op3	Operation
IMPDEP1	11 0110	Implementation-Dependent Instruction 1
IMPDEP2	11 0111	Implementation-Dependent Instruction 2

Format (3):

10	*impl-dep*	op3	*impl-dep*

31 30 29   25 24   19 18   0
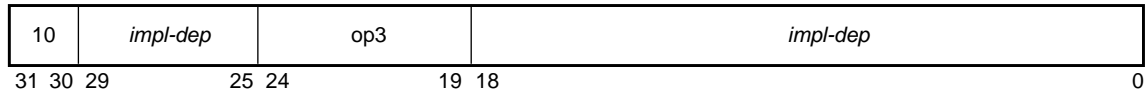
Description:

The IMPDEP1 and IMPDEP2 instructions are completely implementation-dependent. Implementation-dependent aspects include their operation, the interpretation of bits 29..25 and 18..0 in their encodings, and which (if any) exceptions they may cause.

See I.1.2, "Implementation-Dependent and Reserved Opcodes" in *V9* for information about extending the SPARC-V9 instruction set using the implementation-dependent instructions.

Compatibility Note:
These instructions replace the CPop*n* instructions in SPARC-V8.

Note: SPARC64-III uses IMPDEP2 to encode the Floating-point Multiply Add/Subtract instructions. See A.23.1, "IMPDEP2 (Floating-point Multiply-Add/Subtract)", for details.

SPARC64-III does not use IMPDEP1; attempts to execute an IMPDEP1 opcode cause an *illegal_instruction* exception.

Exceptions:
illegal_instruction (IMPDEP1)
implementation-dependent (IMPDEP2)

A.23.1 IMPDEP2 (Floating-point Multiply-Add/Subtract)

Opcode	Variation	Size†	Operation
FMADDs	00	01	Multiply-Add Single
FMADDd	00	10	Multiply-Add Double
FMSUBs	01	01	Multiply-Subtract Single
FMSUBd	01	10	Multiply-Subtract Double
FNMADDs	11	01	Negative Multiply-Add Single
FNMADDd	11	10	Negative Multiply-Add Double
FNMSUBs	10	01	Negative Multiply-Subtract Single
FNMSUBd	10	10	Negative Multiply-Subtract Double

† 11 is reserved for quad.

Format (5):

10	rd	110111	rs1	rs3	var	size	rs2

31 30 29   25 24   19 18   14 13   9 8   7 6   5 4   0

Operation	Implementation
Multiply-Add	$rs1 \times rs2 + rs3 \rightarrow rd$
Multiply-Subtract	$rs1 \times rs2 - rs3 \rightarrow rd$
Negative Multiply-Subtract	$-(rs1 \times rs2 - rs3) \rightarrow rd$
Negative Multiple-Add	$-(rs1 \times rs2 + rs3) \rightarrow rd$

Assembly Language Syntax	
fmadds	$freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}$
fmaddd	$freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}$
fmsubs	$freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}$
fmsubd	$freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}$
fnmadds	$freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}$
fnmaddd	$freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}$
fnmsubs	$freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}$
fnmsubd	$freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}$

Description:

The floating-point multiply-add instructions multiply the registers specified by the *rs1* field by the registers specified by the *rs2* field, then add that product to the registers specified by the *rs3* field and write the result into the registers specified by the *rd* field.

The floating-point multiply-subtract instructions multiply the registers specified by the *rs1* field by the registers specified by the *rs2* field, then subtract from that product the registers specified by the *rs3* field and write the result into the registers specified by the *rd* field.

The floating-point negative multiply-add instructions multiply the registers specified by the *rs1* field by the registers specified by the *rs2* field, then add that product to the registers specified by the *rs3* field and write the **negation** of the result into the registers specified by the *rd* field.

The floating-point negative multiply-subtract instructions multiply the registers specified by the *rs1* field by the registers specified by the *rs2* field, then subtract from that product the registers specified by the *rs3* field and write the **negation** of the result into the registers specified by the *rd* field.

All of the operations above can incur at most one rounding error.

Programming Note:

The Multiply Add/Subtract instructions use the SPARC-V9 IMPDEP2 opcode, and they are specific to the SPARC64-III implementation. They **cannot** be used in any programs that will be exe-

cuted on any other SPARC-V9 processor, unless that implementation exactly matches the SPARC64-III for the IMPDEP2 opcode.

Traps:

fp_disabled
fp_exception_ieee_754 (NV,NX,OF,UF)
illegal_instruction (size = 00_2 or 11_2) (*fp_disabled* is not checked for these encodings)

A.24 Jump and Link

Opcode	op3	Operation
JMPL	11 1000	Jump and Link

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
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* field = 0, or "*r*[*rs1*] + sign_ext(*simm13*)" if *i* = 1.

The JMPL instruction copies the PC, which contains the address of the JMPL instruction, into register *r*[*rd*]. All 64 bits of the PC are stored into *r[rd]* regardless of the state of PSTATE.AM. The value written into *r*[*rd*] is visible to the instruction in the delay slot.

If either of the low-order two bits of the jump address is nonzero, a *mem_address_not_aligned* exception occurs.

If the JMPL instruction has *r*[*rd*] = 15, SPARC64-III stores PC + 8 in a hardware table. When a *ret* (*jmpl %i7+8, %g0*) or *retl* (*jmpl %o7+8, %g0*) is executed, the value in the table is used to predict the return address. See 6.3.4.3, "CALL and JMPL Instructions", for details of how this hardware table works.

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

The LDFSR 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. It is recommended that the LDXFSR instruction be used in its place.

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
LDFSR^D	10 0001	0	Load Floating-point State Register Lower
LDXFSR	10 0001	1	Load Floating-point State Register
—	10 0001	2..31	*Reserved*

† Encoded floating-point register value, as described in 5.1.4.1

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
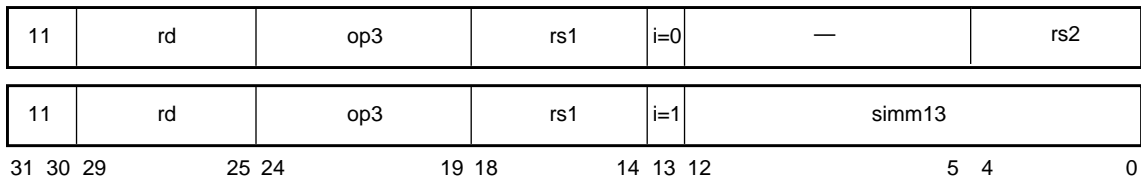
Assembly Language Syntax	
ld	[*address*], $freg_{rd}$
ldd	[*address*], $freg_{rd}$
ldq	[*address*], $freg_{rd}$
ld	[*address*], %fsr
ldx	[*address*], %fsr

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) copies a word-aligned quadword from memory into a quad-precision floating-point register.

Note:

SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; an attempt to execute this instruction causes an illegal instruction exception. Supervisor software then emulates these instructions.

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 lower 32 bits of the FSR. The upper 32 bits of FSR are unaffected by LDFSR.

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.

Compatibility Note:
> SPARC-V9 supports two different instructions to load the FSR; the SPARC-V8 LDFSR instruction is defined to load only the lower 32 bits into the FSR, whereas LDXFSR allows SPARC-V9 programs to load all 64 bits of 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 "$r[rs1] + \text{sign\_ext}(simm13)$" if $i = 1$.

LDF and LDFSR cause a *mem_address_not_aligned* exception if the effective memory address is not word-aligned; LDDF causes an *LDDF_mem_address_not_aligned* exception if the effective memory address is not doubleword-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, a load floating-point instruction causes an *fp_disabled* exception. **Note:** This check is not made for LDQF.

Programming Note:
> In SPARC-V8, some compilers issued sequences of single-precision loads when they could not determine that double- or quadword operands were properly aligned. For SPARC-V9, since emulation of misaligned loads is expected to be fast, it is recommended that compilers issue sets of single-precision loads only when they can determine that double- or quadword operands are **not** properly aligned.

Implementation Note:
> If a load floating-point instruction traps with any type of access error, the contents of the destination floating-point register(s) remain unchanged.

Exceptions:
> *illegal_instruction* ($op3 = 21_{16}$ and $rd = 2..31$ or LDQF)
> *fp_disabled*
> *LDDF_mem_address_not_aligned* (LDDF only)
> *mem_address_not_aligned*
> *data_access_exception*
> *data_access_error*
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.26 Load Floating-point from Alternate Space

Opcode	op3	rd	Operation
LDFA^{PASI}	11 0000	0..31	Load Floating-Point Register from Alternate space
LDDFA^{PASI}	11 0011	†	Load Double Floating-Point Register from Alternate space
LDQFA^{PASI}	11 0010	†	Load QuadFloating-Point Register from Alternate space

† Encoded floating-point register value, as described in 5.1.4.1

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	
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 doubleword 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) copies a word-aligned quadword from memory into a quad-precision floating-point register.

Implementation Note:

> SPARC64-III does not implement the LDQFA instruction in hardware; an attempt to execute this instruction causes an *illegal_instruction* exception. Supervisor software will then emulate the LDQFA.

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 "$r[rs1] + \text{sign\_ext}(simm13)$" if $i = 1$.

LDFA causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned; LDDFA causes an *LDDF_mem_address_not_aligned* exception if the effective memory 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, load floating-point from alternate space instructions cause an *fp_disabled* exception.

Implementation Note:
> This check is not made for LDQFA. LDFA, and LDDFA 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 sequences of single-precision loads when they could not determine that double- or quadword operands were properly aligned. For SPARC-V9, since emulation of misaligned loads is expected to be fast, it is recommended that compilers issue sets of single-precision loads only when they can determine that double- or quadword operands are **not** properly aligned.

Implementation Note:
> If a load floating-point instruction traps with any type of access error, the destination floating-point register(s) remain unchanged.

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*
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.27 Load Integer

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. It is recommended that the LDX instruction be used in its place.

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
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
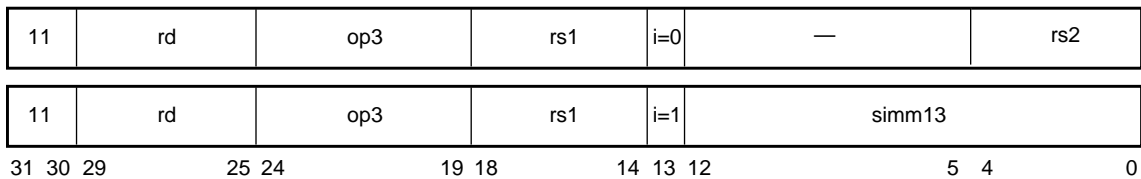
Assembly Language Syntax	
ldsb	[*address*], *reg$_{rd}$*
ldsh	[*address*], *reg$_{rd}$*
ldsw	[*address*], *reg$_{rd}$*
ldub	[*address*], *reg$_{rd}$*
lduh	[*address*], *reg$_{rd}$*
lduw	[*address*], *reg$_{rd}$* (*synonym:* ld)
ldx	[*address*], *reg$_{rd}$*
ldd	[*address*], *reg$_{rd}$*

Description:

The load integer instructions copy a byte, a halfword, a word, an extended word, or a doubleword from memory. All except LDD 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 doubleword integer instructions (LDD) copy 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. **Note**: 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.

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 and load doubleword) instruction operates atomically.

LDUH and LDSH cause a *mem_address_not_aligned* exception if the address is not half-word-aligned. LDUW and LDSW cause a *mem_address_not_aligned* exception if the effective address is not word-aligned. LDX and LDD cause a *mem_address_not_aligned* exception if the address is not doubleword-aligned.

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.

Compatibility Note:
> The SPARC-V8 LD instruction has been renamed LDUW in SPARC-V9. The LDSW instruction is new in SPARC-V9.

Exceptions:
> *illegal_instruction* (LDD with odd *rd*)
> *mem_address_not_aligned* (all except LDSB, LDUB)
> *data_access_exception*
> *data_access_error*
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.28 Load Integer from Alternate Space

The LDDA 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. It is recommended that the LDXA instruction be used in its place.

Opcode	op3	Operation
LDSBA^{PASI}	01 1001	Load Signed Byte from Alternate space
LDSHA^{PASI}	01 1010	Load Signed Halfword from Alternate space
LDSWA^{PASI}	01 1000	Load Signed Word from Alternate space
LDUBA^{PASI}	01 0001	Load Unsigned Byte from Alternate space
LDUHA^{PASI}	01 0010	Load Unsigned Halfword from Alternate space
LDUWA^{PASI}	01 0000	Load Unsigned Word from Alternate space
LDXA^{PASI}	01 1011	Load Extended Word from Alternate space
LDDA^{D, PASI}	01 0011	Load Doubleword from Alternate space

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
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 $_{rd}$*
ldda   [*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 $_{rd}$*
ldda   [*reg_plus_imm*] %asi, *reg $_{rd}$*

Description:

The load integer from alternate space instructions copy a byte, a halfword, a word, an extended word, or a doubleword from memory. All except LDDA 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 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. **Note**: 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.

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 seven 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 and load doubleword) 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 and LDDA cause 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.

LDDA with ASI=24_{16} or $2C_{16}$ has a special feature. See A.28.1, "Atomic Quad Load" 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.

Compatibility Note:
>
> The SPARC-V8 instruction LDA has been renamed LDUWA in SPARC-V9. The LDSWA instruction is new in SPARC-V9.

Exceptions:
>
> *privileged_action*
> *illegal_instruction* (LDDA with odd rd)
> *mem_address_not_aligned* (all except LDSBA and LDUBA)
> *data_access_exception*
> *data_access_error*

A.28.1   Atomic Quad Load

opcode	*ASI*	*operation*
LDDA	0x24	Cacheable, 128-bit atomic load
LDDA	0x2c	Cacheable, 128-bit atomic load, little endian

Description:

SPARC64-III treats an LDDA with an ASI value of 0x24 or 0x2c differently from other LDDA's. An LDDA made with an ASI of 0x24 or 0x2c will do a cacheable, 128-bit, atomic load from memory and store the result in an even/odd 64-bit integer register pair.

ASI 0x24 causes the 128-bit access to be made in big endian mode using the nucleus context. The doubleword at the effective memory address is copied into the even r register in big endian mode. The doubleword at the effective memory address + 8 is copied into the following odd-numbered r register in big endian mode.

ASI 0x2c causes the 128-bit access to be made in little endian mode using the nucleus context. The doubleword at the effective memory address is copied into the even r register in little endian mode. The doubleword at the effective memory address + 8 is copied into the following odd-numbered r register in little endian mode.

These privileged ASIs will cause a *privileged_action* trap if executed when PSTATE.PRIV = 0.

Accesses to non cacheable locations will cause a *data_access_exception* trap with ftype=E_{16} (illegal access to Noncacheable Page). Accesses to addresses that are not 128-bit aligned will cause a *mem_address_not_aligned* trap.

Programming Note:

If ASI's 0x24 or 0x2c are used with any instruction other than an LDDA, a *data_access_exception* trap with ftype=F_{16} (invalid ASI) will be taken.

The quadword litlle endian address convention in LDDA with ASI 0x2c differs from the one which SPARC V9 (page 70) specifies.

Exceptions:

privileged_action
mem_address_not_aligned
data_access_exception
data_access_error
32i_data_access_MMU_miss
32i_data_access_protection
data_access_exception (128-bit atomic loads only: non cacheable access)
mem_address_not_aligned (128-bit atomic loads only: address not 128 bit aligned.)

A.29 Load-store Unsigned Byte

Opcode	op3	Operation
LDSTUB	00 1101	Load-Store Unsigned Byte

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
ldstub   [*address*], *reg*$_{rd}$

Description:

The load-store unsigned byte instruction copies a byte from memory into *r*[*rd*], and 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 are implementation-dependent (impl. dep #120).

Exceptions:
data_access_exception
data_access_error
32i_data_access_MMU_miss
32i_data_access_protection

A.30 Load-store Unsigned Byte to Alternate Space

Opcode	op3	Operation
LDSTUBA^{PASI}	01 1101	Load-Store Unsigned Byte into Alternate space

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
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 "$r[rs1] + \text{sign\_ext}(simm13)$" if $i = 1$.

LDSTUBA causes a *privileged_action* exception if PSTATE.PRIV = 0 and bit 7 of the ASI is zero.

For information about the coherence and atomicity of memory operations between processors and I/O DMA memory accesses, see HaL-specific documents described in the Bibliography.

Exceptions:

> *privileged_action*
> *data_access_exception*
> *data_access_error*
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.31 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):

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
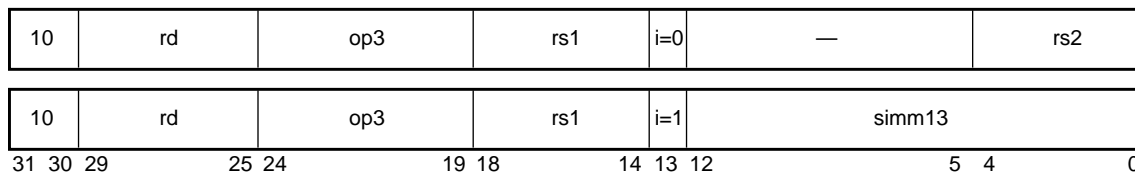
Assembly Language Syntax
and reg_{rs1}, reg_or_imm, reg_{rd}
andcc reg_{rs1}, reg_or_imm, reg_{rd}
andn reg_{rs1}, reg_or_imm, reg_{rd}
andncc reg_{rs1}, reg_or_imm, reg_{rd}
or reg_{rs1}, reg_or_imm, reg_{rd}
orcc reg_{rs1}, reg_or_imm, reg_{rd}
orn reg_{rs1}, reg_or_imm, reg_{rd}
orncc reg_{rs1}, reg_or_imm, 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 *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 1 if bits 31:0 of the result are zero (otherwise to 0), and *xcc.z* to 1 if all 64 bits of the result are zero (otherwise to 0).

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.32 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 bitwise OR 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 below. 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 56 specifies the order constraint that each bit of *mmask* (selected when set to 1) imposes on memory refer-

ences appearing before and after the MEMBAR. From zero to four mask bits may be selected in the *mmask* field

Table 56: MEMBAR *mmask* Encodings (V9=25)

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, illustrated in Table 57, 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 as well as partial order constraints are applied.

Table 57: MEMBAR *cmask* Encodings (V9=26)

Mask Bit	Function	Name	Description
cmask<2>	Synchronization barrier	#Sync	All operations (including nonmemory reference operations) appearing prior to the MEMBAR must have been performed and the effects of any exceptions become 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.

For information on the use of MEMBAR, see 8.4.3, "MEMBAR Instruction" in *V9*, and Appendix J, "Programming With the Memory Models" in *V9*. Chapter 8, "Memory Models" contains additional information about the memory models themselves.

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 are implementation-dependent.

Note:

  MEMBAR#Lookaside, MEMBAR#StoreStore, and MEMBAR#LoadStore are treated as NOPs in SPARC64-III since the hardware memory models always enforce the semantics of these MEMBAR's for all memory accesses. MEMBAR#StoreLoad and MEMBAR#LoadLoad enforce the ordering specified by the instruction. MEMBAR#Sync and MEMBAR#MemIssue cause the pro-

cessor to sync and cause the effects of all cacheable and noncacheable memory accesses made before the MEMBAR to be visible from the other processors in the system.

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.

Exceptions:

(none)

A.33   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 (also see):

Table 58: **Floating-point Move on Condition *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):

Table 59: **Floating-point Move on Condition *opf* Field**

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.

Note: SPARC64-III does not implement in hardware any instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt = unimplemented_FPop*). Supervisor software then emulates the FMOVQcc.

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

```
double A, B, X;
if (A > B) then X  = 1.03; else X  = 0.0;
```

> can be coded as

```
! assume A is in %f0; B is in %f2; %xx points to constant area
    ldd      [%xx+C_1.03],%f4  ! X  = 1.03
    fcmpd    %fcc3,%f0,%f2     ! A > B
    fble ,a  %fcc3,label
    ! following only executed if the branch is taken
    fsubd    %f4,%f4,%f4       ! X  = 0.0
label:...
```

> This takes four instructions including a branch.

> Using 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 also takes four instructions but requires no branches and may boost performance significantly. It is suggested that MOVcc and FMOVcc be used instead of branches wherever they would improve performance.

Exceptions:
> *fp_disabled*
> *fp_exception_other* (*ftt = unimplemented_FPop* (*opf_cc* $= 101_2$ or 111_2 and quad forms))

A.34 Move F-P 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	$r[rs1] \leq 0$
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	$r[rs1] \neq 0$
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	$r[rs1] \geq 0$

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:

Table 60: **Floating-point Move on Integer Register Condition *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.

Note: SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; it traps them with *fp_exception_other* (with *ftt = unimplemented_FPop*). Supervisor software then emulates the FMOVQrcond.

Implementation Note:
   If this instruction is implemented by tagging each register value with an N (negative) and a Z (zero) bit, use the following table to determine whether *rcond* is TRUE:

Branch	Test
FMOVRNZ	**not** Z
FMOVRZ	Z
FMOVGEZ	**not** N
FMOVRLZ	N
FMOVRLEZ	N **or** Z
FMOVRGZ	N **nor** Z

Exceptions:
   fp_disabled
   fp_exception_other (*unimplemented_FPop* (*rcond* $= 000_2$ or 100_2 and quad forms))

A.35 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
```

Table 61: **Move Integer Register on Condition** *ccn* **Encodings**

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	%fcc*n*, *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:

> Branches cause many implementations' performance to degrade significantly. Frequently, the MOVcc and FMOVcc instructions can be used to avoid branches. For example, the C language if-then-else statement
>
> ```
>     if (A > B) then X = 1; else X = 0;
> ```
>
> can be coded as
>
> ```
>     cmp       %i0,%i2
>     bg,a      %xcc,label
>     or        %g0,1,%i3        ! X  = 1
>     or        %g0,0,%i3        ! X  = 0
> ```

```
label:...
```

This takes four instructions including a branch. Using 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 takes only three instructions and no branches and may boost performance significantly. It is suggested that MOVcc and FMOVcc be used instead of branches wherever they would increase performance.

Exceptions:

illegal_instruction ($cc2 \square cc1 \square cc0 = 101_2$ or 111_2)

fp_disabled ($cc2 \square cc1 \square cc0 = 000_2$, 001_2, 010_2, or 011_2 and the FPU is disabled)

A.36 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	$r[rs1] \leq 0$
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	$r[rs1] \geq 0$

Format (3):

10	rd	op3	rs1	i=0	rcond	—	rs2

10	rd	op3	rs1	i=1	rcond	simm10

31 30 29   25 24   19 18   14 13 12   10 9   5 4   0

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]* satisfies 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]* does 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:

If this instruction is implemented by tagging each register value with an N (negative) and a Z (zero) bit, use the table below to determine if *rcond* is TRUE:

Move	Test
MOVRNZ	**not** Z
MOVRZ	Z
MOVRGEZ	**not** N
MOVRLZ	N
MOVRLEZ	N **or** Z
MOVRGZ	N **nor** Z

Exceptions:

illegal_instruction ($rcond = 000_2$ or 100_2)

A.37 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):

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	
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 "$r[rs1] \times r[rs2]$" if $i = 0$ or "$r[rs1] \times$ 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 "$r[rs1] \div r[rs2]$" if $i = 0$ or "$r[rs1] \div$ 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:

$$8000\ 0000\ 0000\ 0000_{16} \div \text{FFFF FFFF FFFF FFFF}_{16} = 8000\ 0000\ 0000\ 0000_{16}.$$

These instructions do not modify any condition codes.

Exceptions:
    division_by_zero

A.38 Multiply (32-bit)

> The UMUL, UMULcc, SMUL, and SMULcc 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. It is recommended that the MULX instruction be used in their place.

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):

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	
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 (UMUL, UMULcc) operates on unsigned integer word operands and computes an unsigned integer doubleword product. Signed multiply (SMUL, SMULcc) operates on signed integer word operands and computes 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 62. **Note:** 32-bit negative (*icc*.N) and zero (*icc*.Z) condition codes are set according to the **less** significant word of the product, and not according to the full 64-bit result.

Table 62: 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	Zero
icc.C	Zero
xcc.N	Set if product[63] = 1
xcc.Z	Set if product[63:0] = 0
xcc.V	Zero
xcc.C	Zero

Programming Note:

32-bit overflow after UMUL / UMULcc is indicated by $Y \neq 0$.

32-bit overflow after SMUL / SMULcc is indicated by $Y \neq (r[rd] >> 31)$, where ">>" indicates 32-bit arithmetic right shift.

Implementation Notes:

An implementation may assume that the smaller operand typically will be *r[rs2]* or *simm13*.

These instructions are executed in hardware in the SPARC64-III; however, they sync the CPU before executing. It is recommended that software use 64-bit multiplies if possible.

Exceptions:

(none)

A.39 Multiply Step

> The MULScc 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. It is recommended that the MULX instruction be used in its place.

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 lower 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 the Y = 0, then 0 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.

Note: These instructions sync the CPU before executing; therefore, it is recommend that software use 64-bit multiplies if possible.

Exceptions:
      (none)

A.40 No Operation

Opcode	op	op2	Operation
NOP	00	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
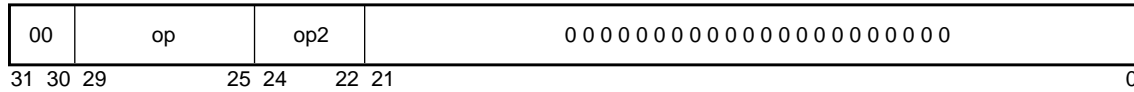
Assembly Language Syntax
`nop`

Description:

The NOP instruction changes no program-visible state (except the PC and nPC).

Note: NOP is a special case of the SETHI instruction, with $imm22 = 0$ and $rd = 0$.

Exceptions:
(none)

A.41 Population Count

Opcode	op3	Operation
POPC	10 1110	Population Count

Format (3):

10	rd	op3	0 0000	i=0	—	rs2

10	rd	op3	0 0000	i=1	simm13

```
31 30 29      25 24      19 18      14 13 12              5 4      0
```

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:** SPARC64-III does not implement this instruction; instead it generates an *illegal_instruction* exception. The instruction is emulated in supervisor software.

Implementation Note:
> Instruction bits 18 through 14 must be zero for POPC. Other encodings of this field (*rs1*) may be used in future versions of the SPARC architecture for other instructions.

Programming Note:
> POPC can be used to "find first bit set" in a register. A C program illustrating how POPC can be used for this purpose follows:

```
int ffs(zz)   /* finds first 1 bit, counting from the LSB */
unsigned zz;
{
    return popc ( zz ^ (~ (-zz)));  /* for nonzero zz */
}
```

> Inline assembly language code for ffs( ) is

```
neg       %IN, %M_IN              ! -zz(2's complement)
xnor      %IN, %M_IN, %TEMP       ! ^ ~ -zz (exclusive nor)
popc      %TEMP,%RESULT           ! result  = popc(zz ^ ~ -zz)
movrz     %IN,%g0,%RESULT         ! %RESULT should be 0 for %IN=0
```

> where IN, M_IN, TEMP, and RESULT are integer registers.

Example:

```
IN                = ...00101000! 1st 1 bit from rt is 4th bit
-IN               = ...11011000
~ -IN             = ...00100111
IN ^ ~ -IN        = ...00001111
popc(IN ^ ~ -IN)  = 4
```

Exceptions:
illegal_instruction

A.42 Prefetch Data

Opcode	op3	Operation
PREFETCH	10 1101	Prefetch Data
PREFETCHA$^{\text{PASI}}$	11 1101	Prefetch Data from Alternate Space

Format (3) PREFETCH:

11	fcn	op3	rs1	i=0	—	rs2

11	fcn	op3	rs1	i=1	simm13	

31 30 29   25 24   19 18   14 13 12   5   4   0

Format (3) PREFETCHA:

11	fcn	op3	rs1	i=0	imm_asi	rs2

11	fcn	op3	rs1	i=1	simm13	

31 30 29   25 24   19 18   14 13 12   5   4   0

Table 63: **SPARC-V9 and SPARC64-III Prefetch Functions**

fcn	SPARC-V9 Prefetch Function	SPARC64-III Prefetch Function
0	Prefetch for several reads	Prefetch for several reads
1	Prefetch for one read	Prefetch for several reads
2	Prefetch for several writes	Prefetch for several writes
3	Prefetch for one write	Prefetch for several writes
4	Prefetch page	Prefetch for several reads
5–15	*Reserved*	*illegal_instruction* trap
16–31	*Implementation-dependent*	NOP

Assembly Language Syntax	
prefetch	[*address*], *prefetch_fcn*
prefetcha	[*regaddr*] *imm_asi*, *prefetch_fcn*
prefetcha	[*reg_plus_imm*] %asi, *prefetch_fcn*

Description:

In nonprivileged code, a prefetch instruction has the same observable effect as a NOP; 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.

Implementation Note:
> Any effects of prefetch in privileged code should be reasonable (for example, handling ECC errors, no page prefetching allowed within code that handles page faults). The benefits of prefetching should be available to most privileged code.

Execution of a prefetch instruction initiates data movement (or preparation for future data movement or address mapping) to reduce the latency of subsequent loads and stores to the specified address range.

A successful prefetch initiates movement of a block of data containing the addressed byte from memory toward the processor. In SPARC64-III the block of data is one 64-byte cache line.

Programming Note:
> Software may prefetch 64 bytes beginning at an arbitrary address *address* by issuing the instructions
>
> ```
> prefetch        [address], prefetch_fcn
> prefetch        [address + 63], prefetch_fcn
> ```

Implementation Note:
> Prefetching may be used to help manage memory cache(s). A prefetch from a nonprefetchable location has no effect. It is up to memory management hardware to determine how locations are identified as not prefetchable.

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 $i = 1$.

Variants of the prefetch instruction can be used to prepare the memory system for different types of accesses.

Note: Even though SPARC64-III does not distinguish among prefetches for one or several reads or writes and does not distinguish prefetch page, programmers should choose the correct variation; future revisions of the CPU will distinguish some or all of the variants. For that reason, all of the SPARC-V9 variants are documented in the following subsections. For SPARC64-III behavior, see A.42.3, "SPARC64-III PREFETCH Behavior" for details.

A.42.1 SPARC-V9 Prefetch Variants

The prefetch variant is selected by the *fcn* field of the instruction. *fcn* values 5..15 are reserved for future extensions of the architecture.

PREFETCH *fcn* values of 16..31 are implementation-dependent; they are treated as NOPs on SPARC64-III.

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. An implementation may implement a prefetch variant by any technique, as long as the intent of the variant is achieved.

The prefetch instruction is designed to treat the common cases well. The variants are intended to provide scalability for future improvements in both hardware and compilers. If a variant is implemented, then it should have the effects described below. In case some of the variants listed below are implemented and some are not, there is a recommended over-loading of the unimplemented variants (see the Implementation Note labeled "Recommended Overloadings" in A.42.2).

A.42.1.1 Prefetch for Several Reads (*fcn* = 0)

The intent of this variant is to cause movement of data into the data cache nearest the processor, with "reasonable" efforts made to obtain the data.

Implementation Note:
> If, for example, some TLB misses are handled in hardware, then they should be handled. On the other hand, a multiple ECC error is reasonable cause for cancellation of a prefetch.

If the addressed data is already present (and owned, if necessary) in the cache, then this variant has no effect.

A.42.1.2 Prefetch for One Read (*fcn* = 1)

This variant indicates that, if possible, the data cache should be minimally disturbed by the data read from the given address, because that data is expected to be read once and not reused (read or written) soon after that.

If the data is already present in the cache, then this variant has no effect.

Programming Note:
> The intended use of this variant is in streaming large amounts of data into the processor without overwriting data in cache memory.

A.42.1.3 Prefetch for Several Writes (and Possibly Reads) (*fcn* = 2)

The intent of this variant is to cause movement of data in preparation for writing.

If the addressed data is already present in the data cache, then this variant has no effect.

Programming Note:
> An example use of this variant is to initialize a cache line in preparation for a partial write.

Implementation Note:
> On a multiprocessor, this variant indicates that exclusive ownership of the addressed data is needed, so it may have the additional effect of obtaining exclusive ownership of the addressed cache line.

A.42.1.4 Prefetch for One Write (*fcn* = 3)

This variant indicates that, if possible, the data cache should be minimally disturbed by the data written to this address, because that data is not expected to be reused (read or written) soon after it has been written once.

A.42.1.5 Prefetch Page (*fcn* = 4)

In a virtual-memory system, the intended action of this variant is for the supervisor software or hardware to initiate asynchronous mapping of the referenced virtual address, assuming that it is legal to do so.

Programming Note:
> The desire is to avoid a later page fault for the given address, or at least to shorten the latency of a page fault.

In a nonvirtual-memory system, or if the addressed page is already mapped, this variant has no effect.

The referenced page need not be mapped when the instruction completes. Loads and stores issued before the page is mapped should block just as they would if the prefetch had never been issued. When the activity associated with the mapping has completed, the loads and stores may proceed.

Implementation Note:
> An example of mapping activity is DMA from secondary storage.

Implementation Note:
> Use of this variant may be disabled or restricted in privileged code that is not permitted to cause page faults.

A.42.1.6 Implementation-dependent Prefetch (*fcn* = 16..31)

These values are available for implementations to use. An implementation shall treat any unimplemented prefetch *fcn* values as NOPs (impl. dep. #103).

A.42.2 General Comments

There is no variant of PREFETCH for instruction prefetching. Instruction prefetching should be encoded using the Branch Never (BPN) form of the BPcc instruction (see A.7, "Branch on Integer Condition Codes with Prediction (BPcc)"l.

One error to avoid in thinking about prefetch instructions is that they should have "no cost to execute." As long as the cost of executing a prefetch instruction is well less than one-third the cost of a cache miss, use of prefetching is a net win. It does not appear that prefetching causes a significant number of useless fetches from memory, though it may increase the rate of **useful** fetches (and hence the bandwidth), because it more efficiently overlaps computing with fetching.

Implementation Note:
> **Recommended Overloadings.** There are four recommended sets of overloadings for the prefetch variants, based on a simplistic classification of SPARC-V9 systems into cost (low-cost *vs.* high-cost) and processor multiplicity (uniprocessor *vs.* multiprocessor) categories. These overloadings are chosen to help ensure efficient portability of software across a range of implementations.
>
> In a uniprocessor, there is no need to support multiprocessor cache protocols; hence, Prefetch for Several Reads and Prefetch for Several Writes may behave identically. In a low-cost implementation, Prefetch for One Read and Prefetch for One Write may be identical to Prefetch for Several

Reads and Prefetch for Several Writes, respectively. The following table shows potential Prefetch overloadings.

Multiplicity	Cost	Prefetch for ..	Could Be Overloaded to Mean the Same as Prefetch for ..
Uniprocessor	Low	One read	Several reads
		Several reads	Several reads
		One write	Several writes
		Several writes	—
Uniprocessor	High	One read	—
		Several reads	Several reads
		One write	—
		Several writes	—
Multiprocessor	Low	One read	Several reads
		Several reads	—
		One write	Several writes
		Several writes	—
Multiprocessor	High	One read	—
		Several reads	—
		One write	—
		Several writes	—

Programming Note:

A SPARC-V9 compiler that generates PREFETCH instructions should generate each of the four variants where it is most appropriate. The overloadings suggested in the previous Implementation Note ensure that such code will be portable and reasonably efficient across a range of hardware configurations.

Implementation Note:

The Prefetch for One Read and Prefetch for One Write variants assume the existence of a "bypass cache," so that the bulk of the "real cache" remains undisturbed. If such a bypass cache is used, it should be large enough to properly shield the processor from memory latency. Such a cache should probably be small, highly associative, and use a FIFO replacement policy.

A.42.3 SPARC64-III PREFETCH Behavior

Prefetch types 0..4 are mapped into two cases: Prefetch for read, Prefetch for Write.

A prefetch will try to load the cache line that contains the effective address of the PREFETCH into the D1-Cache. Below is a description of what will actually happen in the CPU for prefetches:

1. The prefetch address is sent to the µDTLB for translation. If a miss occurs in the µDTLB then the address is sent to the Main TLB (MTLB) for translation. If the MTLB misses then the prefetch will be dropped.

2. If any protection violation occurs in the µDTLB or MTLB the access will be dropped.

3. If translation succeeds and if the requested cache line is already in the D1 cache, the D1 cache will complete the PREFETCH.

4. If the requested cache line is not in the D1 cache and there are no free reload buffers, the D1 cache will retry the PREFETCH until a reload buffer is available.

5. If the requested cache line is not in the D1 cache and there is a free reload buffer then the cache will send the prefetch to the unified second level cache (UC). If the UC is busy or unable to receive a cacheable command at that time, the D1 cache will retry the prefetch.

6. If the UC accepts the request and hits, the requested cache line is sent to the D1 cache.

7. If the UC cache does not have the cache line it will send the request on to the UPA bus which will supply the data from memory. When the data is returned from the UPA it will be sent to the UC and D1 caches.

If a *Prefetch for Write* hits in the D1 cache but the D1 cache doesn't have ownership of the line, the UC may give the ownership to the D1 cache instead of sending the data.

Prefetches will work if the ASI is ASI_PRIMARY, ASI_SECONDARY, or ASI_NUCLEUS.

Exceptions:
illegal_instruction $(fcn = 5 .. 15)$
data_access_error

A.43 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

Note: SPARC64-III does not need or have a floating-point deferred trap queue (FQ). An attempt to read from the FQ causes an *illegal_instruction* exception.

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 a RDPR instruction with *rs1* in that range causes an *illegal_instruction* exception.

Programming Note:
> On an 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. On an implementation with deferred floating-point traps, the address of the trapping instruction might be a value obtained from the FQ.

Exceptions:
> *privileged_opcode*
> *illegal_instruction* (($rs1 = 16..30$) or ((rs1$\leq$3) and (TL = 0)))
> *illegal_instruction* (RDPR of FQ)

A.44 Read State Register

> The RDY 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. It is recommended that all instructions which reference the Y register be avoided.

Opcode	op3	rs1	[12:8]	Operation
RDY[D]	10 1000	0	---	Read Y Register
—	10 1000	1	---	*reserved*
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*
See text	10 1000	15	---	*See text*
—	10 1000	16−17	---	*reserved*
RHDW_MODE	10 1000	18	---	Read Hardware Mode Register
RGSR	10 1000	19	---	Read Graphic Status Register
—	10 1000	20−21		*reserved*
RSCHED_INT	10 1000	22	---	Read SCHED_INT Register.
RTICK_MATCH	10 1000	23	---	Read Tick Match Register
RIFTYPE	10 1000	24	---	Read Instruction Fault Type Register
RSCRATCH	10 1000	25	0-3	Read CPU Scratch Registers
RBRKPT_ADDR	10 1000	26	0	Read Data Breakpoint Address Reg.
RBRKPT_MASK	10 1000	26	1	Read Data Breakpoint Mask Reg.
—	10 1000	27		*reserved*
RDFAULT_ADDR	10 1000	28	---	Read Data Fault Address Register
RDFTYPE	10 1000	29	---	Read Data Fault Type Register
RD_PM_VN	10 1000	30	0	Read Perf Monitor View Number
RD_PM_REG0	10 1000	30	1	Read Perf Monitor Reg. #0
RD_PM_REG1	10 1000	30	2	Read Perf Monitor Reg. #1
RD_PM_REG2	10 1000	30	3	Read Perf Monitor Reg. #2
RD_PM_REG3	10 1000	30	4	Read Perf Monitor Reg. #3
RD_PM_REG4	10 1000	30	5	Read Perf Monitor Reg. #4
RD_PM_REG5	10 1000	30	6	Read Perf Monitor Reg. #5
RDSCR	10 1000	31	---	Read State Control Register

RDPM (ASR30) please refer to Appendix Q, "Performance Monitoring"

Format (3):

10	rd	op3	rs1	i=0	—
31 30 29	25 24	19 18	14 13 12		0
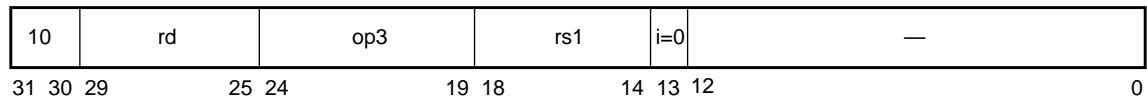
Format 3 (rd %asr25 (SCRATCH) only):

10	rd	101000	rs1=11001	i=0	register #	[7:0]=00000000
31	29	24	18	13	12	7
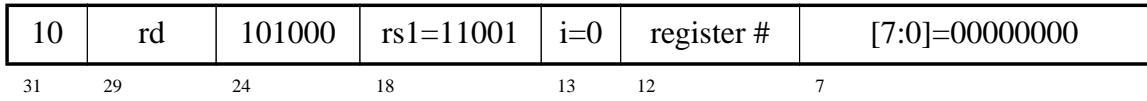
Format 3 (rd %asr26 only (Data Breakpoint Registers)):

10	rd	101000	rs1=11010	i=0	register #	[7:0]=00000000
31	29	24	18	13	12	7
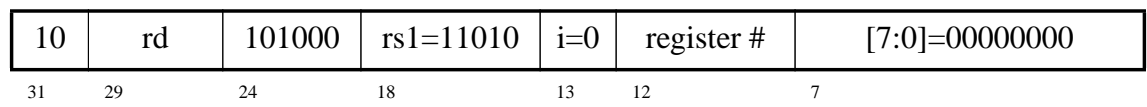
Format 3 (rd %asr30 only (Performance Monitors)):

10	rd	101000	rs1=11110	i=0	pm reg. #	[7:0]=00000000
31	29	24	18	13	12	7
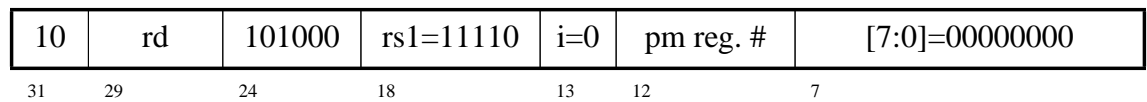
Assembly Language Syntax
rd   %y, reg_{rd}
rd   %ccr, reg_{rd}
rd   %asi, reg_{rd}
rd   %tick, reg_{rd}
rd   %pc, reg_{rd}
rd   %fprs, reg_{rd}
rd   %hardware_mode, reg_{rd}
rd   %graphic_status, reg_{rd}
rd   %sched_int, reg_{rd}
rd   %tick_match, reg_{rd}
rd   %iftype, reg_{rd}
rd   %scratch[0-3], reg_{rd}
rd   %dbreak_addr, reg_{rd}
rd   %dbreak_mask, reg_{rd}
rd   %dfaddr, reg_{rd}
rd   %dftype, reg_{rd}
rd   %pm[0-6], reg_{rd}
rd   %scr, reg_{rd}

Description:

These instructions read the specified state register into *r*[*rd*].

Note: RDY, RDCCR, RDASI, RDPC, RDTICK, RDFPRS, RDASR, RHDW_MODE, RGSR, RSCHED_INT, RTICK_MATCH, RIFTYPE, RSCRATCH, RBRKPT_ADDR, RBRKPT_MASK, DRFAULT_ADDR, RDFYPE, RD_PMs, and RDSCR are distinguished only by the value in the *rs1* field.

If *rs1* ≥ 7, an ancillary state register is read. The following values of *rs1* are reserved for future versions of the architecture: 7..14, and 16..17. An RDASR instruction with *rs1* = 15, *rd* = 0, and *i* = 0 is defined to be a STBAR instruction (see A.51). An RDASR instruction with *rs1* = 15, *rd* = 0, and *i* = 1 is defined to be a MEMBAR instruction (see A.32). RDASR with *rs1* = 15 and *rd* ≠ 0 is reserved for future versions of the architecture; it causes an *illegal_instruction* exception.

RDTICK causes a *privileged_action* exception if PSTATE.PRIV = 0 and TICK.NPT = 1.

For RDPC, all 64 bits of the PC value are stored in *r*[*rd*], regardless of the setting of PSTATE.AM.

RDFPRS waits for all pending FPops and loads of floating-point registers to complete before reading the FPRS register.

SPARC64-III implements these additional ASRs: RHDW_MODE, RGSR, RSCHED_INT, RTICK_MATCH, RIFTYPE, RSCRATCH, RBRKPT_ADDR, RBRKPT_MASK, DRFAULT_ADDR, RDFYPE, RD_PMs, and RDSCR. All of these registers are privileged; an attempt to read any of these registers in nonprivileged mode causes a *privileged_opcode* exception.

A *rd %pm[0-6]*, *rd %dbreak_addr*, *rd %dbreak_mask*, or *rd %scratch[0-3]* which uses a reserved or undocumented value for bits [12:8] or does not contain all zeroes in bits [7:0] will cause an *illegal_ instruction* trap.

⟨V9⟩ See I.1.1, "Read/Write Ancillary State Registers (ASRs)" in *V9* for a discussion of extending the SPARC-V9 instruction set using read/write ASR instructions.

Implementation Note:
> Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers. See *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, a document available from SPARC International, for information on implemented ancillary state 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* (RDASR with *rs1* = 15, 18, 22..26, 28..30)
> *illegal_instruction* (RDASR with *rs1* = 1 or 7..14; RDASR with *rs1* = 15 and *rd* ≠ 0; RDASR with *rs1* = 16..17, 20..21, 27)
> *privileged_action* (RDTICK only)

A.45 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
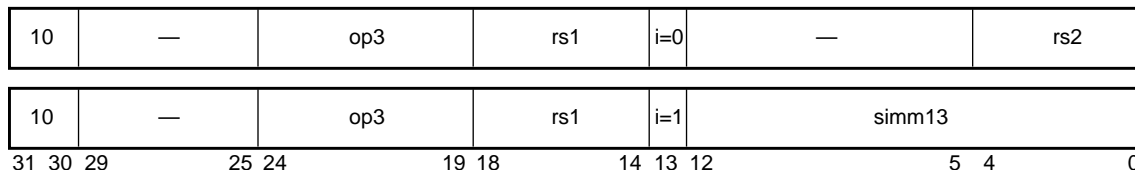
Assembly Language Syntax
return   *address*

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 are nonzero.

Programming Note:

To reexecute the trapped instruction when returning from a user trap handler, use the RETURN instruction in the delay slot of a JMPL instruction, for example:

```
jmpl     %l6,%g0   ! Trapped PC supplied to user trap handler
return   %l7       ! Trapped nPC supplied to user trap handler
```

Programming Note:

A routine that uses a register window may be structured either as

```
save     %sp,-framesize, %sp
. . .
ret                ! Same as jmpl %i7 + 8, %g0
restore            ! Something useful like "restore %o2,%l2,%o0"
```

or as

```
save     %sp,-framesize, %sp
. . .
return   %i7 + 8
nop                ! Could do some useful work in the caller's
                   ! window e.g. "or %o1, %o2,%o0"
```

Exceptions:

mem_address_not_aligned
fill_n_normal ($n = 0..7$)
fill_n_other ($n = 0..7$)

A.46 SAVE and RESTORE

Opcode	op3	Operation
SAVE	11 1100	Save caller's window
RESTORE	11 1101	Restore caller's window

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	
save	reg_{rs1}, reg_or_imm, reg_{rd}
restore	reg_{rs1}, reg_or_imm, reg_{rd}

Description (Effect on Nonprivileged 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 $r[rs1]$ and/or $r[rs2]$ are read from the **old** window (that is, the window addressed by the original CWP) and the sum is written into $r[rd]$ of the **new** window (that is, the window addressed by the new CWP).

Note: CWP arithmetic is performed modulo the number of implemented windows, NWINDOWS.

Programming Note:
> Typically, if a SAVE (RESTORE) instruction traps, the spill (fill) trap handler returns to the trapped instruction to reexecute it. So, although the ADD operation is not performed the first time (when the instruction traps), it is performed the second time the instruction executes. The same applies to changing the CWP.

Programming Note:
> The SAVE instruction can be used to atomically allocate a new window in the register file and a new software stack frame in memory. See H.1.2, "Leaf-Procedure Optimization" in *V9* for details.

Programming Note:
> There is a performance trade-off to consider between using SAVE/RESTORE and saving and restoring selected registers explicitly.

Description (effect on privileged state):

If the SAVE instruction does not trap, it increments the CWP (**mod** NWINDOWS) to provide a new register window and updates the state of the register windows by decrementing CANSAVE and incrementing CANRESTORE.

If the new register window is occupied (that is, CANSAVE = 0), a spill trap is generated. The trap vector for the spill trap is based on the value of OTHERWIN and WSTATE. The spill trap handler is invoked with the CWP set to point to the window to be spilled (that is, old CWP + 2).

If CANSAVE ≠ 0, the SAVE instruction checks whether the new window needs to be cleaned. It causes a *clean_window* trap if the number of unused clean windows is zero, that is, (CLEANWIN – CANRESTORE) = 0. The *clean_window* trap handler is invoked with the CWP set to point to the window to be cleaned (that is, old CWP + 1).

If the RESTORE instruction does not trap, it decrements the CWP (**mod** NWINDOWS) to restore the register window that was in use prior to the last SAVE instruction executed by the current process. It also updates the state of the register windows by decrementing CANRESTORE and incrementing CANSAVE.

If the register window to be restored has been spilled (CANRESTORE = 0), a fill trap is generated. The trap vector for the fill trap is based on the values of OTHERWIN and WSTATE, as described in 7.5.2.1, "Trap Type for Spill/Fill Traps". 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. For details, see the unnumbered subsection titled "Splitting the Register Windows" in H.2.3, "Client-Server Model" in *V9*.

Programming Note:
> 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.47 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):

10	fcn	op3	—

| 31 30 29 | 25 24 | 19 18 | 0 |
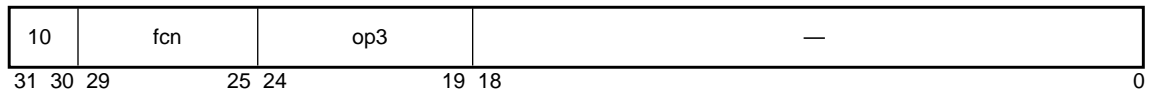
Assembly Language Syntax
`saved`
`restored`

Description:

SAVED and RESTORED adjust the state of the register-windows control registers.

SAVED increments CANSAVE. If OTHERWIN = 0, it decrements CANRESTORE. If OTHERWIN≠0, it decrements OTHERWIN.

RESTORED increments CANRESTORE. If CLEANWIN < (NWINDOWS−1), RESTORED increments CLEANWIN. If OTHERWIN = 0, it decrements CANSAVE. If OTHERWIN ≠ 0, it decrements OTHERWIN.

Programming Note:

⚠️*V9*
  The spill (fill) handlers use the SAVED (RESTORED) instruction to indicate that a window has been spilled (filled) successfully. See H.2.2, "Example Code for Spill Handler" in *V9* for details.

Programming Note:

  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 does not cause a trap.

Programming Note:

  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.48 SETHI

Opcode	op	op2	Operation
SETHI	00	100	Set High 22 Bits of Low Word

Format (2):

00	rd	100	imm22

31 30 29   25 24   22 21   0
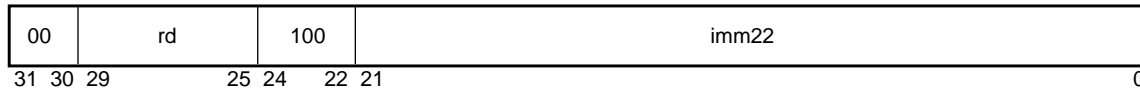
Assembly Language Syntax
sethi $const22$, reg_{rd}
sethi   %hi ($value$), reg_{rd}

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.

A SETHI instruction with $rd = 0$ and $imm22 = 0$ is defined to be a NOP instruction, which is defined in A.40.

Programming Note:

The most common form of 64-bit constant generation is creating stack offsets whose magnitude is less than 2^{32}. The code below can be used to create the constant 0000 0000 ABCD 1234_{16}:

```
sethi     %hi(0xabcd1234),%o0
or        %o0, 0x234, %o0
```

The following code shows how to create a negative constant. **Note**: The immediate field of the xor instruction is sign extended and can be used to get 1s 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.49 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):

10	rd	op3	rs1	i=0	x	—	rs2

10	rd	op3	rs1	i=1	x=0	—	shcnt32

10	rd	op3	rs1	i=1	x=1	—	shcnt64

31 30 29   25 24   19 18   14 13 12   6   5   4   0

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 64 shows the shift count encodings for all values of i and x.

Table 64: 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.

Programming Note:
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:
software_initiated_reset

A.50 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:

On SPARC-V9 systems, 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.

See 7.6.2.5, "Software-initiated Reset (SIR) Traps", for more information about software-initiated resets.

When executed in user mode, the action of SIR is conditional on the SIR_enable control flag.

The location of the SIR_enable control flag and the means of accessing the SIR_enable control flag are implementation-dependent. In SPARC64-III it is permanently zero, therefore an SIR executes without effect (as a NOP) in user mode.

A privileged WRSIR instruction can be used to cause a software initiated reset (SIR) on SPARC64-III. The SIR instruction is actually a WRASR with $rd = 15$, $rs1 = 0$, and $i = 1$. See A.63, "Write State Register", for more information.

Exceptions:
        (none)

A.51 Store Barrier

The STBAR 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. It is recommended that the MEMBAR instruction be used in its place.

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:

In SPARC64-III, STBAR behaves as NOP since the hardware memory models always enforce the semantics of these MEMBARs for all memory accesses.

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.52 Store Floating-point

> The STFSR 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. It is recommended that the STXFSR instruction be used in its place.

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
STFSR^D	10 0101	0	Store Floating-point State Register Lower
STXFSR	10 0101	1	Store Floating-point State Register
—	10 0101	2..31	*Reserved*

† Encoded floating-point register value, as described in 5.1.4.1

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
st $freg_{rd}$, [*address*]
std $freg_{rd}$, [*address*]
stq $freg_{rd}$, [*address*]
st   %fsr, [*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) copies the contents of a quad floating-point register into a word-aligned quadword in memory. **Note:** SPARC64-III does not implement in hardware the instructions that specify a quad floating-point register; an attempt to execute this instruction causes an *illegal_instruction* exception. Supervisor software then emulates these instructions.

The store floating-point state register lower instruction (STFSR) waits for any currently executing FPop instructions to complete, and then it writes the lower 32 bits of the FSR into memory.

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.

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 and STXFSR zero 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 "$r[rs1] + \text{sign\_ext}(simm13)$" if $i = 1$.

STF and STFSR cause a *mem_address_not_aligned* exception if the effective memory address is not word-aligned; STDF causes an *STDF_mem_address_not_aligned* exception if the effective address is not doubleword-aligned;STXFSR causes a *mem_address_not_aligned* exception if the address is not doubleword-aligned. 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, a store floating-point instruction causes an *fp_disabled* exception.

Programming Note:
> In SPARC-V8, some compilers issued sets of single-precision stores when they could not determine that double- 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 double- or quadword operands are **not** properly aligned.

Exceptions:
> *fp_disabled*
> *mem_address_not_aligned*
> *STDF_mem_address_not_aligned* (STDF only)
> *data_access_exception*
> *data_access_error*
> *illegal_instruction* ($op3 = 25_{16}$ and $rd = 2..31$)
> *illegal_instruction* (STQF)
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.53 Store Floating-point into Alternate Space

Opcode	op3	rd	Operation
STFA^{PASI}	11 0100	0..31	Store Floating-point Register to Alternate Space
STDFA^{PASI}	11 0111	†	Store Double Floating-point Register to Alternate Space
STQFA^{PASI}	11 0110	†	Store Quad Floating-point Register to Alternate Space

† Encoded floating-point register value, as described in 5.1.4.1

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	
sta	$freg_{rd}$, [*regaddr*] *imm_asi*
sta	$freg_{rd}$, [*reg_plus_imm*] %asi
stda	$freg_{rd}$, [*regaddr*] *imm_asi*
stda	$freg_{rd}$, [*reg_plus_imm*] %asi
stqa	$freg_{rd}$, [*regaddr*] *imm_asi*
stqa	$freg_{rd}$, [*reg_plus_imm*] %asi

Description:

The store single floating-point into alternate space instruction (STFA) copies *f*[*rd*] into memory.

The store double floating-point into alternate space instruction (STDFA) copies a double-word from a double floating-point register into a word-aligned doubleword in memory.

The store quad floating-point into alternate space instruction (STQFA) copies the contents of a quad floating-point register into a word-aligned quadword in memory. **Note:** SPARC64-III does not implement in hardware the instructions that specify a quad float-ing-point register; an attempt to execute this instruction causes an *illegal_intruction* excep-tion. Supervisor software then emulates these instructions.

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 seven 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 causes a *mem_address_not_aligned* exception if the effective memory address is not word-aligned; STDFA causes an *STDF_mem_address_not_aligned* exception if the effective address is not doubleword-aligned. 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 an *fp_disabled* exception.

STFA, STDFA, and STQFA 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 double- 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 double- or quadword operands are **not** properly aligned.

Exceptions:
> *fp_disabled*
> *mem_address_not_aligned*
> *STDF_mem_address_not_aligned* (STDFA only)
> *privileged_action*
> *data_access_exception*
> *data_access_error*
> *illegal_instruction* (STQFA)
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.54 Store Integer

> The STD 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. It is recommended that the STX instruction be used in its place.

Opcode	op3	Operation
STB	00 0101	Store Byte
STH	00 0110	Store Halfword
STW	00 0100	Store Word
STX	00 1110	Store Extended Word
STD^D	00 0111	Store 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		
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*]	
std	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 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 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 and store doubleword) 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 and STD causes a *mem_address_not_aligned* exception if the effective address is not doubleword-aligned.

Programming Notes:

> STD is provided for compatibility with SPARC-V8. It may execute slowly on SPARC-V9 machines because of data path and register-access difficulties. Therefore, STD should be avoided.

> If STD is emulated in software, STX should be used in order to preserve atomicity.

Exceptions:

> *illegal_instruction* (STD with odd *rd*)
> *mem_address_not_aligned* (all except STB)
> *data_access_exception*
> *data_access_error*
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.55 Store Integer into Alternate Space

> The STDA instruction is deprecated; it is provided only for compatibility with pre-
> vious versions of the architecture. It should not be used in new SPARC-V9 soft-
> ware. It is recommended that the STXA instruction be used in its place.

Opcode	op3	Operation
STBA^{PASI}	01 0101	Store Byte into Alternate space
STHA^{PASI}	01 0110	Store Halfword into Alternate space
STWA^{PASI}	01 0100	Store Word into Alternate space
STXA^{PASI}	01 1110	Store Extended Word into Alternate space
STDA$^{D, PASI}$	01 0111	Store Doubleword into Alternate space

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
```
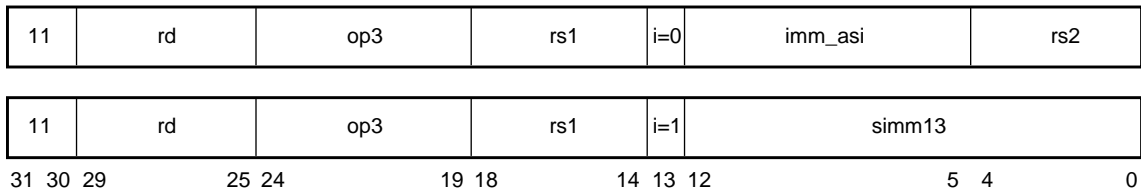
Assembly Language Syntax		
stba	reg_{rd}, [*regaddr*] *imm_asi*	(*synonyms*: stuba, stsba)
stha	reg_{rd}, [*regaddr*] *imm_asi*	(*synonyms*: stuha, stsha)
stwa	reg_{rd}, [*regaddr*] *imm_asi*	(*synonyms*: sta, stuwa, stswa)
stxa	reg_{rd}, [*regaddr*] *imm_asi*	
stda	reg_{rd}, [*regaddr*] *imm_asi*	
stba	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	
stda	reg_{rd}, [*reg_plus_imm*] %asi	

Description:

The store integer into alternate space 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 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 sig-
nificant 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 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 and store doubleword) instruction operates atomically.

STHA causes a *mem_address_not_aligned* exception if the effective address is not half-word-aligned. STWA causes a *mem_address_not_aligned* exception if the effective address is not word-aligned. STXA and STDA cause 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.

Programming Notes:
> 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, STDA should be avoided.

Compatibility Note:
> The SPARC-V8 STA instruction is renamed STWA in SPARC-V9.

Exceptions:
> *unimplemented_STD* (STDA only)
> *illegal_instruction* (STDA with odd *rd*)
> *privileged_action*
> *mem_address_not_aligned* (all except STBA)
> *data_access_exception*
> *data_access_error*
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.56 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 "$r[rs1] - \text{sign\_ext}(simm13)$" 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 "$r[rs1] - \text{sign\_ext}(simm13) - icc.c$," and write the difference into $r[rd]$.

SUBcc and SUBCcc modify the integer condition codes (CCR.*icc* and CCR.*xcc*). 32-bit overflow (CCR.*icc.v*) occurs on subtraction if bit 31 (the sign) of the operands differ and bit 31 (the sign) of the difference differs from $r[rs1]{<}31{>}$. 64-bit overflow (CCR.*xcc.v*) occurs on subtraction if bit 63 (the sign) of the operands differ 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. See the CMP synthetic instruction in Appendix G, "Assembly Language Syntax".

Programming Note:
> 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).

Compatibility Note:
> SUBC and SUBCcc were named SUBX and SUBXcc, respectively, in SPARC-V8.

Exceptions:
> (none)

A.57 Swap Register with Memory

> The SWAP 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. It is recommended that the CASA or CASXA instruction be used in its place.

Opcode	op3	Operation
SWAP^D	00 1111	SWAP register with memory

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
swap   [*address*], *reg$_{rd}$*

Description:

SWAP exchanges the lower 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 (impl. dep #120).

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*
> *32i_data_access_MMU_miss*
> *32i_data_access_protection*

A.58   Swap Register with Alternate Space Memory

> The SWAPA 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. It is recommended that the CASXA instruction be used in its place.

Opcode	op3	Operation
SWAPA$^{D, PASI}$	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 lower 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 double-word 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 (impl. dep #120).

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
32i_data_access_MMU_miss
32i_data_access_protection

A.59 Tagged Add

> The TADDccTV 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. It is recommended that TADDcc followed by BPVS be used in its place
> (with instructions to save the pre-TADDcc integer condition codes, if necessary).

Opcode	op3	Operation
TADDcc	10 0000	Tagged Add and modify cc's
TADDccTV[D]	10 0010	Tagged Add and modify cc's, or Trap on Overflow

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
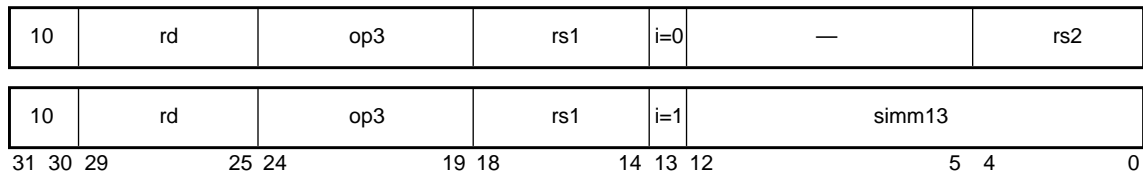
Assembly Language Syntax	
taddcc	reg_{rs1}, reg_or_imm, reg_{rd}
taddcctv	reg_{rs1}, reg_or_imm, reg_{rd}

Description:

These instructions compute 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*), and TADDccTV does so also,
if it does not trap.

A *tag_overflow* exception occurs if bit 1 or bit 0 of either operand is nonzero, or if the addi-
tion generates 32-bit arithmetic overflow (that is, both operands have the same value in bit
31, and bit 31 of the sum is different).

If TADDccTV causes a tag overflow, a *tag_overflow* exception is generated, and $r[rd]$ and
the integer condition codes remain unchanged. If a TADDccTV does not cause a tag over-
flow, the sum is written into $r[rd]$, and the integer condition codes are updated. CCR.*icc.v*
is set to 0 to indicate no 32-bit overflow. If a TADDcc causes a tag overflow, the 32-bit
overflow bit (CCR.*icc.v*) is set to 1; if it does not cause a tag overflow, CCR.*icc.v* is
cleared.

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 par-
ticular, 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.

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 (TADDccTV only)

A.60 Tagged Subtract

> The TSUBccTV 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. It is recommended that TSUBcc followed by BPVS be used in its place (with instructions to save the pre-TSUBcc integer condition codes, if necessary).

Opcode	op3	Operation
TSUBcc	10 0001	Tagged Subtract and modify cc's
TSUBccTV^D	10 0011	Tagged Subtract and modify cc's, or Trap on Overflow

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
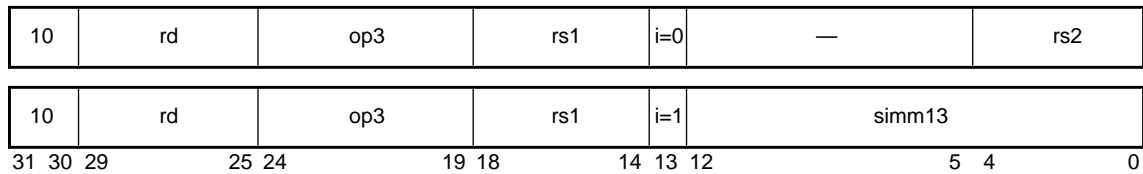
Assembly Language Syntax
tsubcc reg_{rs1}, *reg_or_imm*, reg_{rd}
tsubcctv reg_{rs1}, *reg_or_imm*, reg_{rd}

Description:

These instructions compute "$r[rs1] - r[rs2]$" if $i = 0$, or "$r[rs1] -$ sign_ext(*simm13*)" if $i = 1$.

TSUBcc modifies the integer condition codes (*icc* and *xcc*); TSUBccTV also modifies the integer condition codes, 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, a *tag_overflow* exception is generated and $r[rd]$ and the integer condition codes remain unchanged. If a TSUBccTV does not cause a tag overflow condition, the difference is written into $r[rd]$, and the integer condition codes are updated. CCR.*icc.v* is set to 0 to indicate no 32-bit overflow. If a TSUBcc causes a tag overflow, the 32-bit overflow bit (CCR.*icc.v*) is set to 1; if it does not cause a tag overflow, CCR.*icc.v* is cleared.

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* (TSUBccTV only)

A.61 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):

10	—	cond	op3	rs1	i=0	cc1	cc0	—	rs2

10	—	cond	op3	rs1	i=1	cc1	cc0	—	sw_trap_#

31 30 29 28   25 24   19 18   14 13 12 11 10   7 6 5 4   0

Table 65: Tcc Encodings for *ccn*

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 zeros by software. When $i = 0$, bits 5 through 10 are reserved, and the most significant 57 bits of "$r[rs1] + r[rs2]$" are unused, and both should be supplied as zeros by software.

Description (Effect on Privileged State):

If a *trap_instruction* traps, 256 plus the software trap number is written into TT[TL]. Then the trap is taken, and the processor performs the normal trap entry procedure, as described in Chapter 7, "Traps".

Programming Note:
> Tcc can be used to implement breakpointing, tracing, and calls to supervisor software. It can also be used for run-time 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, so it is believed that, in practice, full software compatibility will be achieved.

Note: In SPARC64-III all Tcc instructions except TA with "`%g0 + software_trap#`" addressing, serialize the CPU.

Programming Note:

> Using a TN (trap never) instruction is the preferred way to synchronize (serialize) the SPARC64-III CPU. Software should use TN to synchronize the machine. Future versions of HAL's CPU *will* synchronize the CPU when a TN is executed; however, future versions may not serialize the machine for other Tcc instructions.

Exceptions:

trap_instruction
illegal_instruction $(cc1 \parallel cc0 = 01_2 \text{ or } 11_2)$

A.62 Write Privileged Register

Opcode	op3	Operation
WRPR[P]	11 0010	Write Privileged 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
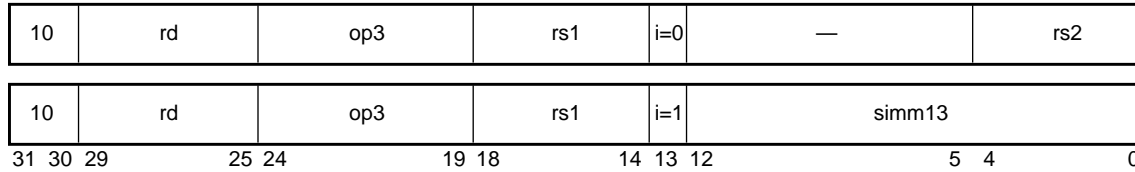
rd	Privileged Register
0	TPC
1	TNPC
2	TSTATE
3	TT
4	TICK
5	TBA
6	PSTATE
7	TL
8	PIL
9	CWP
10	CANSAVE
11	CANRESTORE
12	CLEANWIN
13	OTHERWIN
14	WSTATE
15..31	*Reserved*

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 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, care must be taken 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.

Programming Note:
> SPARC64-III does not have or need a floating-point deferred-trap queue. PSTATE.PEF can be changed from 0 to 1 at any time.

On SPARC64-III the TL register is 3 bits wide, however, the maximum value that can be stored in the TL register is 4 (to coincide with MAXTL). A write to the TL register with values 5, 6, or 7 will result in the value 4 being stored in TL.

Implementation Note:
>   Some WRPR instructions serialize the CPU or have other issue restrictions. See 6.1.3, "Serializing Instructions", and 6.1.4, "Issue Stalling Instructions", for details.

Exceptions:
>   *privileged_opcode*
>   *illegal_instruction* $((rd = 15..31)$ or $((rd \leq 3)$ and $(TL = 0)))$

A.63   **Write State Register**

> The WRY 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. It is recommended that all instructions which reference the Y register be avoided.

Opcode	op3	rd	[12:8]	Operation
WRY^D	11 0000	0	---	Write Y register
—	11 0000	1	---	*Reserved*
WRCCR	11 0000	2	---	Write Condition Codes Register
WRASI	11 0000	3	---	Write ASI register
—	11 0000	4, 5	---	*Reserved*
WRFPRS	11 0000	6	---	Write Floating-Point Registers Status register
—	11 0000	7..14	---	R*eserved*
See text	11 0000	15	---	*See text*
—	11 0000	16..17	---	*Reserved*
WR_HDW_MODE^{PASR}	11 0000	18	---	Write Hardware Mode reg.
WRGSR	11 0000	19	---	Write Graphic Status Register
SET_SCHED_INT^{PASR}	11 0000	20	---	Set bits in SCHED_INT Reg.
CLEAR_SCHED_INT^{PASR}	11 0000	21	---	Clear bits in SCHED_INT Reg.
WR_SCHED_INT^{PASR}	11 0000	22	---	Write SCHED_INT register.
WR_TICK_MATCH^{PASR}	11 0000	23	---	Write Tick Match Register
—	11 0000	24		*Reserved*
WR_SCRATCH^{PASR}	11 0000	25	0-3	Write CPU Scratch Register N
WR_BRK_ADDR^{PASR}	11 0000	26	0	Write Data Brkpt. Address Reg.
WR_BRK_MASK^{PASR}	11 0000	26	1	Write Data Breakpt. Mask Reg.
—	11 0000	27..29		*Reserved*
WR_PM_DIS^{PASR}	11 0000	30	0	Disable all performance ctrs.
WR_PM_CLR_DIS^{PASR}	11 0000	30	1	Clear and disable all perf. ctrs.
WR_PM_EN^{PASR}	11 0000	30	2	Enable all perf, ctrs.
WR_PM_CLR_EN^{PASR}	11 0000	30	3	Clear and enable all perf. ctrs.
WR_PM_VN^{PASR}	11 0000	30	4	Write PM View Number
WRSCR^{PASR}	11 0000	31	---	Write State Control 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

Format 3 (wr %asr25 (SCRATCH) only):

10	rd=11001	op3	rs1	i=0	register #	[7:5]=000	rs2

31   29   24   18   13   12   7   4

Format (3) (wr %asr26 only (Breakpoint registers)):

10	rd=11010	op3	rs1	i=0	register #	[7:5]=000	rs2

31   29   24   18   13   12   7   4

Format (3) (wr %asr30 only (Performance monitors)):

10	rd=11110	op3	rs1	i=0	operation	[7:5]=000	rs2

31   29   24   18   13   12   7   4

Assembly Language Syntax		
wr	reg_{rs1}, *reg_or_imm*,	%y
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*,	%hardware_mode
wr	reg_{rs1}, *reg_or_imm*,	%graphic_status
wr	reg_{rs1}, *reg_or_imm*,	%set_sched_int
wr	reg_{rs1}, *reg_or_imm*,	%clear_sched_int
wr	reg_{rs1}, *reg_or_imm*,	%sched_int
wr	reg_{rs1}, *reg_or_imm*,	%tick_match
wr	reg_{rs1}, reg_{rs2},	%scratch[0-3]
wr	reg_{rs1}, reg_{rs2},	%brk_addr
wr		%pm_dis
wr		%pm_clr_dis
wr		%pm_en
wr		%pm_clr_en
wr	reg_{rs1}, reg_{rs2},	%pm_vn
wr	reg_{rs1}, *reg_or_imm*,	%scr

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 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-depen-

dent (see below). A WRASR instruction is indicated by $op = 2_{16}$, $rd = 4$, 5, or ≥ 7 and $op3 = 30_{16}$.

See I.1.1, "Read/Write Ancillary State Registers (ASRs)" in *V9* for a discussion of extending the SPARC-V9 instruction set using read/write ASR instructions.

The WRY, WRCCR, WRFPRS, WRASI, WRSIR, and WRSCR instructions are **not** delayed-write instructions. The instruction immediately following a WRY, WRCCR, WRFPRS, or WRASI, WRSIR, and WRSCR observes the new value of the Y, CCR, FPRS, ASI, SIR, and or SCR register.

WRFPRS waits for any pending floating-point operations to complete before writing the FPRS register.

See section 5.2.11, "Ancillary State Registers (ASRs)" for details of the ASR registers.

The *wr &scratch[0..3], wr %brk_addr, wr %brk_mask*, and *wr pm* instructions may not be used with the immediate addressing mode, since the top 5 bits of the *simm13* field are used to select a register number or operation.

Implementation Note:
> Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers. See *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, a document available from SPARC International, for information on ancillary state registers provided by specific implementations.

Compatibility Note:
> The SPARC-V8 WRIER, WRPSR, WRWIM, and WRTBR instructions do not exist in SPARC-V9, since the IER, PSR, TBR, and WIM registers do not exist in SPARC-V9.

Implementation Note:
> Some WRASR instructions serialize the CPU or have other issue restrictions. See 6.1.3, "Serializing Instructions", for details.

Exceptions:
> *privileged_opcode* (WRASR with $rd = 18, 20..23, 25, 26, 30, 31$)
> *illegal_instruction* (WRASR with $rd = 1, 4, 5, 7..14, 16, 17, 24, 27..29$, WRASR with $rd = 15$ and $rs1 \neq 0$ or $i \neq 1$, WR_SCRATCH, WR_BRK_ADDR, WR_BRK_MASK, and WR_PM instructions if bits [13:5] do not specify a legal value for the instruction.)
> *software_initiated_reset* (WRSIR only)

B IEEE Std 754-1985 Requirements for SPARC-V9

The IEEE Std 754-1985 floating-point standard contains a number of implementation-dependencies. This appendix specifies choices for these implementation-dependencies, to ensure that SPARC-V9 implementations are as consistent as possible.

B.1 Traps Inhibit Results

As described in 5.1.7, "Floating-point State Register (FSR)", and elsewhere, when a floating-point trap occurs:

■ The destination floating-point register(s) (the *f* registers) are unchanged.

■ The floating-point condition codes (*fcc0*, *fcc1*, *fc*c2, and *fcc3*) are unchanged.

■ The FSR.*aexc* (accrued exceptions) field is unchanged.

■ The FSR.*cexc* (current exceptions) field is unchanged except for *IEEE_754_exception*s; in that case, *cexc* contains a bit set to "1" corresponding to the exception that caused the trap. Only one bit shall be set in *cexc*.

Instructions causing an *fp_exception_other* trap due to unfinished or unimplemented FPops execute as if by hardware; that is, a trap is undetectable by user software, except that timing may be affected. A user-mode trap handler invoked for an *IEEE_754_exception*, whether as a direct result of a hardware *fp_exception_ieee_754* trap or as an indirect result of supervisor handling of an *unfinished_FPop* or *unimplemented_FPop*, can rely on the following:

■ The address of the instruction that caused the exception will be available.

■ The destination floating-point register(s) are unchanged from their state prior to that instruction's execution.

■ The floating-point condition codes (*fcc0*, *fcc1*, *fc*c2, and *fcc3*) are unchanged.

■ The FSR *aexc* field is unchanged.

■ The FSR *cexc* field contains exactly one bit set to 1, corresponding to the exception that caused the trap.

■ The FSR *ftt*, *qne*, and *reserved* fields are zero.

The SPARC64-III hardware in conjunction with kernel fixup or emulation code produces the results required in this section.

B.2   NaN Operand and Result Definitions

An untrapped floating-point result can be in a format that is either the same as, or different from, the format of the source operands. These two cases are described separately below.

B.2.1   Untrapped Result in Different Format from Operands

F[sdq]TO[sdq] with a quiet NaN operand:
> No exception caused; result is a quiet NaN. The operand is transformed as follows:
>
> **NaN transformation**: The most significant bits of the operand fraction are copied to the most significant bits of the result fraction. When converting to a narrower format, excess low-order bits of the operand fraction are discarded. When converting to a wider format, excess low-order bits of the result fraction are set to 0. The quiet bit (the most significant bit of the result fraction) is always set to 1, so the NaN transformation always produces a quiet NaN. The sign bit is copied from the operand to the result without modification.

F[sdq]TO[sdq] with a signaling NaN operand:
> Invalid exception; result is the signaling NaN operand processed by the **NaN transformation** above to produce a quiet NaN.

FCMPE[sdq] with any NaN operand:
> Invalid exception; the selected floating-point condition code is set to unordered.

FCMP[sdq] with any signaling NaN operand:
> Invalid exception; the selected floating-point condition code is set to unordered.

FCMP[sdq] with any quiet NaN operand but no signaling NaN operand:
> No exception; the selected floating-point condition code is set to unordered.

B.2.2   Untrapped Result in Same Format as Operands

No NaN operand:
> For an invalid operation such as **sqrt**(–1.0) or 0.0 ÷ 0.0, the result is the quiet NaN with sign = zero, exponent = all ones, and fraction = all ones. The sign is zero to distinguish such results from storage initialized to all ones.

One operand, a quiet NaN:
> No exception; result is the quiet NaN operand.

One operand, a signaling NaN:
> Invalid exception; result is the signaling NaN with its quiet bit (most significant bit of fraction field) set to 1.

Two operands, both quiet NaNs:
> No exception; result is the *rs2* (second source) operand.

Two operands, both signaling NaNs:

Invalid exception; result is the *rs2* operand with the quiet bit set to 1.

Two operands, only one a signaling NaN:

Invalid exception; result is the signaling NaN operand with the quiet bit set to 1.

Two operands, neither a signaling NaN, only one a quiet NaN:

No exception; result is the quiet NaN operand.

In Table 66 NaN*n* means that the NaN is in rs*n*, Q means quiet, S signaling.

Table 66: Untrapped Floating-point Results (V9=27)

		rs2 Operand		
		Number	**QNaN2**	**SNaN2**
rs1 Operand	**None**	IEEE 754	QNaN2	QSNaN2
	Number	IEEE 754	QNaN2	QSNaN2
	QNaN1	QNaN1	QNaN2	QSNaN2
	SNaN1	QSNaN1	QSNaN1	QSNaN2

QSNaN*n* means a quiet NaN produced by the **NaN transformation** on a signaling NaN from rs*n*; the invalid exception is always indicated. The QNaN*n* results in the table never generate an exception, but IEEE 754 specifies several cases of invalid exceptions, and QNaN results from operands that are both numbers.

B.3 Trapped Underflow Definition (UFM = 1)

Underflow occurs if the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format.

In the SPARC64-III CPU, tininess is always detected before rounding. See 5.1.7.6, "FSR_floating-point_trap_type (ftt)", for details on how the divider handles trapped underflows.

Note:

The wrapped exponent results intended to be delivered on trapped underflows and overflows in IEEE 754 are irrelevant to SPARC-V9 at the hardware and supervisor software levels; if they are created at all, it would be by user software in a user-mode trap handler.

B.4 Untrapped Underflow Definition (UFM = 0)

Underflow occurs if the exact unrounded result has magnitude between zero and the smallest normalized number in the destination format, **and** the correctly rounded result in the destination format is inexact.

Table 67 summarizes what happens when an exact **unrounded** value u satisfying

$$0 \le |u| \le \textit{smallest normalized number}$$

would round, if no trap intervened, to a **rounded** value r which might be zero, subnormal, or the smallest normalized value. "UF" means underflow trap (with ufc set in *cexc*), "NX"

means inexact trap (with nxc set in *cexc*), "uf" means untrapped underflow exception (with ufc set in *cexc* and ufa in *aexc*), and "nx" means untrapped inexact exception (with nxc set in *cexc* and nxa in *aexc*).

Table 67: Untrapped Floating-Point Underflow (<u>V9=28</u>)

		UFM = 1 NXM = ?	UFM = 0 NXM = 1	UFM = 0 NXM = 0
	Underflow trap: Inexact trap:			
u = r	*r* is minimum normal	*None*	*None*	*None*
	r is subnormal	UF	*None*	*None*
	r is zero	*None*	*None*	*None*
u ≠ r	*r* is minimum normal	UF	NX	uf nx
	r is subnormal	UF	NX	uf nx
	r is zero	UF	NX	uf nx

See <u>5.1.7.6.2</u>, "ftt = unfinished_FPop", for details on how the divider handles untrapped underflows.

B.5 Integer Overflow Definition

F[sdq]TOi:

When a NaN, infinity, large positive argument ≥ 2147483648.0, or large negative argument ≤ –2147483649.0 is converted to an integer, the invalid_current (*nvc*) bit of FSR.*cexc* should be set and *fp_exception_IEEE_754* should be raised. If the floating-point invalid trap is disabled (FSR.TEM.NVM = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is 2147483647; if the sign bit of the operand is 1, the result is –2147483648.

F[sdq]TOx:

When a NaN, infinity, large positive argument ≥ 2^{63}, or large negative argument ≤ – $(2^{63} + 1)$, is converted to an extended integer, the invalid_current (*nvc*) bit of FSR.*cexc* should be set and *fp_exception_IEEE_754* should be raised. If the floating-point invalid trap is disabled (FSR.TEM.NVM = 0), no trap occurs and a numerical result is generated: if the sign bit of the operand is 0, the result is $2^{63} - 1$; if the sign bit of the operand is 1, the result is -2^{63}.

B.6 Floating-Point Nonstandard Mode

SPARC64-III does not implement any nonstandard IEEE operations and, thus, does not support a nonstandard mode.

C SPARC-V9 Implementation Dependencies

This appendix provides a summary of all implementation dependencies in the SPARC-V9 standard. In SPARC-V9 the notation "**IMPL. DEP. #*nn*:**" is used to identify the definition of an implementation dependency; the notation "(impl. dep. #*nn*)" is used to identify a reference to an implementation dependency. These dependencies are described by their number *nn* in <u>Table 68 on page 347</u>. These numbers have been removed from the body of this document for SPARC64-III to make the document more readable. <u>Table 68</u> has been modified to include a description of the manner in which SPARC64-III has resolved each implementation dependency.

SPARC International maintains a document, *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, which describes the implementation-dependent design features of all SPARC-V9-compliant implementations. Contact SPARC International for this document at

<div align="center">

SPARC International, Inc.
535 Middlefield Rd, Suite 210
Menlo Park, CA 94025
(415) 321-8692

</div>

C.1 Definition of an Implementation Dependency

The SPARC-V9 architecture is a **model** that specifies unambiguously the behavior observed by **software** on SPARC-V9 systems. Therefore, it does not necessarily describe the operation of the **hardware** of any actual implementation.

An implementation is **not** required to execute every instruction in hardware. An attempt to execute a SPARC-V9 instruction that is not implemented in hardware generates a trap. Whether an instruction is implemented directly by hardware, simulated by software, or emulated by firmware is implementation-dependent.

The two levels of SPARC-V9 compliance are described in <u>1.2.6, "SPARC-V9 Compliance" in *V9*</u>.

Some elements of the architecture are defined to be implementation-dependent. These elements include certain registers and operations that may vary from implementation to implementation, and are explicitly identified as such in this appendix.

Implementation elements (such as instructions or registers) that appear in an implementation but are not defined in this document (or its updates) are not considered to be SPARC-V9 elements of that implementation.

C.2 Hardware Characteristics

Hardware characteristics that do not affect the behavior observed by software on SPARC-V9 systems are not considered architectural implementation dependencies. A hardware characteristic may be relevant to the user system design (for example, the speed of execution of an instruction) or may be transparent to the user (for example, the method used for achieving cache consistency). The SPARC International document, *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, provides a useful list of these hardware characteristics, along with the list of implementation-dependent design features of SPARC-V9-compliant implementations.

In general, hardware characteristics deal with

■ Instruction execution speed

■ Whether instructions are implemented in hardware

■ The nature and degree of concurrency of the various hardware units comprising a SPARC-V9 implementation.

C.3 Implementation Dependency Categories

Many of the implementation dependencies can be grouped into four categories, abbreviated by their first letters throughout this appendix:

Value (v):
   The semantics of an architectural feature are well-defined, except that a value associated with it may differ across implementations. A typical example is the number of implemented register windows (Implementation dependency #2).

Assigned Value (a):
   The semantics of an architectural feature are well-defined, except that a value associated with it may differ across implementations and the actual value is assigned by SPARC International. Typical examples are the *impl* field of Version register (VER) (Implementation dependency #13) and the FSR.*ver* field (Implementation dependency #19).

Functional Choice (f):
   The SPARC-V9 architecture allows implementors to choose among several possible semantics related to an architectural function. A typical example is the treatment of a catastrophic error exception, which may cause either a deferred or a disrupting trap (Implementation dependency #31).

Total Unit (t):
   The existence of the architectural unit or function is recognized, but details are left to each implementation. Examples include the handling of I/O registers (Imple-

mentation dependency #7) and some alternate address spaces (Implementation dependency #29).

C.4 List of Implementation Dependencies

Table 68 provides a complete list of the implementation dependencies in the architecture, the definition of each, and references to the page numbers in the standard where each is defined or referenced. Most implementation dependencies occur because of the address spaces, I/O registers, registers (including ASRs), the type of trapping used for an exception, the handling of errors, or miscellaneous non-SPARC-V9-architectural units such as the MMU or caches (which affect the FLUSH instruction).

Table 68: SPARC64-III Implementation Dependencies (V9=29)

Nbr	Description	SPARC64-III Implementation Notes
1	**Software emulation of instructions** Whether an instruction is implemented directly by hardware, simulated by software, or emulated by firmware is implementation-dependent.	See 6.3.12, "Summary of Unimplemented Instructions", for details of unimplemented instructions. The operating system emulates all instructions that generate *illegal_instruction* or *unimplemented_FPop* exceptions.
2	**Number of IU registers** An implementation of the IU may contain from 64 to 528 general-purpose 64-bit *r* registers. This corresponds to a grouping of the registers into two sets of eight global *r* registers, plus a circular stack of from three to 32 sets of 16 registers each, known as register windows. Since the number of register windows present (NWINDOWS) is implementation-dependent, the total number of registers is also implementation-dependent.	The CPU has 5 register windows (NWINDOWS = 5) for a total of 96 integer registers.
3	**Incorrect IEEE Std 754-1985 results** An implementation may indicate that a floating-point instruction did not produce a correct IEEE Std 754-1985 result by generating a special floating-point unfinished or unimplemented exception. In this case, privileged mode software shall emulate any functionality not present in the hardware.	FDIV and FSQRT generate *unfinished_FPop* an exception under certain conditions. See 5.1.7.6.2, "ftt = unfinished_FPop" for details. All of the quad floating -point instructions are not implemented, and they generate an unimplemented exception.
4-5	*Reserved*	—
6	**I/O registers privileged status** Whether I/O registers can be accessed by non-privileged code is implementation-dependent.	This is beyond the scope of this publication. It should be defined in a system which uses SPARC64-III.
7	**I/O register definitions** The contents and addresses of I/O registers are implementation-dependent.	This is beyond the scope of this publication. It should be defined in a system which uses SPARC64-III.

Table 68: SPARC64-III Implementation Dependencies (<u>V9=29</u>) (*Continued*)

Nbr	Description	SPARC64-III Implementation Notes
8	**RDASR/WRASR target registers** Software can use read/write ancillary state register instructions to read/write implementation-dependent processor registers (ASRs 16-31).	See <u>A.44</u>, "Read State Register", and <u>A.63</u>, "Write State Register", for details of implementation-dependent RDASR/WRASR instructions.
9	**RDASR/WRASR privileged status** Whether each of the implementation-dependent read/write ancillary state register instructions (for ASRs 16-31) is privileged is implementation-dependent.	See <u>A.44</u>, "Read State Register", and <u>A.63</u>, "Write State Register", for details of implementation-dependent RDASR/WRASR instructions.
10-12	*Reserved*	—
13	**VER.*impl*** VER.*impl* uniquely identifies an implementation or class of software-compatible implementations of the architecture. Values $FFF0_{16}..FFFF_{16}$ are reserved and are not available for assignment.	VER.*impl* = 3 for the SPARC64-III CPU.
14-15	*Reserved*	—
16	**IU deferred-trap queue** The existence, contents, and operation of an IU deferred-trap queue are implementation-dependent; it is not visible to user application programs under normal operating conditions.	SPARC64-III does not have or need an IU deferred-trap queue.
17	*Reserved*	—
18	**Nonstandard IEEE 754-1985 results** Bit 22 of the FSR, FSR_nonstandard_fp (NS), when set to 1, causes the FPU to produce implementation-defined results that may not correspond to IEEE Standard 754-1985.	SPARC64-III always produces correct ANSI/IEEE-754 results; thus, writes to the NS bit are ignored and reads from it always return zero.
19	**FPU version, FSR.ver** Bits 19:17 of the FSR, FSR.*ver*, identify one or more implementations of the FPU architecture.	FSR.*ver* = 0 for SPARC64-III.
20-21	*Reserved*	—
22	**FPU TEM, cexc, and aexc** An implementation may choose to implement the TEM, *cexc*, and *aexc* fields in hardware in either of two ways (see <u>5.1.7.1</u> for details).	SPARC64-III implements all bits in the TEM, *cexc*, and *aexc* fields in hardware
23	**Floating-point traps** Floating-point traps may be precise or deferred. If deferred, a floating-point deferred-trap queue (FQ) must be present.	In SPARC64-III floating-point traps are always precise; no FQ is needed.
24	**FPU deferred-trap queue (FQ)** The presence, contents of, and operations on the floating-point deferred-trap queue (FQ) are implementation-dependent.	SPARC64-III does not have or need a floating-point deferred-trap queue.

Table 68: SPARC64-III Implementation Dependencies (<u>V9=29</u>) (*Continued*)

Nbr	Description	SPARC64-III Implementation Notes
25	**RDPR of FQ with nonexistent FQ** On implementations without a floating-point queue, an attempt to read the FQ with an RDPR instruction shall cause either an *illegal_instruction* exception or an *fp_exception_other* exception with FSR.*ftt* set to 4 (*sequence_error*).	Attempting to execute an RDPR of the FQ causes an *illegal_instruction* exception.
26-28	*Reserved*	—
29	**Address space identifier (ASI) definitions** The following ASI assignments are implementation-dependent: restricted ASIs $00_{16}..03_{16}$, $05_{16}..0B_{16}$, $0D_{16}..0F_{16}$, $12_{16}..17_{16}$, and $1A_{16}..7F_{16}$; and unrestricted ASIs $C0_{16}..FF_{16}$.	The ASIs that are supported by SPARC64-III are defined in <u>Appendix L, "ASI Assignments"</u>.
30	**ASI address decoding** An implementation may choose to decode only a subset of the 8-bit ASI specifier; however, it shall decode at least enough of the ASI to distinguish ASI_PRIMARY, ASI_PRIMARY_LITTLE, ASI_AS_IF_USER_PRIMARY, ASI_AS_IF_USER_PRIMARY_LITTLE, ASI_PRIMARY_NOFAULT, ASI_PRIMARY_NOFAULT_LITTLE, ASI_SECONDARY, ASI_SECONDARY_LITTLE, ASI_AS_IF_USER_SECONDARY, ASI_AS_IF_USER_SECONDARY_LITTLE, ASI_SECONDARY_NOFAULT, and ASI_SECONDARY_NOFAULT_LITTLE. If ASI_NUCLEUS and ASI_NUCLEUS_LITTLE are supported (impl. dep. #124), they must be decoded also. Finally, an implementation must always decode ASI bit<7> while PSTATE.PRIV = 0, so that an attempt by non-privileged software to access a restricted ASI will always cause a *privileged_action* exception.	SPARC64-III supports all of the listed ASIs.
31	**Catastrophic error exceptions** The causes and effects of catastrophic error exceptions are implementation-dependent. They may cause precise, deferred, or disrupting traps.	SPARC64-III contains a watchdog timer that times out after no instruction has been committed for the number of cycles required to count down a 31-bit register. If the timer times out, the CPU enters error_state and outputs P_FERR to the UPA bus.
32	**Deferred traps** Whether any deferred traps (and associated deferred-trap queues) are present is implementation-dependent.	See <u>7.3.2, "Deferred Traps"</u>. SPARC64-III does not contain a deferred trap queue.

Table 68: **SPARC64-III Implementation Dependencies (<u>V9=29</u>)** (*Continued*)

Nbr	Description	SPARC64-III Implementation Notes
33	**Trap precision** Exceptions that occur as the result of program execution may be precise or deferred, although it is recommended that such exceptions be precise. Examples include *mem_address_not_aligned* and *division_by_zero*.	The only deferred trap in SPARC64-III is the *data_breakpoint* trap. All other traps that occur as the result of program execution are precise
34	**Interrupt clearing** How quickly a processor responds to an interrupt request and the method by which an interrupt request is removed are implementation-dependent.	For details of interrupt handling see <u>Appendix N,</u> <u>"Interrupt Handling"</u>.
35	**Implementation-dependent traps** Trap type (TT) values $060_{16}..07F_{16}$ are reserved for implementation-dependent exceptions. The existence of *implementation_dependent_n* traps and whether any that do exist are precise, deferred, or disrupting is implementation-dependent.	SPARC64-III supports the following implementation-dependent traps: – *interrupt_vector* (tt = 060_{16}) – *data_breakpoint* (tt = 061_{16}) – *programmed_emulation_trap* (tt = 062_{16}) – *async_error* (tt = 063_{16}) – *32i_instruction_access_MMU_miss* (tt = 064_{16} through 067_{16}) – *32i_data_access_MMU_miss* (tt = 068_{16} through $06B_{16}$) – *32i_data_access_protection* (tt = $06C_{16}$ through $06F_{16}$) – *watchdog* (tt = $07F_{16}$)
36	**Trap priorities** The priorities of particular traps are relative and are implementation-dependent, because a future version of the architecture may define new traps, and implementations may define implementation-dependent traps that establish new relative priorities.	SPARC64-III's implementation-dependent traps have the following priorities: – *interrupt_vector* (priority=16) – *data_breakpoint* (priority=14) – *programmed_emulation_trap* (priority=6) – *async_error* (priority=2) – *32i_instruction_access_MMU_miss* (priority=2) – *32i_data_access_MMU_miss* (priority=12) – *32i_data_access_protection* (priority=12) – *watchdog* (priority=1)
37	**Reset trap** Some of a processor's behavior during a reset trap is implementation-dependent.	SPARC64-III implements Power On Reset (POR) through SCAN. Watchdog Reset (WDR) is not implemented. Externally Initiated Reset (XIR) with TL=MAXTL causes the CPU to enter error_state.
38	**Effect of reset trap on implementation-dependent registers** Implementation-dependent registers may or may not be affected by the various reset traps.	See <u>O.3, "Processor State after Reset and in</u> <u>RED_state"</u>.

Table 68: SPARC64-III Implementation Dependencies (<u>V9=29</u>) (*Continued*)

Nbr	Description	SPARC64-III Implementation Notes
39	**Entering error_state on implementation-dependent errors** The processor may enter error_state when an implementation-dependent error condition occurs.	A CPU watchdog timeout or any trap with TL=MAXTL cause entry to error_state.
40	**Error_state processor state** What occurs after error_state is entered is implementation-dependent, but it is recommended that as much processor state as possible be preserved upon entry to error_state.	SPARC64-III outputs P_FERR on entry to error_state. Most error logging register state will be preserved and can be read after a Power On Reset.
41	*Reserved*	—
42	**FLUSH instruction** If FLUSH is not implemented in hardware, it causes an *illegal_instruction* exception, and its function is performed by system software. Whether FLUSH traps is implementation-dependent.	SPARC64-III implements the FLUSH instruction in hardware.
43	*Reserved*	—
44	**Data access FPU trap** If a load floating-point instruction traps with any type of access error exception, the contents of the destination floating-point register(s) either remain unchanged or are undefined.	The destination register(s) are unchanged if an access error occurs.
45 - 46	*Reserved*	—
47	**RDASR** RDASR instructions with *rd* in the range 16..31 are available for implementation-dependent uses (impl. dep. #8). For an RDASR instruction with *rs1* in the range 16..31, the following are implementation-dependent: the interpretation of bits 13:0 and 29:25 in the instruction, whether the instruction is privileged (impl. dep. #9), and whether it causes an *illegal_instruction* trap.	See <u>A.44, "Read State Register"</u>, for details.
48	**WRASR** WRASR instructions with *rd* in the range 16..31 are available for implementation-dependent uses (impl. dep. #8). For a WRASR instruction with *rd* in the range 16..31, the following are implementation-dependent: the interpretation of bits 18:0 in the instruction, the operation(s) performed (for example, **xor**) to generate the value written to the ASR, whether the instruction is privileged (impl. dep. #9), and whether it causes an *illegal_instruction* trap.	See <u>A.63, "Write State Register"</u>, for details.
49-54	*Reserved*	—

Table 68: **SPARC64-III Implementation Dependencies (<u>V9=29</u>)** (*Continued*)

Nbr	Description	SPARC64-III Implementation Notes
55	**Floating-point underflow detection** Whether "tininess" (in IEEE 754 terms) is detected before or after rounding is implementation-dependent. It is recommended that tininess be detected before rounding.	SPARC64-III detects "tininess" before rounding.
56-100	*Reserved*	—
101	**Maximum trap level** It is implementation-dependent how many additional levels, if any, past level 4 are supported.	MAXTL = 4.
102	**Clean windows trap** An implementation may choose either to implement automatic "cleaning" of register windows in hardware, or generate a *clean_window* trap, when needed, for window(s) to be cleaned by software.	SPARC64-III generates a *clean_window* trap.
103	**Prefetch instructions** The following aspects of the PREFETCH and PREFETCHA instructions are implementation-dependent: (1) whether they have an observable effect in privileged code; (2) whether they can cause a *data_access_MMU_miss* exception; (3) the attributes of the block of memory prefetched: its size (minimum = 64 bytes) and its alignment (minimum = 64-byte alignment); (4) whether each variant is implemented as a NOP, with its full semantics, or with common-case prefetching semantics; (5) whether and how variants 16..31 are implemented.	SPARC64-III implements PREFETCH variations 0 thru 4 with the following implementation-dependent characteristics: – The prefetches have observable affects in privileged code. – A prefetch does not cause a *data_access_MMU_miss* trap, because the prefetch is dropped when a *data_access_MMU_miss* condition happens. – All prefetches are for 64-byte cache lines, which are aligned on a 64-byte boundary. – See <u>A.42</u>, "Prefetch Data", for implemented variations and their characteristics. – Variants 16..31 are treated as NOPs. Prefetches will work normally if the ASI is ASI_PRIMARY, ASI_SECONDARY, or ASI_NUCLEUS.
104	**VER.manuf** VER.*manuf* contains a 16-bit semiconductor manufacturer code. This field is optional, and if not present reads as zero. VER.*manuf* may indicate the original supplier of a second-sourced chip in cases involving mask-level second-sourcing. It is intended that the contents of VER.*manuf* track the JEDEC semiconductor manufacturer code as closely as possible. If the manufacturer does not have a JEDEC semiconductor manufacturer code, SPARC International will assign a VER.*manuf* value.	VER.*manuf* = 0004_{16}. The lower 8 bits are Fujitsu's JEDEC manufacturing code.

Table 68: SPARC64-III Implementation Dependencies (<u>V9=29</u>) (*Continued*)

Nbr	Description	SPARC64-III Implementation Notes
105	**TICK register** The difference between the values read from the TICK register on two reads should reflect the number of processor cycles executed between the reads. If an accurate count cannot always be returned, any inaccuracy should be small, bounded, and documented. An implementation may implement fewer than 63 bits in TICK.*counter*; however, the counter as implemented must be able to count for at least 10 years without overflowing. Any upper bits not implemented must read as zero.	SPARC64-III implements 63 bits of the TICK register; it increments on every clock cycle.
106	**IMPDEP*n* instructions** The IMPDEP1 and IMPDEP2 instructions are completely implementation-dependent. Implementation-dependent aspects include their operation, the interpretation of bits 29:25 and 18:0 in their encodings, and which (if any) exceptions they may cause.	SPARC64-III uses the IMPDEP2 opcode for the Multiply Add/Subtract instructions.
107	**Unimplemented LDD trap** It is implementation-dependent whether LDD and LDDA are implemented in hardware. If not, an attempt to execute either will cause an *unimplemented_LDD* trap.	SPARC64-III implements LDD in hardware.
108	**Unimplemented STD trap** It is implementation-dependent whether STD and STDA are implemented in hardware. If not, an attempt to execute either will cause an *unimplemented_STD* trap.	SPARC64-III implements STD in hardware.
109	**LDDF_mem_address_not_aligned** LDDF and LDDFA require only word alignment. However, if the effective address is word-aligned but not doubleword-aligned, either may cause an *LDDF_mem_address_not_aligned* trap, in which case the trap handler software shall emulate the LDDF (or LDDFA) instruction and return.	If the address is word-aligned but not double-word aligned, SPARC64-III generates the *LDDF_mem_address_not_aligned* exception. The trap handler software emulates the instruction.
110	**STDF_mem_address_not_aligned** STDF and STDFA require only word alignment in memory. However, if the effective address is word-aligned but not doubleword-aligned, either may cause an *STDF_mem_address_not_aligned* trap, in which case the trap handler software shall emulate the STDF or STDFA instruction and return.	If the address is word-aligned but not double-word aligned, SPARC64-III generates the *STDF_mem_address_not_aligned* exception. The trap handler software emulates the instruction.

Table 68: **SPARC64-III Implementation Dependencies (<u>V9=29</u>)** (*Continued*)

Nbr	Description	SPARC64-III Implementation Notes
111	**LDQF_mem_address_not_aligned** LDQF and LDQFA require only word alignment. However, if the effective address is word-aligned but not quadword-aligned, either may cause an *LDQF_mem_address_not_aligned* trap, in which case the trap handler software shall emulate the LDQF (or LDQFA) instruction and return.	SPARC64-III generates an *illegal_instruction* exception for all LDQFs. The CPU does not perform the check for *fp_disabled*. The trap handler software emulates the instruction.
112	**STQF_mem_address_not_aligned** STQF and STQFA require only word alignment in memory. However, if the effective address is word-aligned but not quadword-aligned, either may cause an *STQF_mem_address_not_aligned* trap, in which case the trap handler software shall emulate the STQF or STQFA instruction and return.	SPARC64-III generates an *illegal_instruction* exception for all STQFs. The CPU does not perform the check for *fp_disabled*. The trap handler software emulates the instruction.
113	**Implemented memory models** Whether the Partial Store Order (PSO) or Relaxed Memory Order (RMO) models are supported is implementation-dependent.	SPARC64-III implements PSO with Total Store Order (TSO) or Load/Store Order (LSO), which are stronger models. See <u>Chapter 8, "Memory Models"</u>, for details.

114	**RED_state trap vector address (RSTVaddr)**
	The RED_state trap vector is located at an implementation-dependent address referred to as RSTVaddr.
		RSTVaddr is a constant, where:
		    $VA = \text{FFFF FFFF F000 0000}_{16}$ and
		    $PA = \text{1FF F000 0000}_{16}$

115	**RED_state processor state** What occurs after the processor enters RED_state is implementation-dependent.	See <u>7.2.1, "RED_state"</u>, for details of implementation-specific actions in RED_state.
116	**SIR_enable control flag** The location of the SIR_enable control flag and the means of accessing the SIR_enable control flag are implementation-dependent. In some implementations, it may be permanently zero.	In SPARC64-III the SIR_enable control flag is hard-wired to 0; thus it always treats the SIR instruction as a NOP if PSTATE.PRIV=0.
117	**MMU disabled prefetch behavior** Whether Prefetch and Non-faulting Load always succeed when the MMU is disabled is implementation-dependent.	Prefetch and Non-faulting Load always succeed when the MMU is disabled.
118	**Identifying I/O locations** The manner in which I/O locations are identified is implementation-dependent.	This is beyond the scope of this publication. It should be defined in a system which uses SPARC64-III.
119	**Unimplemented values for PSTATE.MM** The effect of writing an unimplemented memory-mode designation into PSTATE.MM is implementation-dependent.	Writing 11_2 into PSTATE.MM causes the machine to use the STO memory model. However, the encoding 11_2 should not be used, since future versions of SPARC64-III may use this encoding for a new memory model.

Table 68: SPARC64-III Implementation Dependencies (<u>V9=29</u>) (*Continued*)

Nbr	Description	SPARC64-III Implementation Notes
120	**Coherence and atomicity of memory operations** The coherence and atomicity of memory operations between processors and I/O DMA memory accesses are implementation-dependent.	This is beyond the scope of this publication. It should be defined in a system which uses SPARC64-III.
121	**Implementation-dependent memory model** An implementation may choose to identify certain addresses and use an implementation-dependent memory model for references to them.	SPARC64-III implements Load/Store Order (LSO), Total Store Order (TSO), and Store Order (STO) memory models. See <u>Chapter 8, "Memory Models"</u>, for details. Accesses to pages with the SO (Strongly Ordered) bit of their MMU page table entry set are also made in Program Order.
122	**FLUSH latency** The latency between the execution of FLUSH on one processor and the point at which the modified instructions have replaced outdated instructions in a multiprocessor is implementation-dependent.	This is beyond the scope of this publication. It should be defined in a system which uses SPARC64-III.
123	**Input/output (I/O) semantics** The semantic effect of accessing input/output (I/O) registers is implementation-dependent.	This is beyond the scope of this publication. It should be defined in a system which uses SPARC64-III.
124	**Implicit ASI when TL > 0** When TL > 0, the implicit ASI for instruction fetches, loads, and stores is implementation-dependent. See <u>F.4.4, "Contexts" in *V9*</u> for more information.	SPARC64-III uses ASI_NUCLEUS for instruction fetches and ASI_NUCLEUS{_LITTLE} for data fetches as the implicit ASI when TL > 0.
125	**Address masking** When PSTATE.AM = 1, the value of the high-order 32-bits of the PC transmitted to the specified destination register(s) by CALL, JMPL, RDPC, and on a trap is implementation-dependent.	When PSTATE.AM = 1, SPARC64-III *does **not*** mask out the high-order 32 bits of the PC when transmitting it to the destination register; all 64-bits of the PC are transmitted.
126	**Register Windows State Registers Width** Privileged registers CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN contain values in the range 0..NWINDOWS−1. The effect of writing a value greater than NWINDOWS-1 to any of these registers is undefined.Although the width of each of these five registers is nominally 5 bits, the width is implementation-dependent and shall be between $\lceil \log_2(\text{NWINDOWS}) \rceil$ and 5 bits, inclusive. If fewer than 5 bits are implemented, the unimplemented upper bits shall read as 0 and writes to them shall have no effect. All five registers should have the same width.	NWINDOWS for SPARC64-III is 5, therefore only 3 bits are implemented for the following registers: CWP, CANSAVE, CANRESTORE, OTHERWIN. If an attempt is made to write a value greater than NWINDOWS−1 to any of these registers, the extraneous upper bits are discarded. The CLEANWIN register contains 3 bits, but software must not write the values 5, 6, or 7 to the register. Setting CLEANWIN > 4 violates the register window state definition in <u>6.4.1, "Register Window State Definition"</u>, **Note**: Hardware does not enforce this restriction; system software must keep the window state consistent.

D Formal Specification of the Memory Models

V9 *Consult V9 for the text of this appendix.*

E Opcode Maps

E.1 Overview

This appendix contains the SPARC64-III instruction opcode maps.

Opcodes marked with a dash '—' are reserved; an attempt to execute a reserved opcode shall cause a trap, unless it is an implementation-specific extension to the instruction set. See 6.3.11, "Reserved Opcodes and Instruction Fields", for more information.

In this appendix and in Appendix A, "Instruction Definitions", certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in Table 49 on page 214. For deprecated opcodes, see the appropriate instruction pages in Appendix A for preferred substitute instructions.

E.2 Tables

In the tables in this appendix, *reserved* (—) and shaded entries indicate opcodes that are not implemented in SPARC64-III.

Table 69: *op*[1:0] (V9=30)

op [1:0]			
0	**1**	**2**	**3**
Branches & SETHI *See Table 70*	CALL	Arithmetic & Misc. *See Table 71*	Loads/Stores *See Table 72*

Table 70: *op2*[2:0] (*op* = 0) (V9=31)

op2 [2:0]							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ILLTRAP	BPcc *See Table 75*	Bicc^D *See Table 75*	BPr *See Table 76*	SETHI NOP†	FBPfcc *See Table 75*	FBfcc^D *See Table 75*	—

$^†rd = 0$, *imm22* = 0

The ILLTRAP and *reserved* (—) encodings generate an *illegal_instruction* trap.

Table 71: *op3*[5:0] (*op* = 2) (<u>**V9=32**</u>)

		op3 [5:4]			
		0	**1**	**2**	**3**
op3 [3:0]	**0**	ADD	ADDcc	TADDcc	WRY^D (*rd* = 0) — (*rd*=1) WRCCR (*rd*=2) WRASI (*rd*=3) — (*rd*=4, 5) WRFPRS (*rd*=6) WRASR^{PASR} (*7≤rd≤14*) SIR (*rd*=15, *rs1*=0, *i*=1)
	1	AND	ANDcc	TSUBcc	SAVED^P (*fcn* = 0), RESTORED^P (*fcn* = 1)
	2	OR	ORcc	TADDccTV^D	WRPR^P
	3	XOR	XORcc	TSUBccTV^D	—
	4	SUB	SUBcc	MULScc^D	FPop1 *See Table 73*
	5	ANDN	ANDNcc	SLL (*x* = 0), SLLX (*x* = 1)	FPop2 *See Table 74*
	6	ORN	ORNcc	SRL (*x* = 0), SRLX (*x* = 1)	IMPDEP1
	7	XNOR	XNORcc	SRA (*x* = 0), SRAX (*x* = 1)	IMPDEP2 (FMADD / FMSUB)
	8	ADDC	ADDCcc	RDY^D (*rs1* = 0) — (*rs1*=1) RDCCR (*rs1*=2) RDASI (*rs1*=3) RDTICK^{PNPT} (*rs1*=4) RDPC (*rs1*=5) RDFPRS (*rs1*=6) RDASR^{PASR} (*7≤rd≤14*) MEMBAR (*rs1* = 15, *rd*=0, *i* = 1) STBAR^D (*rs1* = 15, *rd*=0, *i* = 0)	JMPL
	9	MULX	—	—	RETURN
	A	UMUL^D	UMULcc^D	RDPR^P	Tcc *See Table 75*
	B	SMUL^D	SMULcc^D	FLUSHW	FLUSH
	C	SUBC	SUBCcc	MOVcc	SAVE
	D	UDIVX	—	SDIVX	RESTORE
	E	UDIV^D	UDIVcc^D	POPC (*rs1* = 0) — (*rs1*>0)	DONE^P (*fcn* = 0) RETRY^P (*fcn* = 1)
	F	SDIV^D	SDIVcc^D	MOVr *See Table 76*	—

POPC and the reserved (—) opcodes cause an illegal_instruction trap.

Table 72: *op3*[5:0] (*op* = 3) (**V9=33**)

		op3 [5:4]			
		0	**1**	**2**	**3**
op3 [3:0]	**0**	LDUW	LDUWA[PASI]	LDF	LDFA[PASI]
	1	LDUB	LDUBA[PASI]	LDFSR[D], LDXFSR	—
	2	LDUH	LDUHA[PASI]	LDQF	LDQFA[PASI]
	3	LDD[D]	LDDA[D, PASI]	LDDF	LDDFA[PASI]
	4	STW	STWA[PASI]	STF	STFA[PASI]
	5	STB	STBA[PASI]	STFSR[D], STXFSR	—
	6	STH	STHA[PASI]	STQF	STQFA[PASI]
	7	STD[D]	STDA[PASI]	STDF	STDFA[PASI]
	8	LDSW	LDSWA[PASI]	—	—
	9	LDSB	LDSBA[PASI]	—	—
	A	LDSH	LDSHA[PASI]	—	—
	B	LDX	LDXA[PASI]	—	—
	C	—	—	—	CASA[PASI]
	D	LDSTUB	LDSTUBA[PASI]	PREFETCH	PREFETCHA[PASI]
	E	STX	STXA[PASI]	—	CASXA[PASI]
	F	SWAP[D]	SWAPA[D, PASI]	—	—

LDQF, LDQFA, STQF, STQFA, and the *reserved* (—) opcodes cause an *illegal_instruction* trap.

Table 73: *opf*[8:3] (*op* = 2,*op3* = 34_{16} = **FPop1**) (**V9=34**)

opf[8:3]	opf[3:0]							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
00_{16}	—	FMOVs	FMOVd	FMOVq	—	FNEGs	FNEGd	FNEGq
01_{16}	—	FABSs	FABSd	FABSq	—	—	—	—
02_{16}	—	—	—	—	—	—	—	—
03_{16}	—	—	—	—	—	—	—	—
04_{16}	—	—	—	—	—	—	—	—
05_{16}	—	FSQRTs	FSQRTd	FSQRTq	—	—	—	—
06_{16}	—	—	—	—	—	—	—	—
07_{16}	—	—	—	—	—	—	—	—
08_{16}	—	FADDs	FADDd	FADDq	—	FSUBs	FSUBd	FSUBq
09_{16}	—	FMULs	FMULd	FMULq	—	FDIVs	FDIVd	FDIVq
$0A_{16}$	—	—	—	—	—	—	—	—
$0B_{16}$	—	—	—	—	—	—	—	—
$0C_{16}$	—	—	—	—	—	—	—	—
$0D_{16}$	—	FsMULd	—	—	—	—	FdMULq	—
$0E_{16}$	—	—	—	—	—	—	—	—
$0F_{16}$	—	—	—	—	—	—	—	—
10_{16}	—	FsTOx	FdTOx	FqTOx	FxTOs	—	—	—
11_{16}	FxTOd	—	—	—	FxTOq	—	—	—
12_{16}	—	—	—	—	—	—	—	—
13_{16}	—	—	—	—	—	—	—	—
14_{16}	—	—	—	—	—	—	—	—
15_{16}	—	—	—	—	—	—	—	—
16_{16}	—	—	—	—	—	—	—	—
17_{16}	—	—	—	—	—	—	—	—
18_{16}	—	—	—	—	FiTOs	—	FdTOs	FqTOs
19_{16}	FiTOd	FsTOd	—	FqTOd	FiTOq	FsTOq	FdTOq	—
$1A_{16}$	—	FsTOi	FdTOi	FqTOi	—	—	—	—
$1B_{16}..3F_{16}$	—	—	—	—	—	—	—	—

Shaded boxes and *reserved* (—) opcodes cause an *fp_exception_other* trap with *ftt = unimplemented_FPop*.

Table 74: *opf*[8:0] (*op* = 2, *op3* = 35$_{16}$ = FPop2) (**V9=35**)

opf[8:4]	opf[3:0]								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8..F**
00	—	FMOVs (fcc0)	FMOVd (fcc0)	FMOVq (fcc0)	—	†	†	†	—
01	—	—	—	—	—	—	—	—	—
02	—	—	—	—	—	FMOVsZ	FMOVdZ	FMOVqZ	—
03	—	—	—	—	—	—	—	—	—
04	—	FMOVs (fcc1)	FMOVd (fcc1)	FMOVq (fcc1)	—	FMOVsLEZ	FMOVdLEZ	FMOVqLEZ	—
05	—	FCMPs	FCMPd	FCMPq	—	FCMPEs	FCMPEd	FCMPEq	—
06	—	—	—	—	—	FMOVsLZ	FMOVdLZ	FMOVqLZ	—
07	—	—	—	—	—	—	—	—	—
08	—	FMOVs (fcc2)	FMOVd (fcc2)	FMOVq (fcc2)	—	†	†	†	—
09	—	—	—	—	—	—	—	—	—
0A	—	—	—	—	—	FMOVsNZ	FMOVdNZ	FMOVqNZ	—
0B	—	—	—	—	—	—	—	—	—
0C	—	FMOVs (fcc3)	FMOVd (fcc3)	FMOVq (fcc3)	—	FMOVsGZ	FMOVdGZ	FMOVqGZ	—
0D	—	—	—	—	—	—	—	—	—
0E	—	—	—	—	—	FMOVsGEZ	FMOVdGEZ	FMOVqGEZ	—
0F	—	—	—	—	—	—	—	—	—
10	—	FMOVs (icc)	FMOVd (icc)	FMOVq (icc)	—	—	—	—	—
11..17	—	—	—	—	—	—	—	—	—
18	—	FMOVs (xcc)	FMOVd (xcc)	FMOVq (xcc)	—	—	—	—	—
19..1F	—	—	—	—	—	—	—	—	—

†Undefined variation of FMOVR

Shaded boxes and *reserved* (—) opcodes cause an *fp_exception_other* trap with *ftt* = *unimplemented_FPop*.

Table 75: *cond*[3:0] (**V9=36**)

		BPcc	Bicc[D]	FBPfcc	FBfcc[D]	Tcc
		op = 0 op2 = 1	op = 0 op2 = 2	op = 0 op2 = 5	op = 0 op2 = 6	op = 2 op3 = $3A_{16}$
cond [3:0]	0	BPN	BN[D]	FBPN	FBN[D]	TN
	1	BPE	BE[D]	FBPNE	FBNE[D]	TE
	2	BPLE	BLE[D]	FBPLG	FBLG[D]	TLE
	3	BPL	BL[D]	FBPUL	FBUL[D]	TL
	4	BPLEU	BLEU[D]	FBPL	FBL[D]	TLEU
	5	BPCS	BCS[D]	FBPUG	FBUG[D]	TCS
	6	BPNEG	BNEG[D]	FBPG	FBG[D]	TNEG
	7	BPVS	BVS[D]	FBPU	FBU[D]	TVS
	8	BPA	BA[D]	FBPA	FBA[D]	TA
	9	BPNE	BNE[D]	FBPE	FBE[D]	TNE
	A	BPG	BG[D]	FBPUE	FBUE[D]	TG
	B	BPGE	BGE[D]	FBPGE	FBGE[D]	TGE
	C	BPGU	BGU[D]	FBPUGE	FBUGE[D]	TGU
	D	BPCC	BCC[D]	FBPLE	FBLE[D]	TCC
	E	BPPOS	BPOS[D]	FBPULE	FBULE[D]	TPOS
	F	BPVC	BVC[D]	FBPO	FBO[D]	TVC

Table 76: Encoding of *rcond*[2:0] Instruction Field (**V9=37**)

		BPr	MOVr	FMOVr
		op = 0 op2 = 3	op = 2 op3 = $2F_{16}$	op = 2 op3 = 35_{16}
rcond [2:0]	0	—	—	—
	1	BRZ	MOVRZ	FMOVRZ
	2	BRLEZ	MOVRLEZ	FMOVRLEZ
	3	BRLZ	MOVRLZ	FMOVRLZ
	4	—	—	—
	5	BRNZ	MOVRNZ	FMOVRNZ
	6	BRGZ	MOVRGZ	FMOVRGZ
	7	BRGEZ	MOVRGEZ	FMOVRGEZ

Table 77: *cc / opf_cc* **Fields (MOVcc and FMOVcc) (V9=38)**

opf_cc			Condition Code Selected
cc2	cc1	cc0	
0	0	0	*fcc0*
0	0	1	*fcc1*
0	1	0	*fcc2*
0	1	1	*fcc3*
1	0	0	*icc*
1	0	1	—
1	1	0	*xcc*
1	1	1	—

Table 78: *cc* **Fields (FBPfcc, FCMP and FCMPE) (V9=39)**

cc1	cc0	Condition Code Selected
0	0	*fcc0*
0	1	*fcc1*
1	0	*fcc2*
1	1	*fcc3*

Table 79: *cc* **Fields (BPcc and Tcc) (V9=40)**

cc1	cc0	Condition Code Selected
0	0	*icc*
0	1	—
1	0	*xcc*
1	1	—

F MMU Architecture

F.1 Introduction

Appendix F, "SPARC-V9 MMU Requirements" in *V9*, describes the boundary conditions that all SPARC-V9 MMUs must satisfy. This version of the appendix describes the architecture of HAL's SPARC64-III memory management unit. It is intended to provide the information needed to port the Solaris O/S to the SPARC64-III.

You should read and understand the concepts introduced in Appendix F, "SPARC-V9 MMU Requirements" in *V9* before proceeding.

F.1.1 Abbreviations and Acronyms

The following abbreviations and acronyms are used extensively throughout this appendix.

μ**ITLB**

> Instruction Micro Translation Look-aside Buffer. A 32-entry fully-associative TLB used to translate virtual instruction addresses to physical instruction addresses.

μ**DTLB:**

> Data Micro Translation Look-aside Buffer. A 32-entry fully-associative TLB used to translate virtual operand addresses to physical operand addresses.

MTLB:

> Main Translation Look-aside Buffer. A 256 entry fully-associative TLB that holds the address translations for both instruction and operand references. MTLB is accessed on μITLB or μDTLB misses.

VA:

> Virtual Address.

PA:

> Physical Address.

PTE:

> Page Table Entry.

TR:

> Translation region. A Portion of the MMU containing the MTLB hardware.

F.2   MMU and TLB Overview

The SPARC64-III MMU comprises:

■ An instruction micro-TLB (μITLB)

■ A data micro-TLB (μDTLB) and

■ A main TLB (MTLB).

The μTLBs are small (32 entry), fully associative TLBs; they perform translation in parallel with cache access. If an access cannot be translated by the μTLB, the hardware references the MTLB. The MTLB is a relatively large (256 entries), fully associative TLB. If the MTLB contains the PTE for an access, MMU hardware copies it into the appropriate μTLB. The μTLBs are invisible to software, except for instructions that invalidate the entire μITLB and μDTLB. Both the μITLB and μDTLB should be invalidated before enabling virtual address translation. The software operates on the MTLB and the hardware takes care of reflecting changes in μTLBs due to operations performed on the MTLB. The hardware ensures that all entries in the μTLBs (both μITLB and μDTLB) are also present in the MTLB.

This appendix concentrates on MTLB operations.

Figure 80 illustrates the MMU hardware. The μITLB, μDTLB and D1 Cache are connected to the MTLB. Activities related to translation are carried over the interfaces between the μTLBs and the MTLB. The MTLB receives its instructions via the D1 Cache interface and sends back its responses to the D1 Cache interface. The D1 Cache interface exists simply to provide a physical path to execute MMU related load and stores.
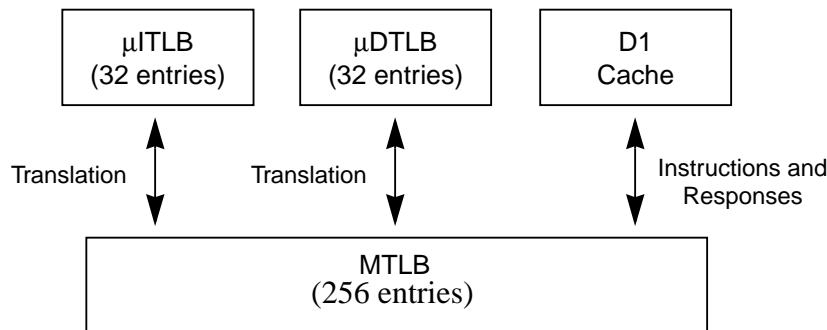
Figure 80: **SPARC64-III MMU Organization**

The following are the main features of the MMU:

■ Full 64-bit virtual address (VA) support. All virtual addresses are qualified by a 12-bit context. Taken together, these two items make up a complete reference of "context:VA".

■ Primary, Secondary and Nucleus contexts are supported. A context is represented by a 12-bit number.

■ 2 terabytes of physical memory may be addressed through the supported 41 bit physical address. SPARC64-III-based multi-processor systems will be based upon a shared main memory.

■ The MTLB is a unified, 256 entry, fully associative TLB. It has translation entries for both instruction and data accesses. Each entry translates a page, which may be of 15 different sizes.

■ Software tablewalk is used to update the MTLB with a new PTE on an MTLB miss.

F.3 MTLB Organization

The MTLB caches mappings of context:VA to PA. It is organized as a 256 entry fully associative structure. Comparison is performed as follows:

If the Universal (U) bit is not set (=0) for a page:
Translate context:VA $\rightarrow$ PA

If the Universal (U) bit is set (=1) for a page:
Translate VA $\rightarrow$ PA

In simple terms, the Universal bit means that the translation is valid for all contexts in the system.

The MTLB supports variable sized pages and has a facility to lock entries. The contiguous block of lockable entries always starts at entry 0. Entries can be locked or unlocked by privileged software only. Locked entries are excluded from the normal insertion algorithm. Privileged software has the capability of reading, writing or invalidating the MTLB entries, including the locked entries.

Figure 81 illustrates the MTLB internal organization.
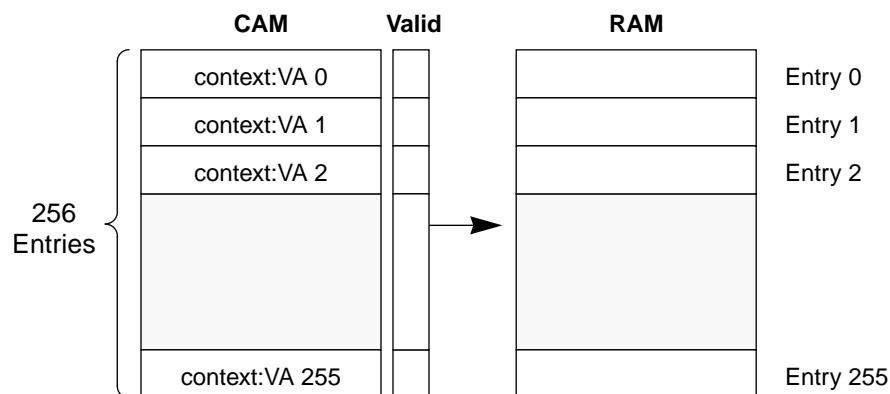
Figure 81: 256 Entry Fully Associative Main TLB (MTLB) Organization

F.3.1 Contexts

Contexts provide a set of 64-bit address spaces and have the following properties:

■ Contexts are 12-bit values in SPARC64-III; that is, there are 4,096 distinct contexts.

■ Context:VA is the fully qualified virtual address for translation.

■ The MMU supports three context types:

    A. Primary Context

    B. Secondary Context

    C. Nucleus Context

Each of the contexts is stored in a register. Only one of these contexts is used for a given translation operation. A context switch—that is, a write into the context register—does not cause invalidation of the MTLB or uTLBs. When a context number is reused—that is,. when the context rolls over from its maximum value of 2^{12} to 0—the software takes the responsibility of invalidating the MTLB. The hardware is responsible for invalidating the corresponding entry in either the μITLB or the μDTLB. The hardware makes sure that all entries present in the μITLB and μDTLB are also present in the MTLB. There is no mechanism to invalidate the entire MTLB using a single instruction; if the entire MTLB must be invalidated, software invalidates it one entry at a time. An entry is invalidated by first writing a 0 into the tlb_data register and then issuing a write_data_mtlb_specified_entry or write_data_mtlb_fifo_counter instruction.

Figure 82 illustrates the MTLB access mechanism.
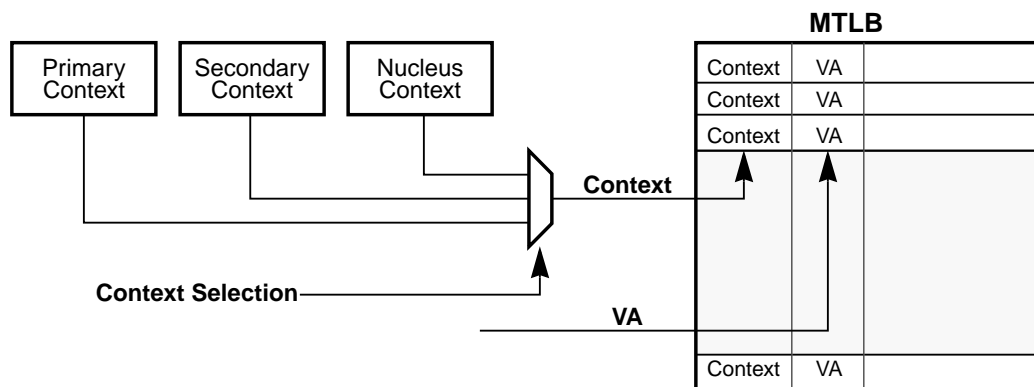
Figure 82: MTLB Access

F.3.2 Page Table Entry (PTE) Format

The SPARC64-III processor hardware does not assume any particular data structure in memory for the MMU to work correctly. To speed up the TLB miss handler, however, SPARC64-III does provide the hardware calculation of pointers. Operating system software can either use the hardware support or ignore it. If system software uses the TLB miss hardware support, the following data structure should be used in memory:

On an MTLB miss, the miss trap handler is invoked. This accesses the Translation Memory Buffer (TMB), which is resident in memory.

■ TMB is organized as a direct mapped cache. The index consists of the lower bits of the miss VA. The tag consists of the remaining upper bits of the miss VA.

■ The TMB contains between 512 and 64K Page Table Entries (PTEs).

■ The TMB can have a mixture of 8Kb and 64Kb pages, or it can be split into 8Kb (lower half) and 64Kb (upper half) pages. A split TMB is more efficient.

■ Accesses to the TMB are optimized for 8Kb and 64Kb page sizes.

■ Each PTE is 16 bytes long. These 16 bytes can be accessed atomically using the quad load (LDDA) instruction.

■ If a PTE is not found in the TMB, a slow, but detailed, trap handler is invoked. There is no restriction for the data structure accessed by the slow handler.

F.3.2.1 PTE Format (within TMB)

Figure 83 illustrates the Page Table Entry (PTE) as it is stored in the TMB. The following subsections describe each PTE field in detail.
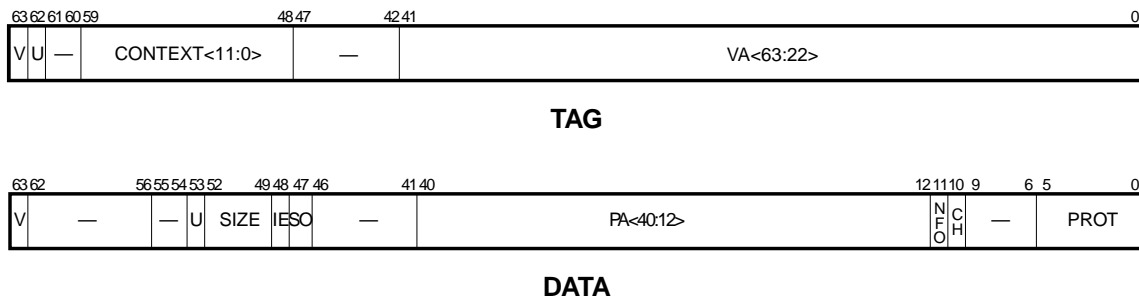
Figure 83: PTE Format Within the TMB

V

V=1 indicates that the PTE is valid. This bit is present in both TAG and DATA portions of the PTE to speed up the TLB miss handler.

U

Universal bit. If U=1, the context is ignored; that is, this associated VA is valid for all contexts, and is thus, universal. The U bit is present in both TAG and DATA portions of the PTE to speed up the TLB miss handler.

CONTEXT<11:0>

Context number.

VA<63:22>

Tag portion of the VA. These bits of the VA are used to determine if an address matches the TMB entry.

SIZE<3:0>

Encodes the page size associated with this PTE, as specified in Table 80.

Table 80: Page Size Encoding in PTE.SIZE

SIZE	Page Size	Page Size (2^n)
0000_2	4Kb	2^{12}
0001_2	8Kb	2^{13}
0010_2	16 Kb	2^{14}
0011_2	32 Kb	2^{15}
0100_2	64 Kb	2^{16}
0101_2	128 Kb	2^{17}
0110_2	256 Kb	2^{18}
0111_2	512 Kb	2^{19}
1000_2	1 Mb	2^{20}
1001_2	4 Mb	2^{22}
1010_2	16 Mb	2^{24}
1011_2	64 Mb	2^{26}
1100_2	256 Mb	2^{28}
1101_2	1 Gb	2^{30}
1110_2	4 Gb	2^{32}
1111_2	4 Gb	2^{32}

IE

Invert Endianness. IE=1 means that the endianness of data accessed through this PTE is to be inverted. This does not imply that the associated data is little-endian.

SO

Strongly ordered. SO=1 means that the accesses through this PTE are strongly ordered.

PA<40:12>

Physical Address.

NFO

Non-Faulting-Only bit, used for loads. If NFO=1, protection violations do not generate exceptions. The only side effect they have is that they cause the data returned to be zero.

CH

Cacheable. When CH=1, the information at the associated address is cacheable; when CH=0, it is not cacheable. Since the SPARC64-III follows the Sun Microsystems' UPA protocol, when CH=0, not only are the accesses non cacheable, but they must also come from the I/O space. In other words, it is not possible to access information from memory when CH=0.

PROT<5:0>

> Protection bits. The table below shows the bit definition. When a protection bit is zero, the access is denied. When a protection bit is set (=1), the associated access is allowed. Table 81 enumerates the encodings for the PTE.PROT field.

Table 81: PTE.PROT Field Encoding

PROT	Meaning
Bit 5	Supervisor Read Permission
Bit 4	Supervisor Write Permission
Bit 3	Supervisor Execute Permission
Bit 2	User Read Permission
Bit 1	User Write Permission
Bit 0	User Execute Permission

All fields marked with a '—' are *reserved*. The software should write zero to these fields.

F.3.3 Translation Lookaside Buffer (TLB) Entry Format (within MTLB)

Figure 84 illustrates the TLB entry format.
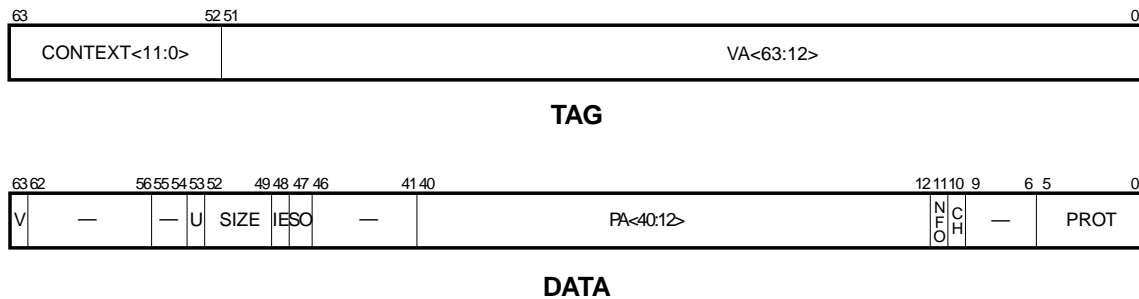
Figure 84: TLB Entry Format

The Translation Lookaside Buffer Entry has almost exactly the same format as the Page Table Entry. Because of this, the TLB Entry fields are not documented here; refer to the PTE fields above for their definitions.

All fields marked with a '—' are *reserved*. They are read as zero and writes to them are ignored.

This format is used in the following instruction:

- Write Instruction Main TLB Into Specified Entry (ASI=33, W)

- Write Instruction Main TLB Into FIFO Entry (ASI=34, W)

- Write Data Main TLB Into Specified Entry (ASI=35, W)

- Write Data Main TLB Into FIFO Entry (ASI=36, W)

- Read Main TLB Context and VA X (ASI=37, R)

■ Read Main TLB Context and VA Y (ASI=41, R)

F.4 MMU Registers

The table below lists the MMU-related registers accessed using special ASIs. A write to these registers should take place using appropriate memory barriers or using syncing or serializing instructions.

Table 82: MMU Registers

Register Name	ASI/ asr	Addr hex	VA hex	Function
primary_context	ASI	40(r/w)	10	12-bit primary context register
secondary_context	ASI	40(r/w)	20	12-bit secondary context register.
nucleus_context	ASI	40(r/w)	30	12-bit nucleus context register.
inst_tmb_base	ASI	3F(r/w)	30	Base address (VA) of the instruction TMB
data_tmb_base	ASI	3F(r/w)	40	Base address (VA) of the data TMB
inst_tlb_match_data	ASI	3F(r/w)	10	Instruction context:va for an MTLB miss
data_tlb_match_data	ASI	3F(r/w)	20	Data context:va for an MTLB miss
tlb_lock_entries	ASI	40(r/w)	50	Entries 0 to (tlb_lock_entries - 1) are locked
tlb_fifo_counter	ASI	40(r/w)	60	MTLB entry number to be written into.
inst_tmb_tag	ASI	39(r)	0	inst_tlb_match_data reformatted for comparison
data_tmb_tag	ASI	3A(r)	0	data_tlb_match_data reformatted for comparison
inst_8KB_tmb_pointer	ASI	3B(r)	0	inst TMB pointer (VA) for 8KB page size
data_8KB_tmb_pointer	ASI	3D(r)	0	data TMB pointer (VA) for 8KB page size
inst_64KB_tmb_pointer	ASI	3C(r)	0	inst TMB pointer (VA) for 64KB page size
data_64KB_tmb_pointer	ASI	3E(r)	0	data TMB pointer (VA) for 64KB page size
asi_scratch_reg	ASI	44 (r/w)	5:3	8 Scratch registers for software use. address [5:3]

F.5 MMU Instructions

The table below lists the instructions associated with MMU processing. STXA/STDFA must be used to send these instructions to MMU and LDXA/LDDFA must be used to read

the information from the MMU. Care should be taken that appropriate memory barriers are inserted or instructions are syncing or serializing when these instructions are issued.

Table 83: MMU Instructions

Instruction	Addr ASI (hex)	VA (hex)	Description
match_and_invalidate_tlb_entry	30(w)	0	Match the context:VA with all entries of the MTLB and invalidate the matched entry. This includes both locked and unlocked entries. All matched entries are invalidated. The context:VA against which match is to be performed is sent as the data portion of the instruction.
invalidate_udtlb	31(w)	0	Invalidates all entries of the µDTLB. Data is undefined.
invalidate_uitlb	32(w)	0	Invalidates all entries of the µITLB. Data is undefined.
write_instr_mtlb_specified_entry	33(w)	11:4	Contents of instr_tlb_match_data register and data specified in the instruction are written into the MTLB entry specified in the VA[11:4].
write_instr_mtlb_fifo_counter	34(w)	0	Contents of instr_tlb_match_data register and data specified in the instruction are written into the MTLB entry specified in the tlb_fifo_counter register.
write_data_mtlb_specified_entry	35(w)	11:4	Contents of data_tlb_match_data register and data specified in the instruction are written into the MTLB entry specified in the VA[11:4].
write_data_mtlb_fifo_counter	36(w)	0	Contents of data_tlb_match_data register and data specified in the instruction are written into the MTLB entry specified in the tlb_fifo_counter register.
read_mtlb_context_va_x	37(r)	11:4	The x-field of the context:va part of the MTLB entry specified in the VA[11:4] is read.[a]
read_mtlb_context_va_y	41(r)	11:4	For diagnostics only. Read the y-field of the MTLB CAM entry specified in the VA[11:4].
read_mtlb_pa_attributes	38(r)	11:4	The PA and attributes part of the MTLB entry specified in the VA[11:4] is read.

a. The information about context:va is stored in the MTLB in a format different than the one specified by the programmer. This is transparent to the software except when reading the contents of the context:va portion of an entry. Each bit of context:va is encoded by 2 bits as follows:

Table 84: x and y fields of CAM

x field	y field	Value to be stored in
0	1	0
1	0	1
0	0	Don't-care (matches or 1)
1	1	Neither 0 nor 1 (testing only)

Example: For 8KB page, we need context[11:0] and VA[40:13], but the context:va field holds context[11:0] and VA[40:12]; VA[12] is a don't care. The don't care state is represented by setting bits in both x and y fields to be 0.

F.6 MMU Exceptions

Table 85 lists the MMU exceptions:

Table 85: MMU Exceptions

Exception Mnemonic	Trap Type	Description
32i_data_access_mmu_miss	0x68-0x6B	PTE is not cached in the MMU.
32i_instruction_access_mmu_miss	0x64-0x67	PTE is not cached in the MMU.
fast_data_access_protection	0x6C-0x6F	Access rights violation. It includes write to clean page
data_access_exception	0x30	ASR 29 specifies the cause of exception.
instruction_access_exception	0x08	PTE cached in MMU. Execute permission is denied.
data_access_error	0x32	Data Fault Access Type Register (ASR 29) specifies the cause of error.
instruction_access_error	0x0A	Instruction Fault Type Register (ASR 24) specifies the cause of error.

F.7 Disable Main and Micro TLB Function

F.7.1 Disable TLB Bits in SCR (ASR31)

The following Disable TLB bits are defined in SCR.

> Bit 29: Disable Main Instruction TLB (D_MITLB)
>
> Bit 27: Disable Micro Instruction TLB (D_UITLB)
>
> Bit 30: Disable Main Data TLB (D_MDTLB)
>
> Bit 28: Disable Micro Data TLB (D_UDTLB)

The table below describes the modes which are specified by the bit. (The table is common to the D_MITLB and D_UITLB bit pair and the D_MDTLB andD_UDTLB bit pair.)

D_MTLB	D_UTLB	Function
0	0	Translation is on, Micro TLB is enabled.
0	1	Translation is on, Micro TLB is disabled. (*For bringup use only*)
1	0	Translation is off, Micro TLB is enabled. (*For bringup use only*)
1	1	Translation is off, Micro TLB is disabled.

The pattern "01" and "10" in the above table should not be used in normal operations. They are provided only for bringup just in case where Main/Micro TLB's have problems in hardware.

F.7.2 Translation Off Mode

When the translation is off, the following things are done by the hardware.

- VA[40:0] is passed to PA[40:0], and VA[63:41] and CONTEXT values are disregarded.

- Memory accesses by instruction fetch and data load/store behave as follows:

  CH (Cacheable) = '0'

  SO (Strong Order) = '1'

  IE (Invert Endianness) = '0'

  NFO (Nonfaulting Only) = '0'

  PROT (SR/SW/SX/UR/UW/UX) = '111111'

in the page table entry.

F.7.3 Notes

The micro TLB's are invisible from the software.

When D_MITLB value is going to be changed, the hardware invalidates all of instructions from I0-Cache and Instruction-Buffer.

On the entry of RED_state, D_MITLB, D_UITLB, D_MDTLB, and D_UDTLB bits in SCR are set to "1" by the hardware.

In RED_state, regardless of D_MITLB, D_UITLB values in SCR, the hardware behaves as if these bits are "1". Therefore the software can't turn on the translation for instruction fetch in RED_state by any means.

It's the software's responsibility of resetting D_MITLB, D_UITLB, D_MDTLB, and D_UDTLB bits in SCR on the exit of or during RED_state if the translation needs to be turned on again.

Any operations other than translations by instruction fetch and data access are not disabled by Main/Micro TLB disable bits. For example, Main TLB write operations or Micro TLB invalidation operations by STXA/STDFA are still operational when Main or Micro TLB are disabled.

F.8 Locking Entries

A mechanism (illustrated in Figure 85) has been provided to lock a set of entries in the MTLB. When the *tlb_lock_entries_reg* is programmed with a value i (0<= i <= 255), 0 to

(i-1) entries are locked. If tlb_lock_entries_reg is 0, no entry is locked. Note that there should be at least one entry that remains unlocked.

The *tlb_fifo_counter_reg* that takes care of the MTLB entry replacement should be programmed such that it does not match an entry that has been locked. When entries are replaced the *tlb_fifo_counter_reg* is incremented till it reaches the max_tlb_entry_count (255 in this case). For the next replacement, the *tlb_fifo_counter_reg* acquires the value programmed in *tlb_lock_entries_reg*.
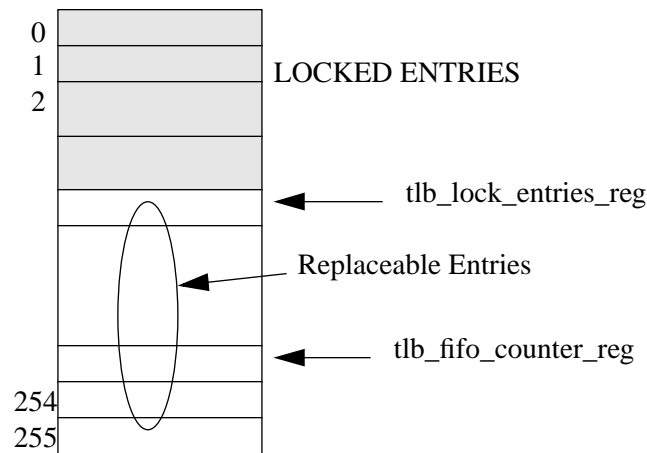
Figure 85: **Locking Entries**

F.9 Data MTLB Miss

This section gives an overview of the Data MTLB Miss operation. The data MTLB miss causes the *32i_data_access_mmu_miss* exception. The trap handler consists of the steps shown in Figure 86:

```
┌──────────────────────┐
│  Load the TMB Pointer │
│    from MMU           │
└──────────────────────┘
           │
┌──────────────────────┐
│ Load the tag for miss ctx:va │
│   from MMU            │
└──────────────────────┘
           │
┌──────────────────────┐
│ load the PTE from memory │
│ using the TMB pointer │
└──────────────────────┘
           │
        ╱  TAG  ╲        NO
       ◇ MATCH? ◇ ───────────────┐
        ╲      ╱                   │
          YES                      │
┌──────────────────────┐           │
│     Insert the PTE    │          │
└──────────────────────┘           │
           │                        │
┌──────────────────────┐           │
│        FLUSH          │          │
└──────────────────────┘           │
           │                        │
┌──────────────────────┐           │
│        RETRY          │          │
└──────────────────────┘           │
                        ┌──────────────────────┐
                        │ Execute the slow handler │
                        └──────────────────────┘
```
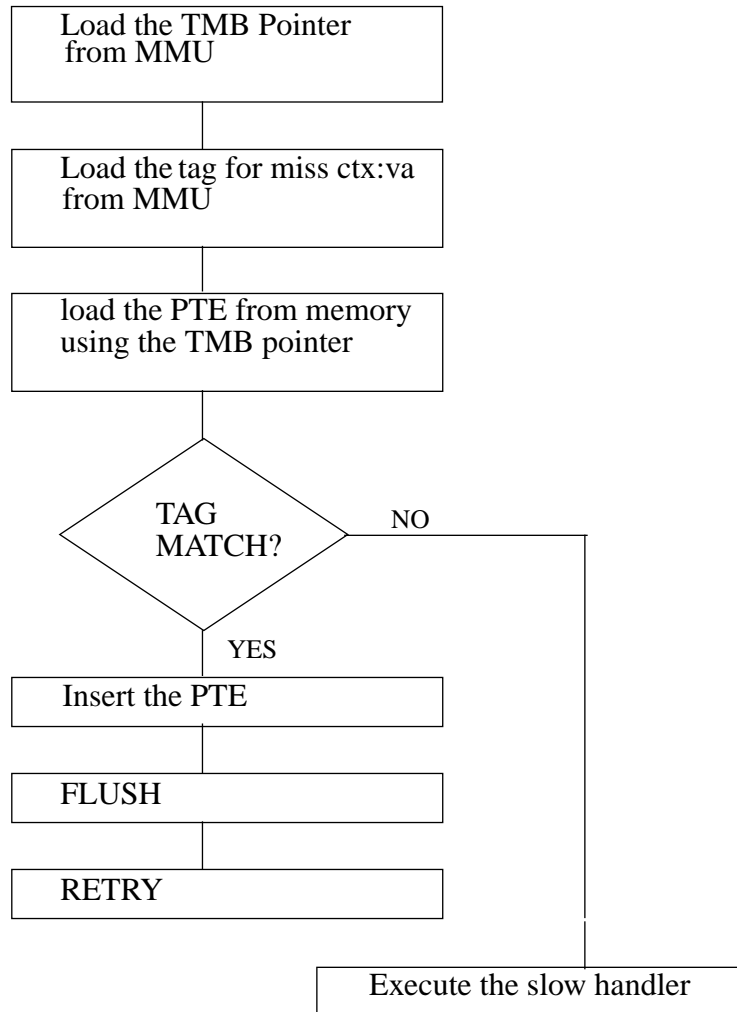
Figure 86: Data MMU Miss Trap Handler

We assume in the above example that we are operating in trap level 0 and have enough ASR/ASI scratch registers available so that a save and restore of registers is not required.

The first step is to load the TMB pointer into an integer register using an MMU ASI instruction. The pointer gives a 16-byte aligned virtual address. For a direct mapped TMB, if the desired PTE is resident in the TMB, it is present in the 16-byte memory space pointed to by this tmb_pointer. Depending on whether the page size is 8KB or 64KB, this pointer is referred to as *data_8KB_tmb_pointer* or *data_64KB_tmb_pointer* respectively in this document.

Formation of the TMB Pointer

When the processor enters the *32i_data_mmu_access_miss* trap handler, the MMU hardware stores the context and the virtual operand address that caused the MMU miss in the *data_tlb_match_data* register. Software has already initialized the *data_tmb_base* register before the first MMU miss is detected. The *data_tmb_base* register contains information about the base address of the TMB (BASE[63:13]), size (N, where N = 0,1..,7) of the TMB and whether it is split or not. The pointer is formed as follows:

If (split == 0)

- data_8KB_tmb_pointer = BASE[63:13+N] [] VA[21+N:13][]0000

- data_64KB_tmb_pointer = BASE[63:13+N][] VA[24+N:16][]0000

If (split == 1)

- data_8KB_tmb_pointer = BASE[63:14+N][] 0 [] VA[21+N:13][]0000

- data_64KB_tmb_pointer = BASE[63:14+N][] 1 [] VA[24+N:16][]0000

The second step is to load the tag associated with the miss VA into an integer register using the MMU ASI instruction to read the data_tmb_tag register.

In the third step, the TMB pointer is used to fetch 16-byte information from memory atomically. The first 8 bytes provide the TAG of the VA and the next 8 bytes the physical address and attributes associated with the VA.

The next step is to compare the tag of the miss VA with the one loaded from memory. The figure below illustrates the formats of the data_tmb_tag register and the tag loaded from memory. The valid bit for the data_tmb_tag_register is always a '1'. To ensure a tag match:

■ The valid bit of the TAG read from memory should be a '1'; that is, the PTE should be valid.

■ The U-bit of the TAG read from memory should be a '0'; that is, the match is performed against both context and VA. This is required as we do not know whether or not the page has the U-bit set when we got a *data_MMU_miss* exception. Since the U-bit is not set more often than it is set, the TLB handler is optimized for U=0 case.

■ Context and VA should match.

V	0	0	CONTEXT[11:0]	0	VA[63:22]

63   62   61   60   59   48   47   42   41   0

data_tmb_tag_register

V	U	0	CONTEXT[11:0]	0	VA[63:22]

63   62   61   60   59   48   47   42   41   0
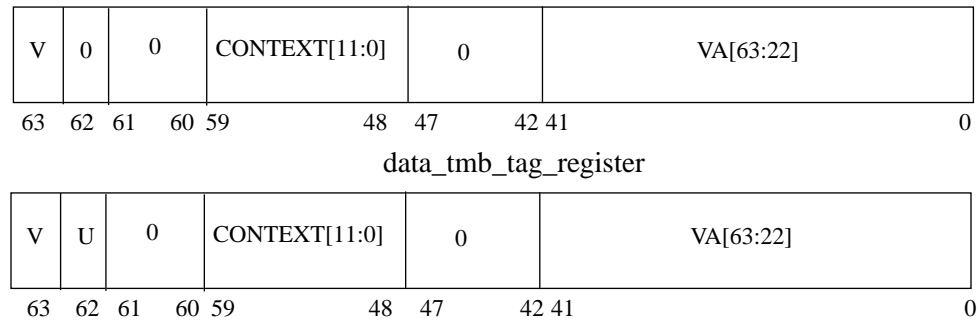
TAG
Table 86: Tag Comparison

If the TAG matches the PTE is inserted in the MTLB using the write_data_mtlb_specified entry or write_data_mtlb_fifo_counter instruction. In the former instruction, the entry to be written into is specified in the instruction itself. In the latter instruction, fifo counter value is used to insert the PTE. The fifo counter is incremented after insertion. The MTLB entries are replaced in FIFO fashion. Therefore, there is no need for software to keep track of the replacement policy.

It should be noted that the miss VA is also present in the ASR28 and context number in ASR29. To assist the software in tablewalk, 4 ASR and 8 ASI registers have been provided.

F.10 Instruction MTLB Miss

The description for handling the instruction MTLB miss is similar to the one for data MTLB miss except that:

1. The word "data" is replaced by "instruction" for instruction and register names.

2. Miss VA can be read out from the TPC. The program was executing in primary context if the trap level is 0 (TL = 0) and in nucleus context if the trap level is greater than 0 (TL > 0). The context number can be read using the *read_primary_context_register* instruction if the program was executing with TL=0 or *read_nucleus_context_register* instruction if the program was executing with TL>0.

F.11 Programming Notes

The software should not alter the Primary Context Register when the instruction MTLB is enabled and the trap level is "0". In other words, the software should not alter the Primary Context Register when the register is being used to fetch instructions. If it is altered in this condition, an unpredictable result is produced.

382 **F MMU Architecture**

The software should not alter the Nucleus Context Register when the instruction MTLB is enabled and the trap level is not "0". In other words, the software should not alter the Nucleus Context Register when the register is being used to fetch instructions. If it is altered in this condition, an unpredictable result is produced.

When the software is writing a context register using the following instructions,

```
STXA ASI=40 VA=0x10 (Write Primary Context Register)
STXA ASI=40 VA=0x20 (Write Secondary Context Register)
STXA ASI=40 VA=0x30 (Write Nucleus Context Register)
```

each has to be followed by

```
A pair of TN and DONE instructions,  OR
A pair of TN and RETRY instructions, OR
FLUSH instruction,                   OR
MEMBAR#Sync instruction,          OR
WR %ASR31<1> = 1 (Invalidate I0)
     (<- normally not recommended due to performance reason.)
```

to make the effects visible to the following instruction fetch, and has to be followed by TN instruction or any other syncing instruction to make the effects visible to the following data access.

When the software is writing a new MTLB entry into MTLB using the following instructions:

```
STXA ASI=33 (Write Instruction Main TLB Into Specified Entry)
STXA ASI=34 (Write Instruction Main TLB Into FIFO Entry)
STXA ASI=35 (Write Data Main TLB Into Specified Entry)
STXA ASI=36 (Write Data Main TLB Into FIFO Entry)
```

each has to be followed by:

```
A pair of TN and DONE instructions,  OR
A pair of TN and RETRY instructions, OR
FLUSH instruction,                   OR
MEMBAR#Sync instruction,       OR
WR %ASR31<1> = 1 (Invalidate I0)
     (<- normally not recommended due to performance reason.)
```

to make the effects visible to the following instruction fetch, and has to be followed by a TN instruction or any other syncing instruction to make the effects visible to the following data access.

When the software is invalidating or modifying a MTLB entry which already exists in MTLB using the following instructions,

STXA ASI=30 (Match And Invalidate TLB Entry)

STXA ASI=33 (Write Instruction Main TLB Into Specified Entry)

STXA ASI=34 (Write Instruction Main TLB Into FIFO Entry)

STXA ASI=35 (Write Data Main TLB Into Specified Entry)

STXA ASI=36 (Write Data Main TLB Into FIFO Entry)

the software should do the following steps.

1) Invalidate or modify TMB in the memory

2) Invalidate or modify MTLB

3) WRASR %31 with bit-1 = 1 (Invalidate I0)

to make the effects visible to the following instruction fetch and data access.

F.12 MMU Reference

F.12.1 ASI MMU Registers

F.12.1.1 PRIMARY_CONTEXT Register

Address $ASI = 40_{16}$, $VA = 10_{16}$
Access Modes: Supervisor read/write.
Function: CONTEXT[11:0].
Data Format: Reserved bits ignored on write, read as '0'.

Reserved	CONTEXT
63 12	11 0
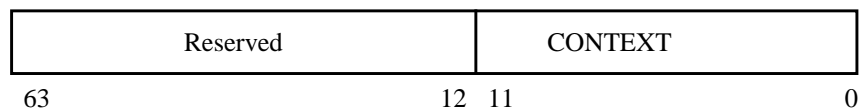
F.12.1.2 SECONDARY_CONTEXT Register

Address: $ASI = 40_{16}$, $VA = 20_{16}$
Access Modes: Supervisor read/write.
Function: CONTEXT[11:0].
Data Format: Reserved bits ignored on write, read as '0'.

Reserved	CONTEXT
63 12	11 0

F.12.1.3 NUCLEUS_CONTEXT Register

Address: $ASI = 40_{16}$, $VA = 30_{16}$
Access Modes: Supervisor read/write.
Function: CONTEXT[11:0].
Data Format: Reserved bits ignored on write, read as '0'..

Reserved	CONTEXT
63 12	11 0

F.12.1.4 INST_TMB_BASE Register

Address: $ASI = 3F_{16}$, $VA = 30_{16}$

Access Modes: Supervisor read/write.

Function: Contains information about an instruction Translation Memory Buffer (TMB) entry.

Data Format: Reserved bits ignored on write, read as '0'. If SPLIT = 1, the lower half of the TMB is 8K pages, and the upper half is 64K pages. TMB_BASE points to the base of the TMB structure in memory..

TMB_BASE	SPLIT	RESERVED	SIZE
63	13 12	11	3 2 0

SIZE	# of TMB Entries
000	512
001	1024
010	2K
011	4K
100	8K
101	16K
110	32K
111	64K

Table 87: TMB Sizes

F.12.1.5 DATA_TMB_BASE Register

Address $ASI = 3F_{16}$, $VA = 40_{16}$

Access Modes: Supervisor read/write.

Function: Contains information about a data TMB entry.

Data Format: Reserved bits ignored on write, read as '0'. If SPLIT = 1, the lower half of the TMB is 8K pages, and the upper half is 64K pages. TMB_BASE points to the base of the TMB structure in memory.

TMB_BASE	SPLIT	RESERVED	SIZE
63	13 12	11	3 2 0

Table 88: TMB Sizes

SIZE	# of TMB ENTRIES
000	512
001	1024
010	2K
011	4K
100	8K
101	16K
110	32K
111	64K

F.12.1.6 INST_TLB_MATCH_DATA Register

Address: $ASI = 3F_{16}$, $VA = 10_{16}$

Access Modes: Supervisor read/write. Also loaded by hardware when a trap for instruction MTLB miss is taken

Function: Data used for matching an entry in the TLB.

Data Format:

CONTEXT[11:0]	VA[63:12]

63   52 51   0
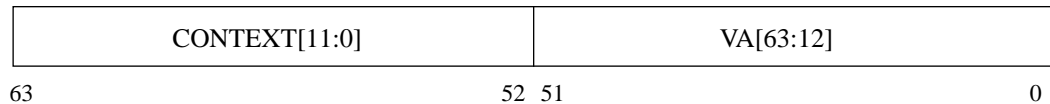
F.12.1.7 DATA_TLB_MATCH_DATA Register

Address: $ASI = 3F_{16}$, $VA = 20_{16}$

Access Modes: Supervisor read/write. Also loaded by hardware when a trap for data MTLB miss is taken.

Function: Data used for matching an entry in the TLB.

Data Format:

CONTEXT[11:0]	VA[63:12]

63   52 51   0

F.12.1.8 TLB_LOCK_ENTRIES Register

Address: $ASI = 40_{16}$, $VA = 50_{16}$

Access Modes: Supervisor read/write

Function: Number of TLB entries to lock.(e.g. '8' means entries 0-7 are "locked"). This register is initialized by hardware to the value of 0xFF. See also TLB_FIFO_COUNTER_REG below.

Data Format: Reserved bits ignored on write, read as '0'.

Reserved	Number_of_entries

63   8 7   0
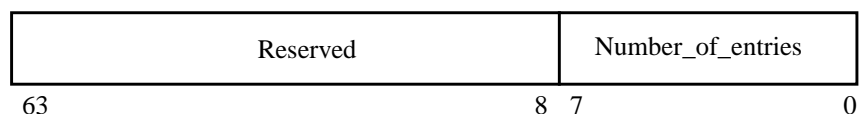
F.12.1.9 TLB_FIFO_COUNTER Register

Address: $ASI = 40_{16}$, $VA = 60_{16}$

Access Modes: Supervisor read/write

Function: This register is initialized to 0xff by hardware. It increments by one when the **WRITE_TLB_ENTRY** register is written to with the V bit set to 1, except when wrapping from 0xff. Instead of wrapping to 0x0, it loads the value from the TLB_LOCK_ENTRIES Register. Software should make sure that this TLB_FIFO_COUNTER Register is greater than the TLB_LOCK_ENTRIES Register when changing the TLB_LOCK_ENTRIES value.

Data Format: Reserved bits ignored on write, read as '0'.

Reserved	TLB Address
63 8	7 0
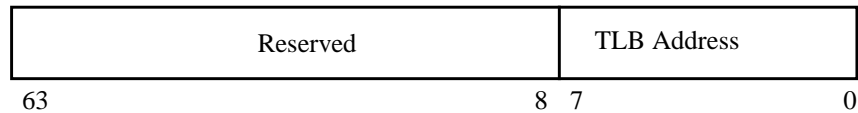
F.12.1.10 INST_TMB_TAG Register (Not A Real Register)

Address: $ASI = 39_{16}$, $VA = 0$

Access Modes: Supervisor read

Function: Used to assist in indexing instruction TMB.

Data Format:

VALID	0	CONTEXT	0	VATAG
63	62 60	59 48	47 42	41 0
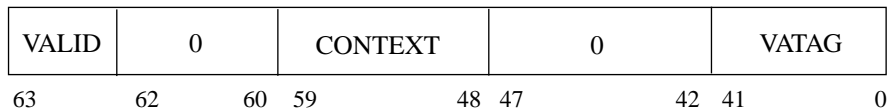
F.12.1.11 DATA_TMB_TAG Register (Not A Real Register)

Address: $ASI = 3A_{16}$, $VA = 0$

Access Modes: Supervisor read

Function: Used to assist in indexing data TMB.

Data Format:

VALID	0	CONTEXT	0	VATAG
63	62 60	59 48	47 42	41 0
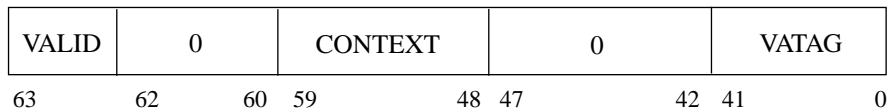
F.12.1.12 INST_8KB_TMB_POINTER Register (Not A Real Register)

Address: ASI = $3B_{16}$, VA = 0

Access Modes: Supervisor read

Function: Pointer to the entry in TMB that may contain the translation for 8KB pages for an instruction MTLB miss.

Data Format:

VA
63   0
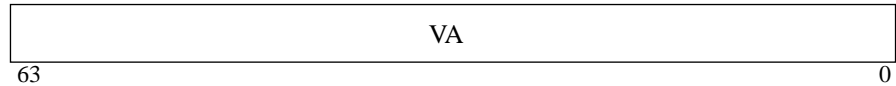
F.12.1.13 DATA_8KB_TMB_POINTER Register (Not A Real Register)

Address: ASI = $3D_{16}$, VA=0

Access Modes: Supervisor read

Function: Pointer to the entry in TMB that may contain the translation for 8KB pages for a data MTLB miss.

Data Format:

VA
63   0
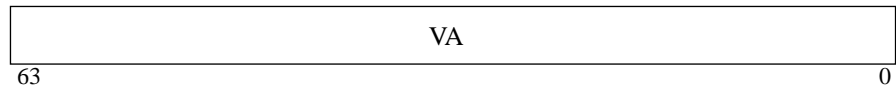
F.12.1.14 INST_64KB_TMB_POINTER Register (Not A Real Register)

Address: ASI = $3C_{16}$, VA = 0

Access Modes: Supervisor read

Function: Pointer to the entry in TMB that may contain the translation for 64KB pages for an instruction MTLB miss.

Data Format: Reserved bits ignored on write, read as '0'.

VA
63   0
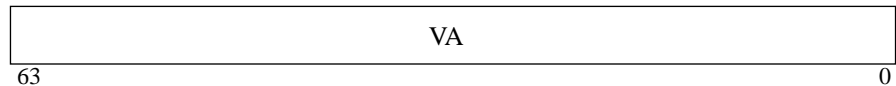
F.12.1.15 DATA_64KB_TMB_POINTER Register (Not A Real Register)

Address: ASI = $3E_{16}$, VA = 0

Access Modes: Supervisor read

Function: Pointer to the entry in TMB that may contain the translation for 64KB pages for a data MTLB miss.

Data Format:

VA
63   0
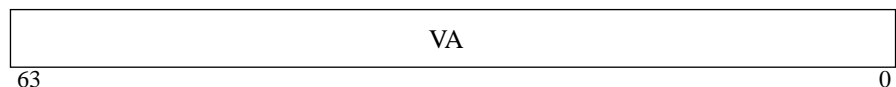
F.12.1.16   ASI Scratch Registers 0-7

Address: ASI = 44_{16}, VA[5:3] = ASI Scratch Register Number (0-7). Other VA bits have to be zero.

Access Modes: Supervisor read/write

Function: Scratch registers.

Data Format:

DATA
63   0

F.12.2   ASI MMU Instructions

These instructions are mapped into the ASI_MMU address space. They are executed for their "side-effects" rather than loading/storing a specific register in the TR.

F.12.2.1   MATCH_AND_INVALIDATE_TLB_ENTRY Instruction

Address: ASI = 30_{16}, VA = 0

Access Modes: Supervisor write only.

Function: Match the context:va with all the entries of MTLB and invalidate the matched entries. This includes both locked and unlocked entries. If more than one entries match, it is a hardware or software bug.

Data Format:

Context[11:0]	VA[63:12]
63   52 51   0
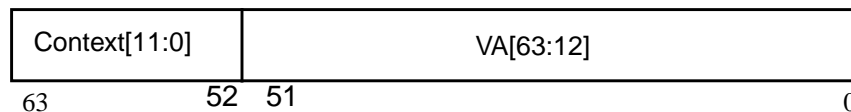
F.12.2.2   INVALIDATE_UDTLB Instruction

Address: ASI = 31_{16}, VA = 0

Access Modes: Supervisor write only.

Function: Invalidate all the data micro TLB entries.

Data Format:

undefined
63   0
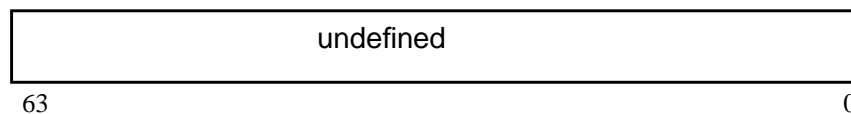
F.12.2.3 INVALIDATE_UITLB Instruction

Address: ASI = 32_{16}, VA = 0

Access Modes: Supervisor write only.

Function: Invalidate all the instruction micro TLB entries.

Data Format:

undefined

63   0
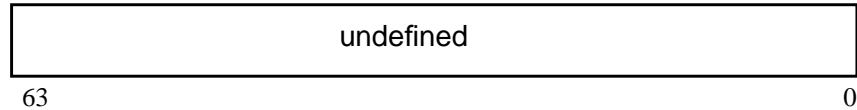
F.12.2.4 WRITE_INSTR_MTLB_SPECIFIED_ENTRY Instruction

Address: ASI = 33_{16}, VA = [11:4]

Access Modes: Supervisor write only.

Function: Inserts the TLB entry specified in the VA[11:4]. Contents of the instr_tlb_match_data register specify the context:va. The data field of the instruction specifies the PA and attributes associated with it.

Data Format:

see TLB entry format

63   0
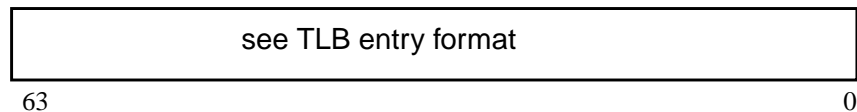
F.12.2.5 WRITE_INSTR_MTLB_FIFO_COUNTER Instruction

Address: ASI = 34_{16}, VA = 0

Access Modes: Supervisor write only.

Function: Inserts the TLB entry specified in the fifo_counter_register. Contents of the instr_tlb_match_data register specify the context:va. The data field of the instruction specifies the PA and attributes associated with it.

Data Format:

see TLB entry format

63   0
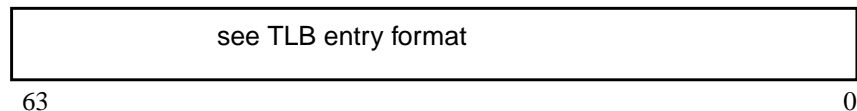
F.12.2.6 WRITE_DATA_MTLB_SPECIFIED_ENTRY Instruction

Address: ASI = 35_{16}, VA = [11:4]

Access Modes: Supervisor write only.

Function: Inserts the TLB entry specified in the VA[11:4]. Contents of the data_tlb_match_data register specify the context:va. The data field of the instruction specifies the PA and attributes associated with it.

Data Format:

see TLB entry format

63   0
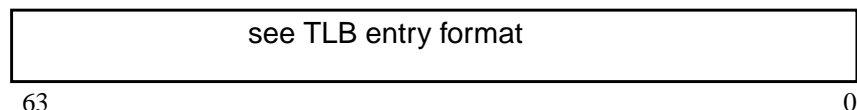
F.12.2.7 WRITE_DATA_MTLB_FIFO_COUNTER Instruction

Address: ASI = 36_{16}, VA = 0

Access Modes: Supervisor write only.

Function: Inserts the TLB entry specified in the fifo_counter_register. Contents of the data_tlb_match_data register specify the context:va. The data field of the instruction specifies the PA and attributes associated with it.

Data Format:

see TLB entry format

63   0
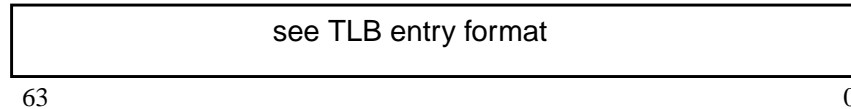
F.12.2.8 READ_MTLB_CONTEXT_VA_X Instruction

Address: ASI = 37_{16}, VA = [11:4]

Access Modes: Supervisor read only.

Function: Read the x-field of the context:va portion of the MTLB entry specified in the VA[11:4].

Data Format:

Context[11:0]	VA[63:12]

63   52 51   0
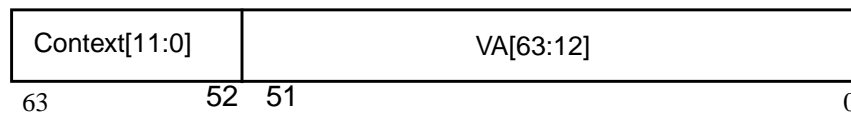
F.12.2.9 READ_MTLB_CONTEXT_VA_Y Instruction

Address: ASI = 41_{16}, VA = [11:4]

Access Modes: Supervisor read only.

Function: Read the y-field of the context:va portion of the MTLB entry specified in the VA[11:4].

Data Format:

Context[11:0]	VA[63:12]

63   52 51   0
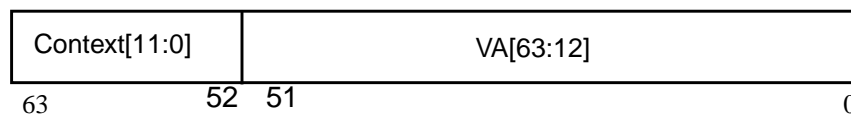
F.12.2.10 READ_MTLB_PA_ATRIBUTES Instruction

Address: $ASI = 38_{16}$, VA = [11:4]

Access Modes: Supervisor read only.

Function: Read the PA and attributes of the MTLB entry specified in the VA[11:4].

Data Format:

DATA

V1	V0	CPRTY	RPRTY	—	U	SIZE[3:0]	IE	SO	—	PA[40:12]	NFO	CH	—	PROT[5:0]
63	62	61 58	57 56	55 54	53	52 49	48	47	46 41	40 12	11	10	9 6	5 0
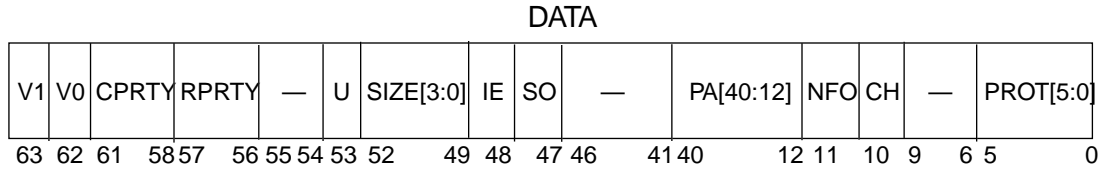
V1: valid bit in Main TLB
V0: valid bit copy in Main TLB
CPRTY[3]: odd parity bit for CONTEXT_X[11:0] and VA_X[63:44] in Main TLB
CPRTY[2]: odd parity bit for VA_X[43:12] in Main TLB
CPRTY[1]: odd parity bit for CONTEXT_Y[11:0] and VA_Y[63:44] in Main TLB
CPRTY[0]: odd parity bit for VA_Y[43:12] in Main TLB
RPRTY[1]: odd parity bit for RSV[1:0], U, SIZE[3:0], IE, SO, PA[40:36] in Main TLB
RPRTY[0]: odd parity bit for PA[35:12], NFO, CH, PROT[5:0] in Main TLB
Other fields: see TLB entry format

G 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 SPARC64-III assembler for the convenience of assembly language programmers.

G.1 Notation Used

The notations defined here are also used in the syntax descriptions in Appendix A.

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 SPARC64-III 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.

G.1.1 Register Names

***reg*:**

A *reg* is an integer register name. It may have any of the following values:[1]

```
%r0..%r31
%g0..%g7      (global registers; same as %r0..%r7)
%o0..%o7      (out    registers; same as %r8 ..%r15)
%l0..%l7      (local  registers; same as %r16..%r23)
%i0..%i7      (in     registers; same as %r24..%r31)
%fp           (frame pointer; conventionally same as %i6)
%sp           (stack pointer; conventionally same as %o6)
```

---

1. In actual usage, the `%sp`, `%fp`, `%g`*n*, `%o`*n*, `%l`*n*, and `%i`*n* forms are preferred over `%r`*n*.

Subscripts identify the placement of the operand in the binary instruction as one of the following:

reg_{rs1} (*rs1* field)
reg_{rs2} (*rs2* field)
reg_{rd} (*rd* field)

freg:

An *freg* is a floating-point register name. It may have the following values:

`%f0, %f1, %f2 .. %f63`   See 5.1.4, "Floating-point Registers"

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

$freg_{rs1}$ (*rs1* field)
$freg_{rs2}$ *(rs2* field)
$freg_{rs3}$ *(rs3* field)
$freg_{rd}$ (*rd* field)

asr_reg:

An *asr_reg* is an Ancillary State Register name. It may have one of the following values:

`%asr16..%asr31`

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

asr_reg_{rs1} (*rs1* field)
asr_reg_{rd} (*rd* field)

i_or_x_cc:

An *i_or_x_cc* specifies a set of integer condition codes**,** those based on either the 32-bit result of an operation (`icc`) or on the full 64-bit result (`xcc`). It may have either of the following values:

`%icc`
`%xcc`

fccn:

An *fccn* specifies a set of floating-point condition codes. It may have any of the following values:

`%fcc0`
`%fcc1`
`%fcc2`
`%fcc3`

G.1.2 Special Symbol Names

Certain special symbols appear in the syntax table in `typewriter font`. They must be written exactly as they are shown, including the leading percent sign (`%`).

The symbol names and the registers or operators to which they refer are as follows:

%asi	Address Space Identifier register
%canrestore	Restorable Windows register
%cansave	Savable Windows register
%cleanwin	Clean Windows register
%cwp	Current Window Pointer register
%fsr	Floating-point State Register
%otherwin	Other Windows register
%pc	Program Counter register
%pil	Processor Interrupt Level register
%pstate	Processor State register
%tba	Trap Base Address register
%tick	Tick (cycle count) register
%tl	Trap Level register
%tnpc	Trap Next Program Counter register
%tpc	Trap Program Counter register
%tstate	Trap State register
%tt	Trap Type register
%ccr	Condition Codes Register
%fprs	Floating-point Registers State register
%ver	Version register
%wstate	Window State register
%y	Y register

The following special symbol names are unary operators that perform the functions described:

%uhi	Extracts bits 63..42 (high 22 bits of upper word) of its operand
%ulo	Extracts bits 41..32 (low-order 10 bits of upper word) of its operand
%hi	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 as follows:

#n_reads	0	(for PREFETCH instruction)
#one_read	1	(for PREFETCH instruction)
#n_writes	2	(for PREFETCH instruction)
#one_write	3	(for PREFETCH instruction)
#page	4	(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   08₁₆        (for MEMBAR instruction mmask field)
        #LoadStore    04₁₆        (for MEMBAR instruction mmask field)
        #StoreLoad    02₁₆        (for MEMBAR instruction mmask field)
        #LoadLoad     01₁₆        (for MEMBAR instruction mmask field)
        #ASI_AIUP     10₁₆        ASI_AS_IF_USER_PRIMARY
        #ASI_AIUS     11₁₆        ASI_AS_IF_USER_SECONDARY
        #ASI_AIUP_L   18₁₆        ASI_AS_IF_USER_PRIMARY_LITTLE
        #ASI_AIUS_L   19₁₆        ASI_AS_IF_USER_SECONDARY_LITTLE
        #ASI_P        80₁₆        ASI_PRIMARY
        #ASI_S        81₁₆        ASI_SECONDARY
        #ASI_PNF      82₁₆        ASI_PRIMARY_NOFAULT
        #ASI_SNF      83₁₆        ASI_SECONDARY_NOFAULT
        #ASI_P_L      88₁₆        ASI_PRIMARY_LITTLE
        #ASI_S_L      89₁₆        ASI_SECONDARY_LITTLE
        #ASI_PNF_L    8A₁₆        ASI_PRIMARY_NOFAULT_LITTLE
        #ASI_SNF_L    8B₁₆        ASI_SECONDARY_NOFAULT_LITTLE
```

The full names of the ASIs may also be defined:

```
        #ASI_AS_IF_USER_PRIMARY                10₁₆
        #ASI_AS_IF_USER_SECONDARY              11₁₆
        #ASI_AS_IF_USER_PRIMARY_LITTLE         18₁₆
        #ASI_AS_IF_USER_SECONDARY_LITTLE 19₁₆
        #ASI_PRIMARY                           80₁₆
        #ASI_SECONDARY                         81₁₆
        #ASI_PRIMARY_NOFAULT                   82₁₆
        #ASI_SECONDARY_NOFAULT                 83₁₆
        #ASI_PRIMARY_LITTLE                    88₁₆
        #ASI_SECONDARY_LITTLE                  89₁₆
        #ASI_PRIMARY_NOFAULT_LITTLE            8A₁₆
        #ASI_SECONDARY_NOFAULT_LITTLE          8B₁₆
```

G.1.3 Values

Some instructions use operand values as follows:

const4	A constant that can be represented in 4 bits
const22	A constant that can be represented in 22 bits
imm_asi	An alternate address space identifier (0..255)
simm7	A signed immediate constant that can be represented in 7 bits
simm10	A signed immediate constant that can be represented in 10 bits
simm11	A signed immediate constant that can be represented in 11 bits
simm13	A signed immediate constant that can be represented in 13 bits
value	Any 64-bit value
shcnt32	A shift count from 0..31

> *shcnt64*   A shift count from 0..63

G.1.4 Labels

A label is a sequence of characters that comprises alphabetic letters (a–z, A–Z [with upper and lower case distinct]), underscores (_), dollar signs ($), periods (.), and decimal digits (0-9). A label may contain decimal digits, but it may not begin with one. A local label contains digits only.

G.1.5 Other Operand Syntax

Some instructions allow several operand syntaxes, as follows:

reg_plus_imm may be any of the following:

> reg_{rs1} (equivalent to reg_{rs1} + %g0)
> reg_{rs1} + *simm13*
> reg_{rs1} − *simm13*
> *simm13*   (equivalent to %g0 + *simm13*)
> *simm13* + reg_{rs1} (equivalent to reg_{rs1} + *simm13*)

address may be any of the following:

> reg_{rs1} (equivalent to reg_{rs1} + %g0)
> reg_{rs1} + *simm13*
> reg_{rs1} − *simm13*
> *simm13*   (equivalent to %g0 + *simm13*)
> *simm13* + reg_{rs1} (equivalent to reg_{rs1} + *simm13*)
> reg_{rs1} + reg_{rs2}

membar_mask is the following:

> *const7*   A constant that can be represented in 7 bits. Typically, this is an expression involving the logical **or** of some combination of #Lookaside, #MemIssue, #Sync, #StoreStore, #LoadStore, #StoreLoad, and #LoadLoad.

prefetch_fcn (prefetch function) may be any of the following:

> #n_reads
> #one_read
> #n_writes
> #one_write
> #page
> 0..31

regaddr (register-only address) may be any of the following:

> reg_{rs1} (equivalent to reg_{rs1} + %g0)
> reg_{rs1} + reg_{rs2}

***reg_or_imm* (register or immediate value) may be either of:**

> *reg$_{rs2}$*
> *simm13*

***reg_or_imm10* (register or immediate value) may be either of:**

> *reg$_{rs2}$*
> *simm10*

***reg_or_imm11* (register or immediate value) may be either of:**

> *reg$_{rs2}$*
> *simm11*

***reg_or_shcnt* (register or shift count value) may be any of:**

> *reg$_{rs2}$*
> *shcnt32*
> *shcnt64*

***software_trap_number* may be any of the following:**

> *reg$_{rs1}$*   (equivalent to *reg$_{rs1}$* + %g0)
> *reg$_{rs1}$* + *simm7*
> *reg$_{rs1}$* − *simm7*
> *simm7*   (equivalent to %g0 + *simm7*)
> *simm7* + *reg$_{rs1}$*   (equivalent to *reg$_{rs1}$* + *simm7*)
> *reg$_{rs1}$* + *reg$_{rs2}$*

> The resulting operand value (software trap number) must be in the range 0..127, inclusive.

G.1.6 Comments

Two types of comments are accepted by the SPARC64-III assembler: C-style "/*...*/" comments, which may span multiple lines, and "!..." comments, which extend from the "!" to the end of the line.

G.2 Syntax Design

The SPARC64-III assembly language syntax is designed so that

■ The destination operand (if any) is consistently specified as the last (rightmost) operand in an assembly language instruction.

■ A reference to the **contents** of a memory location (in a Load, Store, CASA, CASXA, LDSTUB(A), or SWAP(A) instruction) is always indicated by square brackets ([]); a reference to the **address** of a memory location (such as in a JMPL, CALL, or SETHI) is specified directly, without square brackets.

G.3 Synthetic Instructions

Table 89 describes the mapping of a set of synthetic (or "pseudo") instructions to actual instructions. These synthetic instructions are provided by the SPARC64-III assembler for the convenience of assembly language programmers.

Note: Synthetic instructions should not be confused with "pseudo-ops," which typically provide information to the assembler but do not generate instructions. Synthetic instructions always generate instructions; they provide more mnemonic syntax for standard SPARC64-III instructions.

Table 89: Mapping Synthetic to SPARC64-III Instructions (V9=43)

Synthetic Instruction		SPARC64-III Instruction(s)		Comment
`cmp`	reg_{rs1}, reg_or_imm	`subcc`	reg_{rs1}, reg_or_imm, `%g0`	compare
`jmp`	*address*	`jmpl`	*address*, `%g0`	
`call`	*address*	`jmpl`	*address*, `%o7`	
`iprefetch` *label*		`bn,a,pt`	`%xcc`,*label*	instruction prefetch
`tst`	reg_{rs1}	`orcc`	`%g0`, reg_{rs1}, `%g0`	test
`ret`		`jmpl`	`%i7+8`, `%g0`	return from subroutine
`retl`		`jmpl`	`%o7+8`, `%g0`	return from leaf subroutine
`restore`		`restore`	`%g0`, `%g0`, `%g0`	*trivial* `restore`
`save`		`save`	`%g0`, `%g0`, `%g0`	*trivial* `save` (*Warning: trivial* `save` *should only be used in kernel code!*)
`setuw`	*value*, reg_{rd}	`sethi`	`%hi(`*value*`)`, reg_{rd} — **or** — `or` `%g0`, *value*, reg_{rd} — **or** — `sethi` `%hi(`*value*`)`, reg_{rd}; `or` reg_{rd}, `%lo(`*value*`)`, reg_{rd}	(when ((value&$3FF_{16}$) = = 0)) (*when* $0 \leq value \leq 4095$) (otherwise) *Warning: do not use* `setuw` *in the delay slot of a DCTI.*
`set`	*value*, reg_{rd}			synonym for `setuw`
`setsw`	*value*, reg_{rd}	`sethi` `%hi(`*value*`)`, reg_{rd} — **or** — `or` `%g0`, *value*, reg_{rd} — **or** — `sethi` `%hi(`*value*`)`, reg_{rd} `sra` reg_{rd}, `%g0`, reg_{rd} — **or** — `sethi` `%hi(`*value*`)`, reg_{rd}; `or` reg_{rd}, `%lo(`*value*`)`, reg_{rd} — **or** — `sethi` `%hi(`*value*`)`, reg_{rd}; `or` reg_{rd}, `%lo(`*value*`)`, reg_{rd} `sra` reg_{rd}, `%g0`, reg_{rd}		(when (value> = 0) and ((value & $3FF_{16}$) = = 0)) (*when* $-4096 \leq value \leq 4095$) (otherwise, if (value < 0) and ((value & $3FF_{16}$) = = 0)) (otherwise, if value> = 0) (otherwise, if value<0) *Warning: do not use* `setsw` *in the delay slot of a CTI.*
`setx`	*value*, *reg*, reg_{rd}	`sethi` `%uhi(`*value*`)`, *reg* `or` *reg*, `%ulo(`*value*`)`, *reg* `sllx` *reg*,`32`,*reg*		create 64-bit constant ("reg" is used as a temporary register)

Table 89: **Mapping Synthetic to SPARC64-III Instructions (***Continued***)(V9=43)**

Synthetic Instruction		SPARC64-III Instruction(s)		Comment
		sethi	%hi(*value*), *reg$_{rd}$*	*Note:* setx *optimizations are possible, but not enumerated here. The worst-case is shown.Warning: do not use* setx *in the delay slot of a CTI.*
		or	*reg$_{rd}$*, *reg*, *reg$_{rd}$*	
		or	*reg$_{rd}$*, %lo(*value*), *reg$_{rd}$*	
signx	*reg$_{rs1}$*, *reg$_{rd}$*	sra	*reg$_{rs1}$*, %g0, *reg$_{rd}$*	sign-extend 32-bit value to 64 bits
signx	*reg$_{rd}$*	sra	*reg$_{rd}$*, %g0, *reg$_{rd}$*	
not	*reg$_{rs1}$*, *reg$_{rd}$*	xnor	*reg$_{rs1}$*, %g0, *reg$_{rd}$*	one's complement
not	*reg$_{rd}$*	xnor	*reg$_{rd}$*, %g0, *reg$_{rd}$*	one's complement
neg	*reg$_{rs2}$*, *reg$_{rd}$*	sub	%g0, *reg$_{rs2}$*, *reg$_{rd}$*	two's complement
neg	*reg$_{rd}$*	sub	%g0, *reg$_{rd}$*, *reg$_{rd}$*	two's complement
cas	[*reg$_{rs1}$*], *reg$_{rs2}$*, *reg$_{rd}$*	casa	[*reg$_{rs1}$*]#ASI_P, *reg$_{rs2}$*, *reg$_{rd}$*	compare and swap
casl	[*reg$_{rs1}$*], *reg$_{rs2}$*, *reg$_{rd}$*	casa	[*reg$_{rs1}$*]#ASI_P_L, *reg$_{rs2}$*, *reg$_{rd}$*	compare and swap, little-endian
casx	[*reg$_{rs1}$*], *reg$_{rs2}$*, *reg$_{rd}$*	casxa	[*reg$_{rs1}$*]#ASI_P, *reg$_{rs2}$*, *reg$_{rd}$*	compare and swap extended
casxl	[*reg$_{rs1}$*], *reg$_{rs2}$*, *reg$_{rd}$*	casxa	[*reg$_{rs1}$*]#ASI_P_L, *reg$_{rs2}$*, *reg$_{rd}$*	compare and swap extended, little-endian
inc	*reg$_{rd}$*	add	*reg$_{rd}$*, 1, *reg$_{rd}$*	increment by 1
inc	*const13*, *reg$_{rd}$*	add	*reg$_{rd}$*, *const13*, *reg$_{rd}$*	increment by const13
inccc	*reg$_{rd}$*	addcc	*reg$_{rd}$*, 1, *reg$_{rd}$*	incr by 1; set icc & xcc
inccc	*const13*, *reg$_{rd}$*	addcc	*reg$_{rd}$*, *const13*, *reg$_{rd}$*	incr by const13; set icc & xcc
dec	*reg$_{rd}$*	sub	*reg$_{rd}$*, 1, *reg$_{rd}$*	decrement by 1
dec	*const13*, *reg$_{rd}$*	sub	*reg$_{rd}$*, *const13*, *reg$_{rd}$*	decrement by const13
deccc	*reg$_{rd}$*	subcc	*reg$_{rd}$*, 1, *reg$_{rd}$*	decr by 1; set icc & xcc
deccc	*const13*, *reg$_{rd}$*	subcc	*reg$_{rd}$*, *const13*, *reg$_{rd}$*	decr by const13; set icc & xcc
btst	*reg_or_imm*, *reg$_{rs1}$*	andcc	*reg$_{rs1}$*, *reg_or_imm*, %g0	bit test
bset	*reg_or_imm*, *reg$_{rd}$*	or	*reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$*	bit set
bclr	*reg_or_imm*, *reg$_{rd}$*	andn	*reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$*	bit clear
btog	*reg_or_imm*, *reg$_{rd}$*	xor	*reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$*	bit toggle
clr	*reg$_{rd}$*	or	%g0, %g0, *reg$_{rd}$*	clear (zero) register
clrb	[*address*]	stb	%g0, [*address*]	clear byte
clrh	[*address*]	sth	%g0, [*address*]	clear halfword
clr	[*address*]	stw	%g0, [*address*]	clear word
clrx	[*address*]	stx	%g0, [*address*]	clear extended word
clruw	*reg$_{rs1}$*, *reg$_{rd}$*	srl	*reg$_{rs1}$*, %g0, *reg$_{rd}$*	copy and clear upper word
clruw	*reg$_{rd}$*	srl	*reg$_{rd}$*, %g0, *reg$_{rd}$*	clear upper word
mov	*reg_or_imm*, *reg$_{rd}$*	or	%g0, *reg_or_imm*, *reg$_{rd}$*	
mov	%y, *reg$_{rd}$*	rd	%y, *reg$_{rd}$*	
mov	%asr*n*, *reg$_{rd}$*	rd	%asr*n*, *reg$_{rd}$*	
mov	*reg_or_imm*, %y	wr	%g0, *reg_or_imm*, %y	
mov	*reg_or_imm*, %asr*n*	wr	%g0, *reg_or_imm*, %asr*n*	

H Software Considerations

 Consult V9 for the text of this appendix.

I Extending the SPARC-V9 Architecture

Consult V9 for the text of this appendix.

J Programming With the Memory Models

V9 *Consult V9 for the text of this appendix.*

K Changes From SPARC-V8 to SPARC-V9

Consult V9 for the text of this appendix.

L ASI Assignments

L.1 Introduction

Every load or store address in a SPARC V9 processor has an 8-bit Address Space Identifier (ASI) appended to the VA. The VA plus the ASI fully specify the address. For instruction loads and for 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. In practice, ASIs are not only used to differentiate address spaces but are used for other functions like referencing registers in the MMU unit.

L.2 ASI Assignments

For SPARC64-III all accesses made with ASI values in the range $00_{16}..7F_{16}$ when PSTATE.PRIV = 0 will cause a *privileged_action* exception.

Warning:
> The software should follow the ASI assignments and VA assignments in the following SPARC64-III ASI assignments. Some illegal ASI or VA accesses will cause the machine to enter unknown states.

Table 90: SPARC64-III **ASI Assignments**

ASI$_{16}$	R/W	VA$_{16}$	Description	Reference
00..03	—	—	—	—
04	R/W	—	ASI_NUCLEUS	8.3
05..0B	—	—	—	—
0C	R/W	—	ASI_NUCLEUS_LITTLE	8.3
0D..0F	—	—	—	—
10	R/W	—	ASI_PRIMARY_AS_IF_USER	8.3
11	R/W	—	ASI_SECONDARY_AS_IF_USER	8.3
12..13	—	—	—	—
14	R/W	—	ASI_PHYSICAL_CACHEABLE	L.3.1
15	R/W	—	ASI_PHYSICAL_NONCACHEABLE_STRONG_ORDER	L.3.2
16..17	—	—	—	—
18	R/W	—	ASI_PRIMARY_LITTLE_AS_IF_USER	8.3

Table 90: SPARC64-III **ASI Assignments** (*Continued*)

ASI$_{16}$	R/W	VA$_{16}$	Description	Reference
19	R/W	—	ASI_SECONDARY_LITTLE_AS_IF_USER	8.3
1A..1B	—	—	—	—
1C	R/W	—	ASI_PHYSICAL_CACHEABLE_LITTLE	L.3.3
1D	R/W	—	ASI_PHYSICAL_NONCACHEABLE_LITTLE_STRONG_ORDER	L.3.4
1E..23	—	—	—	—
24	R	—	ASI_NUCLEUS_ATOMIC_QUAD	A.28.1
25..2B	—	—	—	—
2C	R	—	ASI_NUCLEUS_ATOMIC_QUAD_LITTLE	A.28.1
2D..2F	—	—	—	—
30	W	—	ASI_MATCH_AND_INVALIDATE_TLB_ENTRY	F.12.2.1
31	W	—	ASI_INVALIDATE_MICRO_DATA_TLB	F.12.2.2
32	W	—	ASI_INVALIDATE_MICRO_INSTRUCTION_TLB	F.12.2.3
33	W	—	ASI_INSTRUCTION_MAIN_TLB_INTO_SPECIFIED_ENTRY	F.12.2.4
34	W	—	ASI_INSTRUCTION_MAIN_TLB_INTO_FIFO_ENTRY	F.12.2.5
35	W	—	ASI_DATA_MAIN_TLB_INTO_SPECIFIED_ENTRY	F.12.2.6
36	W	—	ASI_DATA_MAIN_TLB_INTO_FIFO_ENTRY	F.12.2.7
37	R	—	ASI_MAIN_TLB_CONTEXT_AND_VA_X	F.12.2.8
38	R	—	ASI_MAIN_TLB_PA_AND_ATTRIBUTES	F.12.2.10
39	R	—	ASI_INSTRUCTION_TMB_TAG_REGISTER	F.12.1.10
3A	R	—	ASI_DATA_TMB_TAG_REGISTER	F.12.1.11
3B	R	—	ASI_INSTRUCTION_8KB_TMB_POINTER	F.12.1.12
3C	R	—	ASI_INSTRUCTION_64KB_TMB_POINTER	F.12.1.14
3D	R	—	ASI_DATA_8KB_TMB_POINTER	F.12.1.13
3E	R	—	ASI_DATA_64KB_TMB_POINTER	F.12.1.15
3F	R/W	10	ASI_INSTRUCTION_TLB_MATCH_DATA_REGISTER	F.12.1.6
		20	ASI_DATA_TLB_MATCH_DATA_REGISTER	F.12.1.7
		30	ASI_INSTRUCTION_TMB_BASE_REGISTER	F.12.1.4
		40	ASI_DATA_TMB_BASE_REGISTER	F.12.1.5
40	R/W	10	ASI_PRIMARY_CONTEXT_REGISTER	F.12.1.1
		20	ASI_SECONDARY_CONTEXT_REGISTER	F.12.1.2
		30	ASI_NUCLEUS_CONTEXT_REGISTER	F.12.1.3
		50	ASI_TLB_LOCK_ENTRY_REGISTER	F.12.1.8
		60	ASI_TLB_FIFO_COUNTER	F.12.1.9
41	R	—	ASI_MAIN_TLB_CONTEXT_AND_VA_Y	F.12.1.9
42..43	—	—	—	—
44	R/W	—	ASI_SCRATCH_REGISTER	F.12.1.16
45..47	—	—	—	—
48	R	—	ASI_INTERRUPT_VECTOR_DISPATCH_STATUS_REGISTER	N.4.2
49	R/W	—	ASI_INTERRUPT_VECTOR_RECEIVE_STATUS_REGISTER	N.4.3, N.4.4
4A	R/W	—	ASI_UPA_CONFIGURATION_REGISTER	R.4.1
4B..5F	—	—	—	—

Table 90: SPARC64-III **ASI Assignments** (*Continued*)

ASI$_{16}$	R/W	VA$_{16}$	Description	Reference
60	R/W	—	ASI_PRIMARY_STRONG_ORDER_RESTRICTED	L.3.5
61..67	—	—	—	—
68	R/W	0*	ASI_U2_CACHE_TAG	M.7.1
		1*	ASI_U2_CACHE_DATA	M.7.2
	W	2*	ASI_FLUSH_U2_CACHE_TO_MEMORY_USING_U2_CACHE_INDEX	M.7.3
		3*	ASI_INVALIDATE_I1_CACHE_USING_I1_CACHE_INDEX_AND_WAY	M.7.4
		4*	ASI_INVALDATE_D1_CACHE_USING_D1_CACHE_INDEX_AND_WAY	M.7.5
	R/W	6*	ASI_U2_CACHE_BUFFER_REGISTER_0	M.7.7
		7*	ASI_U2_CACHE_BUFFER_REGISTER_1	M.7.8
69	W	—	ASI_FLUSH_U2_CACHE_TO_MEMORY_USING_PA	M.7.9
6a	W	—	ASI_FLUSH_L1_CACHE_TO_U2_CACHE_USING_U2_CACHE_INDEX	M.7.10
6b	R	—	ASI_SRAM_CONFIGURATION_REGISTER	M.7.11
6c	R/W	—	ASI_TDU_ERROR_LOG_REGISTER	P.7.1
6d	R/W	—	ASI_ICU_ERROR_LOG_REGISTER	P.7.2
6e	R/W	—	ASI_DC_ERROR_LOG_REGISTER	P.7.3
6f	R/W	—	ASI_UC_ECC_ERROR_INJECTION_REGISTER	P.7.4
70 - 76	—	—	—	—
77	W	40	ASI_OUTGOING_INTERRUPT_VECTOR_DATA_REGISTER_0	N.4.5
		50	ASI_OUTGOING_INTERRUPT_VECTOR_DATA_REGISTER_1	
		60	ASI_OUTGOING_INTERRUPT_VECTOR_DATA_REGISTER_2	
		70	ASI_DISPATCH_INTERRUPT_VECTOR	N.4.1
78-7e	—	—	—	—
7F	R	40	ASI_INCOMING_INTERRUPT_VECTOR_DATA_REGISTER_0	N.4.6
		50	ASI_INCOMING_INTERRUPT_VECTOR_DATA_REGISTER_1	
		60	ASI_INCOMING_INTERRUPT_VECTOR_DATA_REGISTER_2	
80	R/W	—	ASI_PRIMARY	8.3
81	R/W	—	ASI_SECONDARY	
82	R	—	ASI_PRIMARY_NO_FAULT	
83	R	—	ASI_SECONDARY_NO_FAULT	
84-87	—	—	—	—
88	R/W	—	ASI_PRIMARY_LITTLE	8.3
89	R/W	—	ASI_SECONDARY_LITTLE	
8A	R	—	ASI_PRIMARY_NO_FAULT_LITTLE	
8B	R	—	ASI_SECONDARY_NO_FAULT_LITTLE	
8C-E1	—	—	—	—
E2	R/W	—	ASI_PRIMARY_STRONG_ORDER	L.3.6
E3-FF	—	—	—	—

Note:

In the ASI $3F_{16}$, 40_{16}, 77_{16}, $7F_{16}$ definitions, there are sub definitions which are specified by VA[63:0] values.

Note:

In the ASI 68_{16} definitions, there are sub definitions which are specified by VA[30:28] values.

Note:

"Flush U2 Cache To Memory Using U2 Cache Index" (ASI=68_{16}, VA[30:28]=2_{16}) and "Flush U2 Cache To Memory Using PA" (ASI=69_{16}) operations may corrupt system memory coherency if issued during other memory activities, or overlapped with other requests. These operations shall not be used in the system programs. They are only for bringup/diagnostic purposes.

L.3 Special Memory Access ASI's

This section describes special memory access ASI's which are not specified in SPARC-V9.

L.3.1 ASI 14_{16} (ASI_PHYSICAL_CACHEABLE)

When this ASI is specified in any memory access instructions, the following things are done by the hardware.

■ VA[40:0] is passed to PA[40:0], and VA[63:41] and CONTEXT values are disregarded.

■ Memory access behaves as if:

 ● CH (Cacheable) = "1"

 ● SO (Strong Order) = "0"

 ● NFO (Nonfaulting Only) = "0"

 ● PROT (SR/SW/SX/UR/UW/UX) = "111111"

 ● Big Endian

Even if D_MDTLB (Disable Main Data TLB) bit in SCR is set, the access with this ASI is still a cacheable access.

L.3.2 ASI 15_{16} (ASI_PHYSICAL_NONCACHEABLE_STRONG_ORDER)

When this ASI is specified in any memory access instructions, the following things are done by the hardware.

■ VA[40:0] is passed to PA[40:0], and VA[63:41] and CONTEXT values are disregarded.

■ Memory access behaves as if:

 ● CH (Cacheable) = "0"

 ● SO (Strong Order) = "1"

 ● NFO (Nonfaulting Only) = "0"

- PROT (SR/SW/SX/UR/UW/UX) = "111111"

- Big Endian

L.3.3 ASI 1C$_{16}$ (ASI_PHYSICAL_CACHEABLE_LITTLE)

When this ASI is specified in any memory access instructions, the following things are done by the hardware.

■ VA[40:0] is passed to PA[40:0], and VA[63:41] and CONTEXT values are disregarded.

■ Memory access behaves as if:

- CH (Cacheable) = "1"

- SO (Strong Order) = "0"

- NFO (Nonfaulting Only) = "0"

- PROT (SR/SW/SX/UR/UW/UX) = "111111"

- Little Endian

L.3.4 ASI 1D$_{16}$ (ASI_PHYSICAL_NONCACHEABLE_LITTLE_STRONG_ORDER)

When this ASI is specified in any memory access instructions, the following things are done by the hardware.

■ VA[40:0] is passed to PA[40:0], and VA[63:41] and CONTEXT values are disregarded.

■ Memory access behaves as if:

- CH (Cacheable) = "0"

- SO (Strong Order) = "1"

- NFO (Nonfaulting Only) = "0"

- PROT (SR/SW/SX/UR/UW/UX) = "111111"

- Little Endian

L.3.5 ASI 60$_{16}$ (ASI_PRIMARY_STRONG_ORDER_RESTRICTED)

When this ASI is specified in any memory access instructions, the following things are done by the hardware.

■ Primary Context is used for the address translation.

■ The access is strongly ordered regardless of load or store, and the memory model.

L.3.6   ASI E2$_{16}$ (ASI_PRIMARY_STRONG_ORDER)

When this ASI is specified in any memory access instructions, the following things are done by the hardware.

■ Primary Context is used for the address translation.

■ The access is strongly ordered regardless of load or store, and the memory model.

M Cache Organization

M.1 Introduction

All caches in the CPU keep one cache line size, that is, 64 bytes. All caches in the CPU keep *inclusive relations* by hardware, that is, if a cache line resides in Level-0, it must reside in Level-1 and Level-2. Once a cache line is evicted from Level-2 Cache, the hardware is responsible for flush-and-invalidating Level-1 and Level-0 Caches.

M.2 Level-0 Instruction Cache (I0 Cache)

The I0 Cache is virtually-indexed virtually-tagged (VIVT), 16 KBytes direct-mapped cache. An I0 cache tag contains virtual address and context-ID. An I0 Cache hit occurs when both virtual address and context match.

The I0 Cache can *not* be bypassed. When the Invalidate_I0 bit in ASR31 is set, the whole I0 cache is invalidated.

I0 Cache Tag and I0 Cache Data are parity-protected.

M.3 Level-1 Instruction Cache (I1 Cache)

The I1 Cache is a virtually-indexed physically-tagged (VIPT), 64 KBytes 4-way set-associative cache.

Instruction fetches bypass the I1 Cache when they are noncacheable accesses. Noncacheable accesses are specified under the conditions A OR B OR C, where:

A. The processor is in RED_state.

B. D_MITLB bit in SCR(ASR31) is set.

C. The page table entry for the address specifies noncacheable memory.

The I1 Cache Tag is parity-protected and the I1 Cache Data is ECC-protected.

M.4 Level-1 Data Cache (D1 Cache)

The D1 Cache is a virtually-indexed physically-tagged (VIPT), write-back, allocating-on-write-miss, 64 KBytes 4-way set-associative cache.

Data accesses bypass the D1 Cache when they are noncacheable accesses. Noncacheable accesses are specified under the conditions A AND (B OR C OR D), where:

A. The ASI used for the access is neither ASI_PHYSICAL_CACHEABLE (ASI 14_{16}) nor ASI_PHYSICAL_CACHEABLE_LITTLE (ASI $1C_{16}$).

B. The ASI used for the access is either ASI_PHYSICAL_NONCACHEABLE_STRONG_ORDER (15_{16}) or ASI_PHYSICAL_NONCACHEABLE_LITTLE_STRONG_ORDER (ASI $1D_{16}$).

C. D_MDTLB bit in the SCR (ASR31) is set.

D. The page table entry for the address specifies a noncacheable memory location.

The D1 Cache Tag is parity-protected and the D1 Cache Data is ECC-protected.

M.5 Level-2 External Unified Cache (U2 Cache)

The CPU's level-2 External Cache is a physically-indexed physically-tagged (PIPT), unified, write-back, allocating, direct-mapped cache of variable size (1M, 2M, 4M, 8M, 16M). The U2 Cache has no references to virtual address and context information.

When the access is noncacheable, the U2 Cache is bypassed. The U2 Cache Tag is parity-protected and U2 Cache Data is ECC-protected.

M.6 Cache Coherency Protocols

The CPU uses the UPA MOESI Cache coherence protocol; these letters are acronyms for cache line states as follows:

M	Exclusive Modified
O	Shared Modified (owned)
E	Exclusive clean
S	Shared clean
I	Invalid

A subset of the MOESI protocol is used in the I1-Cache, the D1-Cache, and D-Tag in the system controller. The table below shows the relations between the protocols.

U2-Cache	D-Tag in SC	D1-Cache	I1-Cache
Invalid (I)	Invalid (I)	Invalid (I)	Invalid (I)
Shared Clean (S)	Shared Clean (S)	Clean (C)	Clean (C)
	Shared Modified (O)		
Shared Modified (O)	Shared Modified (O)		
Exclusive Clean (E)	Exclusive Modified (M)		
Exclusive Modified (M)		Clean (C)	
		Exclusive Modified (M)	

The table below shows the encoding of the MOESI states in the U2 Cache.

Bit 2 (Valid)	Bit 1 (Exclusive)	Bit 0 (Modified)	State
0	—	—	Invalid (I)
1	0	0	Shared Clean (S)
1	1	0	Exclusive Clean (E)
1	0	1	Shared Modified (O)
1	1	1	Exclusive Modified (M)

M.7 ASI Cache Instructions

Several ASI instructions are defined to manipulate the I1, D1, and U2 Caches

The following are common to all of the ASI instructions defined in this section:

1. The opcode of the instructions should be either `ldx(a)`, `lddf(a)`, `stx(a)`, or `stdf(a)`. Otherwise, a *data_access_exception* with FTYPE=F_{16} (Invalid ASI) is taken.

2. No address translation is performed for these instructions.

3. VA[3:0] of all of the instructions should be 0. Otherwise, a *mem_not_aligned* trap is taken.

4. The don't-care bits (described as '—' in the format) in VA can be of any value. But it is recommended that the software use zero for these bits.

5. The don't-care bits (described as '—' in the format) in DATA are read as zero and ignored on write.

6. The instruction operations are not affected by PSTATE.CLE. They are always treated as in a big endian mode.

7. The instructions do not cause the processor to sync.

8. The instructions are all strongly ordered regardless of load or store, and the memory model. Therefore, no speculative executions are performed.

9. In the following cases, some of the U2 Cache INDEX bits are ignored and treated as zero by the hardware depending on U2 Cache size.

  ● Read/Write U2 Cache Tag (ASI=68_{16}, VA[30:28]=0)

  ● Read/Write U2 Cache Data To/From U2 Cache Buffer Registers (ASI=68_{16}, VA[30:28]=001_2)

  ● Flush U2 Cache To Memory Using U2 Cache Index (ASI=68_{16}, VA[30:28]=010_2)

  ● Flush L1 Cache To U2 Cache Using U2 Cache Index (ASI=$6A_{16}$)

  When U2 Cache is configured as 1Mb, INDEX[23:20] is ignored and treated as zero.

When U2 Cache is configured as 2Mb, INDEX[23:21] is ignored and treated as zero.

When U2 Cache is configured as 4Mb, INDEX[23:22] is ignored and treated as zero.

When U2 Cache is configured as 8Mb, INDEX[23:23] is ignored and treated as zero.

When U2 Cache is configured as 16Mb, all bits of INDEX[23:6] are used.

The following subsections describe each ASI Cache instruction in detail.

M.7.1 READ/WRITE_U2_CACHE_TAG

Function: Read/ Write from/to the specified line of U2 Cache Tag

ASI: 68_{16}

RW: Supervisor Read, Supervisor Write

VA: See below for details.

—	000	—	INDEX	—	0000
63	31 30 28 27 24	23		6 5 4 3	0
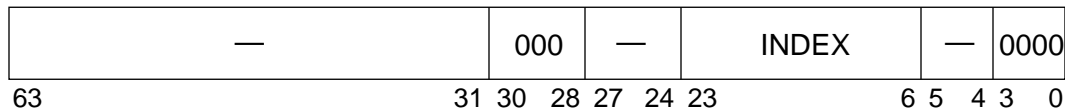
INDEX[23:6]

U2-Cache line index (physical address).

DATA: Below is the format of the U2 Cache Tag:

—	PRTY	DVA	IVA	PA	DWY	IWY	DIN	IIN	MOESI
63 36	35 34	33 32	31 30	29 9	8 7	6 5	4 3	2	0
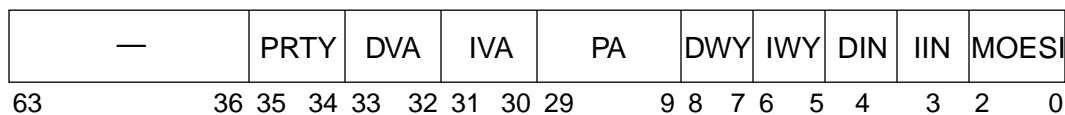
PRTY[1:0]

Parity bits. Bit-1 is the parity for bits 33-17, and bit-0 is the parity for bits 16-0.

DVA[13:12]

D1 Cache VA bits 13-12.

IVA[13:12]

I1 Cache VA bits 13-12.

PA[40:20]

Physical address bits 40-20.

DWY[1:0]

D1 Cache way number.

IWY[1:0]

I1 Cache way number.

DIN

D1 Cache inclusion bit.

IIN

I1 Cache inclusion bit.

MOESI[2:0]

MOESI state of the cache line in U2 Cache. See M.6, "Cache Coherency Protocols" for the encoding.

Note:

During this read operation, the read data of U2 Cache Tag is not parity-checked.

Before executing this instruction, U2-Cache Data ECC check should be disabled (ASR31 bit 50 should be '1'). Otherwise, an *asynchronous_error* trap may be taken, especially if the U2-Cache Data has not been initialized.

M.7.2 Read/Write U2 Cache Data to/from U2 Cache Buffer Registers

Function: Read the specified 16 byte U2 Cache data to U2 Cache Buffer Registers 0 and 1. Write the specified 16 byte U2 Cache data from U2 Cache Buffer Registers 0 and 1. U2 Cache Buffer Register 0 corresponds to the upper 8 bytes (bit 127-64), and U2 Cache Buffer Register 1 corresponds to the lower 8 bytes (bit 63-0).

ASI: 68_{16}

RW: Supervisor Read, Supervisor Write.

VA: See below.

—	001	—	INDEX	OFF SET	0000
63	31 30 28 27 24 23			6 5 4 3	0
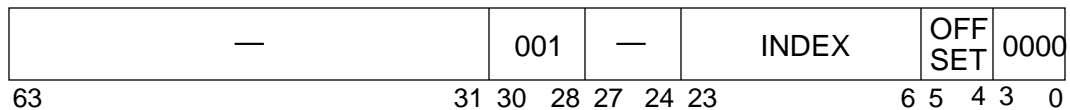
INDEX[23:6]

U2 Cache line index (physical address).

OFFSET[1:0]

Offset in the cache line (equal to PA[5:4]).

DATA: The data in load and store instructions are not used in this operation.

Note:

During the read operation, the read data of U2 Cache Data is not ECC-checked.

M.7.3 Flush U2 Cache To Memory Using U2 Cache Index

Function: Flush the specified U2 Cache data to memory.

ASI: 68_{16}

RW: Supervisor Read (in the read case, it simply behaves as a nop), Supervisor Write.
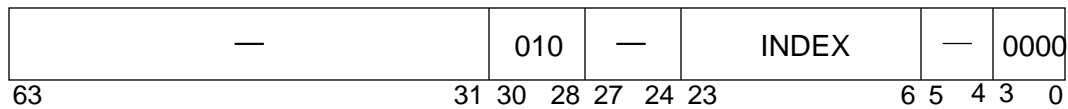
VA: See below.

—	010	—	INDEX	—	0000
63	31 30 28 27 24 23			6 5 4 3 0	

INDEX[23:6]
U2 Cache line index (physical address).

DATA: The data in load and store instructions are not used in this operation.

Note:

This operation may corrupt system memory coherency if issued during other memory activities, or overlapped with other requests. It shall not be used in the system program; it is intended for bringup/diagnostic purposes only.

M.7.4 Invalidate I1 Cache Using I1 Cache Index and Way

Function: Invalidate an I1 Cache line specified by its way number and index. The U2 Cache is not updated by this operation.

ASI: 68_{16}

RW: Supervisor Read (in read case, it simply behaves as a nop), Supervisor Write.

VA: See below.

—	011	—	WAY	—	INDEX	—	0000
63	31 30 28 27 18 17 16 15 14 13					6 5 4 3 0	
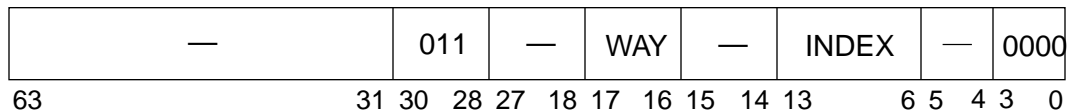
WAY[1:0]
I1 Cache way number.

INDEX[13:6]
I1 Cache line index. (virtual address)

DATA: The data in load and store instructions are not used in this operation.

Note:

This operation corrupts system memory coherency if used in the normal OS environment. This is only for I1 and D1 Cache initialization by the OBP software.

M.7.5 Invalidate D1 Cache Using D1 Cache Index and Way

Function: Invalidate a D1 Cache line specified by its way number and index. U2 Cache is not updated by this operation.

ASI: 68_{16}

RW: Supervisor Read (in read case, it simply behaves as a nop), Supervisor Write.

VA: See below.

—	100	—	WAY	—	INDEX	—	0000

63   31 30   28 27   18 17   16 15   14 13   6 5   4 3   0

WAY[1:0]

D1 Cache way number.

INDEX[13:6]

D1 Cache line index. (virtual address)

DATA: The data in load and store instructions are not used in this operation.

Note:

This operation corrupts system memory coherency if used in the normal OS environment. This is only for I1 and D1 Cache initialization by the OBP software.

M.7.6 Nop

Function: No operation.

RW: Supervisor Read, Supervisor Write.

ASI: 68_{16}

VA: See below.

—		101	—		0000
63		31 30 28	27		4 3 0
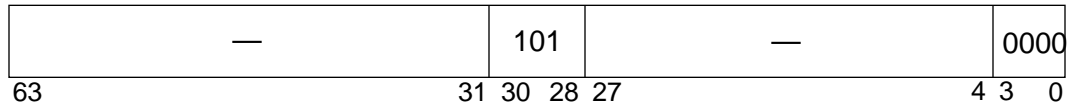
DATA: The data in load and store instructions are not used in this operation.

M.7.7 Read/Write U2 Cache Buffer Register 0

Function: Read/Write U2 Cache Buffer Register 0. Read from U2 Cache Buffer Register 0. Write into U2 Cache Buffer Register 0.

ASI: 68_{16}

RW: Supervisor Read, Supervisor Write.

VA: See below.

—		110	—		0000
63		31 30 28	27		4 3 0
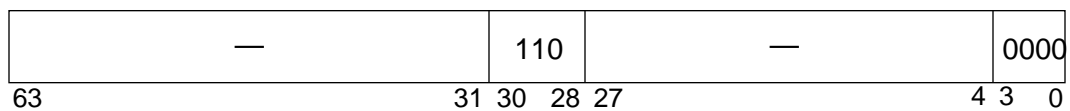
DATA: See below.

DATA
63   0

DATA

Bits[63:0] are the data to be read or written by this operation.

M.7.8 Read/Write U2 Cache Buffer Register 1

Function: Read U2 Cache Buffer Register 1. Write U2 Cache Buffer Register 1.

ASI: 68_{16}

RW: Supervisor Read, Supervisor Write.

VA: See below.

—	111	—	0000
63	31 30 28 27	4 3	0
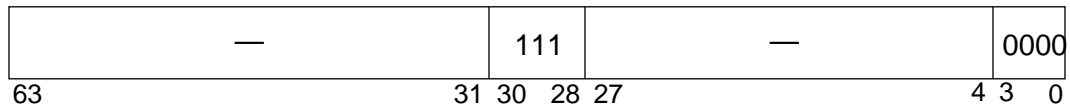
DATA: See below.

DATA
63   0

DATA

Bits [63:0] contain the data to be read or written in this operation.

M.7.9 Flush U2 Cache To Memory Using PA

Function: Flush the specified PA U2 Cache data to the memory.

ASI: 69_{16}

RW: Supervisor Write.

VA: See below.

—	PA	—	0000
63	41 40	6 5 4 3	0
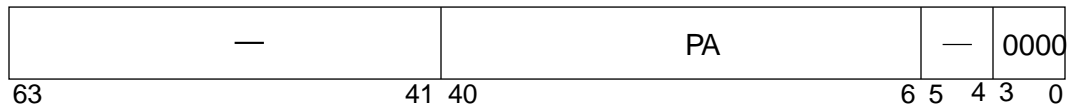
PA[40:6]

Cache line physical address to be flushed

DATA: The data in load and store instruction is not used in this operation.

Note:

This operation may corrupt system memory coherency if issued during other memory activities, or overlapped with other requests. These operations shall not be used in the system program. It's only for bringup/diagnostic purposes.

M.7.10 Flush L1 Cache to U2 Cache Using U2 Cache Index

Function: Flush the specified I1 and D1 Cache line to U2 Cache.

ASI: $6A_{16}$

RW: Supervisor Write.
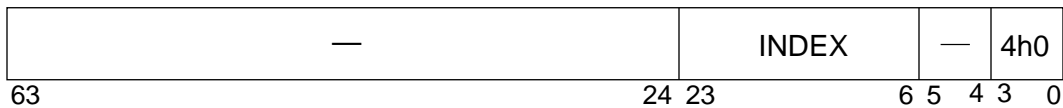
VA: See below.

—		INDEX	—	4h0
63	24 23		6 5 4 3	0

INDEX[23:6]
U2 Cache line index. (physical address)

DATA: The data in load and store instruction is not used in this operation.

M.7.11 Read SRAM Configuration Register

Function: Read SRAM Configuration Register. This register holds the configuration
information for U2 Cache Data SRAM and Tag SRAM. The register is read only;
values are scanned into the register during initialization.

ASI: $6B_{16}$

RW: Supervisor Read.

VA: See below.

—	0000
63	4 3 0

DATA: See below.

—	L2_2BANKS	NEEDS_ DEAD_CYC	UC_DATA_ DLY_SEL	UC_ADDR_ DLY_SEL	SRAM_K_ FREE_RUN	SRAM_ TCYC	TAG_RDY_ LTNCY	DATA_RDY _LTNCY
63 20	19	18	17 10	9 8	7	6	5 3	2 0
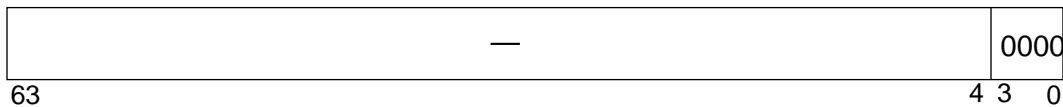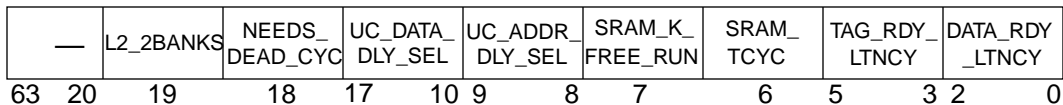
L2_2BANKS
Bank Select.
0: One SRAM data bank.
1: Two SRAM data banks.

NEEDS_DEAD_CYC
Dead Cycle Select.
0: SRAM does not need dead cycle.
1: SRAM needs dead cycle.

UC_DATA_DLY_SEL[7:0]
Max. 8 SRAM's. Each 2 SRAM's are defined below.
UC_DATA_DLY_SEL12[1:0]
UC_DATA_DLY_SEL34[3:2]
UC_DATA_DLY_SEL56[5:3]
UC_DATA_DLY_SEL78[7:6]
00: No data address delay
01: 0.25ns data address delay

10: 0.5ns data address delay

11: 0.75ns data address delay

UC_ADDR_DLY_SEL[1:0]

Address Delay Select.

00: no data address delay

01: 0.25ns data address delay

10: 0.5ns data address delay

11: 0.75ns data address delay

SRAM_K_FREE_RUN

SRAM Free Run Clock Select.

0: Start SRAM clock only for read/writes, then stop clock (logical 0).

1: Free running SRAM clock.

SRAM_TCYC

SRAM Clock Cycle.

1: SRAM Tcyc = 1x CPU Tcyc.

0: SRAM Tcyc = 2x CPU Tcyc.

TAG_RDY_LTNCY[2:0]

Tag ready latency as a multiple of CPU clock cycles.

001: (unused)

010: (unused)

011: 3 cycles

100: 4 cycles

101: 5 cycles

110: 6 cycles

111: 7 cycles

000: 8 cycles

DATA_RDY_LTNCY[2:0]

Data ready latency as a multiple of CPU clock cycles.

001: (unused)

010: (unused)

011: 3 cycles

100: 4 cycles

101: 5 cycles

110: 6 cycles

111: 7 cycles

000: 8 cycles

N Interrupt Handling

N.1 Interrupt Dispatch

When a processor wants to dispatch an interrupt to another UPA port it first sets up the interrupt data registers INTR_DATA_W0, W1, and W2 using ASI instructions. It then sends an INTR_VECTOR_DISPATCH instruction. The interrupt packet and the associated data is forwarded to the target UPA by the system controller. The processor polls the BUSY bit in the INTR_DISPATCH_STATUS register to determine whether the interrupt has been dispatched successfully or not. Below is an example of the interrupt dispatch flow.

```
                              ↓

              read INTR_DISPATCH_STATUS
              if < Busy >     Error
                              ↓
               begin atomic sequence
                  PSTATE.IE <- 0
                  Write INTR_DATA_W0
                  Write INTR_DATA_W1

                  Write INTR_DATA_W2

                  INTR_VECTOR_DISPATCH
                       MEMBAR
                              ↓
                  read INTR_DISPATCH_STATUS
                  if < Busy >  ————
                  PSTATE.IE <- 1
               end atomic sequence
              if < Nack>  ————
              else   dispatch complete

                              ↓
```
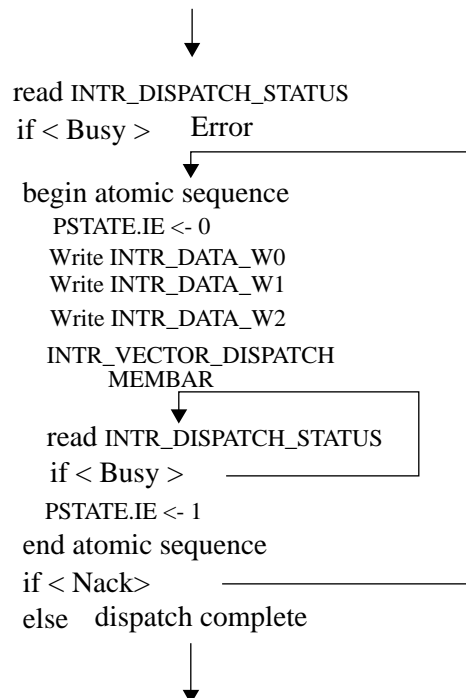
N.2 Interrupt Receive

When an interrupt packet is received, three interrupt data registers are updated with the associated data and the BUSY bit in the INTR_RCV_STATUS is set. If interrupts are enabled, the processor takes a trap and the interrupt data registers are read by the software

to determine the appropriate trap handler. The handler may reprioritize this interrupt packet to a lower priority. Below is an example of the interrupt receive flow.

read INTR_RECEIVE_STATUS
if <! Busy >   Error

Read INTR_DATA_R0
Read INTR_DATA_R1
Read INTR_DATA_R2
Determine Trap Handler

Handle Interrupt
       or
Reprioritize : Set SCHED_INT_REG

CLEAR_INTR_RCV_STATUS

done
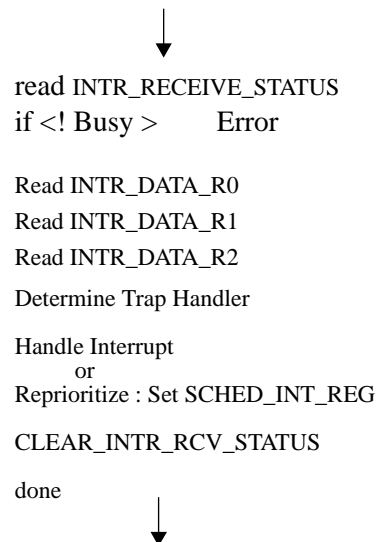
N.3   Interrupt ASI Registers

Interrupt Receive Register

This register reports the status of incoming interrupts.

—	BUSY	MID[4:0]
63	6 5 4	0
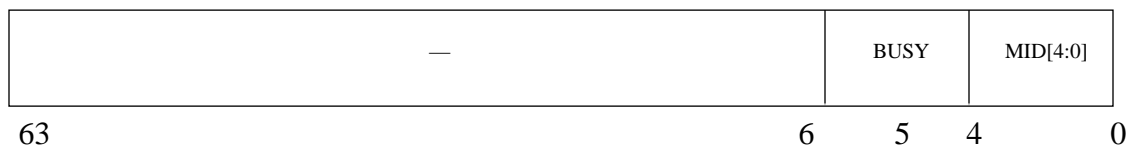
BUSY (read/write)
> This bit is set when an interrupt vector is received. Software has to clear this bit by writing zero.

MID[4:0] (read only)
> Module ID of the interrupter.

Interrupt Vector Dispatch Status Register

This register reports the status of outgoing interrupts.

—	NACK	BUSY
63	2 1	0
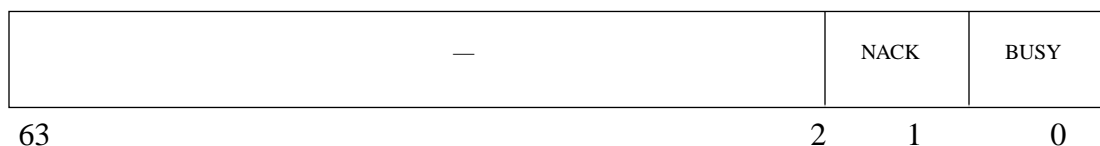
NACK (read only)

   Set when an interrupt dispatch has failed. This is cleared at the start of every inter-
   rupt attempt.

BUSY (read only)

   Set if there is an outstanding dispatch

Interrupt Data Registers

Three interrupt data write registers INTR_DATA_W0-2 and three interrupt data read reg-
isters INTR_DATA_R0-2 are maintained by the UC.

Data

63   0
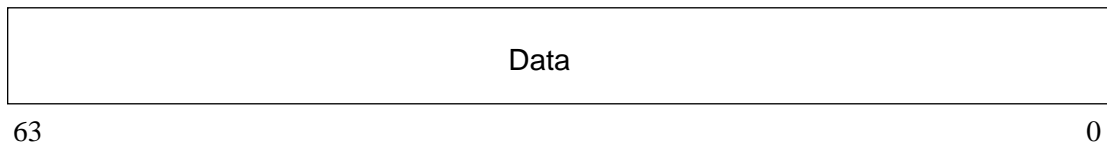
N.4   ASI Instructions for Interrupt Processing

N.4.1   INTR_VECTOR_DISPATCH Instruction

Function: Trigger an interrupt vector dispatch to the target CPU residing at slot MID along
            with the contents of the three interrupt vector data registers

Address: VA<63:19>=don't-care, VA<18:14>=Target-MID, VA<13:6>=don't-care,
            VA<5:0>=0x30 (The software should normally specify all zero for VA<63:19> and
            VA<13:7> and one for VA<6> even though they are don't-care.

	Target-MID	—	0x30

63   19 18   14 13   6 5   0
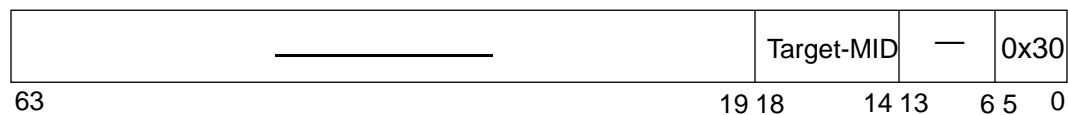
   ASI: 77_{16}
   Instr: STXA
   Data: Not used

N.4.2   READ_INTR_DISPATCH_STATUS Instruction

Function: Read the contents of the Interrupt Vector Dispatch Status register.

Address: VA<63:4>=don't-care, VA<3:0>=0x0 (The software should normally specify all
            zero for VA<63:4> even though they are don't-care.

	0x0

63   4 3   0
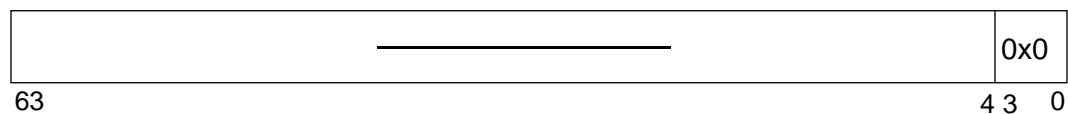
   ASI: 48_{16}
   Instr: LDXA
   Data: See Interrupt Vector Status Register description in N.3, "Interrupt ASI Registers".

N.4.3   READ_INTR_RCV_STATUS Instruction

Function: Returns the contents of the Interrupt Vector Receive Status register

Address: VA<63:4>=don't-care, VA<3:0>=0x0 (The software should normally specify all zero for VA<63:4> even though they are don't-care.

	0x0
63	4 3 0
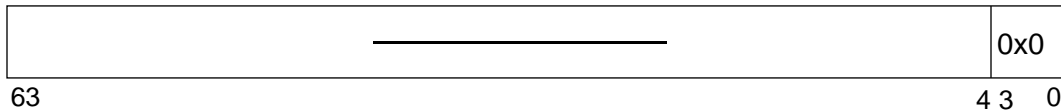
ASI: 49_{16}

Instr: LDXA

Data: See Interrupt Receive Register description in N.3, "Interrupt ASI Registers".

N.4.4   CLEAR_INTR_RCV_STATUS Instruction

Function: When bit 5 of the write data is zero, it clears the busy bit in the Interrupt Vector Receive Status register. When bit 5 of the write data is one, it does nothing.

Address: VA<63:4>=don't-care, VA<3:0>=0x0 (The software should normally specify all zero for VA<63:4> even though they are don't-care.

	0x0
63	4 3 0
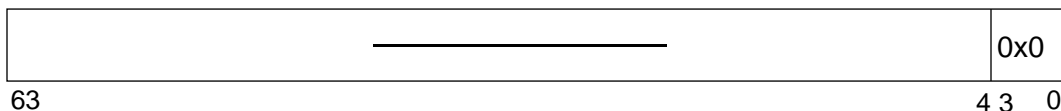
ASI: 49_{16}

Instr: STXA

Data: Only bit 5 is used.

N.4.5   Instructions to Write Interrupt Data Register 0-2

Function: These instructions are used to modify the contents of the three interrupt dispatch data registers, INTR_DATA_W0 to 2.

Address: VA<63:6>=don't-care, VA<5:4>=0x0 for register-0, VA<5:4>=0x1 for register-1, VA<5:4>=0x2 for register-2, VA<3:0>=0x0 (The software should normally specify all zero for VA<63:6> even though they are don't-care.

	0x0 for register-0 0x1 for register-1 0x2 for register-2	0x0
63	6 5 4 3	0
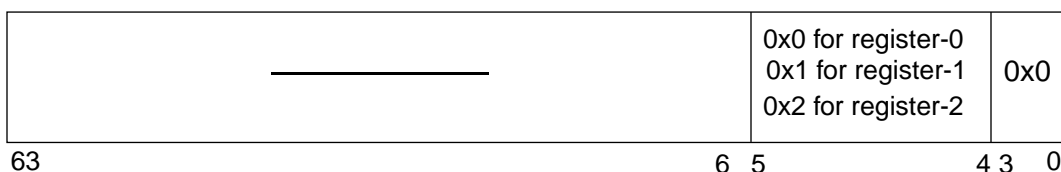
ASI: 77_{16}

Instr: STXA

Data: Data[63:0] to be written to the interrupt data register.

N.4.6   Instructions to Read Interrupt Data Registers

Function: These instructions are used to read the contents of the three interrupt receive data registers: INTR_DATA_R0,1, 2.

Address: VA<63:6>=don't-care, VA<5:4>=0x0 for register-0, VA<5:4>=0x1 for register-1,

VA<5:4>=0x2 for register-2, VA<3:0>=0x0 (The software should normally specify all zero for VA<63:6> even though they are don't-care.

| | 0x0 for register-0
0x1 for register-1
0x2 for register-2 | 0x0 |

63   6 5   4 3   0
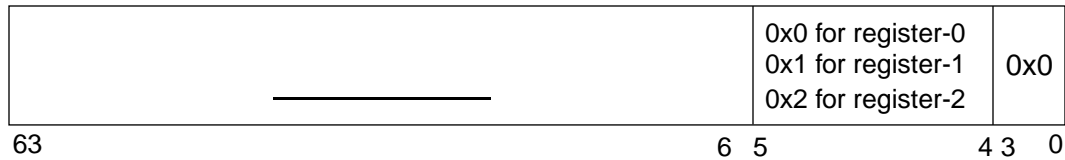
ASI: $7F_{16}$
Instr: LDXA
Data: Data <63:0> to be read from the interrupt data register.

N.5 Interrupt-Related ASR registers

There following subsections describe the ASR registers that are related to interrupt handling.

N.5.1 Set SCHED_INT (ASR20)

WR ASR20 sets bits in the SCHED_INT register. See 5.2.11.3, "Set SCHED_INT Register (ASR20)" for a complete description.

N.5.2 Clear SCHED_INT (ASR21)

WR ASR21 clears bits in SCHED_INT register. See 5.2.11.4, "Clear SCHED_INT Register (ASR21)" for a complete description.

N.5.3 Schedule Interrupt (SCHED_INT) Register (ASR22)

The software uses this register to schedule interrupts. See 5.2.11.5, "Schedule Interrupt (SCHED_INT) Register (ASR22)" for a complete description.

N.5.4 TICK Match Register (ASR23)

This register is compared with the TICK Register, and when they match, an interrupt can be generated. See 5.2.11.6, "TICK Match Register (ASR23)" for a complete description.

O Reset, RED_state, and Error_state

O.1 Reset

O.1.1 Power-on Reset (POR)

The CPU has a Power-On Reset(POR) pin named UPA_RESET_L (asserted low). This pin must be asserted after the power supply reaches full operational voltage. When this signal is asserted, all other resets and traps are ignored. Any pending external transactions are cancelled.

Assertion of POR generates a trap of TT=1 and causes the processor to transfer execution to RSTVaddr + 0x20. This signal must be asserted for at least 4 mS.

POR is supported through Scan by embedded TAP and PROM in the CPU processor module. O.4, "Hardware Power On Reset Sequence" describes this process in detail.

O.1.2 Watchdog Reset (WDR)

Not supported.

SPARC-V9 says that WDR is optional and possibly implementation-dependent.

O.1.3 Externally Initiated Reset (XIR)

The CPU has an Externally Initiated Reset (XIR) pin named UPA_XIR_L (asserted low). This pin must be asserted while the power supply is at full operational voltage and the UPA clock is running.

If TL (Trap Level) < MAXTL (4), the assertion of XIR generates a trap of TT=3 and causes the processor to transfer execution to RSTVaddr + 0x60.

If the processor receives an XIR while TL = 4, it will enter to error_state. The CPU sends P_FERR to UPA, and eventually, the UPA system controller will RESET all processors.

O.1.4 Software Initiated Reset (SIR)

A Software-Initiated RESET is initiated by any processor with a SIR instruction.

If TL (Trap Level) < MAXTL (4), a SIR instruction causes a trap of TT=4 and causes the processor to execute instructions from RSTVaddr + 0x80.

If a processor executes an SIR instruction while TL = 4, it will enter to error_state. The CPU sends P_FERR to the UPA, and eventually, the UPA system controller will RESET all processors.

O.2 RED_state and Error_state

O.2.1 Processor State Transition Diagram

Figure 87 contains the processor state transition diagram.
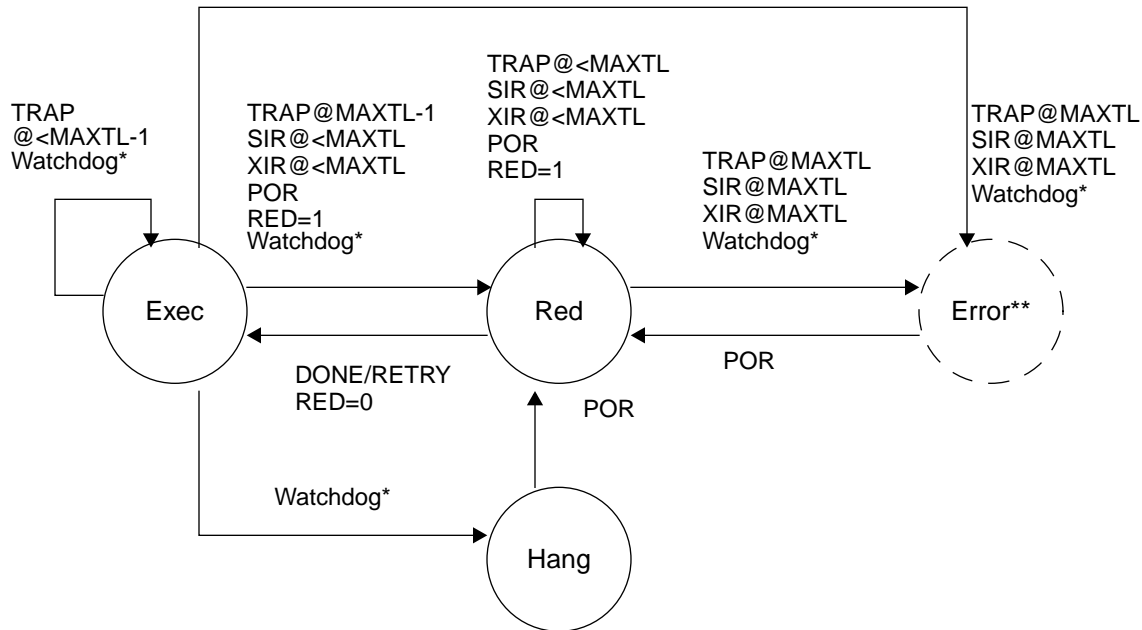
* Behavior on Watchdog trap is dependent on ASR31.
** Error state immediately generates P_FERR and results in POR. Thus the state is transient.

Figure 87: Processor State Diagram

O.2.2 RED_state

Once the processor enters RED_state for any reason except when a power on reset (POR) is performed, the software should not attempt to return to Execute_state; if this is done by software, the state of the processor is unpredictable. The sequence of getting out of RED_state after a power on reset is described in O.4, "Hardware Power On Reset Sequence".

When the processor processes a reset or a trap that enters RED_state, it takes a trap at an offset relative to the RED_state_trap_vector base address (RSTVaddr); in the CPU this is at virtual address VA=$\text{FFFF FFFF F000 0000}_{16}$ and physical address PA = $\text{0000 01FF F000 0000}_{16}$.

The following further describe the processor behavior on the entry of RED_state, and during the RED_state:

■ Whenever the processor enters or exits from RED_state, the I0 cache and prefetch buffers are invalidated.

■ When the processor enters RED_state due to a trap or reset, bit-0 (SM), bit-27 (D_UITLB), bit-28 (D_UDTLB), bit-29 (D_MITLB), and bit-30 (D_MDTLB) in SCR(ASR31) are set by the hardware. It is software's responsibility to reset these bits when required (for example, when the processor exits from RED_state).

■ When the processor enters RED_state *not* due to a trap or reset (that is, when the PSTATE.RED bit has been set using WRPR), all of the SCR register bits are unchanged unlike the case above.

■ When the processor is in RED_state, it behaves as if bit-0 (SM), bit-27 (D_UITLB), and bit-29 (D_MITLB) are set regardless of the actual these values in the SCR register.

O.2.3 Error_state

The processor enters error_state when a trap occurs and TL = MAXTL (4). The SPARC64-III sends P_FERR to the UPA, and eventually, UPA system controller will RESET all processors.

O.3 Processor State after Reset and in RED_state

Table 91 shows the various processor states after the resets and the entry of RED_state.

In this table, it is assumed that RED_state entry happens due to resets or traps. If RED_state entry happens by setting the PSTATE.RED bit using WRPR instruction, no CPU state will be changed except the PSTATE.RED bit itself; the effects of this are described in O.2.2, "RED_state".

Table 91: CPU State after Reset and in RED_state

Name		POR	XIR	SIR	RED_state
Integer registers		Unknown	Unchanged		
Floating Point registers		Unknown	Unchanged		
RSTV value		VA = 0xffff ffff f000 0000 PA = 0x1ff f000 0000			
PC nPC		RSTV | 0x20 RSTV | 0x24	RSTV | 0x60 RSTV | 0x64	RSTV | 0x80 RSTV | 0x84	RSTV | 0xA0 RSTV | 0xA4
PSTATE	AG IE PRIV AM PEF RED MM TLE CLE	1 (Alternate Globals) 0 (Interrupt disable) 1 (Privileged mode) 0 (Full 64-bit address) 1 (FPU on) 1 (Red_state) 00 (LSO) 0 (Trap little endian) 0 (Current little endian)			
TBA[63:15]		Unknown	Unchanged		

Table 91: CPU State after Reset and in RED_state (*Continued*)

Name	POR	XIR	SIR	RED_state
Y	Unknown	Unchanged		
PIL	Unknown	Unchanged		
CWP	Unknown	Unchanged except for register window traps		
TT[TL]	0x1	0x3	0x4	trap_type
CCR	Unknown	Unchanged		
ASI	Unknown	Unchanged		
TL	MAXTL	min(TL+1, MAXTL)		
TPC[TL] TNPC[TL]	Unknown Unknown	PC nPC		
TSTATE CCR ASI PSTATE CWP PC nPC	Unknown	CCR ASI PSTATE CWP PC nPC		
TICK NPT Counter	1 Restart at 0	Unchanged Restart at 0	Unchanged Unchanged (keep counting)	
CANSAVE	Unknown	Unchanged		
CANRESTORE	Unknown	Unchanged		
CANWIN	Unknown	Unchanged		
CELARWIN	Unknown	Unchanged		
WSTATE OTHER NORMAL	Unknown Unknown	Unchanged Unchanged		
VER	0x0004 0x3 mask-dependent 0x4 0x4			
FSR	0	Unchanged		
FPRS	Unknown	Unchanged		
HW MODE TSO (ASR18) PSO RMO BRM DPE	00 (HLSO) 00 (HLSO) 00 (HLSO) 01(2bit br- pred) 00 (Enable DPrefetch)	Unchanged Unchanged Unchanged Unchanged Unchanged		
GSR (ASR19)	Unknown	Unchanged		
SCHED_INT (ASR22)	Unknown	Unchanged		
SW Scratch (ASR25)	Unknown	Unchanged		
Inst Fault Type (ASR24)	Unknown	Unchanged		
Tick Match M_DIS (ASR23) M_VAL	1 Unknown	Unchanged Unchanged		
Data Break Point (ASR26)	Unknown	Unchanged		
Data Fault Address (ASR28)	Unknown	Unchanged		Unchanged or Address of Data fault

Table 91: **CPU State after Reset and in RED_state** (*Continued*)

Name		POR	XIR	SIR	RED_state
Data Fault Type (ASR29)		Unknown	Unchanged		Unchanged or Type of Data fault
Performance (ASR30)		Unknown	Unchanged		
SCR (ASR31)	SM	1 (Sequential Mode)	1 (Sequential Mode)		
	II0	0	Unchanged		
	PM	0	Unchanged		
	TR	0	Unchanged		
	W_SEL	000	Unchanged		
	W_EN/RED	00	Unchanged		
	PM_US	0	Unchanged		
	DB_GSEL	Unknown	Unchanged		
	DB_CSEL	Unknown	Unchanged		
	E_CSE	0	Unchanged		
	D_UITLB	1 (Disable uITLB)	1 (Disable uITLB)		
	D_UDTLB	1 (Disable uDTLB)	1 (Disable uDTLB)		
	D_MITLB	1 (Disable MITLB)	1 (Disable MITLB)		
	D_MDTLB	1 (Disable MDTLB)	1 (Disable MDTLB)		
	D_UAE	1	Unchanged		
	D_ICW3:0	0000	Unchanged		
	D_DCW3:0	0000	Unchanged		
	D_I0P	1	Unchanged		
	D_I1P	1	Unchanged		
	D_D1P	1	Unchanged		
	D_U2P	1	Unchanged		
	D_UITM	1	Unchanged		
	D_UITP	1	Unchanged		
	D_MTP	1	Unchanged		
	D_UPA	1	Unchanged		
	D_I1E	1	Unchanged		
	D_D1E	1	Unchanged		
	D_U2E	1	Unchanged		
	D_AET	1	Unchanged		
	D_ASEET	1	Unchanged		

O.4 Hardware Power On Reset Sequence

When the processor detects assertion of the POR pin, the TAP controller in the processor resets all registers in the scan ring to '0' (by making both scan clock and L2 high). Then the TAP controller reads initialization information from serial PROM and scans it into the scan initialization ring.

Each region of the processor has '0' initialized scan ring(s) called normal scan ring and scan ring(s) which require at least one bit to be set to '1' called scan-init ring. All the scan-init rings are connected into one scan ring within the TAP controller. This is the ring which the TAP controller scans serial PROM data into.

During the process above, all I0 Cache data are invalidated, and the Branch History Table (BHT) and Instruction Lookaside Table (ILT) are initialized. The Micro Instruction TLB, Micro Data TLB, Main TLB, I1-Cache, D1-Cache, and U2-Cache are not invalidated or initialized by power on reset.

The TAP controller then generates a POR trap, depending on which pins are asserted. Since POR has higher priority than XIR, POR trap should be raised when both pins are asserted.

Due to the time limitation available for reset (5mS), the number of bits which can be scanned is limited to around 16Kbits. The processor has been designed to limit the number of bits which must be scan initialized within this limit. The processor has also been designed to be in reset state by the above scan reset sequence.

O.5 Firmware Initialization Sequence

The firmware initialization sequence in this section should be done in the open boot program (OBP) immediately after the hardware power on reset sequence.

When hardware reset is complete, the processor is in RED_state, which disables all Caches (except I0 which can not be disabled) and TLBs. This means VA=RA and UPA accesses can only go to I/O space (since memory access need to be cacheable due to UPA limitation).

With the POR trap, the processor starts fetching instructions from RSTVaddr + 0x20 which is located I/O space and must contain the necessary initialization sequence.

With the hardware reset sequence described in the previous section, only the minimum processor resources are initialized to make it run in RED_state. Following initialization, the firmware should perform the following operations in the sequence shown:

1. TLB Initialization

2. Cache Initialization

3. MSU Initialization

4. Exit from RED_state

O.5.1 TLB Initialization

First both μITLB and μDTLB must be invalidated with a STXA instruction with ASI 30_{16} and 31_{16}.

Next, the Main TLB must be initialized in the following way.

1. Write all 0 value to "Data TLB Match Data Register" using STXA (ASI=$3F_{16}$, VA=20_{16}, "Write Data TLB Match Register").

2. Write all 0 values to all of Main TLB entries (total of 256 entries) using STXA (ASI=35_{16}, "Write Data Main TLB Into Specified Entry). Since this instruction writes only one entry at a time, a write loop is required to initialize all of the Main TLB entries.

Then the minimum Main TLB entries for instruction and data access after exiting RED_state must be written, and TLB Lock Entry Register and TLB FIFO counter register should be properly set.

O.5.2 Cache Initialization

I1 Cache is invalidated by using "Invalidate I1 Cache Using I1 Cache Index and Way" ASI instruction (ASI=68_{16}, VA[30:28]=011_2). Since this instruction invalidates only one cache line entry at a time, for the whole 64KBytes I1 Cache invalidation, it needs to issue 1,024 instructions. Total time required to invalidate I1 is about 15 thousand cycles and is 61.5 uS with 250MHz clock. Similarly, D1 cache is invalidated by using "Invalidate D1 Cache Using D1 Cache Index and Way" ASI instructions (ASI=68_{16}, VA[30:28]=100_2).

U2 Cache is invalidated by writing its tag data with valid bit off into U2 Cache Tag with "Write U2 Cache Tag" ASI instruction(ASI =68_{16}, VA[30:28]=000_2) instruction. Since this instruction writes only one entry at a time (one for every 64 byte cache line), a write loop is required to invalidate the entire U2 Cache. A 1MB U2 Cache requires 16K writes; a 4MB cache requires 64K writes. Each write takes ~15 cycles (since instructions are cached in I0 Cache, I-fetch to PROM occurs only once). Total time required to invalidate all U2 Cache is about 1M cycles in case of 4MB cache and is 4 mS with 250MHz clock.

O.5.3 MSU Initalization

The MSU is initialized with a series of stores, as follows:

Set "Disable U2 Cache Data Error" (D_U2E, bit 50) and "Disable UPA Data Error" (D_UPA, bit 47) in SCR(ASR31). Then do a series of 8-byte stores to all of memory using STXA with ASI=14_{16} to initialize it to zero. Next, reset "Disable U2 Cache Data Error" (D_U2E, bit 50) and "Disable UPA Data Error" (D_UPA, bit 47) in SCR (ASR31).

These stores calculate the correct ECC and so the memory contents are initialized to all-zero values with correct ECC.

It is advisable to set "Enable Next Line Data Prefetch" (DPE) bit in ASR19 during the operations above. This enables next line prefetch and improve speed of writes.

Without the prefetch, cache line (64 byte) read/write from/to MSU takes about 15 UPA cycles (about 200nS). Since writing zeroes into a single cache line requires refill and later write back, speed of initialization is about 64 byte/ 400nS = 160MB/S. This is about 7 Seconds/GB.

With the prefetch, it automatically fetches next cache line when the access misses. This makes load complete in 20 cycles for every 2 lines. 128 byte/ 600nS = 210MB/S. This is about 5 Seconds/GB.

O.5.4 Exit from RED_state

After the TLBs, Caches and memory are initialized, the processor can exit from RED_state and enable TLB and Caches by the following instruction sequence.

          (one of DCTI instructions) target_address   ! Jump to non_RED program

          wrpr %pstate   ! Reset PSTATE.RED bit

The above is the only valid way to get out of RED_state.

Caution:

> Be sure there are no hazards in the instruction fetch when the TLB is enabled. Right after PSTATE.RED is reset, a new instruction address is translated through TLBs and/or the TMB in memory.

P Error Handling

P.1 Overview

The SPARC64-III provides all error checking for all MMU translation paths and cache/ memory access paths between the CPU, 2nd level Cache, and the system bus. Errors are reported in three categories, synchronous access errors, asynchronous access errors, and system fatal errors.

A synchronous error is detected when the CPU can recognize a particular error associated with a particular request (either instruction fetch or data load/store).

For asynchronous errors, it is difficult to determine which request caused the error.

When a system fatal error is reported, the system must be reset before continuing. A synchronous error can be recoverable or non-recoverable. An *asynchronous_access_error* trap is supported only for an error logging purpose.

Synchronous errors that can be recoverable must be logged. Non-recoverable failures require immediate attention and logging, but not system reset. Software trap handlers decides actions of different recoveries based on the error trap types. For system fatal errors, CPU sends P_REPLY of type P_FERR to UPA. The system should generate a UPA POR RESET to all processors. The error register which records a system fatal error will not be cleared during a UPA Power-on RESET.

P.1.1 Synchronous Access Errors

The following errors could happen on instruction fetches and they belong to this category. When these errors happen, the CPU will generate an *instruction_access_error* trap (tt=0x00a, priority=3), and the fault instruction address will be set into TPC in the trap stack. For I1 Cache Data ECC Single or Multiple Error cases, however, the trap may be taken at a different address. But even in those cases, the instruction address that has the error and the trap address which is designated by TPC are within the same cache line address (the same 64 byte boundary address), and there are no problems with its error handling process described later.

- ■ μITLB Multiple Hit (ftype=0x1, priority=03, recoverable)
- ■ MTLB Parity Error (ftype=0x2, priority=05, recoverable)

- MTLB Multiple Hit   (ftype=0x3, priority=04, recoverable)

- I1 Cache Tag Parity Error   (ftype=0x4, priority=06, recoverable)

- I1 Cache Tag Multiple Hit   (ftype=0x5, priority=07, recoverable)

- I1 Cache Data ECC Single Error   ftype=0x6, priority=09, recoverable)

- I1 Cache Data ECC Multiple Error   (ftype=0x7, priority=08, recoverable)

- UPA Bus Error   (ftype=0x8, priority=10, non-recoverable)

- UPA Time-out   (ftype=0x9, priority=11, non-recoverable)

- I0 Cache Tag Parity Error   (ftype=0xe, priority=12, recoverable)

- I0 Cache Data Parity Error   (ftype=0xf, priority=13, recoverable)

The following errors could happen on data loads/stores and they belong to this category. When these errors happen, the CPU will generate a *data_access_error* trap (tt=0x032, priority=12).

- μDTLB Multiple Hit   (ftype=0x1, priority=1, recoverable)

- MTLB Parity Error   (ftype=0x2, priority=3, recoverable)

- MTLB Multiple Hit   (ftype=0x3, priority=2, recoverable)

- D1 Cache Tag Parity Error   (ftype=0x4, priority=4, recoverable)

- D1 Cache Tag Multiple Hit   (ftype=0x5, priority=5, recoverable)

- D1 Cache Data ECC Single Error   (ftype=0x6, priority=7, recoverable)

- D1 Cache Data ECC Multiple Error   (ftype=0x7, priority=6, non-recoverable)

- UPA Bus Error   (ftype=0x8, priority=8, non-recoverable)

- UPA Time-out   (ftype=0x9, priority=9, non-recoverable)

In both the instruction and data access error cases, MTLB multiple hit error is not always detected by the hardware when the condition exists in the Main TLB. Because the hardware caches some of MTLB entries into the μTLB's which are invisible from the software, the error detection depends on the μTLB entries. The software always should avoid the MTLB multiple hit condition and should not rely on the hardware error detection.

P.1.2 Asynchronous Access Errors

The following errors belong to this category. When these errors happen, the CPU will generate an *async_access_error* trap (tt=0x063, priority=2).

- D1 Cache Data ECC Multiple Error   (copyback only) (logging)

- U2 Cache Data ECC Multiple Error   (logging)

- U2 Cache Data ECC Single Error   (logging)

■ UPA data ECC Multiple Error   (logging)

■ UPA data ECC Single Error   (logging)

P.1.3 System Fatal Errors

■ U2-Cache tag parity error

■ UPA system address parity error

■ Trap at MAXTL

■ Watchdog Time-out

P.2 MMU Errors

P.2.1 μTLB Errors

For μITLB, the possible error is a μITLB multiple hit. For uDTLB, the possible error is a μDTLB multiple hit.

The above errors are synchronous errors. When a μTLB error occurs, an *instruction_access_error* or *data_access_error* trap is generated. The error handler should issue an INVALIDATE_UITLB (or INVALIDATE_UDTLB) ASI instruction to invalidate all μITLB (or μDTLB). When the missed request is retried, the μTLB is missed; the correct translation will be refilled from the MTLB.

P.2.2 Main TLB Errors

For MTLB, the possible errors are:

■ MTLB parity error

■ MTLB multiple hit

These are synchronous errors. When an MTLB error occurs, an *instruction_access_error* or *data_access_error* trap is generated. The error handler should issue INVALIDATE_TLB_ENTRY ASI_instruction to invalidate the erroneous entry. When the missed request is retried, the MTLB is missed; the correct translation will be performed as MTLB miss.

P.3 Memory Errors

All caches and memories are protected by either ODD parity or ECC. All of the ECC codes in L1/U2/UPA are the same. The 64-bit ECC code specification can be found in Shigeo Kaneda's correspondence note: "A Class of Odd-Weight-Column SEC-DED-SbED Codes for Memory System Applications", *IEEE Trans. on Computers*, August 1984.

P.3.1   I0 Cache Parity Errors

■ I0-Cache-Tag parity error

■ I0-Cache-Data parity error

P.3.2   I1/D1 Caches TAG Errors

■ I1-Cache-Tag parity error

■ I1-Cache-Tag multiple hit

■ D1-Cache-Tag parity error

■ D1-Cache-Tag multiple hit

P.3.3   I1/D1 Caches Data ECC Errors

■ I1-Cache-Data ECC error

■ D1-Cache-Data ECC error

■ D1-Cache-Data ECC multiple error

■ I1-Cache-Data ECC multiple error

P.3.4   U2 Cache TAG Parity Errors

■ U2-Cache tag parity error

P.3.5   U2 Caches Data ECC Errors

■ U2-Cache-Data SECC error

■ U2-Cache-Data MECC error

P.4   System Errors

P.4.1   System ECC Errors

■ UPA SECC errors

■ UPA MECC errors

P.4.2   System UPA Errors

■ S_ERR that is reported to a UPA master for an error associated with a noncached read.

■ S_RTO is read time out from a slave port for an access to an unimplemented address space.

P.5 Basic Mechanism and Flow of Error Handling

P.5.1 D1-Cache/U2-Cache/UPA ECC Multiple Error

1. When detected, the error is logged into DC/TDU/ICU Error Log Register. Then the CPU takes an *asynchronous_error* trap. Software logs the error information and clears the error log register.

- When updating the error log in the memory, software should use the lock mechanism because it is used by all of processors in the system.

2. The error data is sent with DATA_ERR signal to U2-Cache, I1-Cache, D1-Cache, or ICU.

3. When the data are stored or sent into U2-Cache, I1-Cache, D1-Cache, or UPA, the data and ECC bits which indicates multiple ECC errors are stored or sent (ECC multiple errors are propagated).

- To write ECC multiple error data into those Cache or UPA for the error propagation purpose, bit-63, 35, and 22 are flipped.

- The granularity of ECC error propagation is 16 byte. This means if an ECC error is detected in one of 8 byte data, the ECC multiple error data is generated for the 8 byte data and also the other 8 byte data which is within the same 16 byte boundary. Although the granularity is 16 byte, the software can know the exact original error bit location by looking at ECC syndrome bits from the error log registers.

4. When the error data is being used for real instruction fetch or execution, CPU takes an *instruction_access_error* or *data_access_error* trap (synchronous trap).

- The hardware guarantees that this synchronous error trap is taken after the asynchronous error trap which was the cause of the synchronous error was taken.

5. In the synchronous error trap, the software investigates the cause of the error by looking at error log information stored in memory, then kills the process or go to PANIC and fix the error.

- Before looking at the error log to investigate the cause of the error, the software should wait for several mS because other processors haven't logged error logs which may be the cause of the error.

P.5.2 How to Fix the MECC Error

This scheme can be used for any kind of ECC multiple errors which can be anywhere in the system (i.e. memory, U2-Cache, I1-Cache, D1-Cache of any processors in the system).

1. Disable the *asynchronous_error_trap* (ASR31).

2. Do two 8-byte stores to the error location.

3. Do STXA, ASI=$6A_{16}$ (Flush L1 Cache to U2 Cache Using U2 Cache Index)

4. Clear the error log registers.

5. Enable *asynchronous_error_trap* (ASR31).

P.5.3   U2-Cache/UPA SECC Error

1. If it's detected, the error is logged into TDU/ICU Error Log Register. Then the CPU takes an *asynchronous_error* trap. Software logs the error information and clears the error log registers.

2. The error is corrected by hardware, and doesn't propagate to any other place.

3. Software corrects the original error. (The original error is not corrected by the hardware.)

P.5.4   How to Fix the SECC Error

This scheme can be used for any kind of ECC single errors which can be anywhere in the system (i.e. memory, U2-Cache, I1-Cache, D1-Cache of any processors in the system).

1. Disable the *asynchronous_error_trap* (ASR31).

2. Do LDXA,ASI=14. The address should use the same U2-Cache index PA address of the error (PA[23:6]), but other address bits (PA[40:24]) should be different with those of the error address. By doing this way, the data of the error address is corrected and evicted from I1/D1/U2-Cache.

3. Clear the error log registers.

4. Enable the *asynchronous_error_trap* (ASR31).

P.5.5   Permanent or Intermittent SECC Error

This scheme can be used to check whether the ECC single error is permanent or intermittent in U2-Cache or the memory. This should be done after the error is fixed by the scheme above.

1. Disable *asynchronous_error_trap* (ASR31).

2. Do LDXA,ASI=14. The address should be the same error location address. By fixing the ECC single error, the error data has been already evicted from I1/D1/U2-Cache. Therefore to check whether permanent or intermittent, just reload the data into U2 Cache.

3. Check the error log registers.

4. If UPA ECC single error is logged, this is a memory permanent error.

5. If U2 ECC single error is logged, this is a U2 Cache permanent error.

6. If no error is logged, this is an intermittent error.

7. Clear the error log registers.

8. Enable *asynchronous_error_trap* (ASR31).

When there is a permanent ECC single-bit error in the memory, the program can advance because the corrected data is still in the U2-Cache after the asynchronous error handling. When there is a permanent ECC single-bit error in U2-Cache and the access is not an instruction fetch, the program can advance because the corrected data is still in the D1-Cache after the asynchronous error handling. When there is a permanent ECC single-bit error in U2-Cache and the access is an instruction fetch, the program can not advance because the corrected data is not in the I1-Cache after the asynchronous error handling. To deal with this problem, there are two choices for the software to avoid the loop situation:

■ When a U2 Cache ECC single error is reported by an asynchronous error trap, the software should check whether the error location has been logged as a permanent U2 Cache single error before fixing the error. If it is the case, the software should just skip the error fix and the investigation of the error permanency, and then just clear the error log and do RETRY. By doing this way, the corrected instruction can remain in the I1 Cache, and we can avoid the software loop.

■ If the software detects an infinite synchronous ECC single error trap loop, it should set "Disable Asynchronous ECC Single Error Trap" (D_ASEET) bit in ASR31. It might be better for the software to check whether this bit is set at some fixed interval (perhaps once an hour), and reset it if it's set; the bit should not be set for a long time in order to prevent ECC single errors from becoming ECC multiple errors.

P.6 Hardware Error Trap Processing

P.6.1 Instruction_access_error trap (tt=00a$_{16}$)

■ When this error condition is reported, the CPU waits for sync.

■ After the sync, if it is the earliest and highest trap condition, the CPU is going to take this trap.

■ If TL=MAXTL already, the CPU logs TRAP@MAXTL occurrence into TDU Error Log Register, and assert P_FERR to UPA bus.

■ After taking the trap and before issuing trap routine instructions, all of instructions in I0-Cache and Instruction-Buffer are invalidated.

P.6.2 Data_access_error trap (tt=032$_{16}$)

■ When this error condition is reported, the CPU waits for sync.

■ After the sync, if it is the earliest and highest trap condition, the CPU is going to take this trap.

■ If TL=MAXTL already, the CPU logs TRAP@MAXTL occurrence into TDU Error Log Register, and assert P_FERR to UPA bus.

P.6.3 Asynchronous_error trap (tt=063$_{16}$)

■ When the error condition is detected, the error information (error type, PA, ECC syndrome) is logged into one of DC/TDU/ICU Error Log Registers.

■ In any data ECC single error cases, the data is corrected through ECC correction logic by the hardware and the error doesn't propagate with the data any more. But in the U2 cache single ECC error case, the error in U2 cache itself isn't corrected by the hardware and still will be there.

■ UC starts and keeps asserting asynchronous error signal to CPU core until the error log register gets cleared by the software.

■ The CPU waits for sync.

■ After the sync, if it is the earliest and highest trap condition, the CPU is going to take this trap.

■ If TL=MAXTL already, the CPU logs TRAP@MAXTL occurrence into TDU Error Log Register, and assert P_FERR to UPA bus.

■ Immediately after taking the trap, the CPU sets the "Disable Asynchronous Error Trap" bit in ASR31 to avoid taking the same trap again before the software clears the error log registers. It's the software's responsibility to reset the "Disable Asynchronous Error Trap" bit using WRASR31 instruction after the asynchronous error handling is done. The software also can set this bit using WRASR31 instruction.

P.7 ASI Instructions for Error Handling

This section describes the ASI instructions that are designed for error handling. The following things are common to all of the ASI instructions defined in this section:

■ The opcode of the instructions should be either `ldx(a)`, `lddf(a)`, `stx(a)`, or `stdf(a)`. Otherwise, a *data_access_exception* trap with FTYPE=F$_{16}$ (Invalid ASI) is taken.

■ No address translation is performed for the instructions.

■ VA[3:0] of all of the instructions should be 0; otherwise a *mem_not_aligned* trap is taken.

■ The don't-care bits (described as '—' in the format) in VA can be any value. But it is recommended that the software should use zero for these bits.

■ The don't-care bits (described as '—' in the format) in DATA are read as zero and ignored on write.

■ The instruction operations are not affected by PSTATE.CLE. They are always treated as in a big endian mode.

■ The instructions do not cause the processor to sync.

■ The instructions are all strongly ordered regardless of load or store, and the memory model. Therefore no speculative executions are performed.

The following subsections describe each operation in detail.

P.7.1 Read/Write TDU Error Log Register

Function: Read/Write TDU Error Log Register. This register is used to log U2 Cache parity/ECC errors. It is also used to log "trap at max trap level" and "watchdog time-out" errors.

ASI: $6C_{16}$

RW: Supervisor Read, Supervisor Write.

VA: VA:See below.

—	0000

63   4 3   0

Data: See below.

SYND ROME	U2M	U2MM	U2S	U2SM	U2TG	TRPM	WDTO	PA	—

63   48   47   46   45   44   43   42   41   40   4 3   0
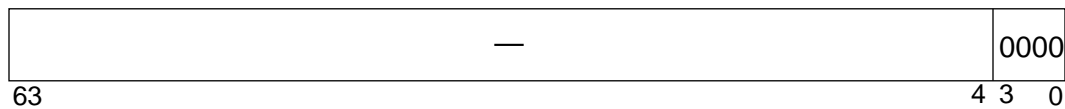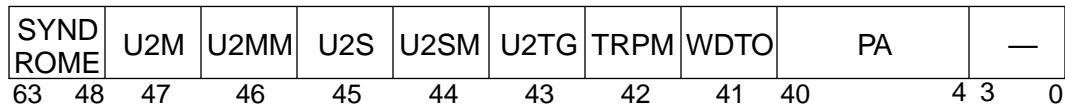
SYNDROME[15:0]

ECC Error Syndrome. ECC error syndrome bits are set when U2-Cache ECC multiple or single error is detected for any kinds of U2-Cache read. Bit 15-8 are the syndrome bits for byte 0-7 (bit 127-64) data, and bit 7-0 are the syndrome bits for byte 8-f (bit 63-0) data. If U2-Cache ECC multiple error bit is already set and another U2-Cache ECC multiple error is detected, this value won't be changed (keep the oldest value). If U2-Cache ECC single error bit is already set and another U2-Cache ECC single error is detected, this value won't be changed (keep the oldest value). If U2-Cache ECC multiple error bit is not set and U2-Cache ECC single error bit is set and U2-Cache ECC multiple error is detected, this value is updated (ECC multiple error has a higher priority over ECC single error).

U2M:

U2-Cache ECC Multiple Error. Set when U2-Cache ECC multiple error is detected for any kinds of U2-Cache read.

U2MM

U2-Cache ECC Multiple Error Multiple Occurrence. Set when U2-Cache ECC Multiple Error bit is already set and another U2-Cache ECC multiple error is detected for any kinds of U2-Cache read.

U2S

U2-Cache ECC Single Error. Set when U2-Cache ECC single error is detected for any kinds of U2-Cache read.

U2SM

U2-Cache ECC Single Error Multiple Occurrence. Set when U2-Cache ECC Single Error bit is already set and another U2-Cache ECC single error is detected for any kinds of U2-Cache read.

U2TG

U2-Cache Tag Parity Error. Set when U2-Cache Tag Parity Error is detected for any kinds of U2-Cache tag read.

TRPM

Trap At MAXTL. Set when a trap at MAXTL is detected.

WDTO

Watchdog Time-out. Set when a watchdog time-out is detected.

PA[40:4]

Error Physical address. The error PA is set when U2-Cache ECC multiple or single error is detected for any kinds of U2-Cache read. If U2-Cache ECC multiple error bit is already set and another U2-Cache ECC multiple error is detected, this value won't be changed (keep the oldest value). If U2-Cache ECC single error bit is already set and another U2-Cache ECC single error is detected, this value won't be changed (keep the oldest value). If U2-Cache ECC multiple error bit is not set and U2-Cache ECC single error bit is set and U2-Cache ECC multiple error is detected, this value is updated (ECC multiple error has a higher priority over ECC single error).

Note:

If both upper and lower 8 byte data within the same 16 byte data have ECC multiple or single errors simultaneously, it is logged as only one error occurrence. In the case of single and multiple error combinations, it is logged as an ECC multiple error. That is, U2M bit is set, U2S bit is unchanged.

This register is reset on a cold POWER_ON_RESET.

This register is *not* reset by UPA_POR.

The SYNDROME bits (bit 63-48) and PA bits (bit 40-4) are reset to "0" on write, regardless of the write data.

The U2M, U2MM, U2S, U2SM, U2TG, TRPM, WDTO bits (bit 47-41) are reset to "0" on write when the corresponding write data is "1" ("echo reset").

The U2M, U2MM, U2S, U2SM, U2TG, TRPM, WDTO bits (bit 47-41) are unchanged on write when the corresponding write data is "0".

P.7.2 Read/Write ICU Error Log Register

Function: Function:Read/Write ICU Error Log Register. This register is used to log UPA bus parity/ECC errors.

ASI: $6D_{16}$

RW: Supervisor Read, Supervisor Write.

VA: See below.

—	4h0
63	4 3 0

Data: DATA:See below.

SYND ROME	UPM	UPMM	UPS	UPSM	UPAD	—	INTR	PA	—
63 48	47	46	45	44	43	42	41	40 4	3 0
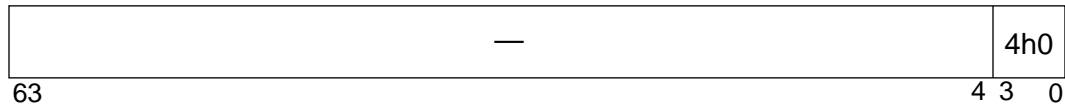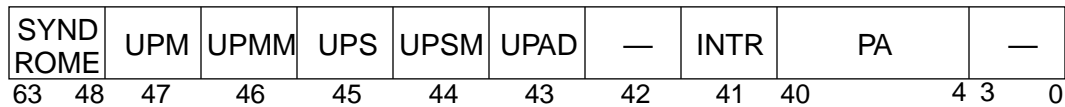
SYNDROME[15:0]

ECC Error Syndrome. ECC error syndrome bits are set when UPA ECC multiple or single error is detected. Bit 15-8 are the syndrome bits for byte 0-7 (bit 127-64) data, and bit 7-0 are the syndrome bits for byte 8-f (bit 63-0) data. If UPA ECC Multiple Error bit is already set and another UPA ECC multiple error is detected, this value won't be changed (keep the oldest value). If UPA ECC Single Error bit is already set and another UPA ECC single error is detected, this value won't be changed (keep the oldest value). If UPA ECC Multiple Error bit is not set and UPA ECC single error bit is set and UPA ECC multiple error is detected, this value is updated (ECC multiple error has a higher priority over ECC single error).

UPM

UPA ECC Multiple Error. Set when UPA ECC multiple error is detected.

UPMM

UPA ECC Multiple Error Multiple Occurrence. Set when UPA ECC Multiple Error bit is already set and another UPA ECC multiple error is detected.

UPS

UPA ECC Single Error. Set when UPA ECC single error is detected.

UPSM

UPA ECC Single Error Multiple Occurrence. Set when UPA ECC Single Error bit is already set and another UPA ECC single error is detected.

UPAD

UPA Address Parity Error. Set when UPA address parity error is detected.

INTR

Interrupt Vector Data Error. Set when UPA ECC multiple or single error is detected and the data is one of interrupt vector data. If UPA ECC multiple error bit is already set and another UPA ECC multiple error is detected, this value won't be changed (keep the oldest value). If UPA ECC single error bit is already set and another UPA ECC single error is detected, this value won't be changed (keep the oldest value). If UPA ECC multiple error bit is not set and UPA ECC single error

bit is set and UPA ECC multiple error is detected, this value is updated (ECC multiple error has a higher priority over ECC single error).

PA[40:4]

Error Physical Address. The error PA is set when UPA ECC multiple or single error is detected. If UPA ECC multiple error bit is already set and another UPA ECC multiple error is detected, this value won't be changed (keep the oldest value). If UPA ECC single error bit is already set and another UPA ECC single error is detected, this value won't be changed (keep the oldest value). If UPA ECC multiple error bit is not set and UPA ECC single error bit is set and UPA ECC multiple error is detected, this value is updated (ECC multiple error has a higher priority over ECC single error). When the error is on the interrupt vector data, PA[40:4] is either 37h0 (Interrupt Vector Data 0), 37h1 (Interrupt Vector Data 1), or 37h2 (Interrupt Vector Data 2).

Note:

If both upper and lower 8 byte data within the same 16 byte data have ECC multiple or single errors simultaneously, it is logged as only one error occurrence. In the case of single and multiple error combinations, it is logged as an ECC multiple error. That is, UPM bit is set, UPS bit is unchanged.

This register is reset on the cold POWER_ON_RESET.

This register is *not* reset by UPA_POR.

The SYNDROME bits (bit 63-48), INTR bit (bit 41), and PA bits (bit 40-4) are reset to "0" on write, regardless of the write data.

The UPM, UPMM, UPS, UPSM, UPAD bits (bit 47-43) are reset to "0" on write when the corresponding write data is "1" ("echo reset").

The UPM, UPMM, UPS, UPSM, UPAD bits (bit 47-43) are unchanged on write when the corresponding write data is "0".

When one of the interrupt data has an ECC error, the processor still tries to take *interrupt_vector* trap regardless of the error. But in this case, the processor always takes an *asynchronous_error* trap first. It's up to the software whether to kill the interrupt request by resetting the busy bit of interrupt data receive register when it recognized that there is an ECC error on the interrupt data by reading ICU Error Log Register in the asynchronous error trap routine.

P.7.3 Read/Write DC Error Log Register

Function: Read/Write DC Error Log Register. This register is used to log D1 Cache ECC error on copyback.

   ASI: $6E_{16}$

   RW: Supervisor Read, Supervisor Write.

   VA: See below.

—	4h0

63   4 3   0

Data: See below.

—	D1M	D1MM	—	PA	—
63 48	47	46 45	41 40		4 3 0

D1M

D1-Cache ECC Multiple ECC Error. Set when D1-Cache ECC multiple ECC error is detected during a copyback from D1-Cache to U2-Cache or UPA.

D1MM

D1-Cache ECC Multiple Error Multiple Occurrence. Set when D1-Cache ECC Multiple Error bit is already set and another D1-Cache ECC multiple error is detected during a copyback from D1-Cache to U2-Cache or UPA.

PA[40:4]

Error Physical Address. The error PA is set when D1-Cache multiple error is detected during a copyback from D1-Cache to U2-Cache or UPA. If D1-Cache ECC multiple error bit is already set and another D1-Cache ECC multiple error is detected during a copyback from D1-Cache to U2-Cache or UPA, this value won't be changed (keep the oldest value).

Note:

If both upper and lower 8 byte data within the same 16 byte data have ECC multiple errors simultaneously, it is logged as only one error occurrence.

This register is reset on cold POWER_ON_RESET.

This register is *not* reset by UPA_POR.

The PA bits (bit 40-4) are reset to "0" on write, regardless of the write data.

The D1M, D1MM bits (bit 47-46) are reset to "0" on write when the corresponding write data is "1" ("echo reset").

The D1M, D1MM bits (bit 47-46) are unchanged on write when the corresponding write data is "0".

P.7.4   Read/Write UC Error Injection Register

Function: Read/Write UC Error Injection Register. This register is used to inject ECC errors into U2 Cache and UPA bus.

ASI: $6F_{16}$

RW: Supervisor Read, Supervisor Write.

VA: See below.

—	4h0
63	4 3 0

Data: See below.

—	DEI	UPA	U2
63 33	32 31	16 15	0
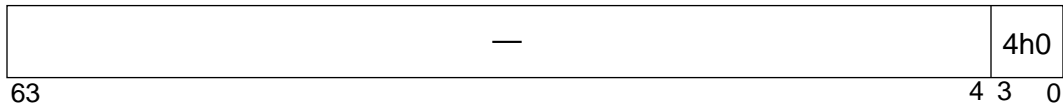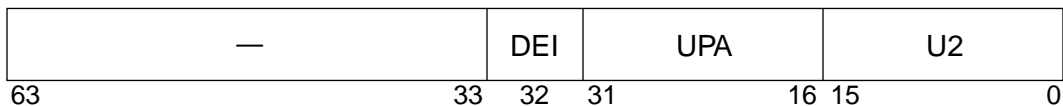
DEI

Disable Error Injection Except U2 Flush. If this bit is "0", any kinds of outgoing data to UPA are error-injected by flipping ECC check bits using "UPA ECC Check Bit Flip" value in this register. If this bit is "1", the outgoing data to UPA caused by the following ASI instructions are error-injected by flipping ECC check bits using "UPA ECC Check Bit Flip" value in this register, and the outgoing data to UPA caused by other than the following ASI instructions are *not* error-injected and "UPA ECC Check Bit Flip" values in this register are ignored.

Flush U2 Cache to Memory Using U2 Cache Buffer Index (ASI=68_{16}, VA[30:28]=010_2)

Flush U2 Cache to Memory Using PA (ASI=69_{16})

UPA[15:0]

UPA ECC Check Bit Flip. Bit 15-8 are used to flip UPA ECC check bits of UPA Data bit 127-64. Bit 7-0 are used to flip UPA ECC check bits of UPA Data bit 63-0. ECC check bit flipping happens depending on these bit values and DEI bit value.

U2[15:0]

U2 Cache ECC Check Bit Flip. Bit 15-8 are used to flip U2 Cache ECC check bits of U2 Cache Data bit 127-64. Bit 7-0 are used to flip U2 Cache ECC check bits of U2 Cache Data bit 63-0. ECC check bit flipping happens depending on these bit values and only when the write to U2 Cache Data is caused by the following ASI instruction:

Write U2 Cache Data From U2 Cache Buffer Registers (ASI=68_{16}, VA[30:28]=001_2)

Q Performance Monitoring

Q.1 Introduction

This appendix describes and specifies performance monitors that have been implemented in the SPARC64-III CPU. Performance monitors have been added in order to improve software tuning and allow cursory system debug. System debug may be performed during bring-up or at customer sites.

Because of complexity related to speculative out-of-order execution, retry and block conditions between the Load Store Unit (LSU) and Cache, and instruction and data prefetch, performance monitors are difficult to specify and implement. For example, consider instruction prefetching. The CPU attempts to prefetch the next cache line from the level-1 instruction cache (I1). This prefetch may miss in the I1. If this cache line is later used by the CPU, I1 miss is counted, even though it may not have resulted in an issue bubble in the CPU. Additionally, the aggressive use of instruction speculation in the CPU complicates performance monitoring. Should stall conditions which occur during an incorrectly executed instruction sequence be counted? How do we measure cache pollution due to speculation? Additionally, we would like to minimize the impact to the processor chip. The CPU has limited silicon resources and additional functionality must be tested and verified.

Therefore, we have attempted to minimize complexity while still providing performance observability. Specifically:

■ No timing paths have been introduced.

■ In some cases, new functionality is leveraged off existing logic in the SPARC64-II CPU.

■ In cases where speculation would complicate performance monitoring design and verification, we have chosen to simplify the monitor.

Q.2 Performance Monitor Description

Q.2.1 Overview

All performance monitors are 32-bits long in SPARC64-III CPU. The performance monitors can be divided into the following groups:

1. Memory Access Latency Counters

These counters measure the latency of certain types of memory accesses.

2. Memory Access Event Counters

   These measure the frequency of certain types of memory accesses from LSU's point of view.

3. L1 Cache Access Event Counters

   The frequency of L1 cache misses can be derived from this set of counters.

4. L2 Cache Access Event Counters

   All kinds of access events to the L2 cache are recorded.

5. UPA Access Event Counters

   Non-cacheable and cacheable UPA requests are counted separately.

6. Cache Coherency Event Counters

   L1 cache coherency events (invalidation, retag and copyback) are counted separately.

7. MMU Event Counters

   TLB misses are counted with respect to µTLB or main TLB, data or instruction side.

8. Instruction Execution Rate Counters

   This set comprises the issue and commit counters which measure instruction execution rate.

9. Issue Stall Counters

   The mixture and number of various issue stall conditions are monitored.

10. Branch Prediction Counters

    The branch mispredict ratio can be calculated from these counters.

11. Machine Sync Counters

    The frequency and latency of machine syncing instructions are measured.

With the exception of the committed instruction counter and main TLB miss counters, all other monitors and counters are updated speculatively. For example, issue stalls (as well as memory accesses) which occur in mispredicted paths may be accumulated into the stall (frequency and latency) counters respectively. Instructions which are issued speculatively may not always modify performance monitors. Consider load/store instructions which are issued to the LSU reservation station, but are not selected for execution due to scheduling constraints. These instructions may later be killed before they are requested from the cache chips.

Finally, counters are not disabled during backups and backsteps, although in some cases (memory access event and latency counters) instructions killed by the Precise State Unit will not update the counter. Refer to the following sections for more detail.

Notes:

Upon saturation, all counters will remain at the maximum value. The maximum value indicates counter overflow.

All performance monitor registers are 32-bits long in SPARC64-III CPU.

Q.2.2 Memory Access Latency Counters

Memory access latency counters measure total latency for different types of load/store accesses which do not "HIT". A "HIT" is defined as a request from the DFMLSU which returns data and CACHE_HIT the third cycle after a memory access has been requested. Hence, a "HIT" may occur on a number of different conditions:

■ The data cache has asserted BLOCK, and the LSU has inhibited any memory requests. After negation of BLOCK, any memory request which "HITs" will increment the data cache hit counter.

■ The data cache has asserted RETRY after the LSU has sent a memory request. After one or more attempts, any memory request which "HITs" will increment the data cache hit counter.

Latency is defined from initiation of a memory request from the LSU to completion status from the data cache. Completion status can be successful data transfer or any error condition.

In order to implement the memory access latency counters, each potential outstanding load/store request from the LSU (maximum of 12) has an associated counter. The counter is 12-bits and upon saturation, will maintain its maximum value. Although the maximum limit for memory accesses is theoretically infinite, the maximum latency which can be measured per memory miss is 4096 cycles. Latency measurements greater than 4096 cycles will be lost. Upon completion of the memory access, the individual latency count is added to the total latency count for the appropriate memory type.

All latency counters are updated speculatively, but load/store instructions which are killed due to backtracks initiated by the Precise State Unit (PSU) will not be accumulated into the latency counters.

Counter 0: Memory Total Latency Counter

Description: Total latency count of all memory access which complete but are not defined as a "HIT". From the memory event counter and the memory latency counter, the average latency for memory accesses can be calculated.

Q.2.3 Memory Access Event Counters

Memory access event counters measure the number of certain types of load/store access from LSU's point of view. All updates to these monitors are performed speculatively.

All memory event counters are updated speculatively, but load/store instructions which are killed due to backtracks initiated by the Precise State Unit (PSU) will not be accumulated into the event counters.

Counter 1:   L1 Data Cache Hit Counter

Description: Incremented when a load/store request to the data cache returns with "HIT". This counter may be incremented a maximum of 2 every cycle.

Counter 2:   Memory Access Event Counter

Description: Any memory access which completes but is not defined as a "HIT," will result be counted in the memory access event counter. This counter may be incremented a maximum of 2 every cycle.

Q.2.4 L1 Cache Access Event Counters

Level-1 cache access event counters record access events which take place in the level-1 data and instruction caches.

Counter 3:   L1 Data Cache Reload for Load Event Counter

Description: Incremented when UC sends reload request, which is corresponding to a load miss, to DC. This counter also indicates the event of a L1 data cache miss for load request, excluding μTLB miss.

Counter 4:   L1 Data Cache Reload for Store Event Counter

Description: Incremented when UC sends reload request, which is corresponding to a store miss, to DC. This counter also indicates the event of a L1 data cache miss for store request, excluding μTLB miss.

Counter 5:   L1 Data Cache Victim Copyback Counter

Description: Accumulates the number of misses that do victim copyback in level-1 data cache.

Counter 6:   L1 Instruction Cache Reload Event Counter

Description: Incremented when UC sends reload requests to IC. This counter also indicates the event of a L1 instruction cache miss, excluding μTLB miss.

Q.2.5 L2 Cache Access Event Counters

Counter 7:   U2 Cache Miss From Instruction Fetch Counter

Description: Incremented when a read request from IC misses in UC.

Counter 8:   U2 Cache Miss From Data Load Counter

Description: Incremented when a load request from DC misses in UC.

Counter 9:   U2 Cache Miss From Data Store Counter

Description: Incremented when a store request from DC misses in UC.

Counter 10: **U2 Cache Miss With Writebacks Counter**

Description: Accumulates the number of events of U2 cache miss that do writeback.

Counter 11: **U2 Cache Invalidate from UPA Transaction Counter**

Description: Incremented when a U2 cache line is invalidated due to the following UPA transactions: S_INV_REQ, S_CPI_REQ.

Counter 12: **U2 Cache Unsolicited Copyback Counter**

Description: Incremented when an unsolicited copyback occurs in U2 cache due to the following UPA transactions: S_CPB_REQ, S_CPI_REQ, S_CPD_REQ, S_CPB_MSI_REQ.

Counter 13: **U2 Cache Hit with "Read to Own" UPA Transaction Counter**

Description: Incremented when U2 cache hits that require "read to own" UPA

Counter 14: **I0 Instruction Cache Miss Counter**

Description: Incremented when level-0 instruction cache line is replaced.

Q.2.6 UPA Access Event Counters

Counter 15: **Non-cacheable Load Counter**

Description: Accumulates the number of non-cacheable load requests from UC to UPA bus.

Counter 16: **Non-cacheable Store Counter**

Description: Accumulates the number of non-cacheable store requests from UC to UPA bus.

Counter 17: **UPA Access Counter**

Description: Incremented when UC sends any kind of request to the system controller.

Q.2.7 Cache Coherency Event Counters

Cache Coherency Event Counters monitor the activities related to the coherency requests. The events that trigger the increment of these counters are certain type of requests from UC to DC or IC. These counters are incremented speculatively, so the event recorded may be related to instructions on a mispredicted path.

Counter 18: **L1 Data Cache Invalidate Event Counter**

Description: Incremented when UC sends invalidate request to DC.

Counter 19: **L1 Data Cache Retag Event Counter**

Description: Incremented when UC sends retag request to DC.

Counter 20: L1 Instruction Cache Invalidate Event Counter

Description: Incremented when UC sends invalidate request to IC.

Counter 21: L1 Data Cache Unsolicited Copyback Counter

Description: Incremented when UC sends DC an unsolicited copyback request.

Q.2.8 MMU Event Counters

Counter 22: Instruction μTLB Miss Counter

Description: Incremented when instruction μTLB miss occurs. On main TLB hit, this counter may be updated speculatively. On main TLB miss, it is updated non-speculatively.

Counter 23: Data μTLB Miss Counter

Description: Incremented when data μTLB miss occurs. On main TLB hit, this counter may be updated speculatively. On main TLB miss, it is updated non-speculatively.

Counter 24: Instruction Main TLB Miss Counter

Description: Incremented when instruction main TLB miss occurs. Non-speculative.

Counter 25: Data Main TLB Miss Counter

Description: Incremented when data main TLB miss occurs. Non-speculative.

Q.2.9 Instruction Execution Rate Counters

The following counters allow measurement of instruction execution rates.

Counter 26: Performance Monitoring Cycle Counter

Description: Accumulates the number of cycles the performance monitors have been enabled. Provides a precise measurement interval. It is separate from the TICK register accessed through the *rd %tick* instruction because it can be enabled and disabled through software. At 250Mhz, 32-bits will result in a monitoring interval exceeding 10 seconds. When monitoring User or System time separately, this counter gives a measure of how long the processor spends with PSTATE.PRIV either 0 (User) or 1 (System) (see Appendix Q.3.5, "Performance Counter Enable").

Counter 27: Instruction Issue Counter

Description: Accumulates the number of instructions issued every cycle. Assuming the machine is running at 250MHZ at the theoretical peak of 4 IPC, a 32-bit counter will provide a monitoring interval of 4 seconds without overflow. Generally, processor IPC will

be closer to ~1.8 over long benchmarks, but the theoretical peak may be reached for short periods. From the performance monitoring cycle counter and the instruction issue counter, the speculative IPC can be obtained.

The instruction issue counter will be larger than the instruction commit counter. Generally, this is due to branch misprediction effects, but can also be due to execution traps (for example, main TLB miss traps). In the latter case, instructions on the correctly predicted path may be killed and reissued on return from the trap.

Special considerations are needed when measuring User and System counts separately (see Appendix Q.3.5, "Performance Counter Enable" for a discussion)

Counter 28: Instruction Commit Counter

Description: Accumulates the number of instructions committed every cycle. Due to speculative instruction issue, the value stored in the committed instruction counter may be less than that of the issue counter. From the performance monitoring cycle counter and the instruction commit counter, the true IPC can be obtained.

Q.2.10 Issue Stall Counters

Issue stall counters measure the frequency of instruction stalls. On any cycle where the machine is unable to meet its peak issue rate of 4 instructions, the first stalling instruction is checked against a prioritized list of stall conditions. Only one out of the seven stall counters is incremented. For example:
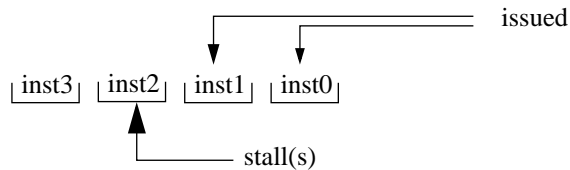
In the above case, the CPU was able to issue the first 2 instructions, but the third instruction (inst2) stalls. In order to decide which one of the stall monitors should be incremented, the stall(s) related to inst2 are checked against the first stall counter. If the instruction stalled due to the first stall type, the counter is incremented and no other stall counter is incremented. Otherwise, the CPU proceeds to the second, and so on, until the last stall monitor is reached. If the last stall monitor is reached, it will always be incremented. Although inst2 may have generated multiple stalls, the prioritized list assures a unique stall count. For example, if the primary instruction cache misses, whether the instruction would have stalled due to lack of free registers is immaterial.

All stall counters are incremented speculatively. They correspond with events in the *issue* cycle of the SPARC64-III CPU. Instructions are issued in-order, so modifications to stall monitors will occur in program order.

A special note is required for *issue_trap*s. Although they are logged in the last stall counter (Counter 34), they are prioritized before Counter 29. On an *issue_trap*, Counter 34 will be incremented if the instruction did not stall due to a fetch stall or instruction invalid stall. This is because when *issue_trap*s occur, whether the instruction would have generated other resource stalls is immaterial.

The sum total of all the stall counters gives a measure of cycles when less than 4 instructions were issued. Subracting this total from the performance monitoring cycle counter gives the total number of 4-issue (speculative IPC 4) cycles. Subsequently, the *average* speculative IPC for the stall cycles can be computed.

Counter 29: Fetch Stall Counter

Description: Incremented when an instructions stalls during the issue cycle due to a fetch bubble. These cases include:

- ◆ primary instruction cache miss
- ◆ prefetch line miss
- ◆ cache-line discontinuity
- ◆ control transfer instruction
- ◆ DONE/RETRY

Counter 30: PSU_KILL Stall Counter

Description: Incremented when an instruction stalls due to an issue kill from the Precise State Unit, and higher prioritized stalls have not occurred. The PSU kills instruction issue due to

- ◆ exception (interrupts, execution trap) handling
- ◆ during the DO_BACKUP cycle as a result of machine backups.

Counter 31: Reservation Station Queue Stall Counter

Description: Incremented when an instruction stalls due to insufficient reservation station queue entries, and higher prioritized stalls have not occurred. Reservation stations comprise:

- ◆ DFMLSU - load/store instructions
- ◆ DFMFXU - fixed-point instructions
- ◆ DFMAGEN - load/store instructions and certain fixed-point instructions
- ◆ DFMFPU - floating-point instructions

Counter 32: Free Register Resource Stall Counter

Description: Incremented when an instruction stalls due to insufficient free registers for register renaming, and higher prioritized stalls have not occurred. The following registers are renamed:

- ◆ fixed-point registers
- ◆ floating-point registers
- ◆ condition code (icc,xcc,fcc0-4) registers

Counter 33: Checkpoint, Serial Number, or Trap Stack Resource Stall

Description: Incremented when an instruction stalls due to insufficient checkpoints, serial numbers, or trap_stack entries, and higher prioritized stalls have not occurred. The following instructions require checkpoints:

- ◆ `BPr(a)`

◆ `FBcc(a)/FBPcc(a)`

◆ `Bcc(a)/BPcc(a)`

◆ `BAa/BPAa/FBAa/FBPAa`

◆ `flushw`

◆ `integer ldd/std(a)`

◆ `saved/restored`

◆ `call`

◆ `jmpl`

◆ `ret/retl`

◆ `return`

◆ `save/restore`

◆ `done/retry`

◆ `ta [%g0+imm]`

◆ `tcc`

◆ `tn`

◆ `rd %asr`

◆ `rd %pr`

◆ `wr %asr`

◆ `wr %pr`

All instructions require serial numbers for issue. Only instructions which generate exceptions require trap_stack entries. The SPARC64-III CPU contains 16 checkpoints, 64 serial numbers, and 4 extra trap_stack entries for renaming.

Counter 34: Other Stall Counter

Description: Incremented when an instruction stalls due to all other issue stall conditions, and a higher prioritized stalls have not occurred. These other stalls include:

◆ syncing stalls due to sequential mode, or instructions which must be issued at machine sync

◆ %pil register stall - following a wr %pil instruction, any further read or write of %pil register will result in a stall until the PIL register has been correctly updated

◆ last_tobe_issued - certain instructions must be the last instruction in the instruction window to be issued.

◆ slot0_only - certain instructions must the first instruction in the issue window.

◆ issue_traps

Q.2.11 Branch Prediction Counters

Counter 35: Branch Issue Counter

Description: The number of branch instructions which have been issued, including speculative branches. If the processor is operating in sequential mode, an accurate non-speculative branch count can be obtained.

Counter 36: Branch Mispredict Counter

Description: Accumulates the number of branch mispredict events. By dividing the value of this counter by the value of "Branch Issue Counter," the branch mispredict ratio can be obtained.

Counter 37: Instruction Lookup Table (ILT) Miss Counter

Description: Accumulates the number of ILT (next fetch PC) misses.

Q.2.12 Machine Sync Counter

Counter 38: Sync Event Counter

Description: Accumulates the number of syncing instructions. Does not include machine syncs due to exceptions (for example, main TLB miss exception).

Counter 39: Sync Cycle Counter

Description: Counts the number of cycles when the machine is waiting for sync to complete. From this counter and the Sync Event Counter, the average latency per syncing instruction can be calculated.

Q.3 Software Interface

Q.3.1 Reading Performance Counters

All performance monitor registers are software visible through the `rd %asr30` instruction (see 6.2, "Instruction Formats" for details of instruction formats). Bits [12:8] of the instruction (pm_reg_#) are used to select the register number within a particular view, which is defined by the value in the View Number Register.

Table 92: Interpretation of bits[12:8]

pm_reg_#	Operation
00000	View Number Register
00001	register #0
00010	register #1
00011	register #2
00100	register #3
00101	register #4
00110	register #5

All rd %asr30 instructions are syncing instructions. On a rd %asr30, all upper bits not specified in the counter are padded with zeroes.

The value of View Number Register can be updated by wr asr%30 with pm_op = "00100." The format of View Number Register is:

pm1_view_#	pm0_view_#

7   3   0
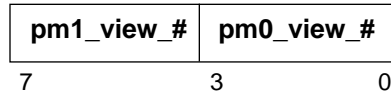
The pm1_view_# indicates the view number selecting the register group to be visible for group 1, which is composed of register# 3 to 5. The pm0_view_# indicates the view number selecting the register group to be visible for group 0, which is composed of register 0 to 2. The counters to be selected according to pm1_view_# are shown in Table 93. Table 94 shows the counter selection according to pm0_view_#. When undefined pm1_view_# or pm0_view_# is specified, the counter value is undefined.

Table 93: View Selection for Monitor Group 1

Register #	Monitor	Counter #
pm1_view_# = 0000		
3	Memory Total Latency Counter	0
4	L1 Data Cache Hit Counter	1
5	Memory Access Event Counter	2
pm1_view_# = 0001		
3	L1 Data Cache Reload for Load Event Counter	3
4	L1 Data Cache Reload for Store Event Counter	4
5	L1 Data Cache Invalidate Event Counter	18
pm1_view_# = 0010		
3	L1 Data Cache Retag EventCounter	19
4	L1 Data Cache Victim Copyback Event Counter	5
5	L1 Data Cache Unsolicited Copyback Counter	21
pm1_view_# = 0011		
3	Data μTLB Miss Counter	23
4	Data main TLB Miss Counter	25
5	UPA Access Counter	17
pm1_view_# = 0100		
3	U2 Cache Miss from Instruction Fetch Counter	7
4	U2 Cache Miss from Data Load Counter	8
5	U2 Cache Miss from Data Store Counter	9
pm1_view_# = 0101		
3	U2 Cache Miss with Writeback Counter	10
4	U2 Cache Invalidate from UPA Transaction	11
5	U2 Cache Unsolicited Copyback Counter	12
pm1_view_# = 0110		
3	U2 Cache Hit with "read to own" UPA Transaction Counter	13
4	Non-cacheable Load Counter	15
5	Non-cacheable Store Counter	16

Table 94: **View Selection for Monitor Group 0**

Register #	Monitor	Counter #
pm0_view_# = 0000		
0	Instruction Commit Counter	28
1	Instruction Issue Counter	27
2	Performance Monitoring Cycle Counter	26
pm0_view_# = 0001		
0	Instruction Commit Counter	28
1	Instruction μTLB Miss Counter	22
2	Instruction main TLB Miss Counter	24
pm0_view_# = 0010		
0	Instruction Commit Counter	28
1	Fetch Stall Counter	29
2	PSU_KILL Stall Counter	30
pm0_view_# = 0011		
0	Instruction Commit Counter	28
1	Reservation Queue Stall Counter	31
2	Free Register Resource Stall Counter	32
pm0_view_# = 0100		
0	Instruction Commit Counter	28
1	Checkpoint, Serial Number, or Trap Stack Resource Stall Counter	33
2	Other Stall Counter	34
pm0_view_# = 0101		
0	Instruction Commit Counter	28
1	Branch Issue Counter	35
2	Branch Mispredict Counter	36
pm0_view_# = 0110		
0	Instruction Commit Counter	28
1	ILT Miss Counter	37
2	I0 Instruction Cache Miss Counter	14
pm0_view_# = 0111		
0	Instruction Commit Counter	28
1	L1 Instruction Cache Reload Counter	6
2	L1 Instruction Cache Invalidate Counter	20
pm0_view_# = 1000		
0	Instruction Commit Counter	28
1	Sync Event Counter	38
2	Sync Cycle Counter	39

Q.3.2 Writing to Performance Registers

Performance monitors may be written by the `wr %asr30` instruction in a restricted operation. See 6.2, "Instruction Formats" for a detailed description of the format. The bits[12:8] are interpreted as the opcode or pm_op field, which is defined in Table 95.

Table 95: Interpretation of op field on WR %asr30

pm_op	Operation
00000	Disable all performance counters
00001	Clear and disable all performance counters
00010	Enable all performance counters
00011	Clear and enable all performance counters
00100	Update view number selecting counter group

The wr %asr30 will not modify software visible register unless the opcode field is '00100_2.' With the PM_OP field equal to '00100_2', the exclusive-or'ed value of rs1 and rs2 will be written to "View Number Register." All writes to %asr30 are syncing.

Q.3.3 Privilege Protection

In order to control access to sensitive performance values, reading and writing to %asr30 may be restricted to privileged code by setting the PM_US field (bit-13) in the State Control Register (ASR31) (see 5.2.11.12, "State Control Register (ASR 31)").

Q.3.4 User Access to Monitors

User access to monitors is most easily accomplished via an operating system interface. This may be done via a loadable module on the Solaris operating system. To prevent counter overflow, the loadable module should periodically accumulate the counters into 64-bit memory locations and clear the counters as needed.

Q.3.5 Performance Counter Enable

Performance counter can be enabled either on User mode only or on Supervisor mode only, or on both User and Supervisor mode through the PMEN_SEL field (bits 15:14) of the State Control Register (ASR31) (see 5.2.11.12, "State Control Register (ASR 31)"). This feature allows the separate measurement of User and Supervisor activity.

Care must be taken when interpreting the separated results. Consider the case of a measurement of Supervisor activity and the switch from User to Supervisor mode occuring via a window spill or fill trap (non-syncing). SPARC64-III can have up to 64 active instructions in the machine, issued before the trap is taken, for example. These instructions will have been issued in User mode and so will not have incremented the Supervisor mode issue count. The processor may stall during the spill or fill trap, due to full queues, for example. Or a main TLB miss may be generated, which will sync the machine. In this case, instructions which were *issued* in *User* mode will be *committed* in *Supervisor* mode. This effect also occurs when transitioning from Supervisor mode to User mode, when

instructions *issued* in *Supervisor* mode will be *committed* in *User* mode. However, the effects may not balance out due to the different nature of User and Supervisor code. Typically Supervisor code contains more syncing instructions on the average than User code which means that the average number of active instructions in the machine in Supervisor mode will be less than User mode. Also the amount of time spent in Supervisor mode is typically much smaller than that spent in User mode. This can bias the Supervisor mode counts for certain applications.

Q.4 Performance Monitor Accuracy

Q.4.1 Syncing Overhead

As specified earlier, reading and writing the performance monitors are performed through rd/wr %asr30 instructions which sync the machine. The actual time required to sync the CPU is not deterministic and is dependent on the number and type of outstanding instructions. Additionally, during the period from detection of a rd/wr %asr30 instruction to when the machine is finally synced and the rd/wr %asr30 is executed, the performance monitors will continue to log specified events.
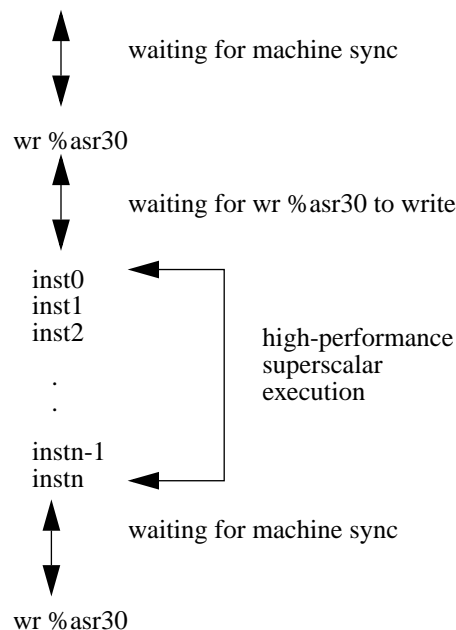
As a result, the syncing overhead from rd/wr %asr30 instructions may affect the accuracy of IPC measurements. We suggest a large enough monitoring interval so that the impact of syncing rd/wr %asr30's is negligible in these cases.

For performance counters based on execution and issue events there is no impact from syncing rd/wr %asr30.

Q.4.2 Monitoring Interval

All CPU performance monitors are enabled and disabled through wr %asr30 instructions. For purposes of performance measurement, the wr %asr30 instruction marks the measurement region. The wr %asr30 instruction itself is never counted. For example, the following code sequence:

```
wr %asr30          ! enable
add
add
load
wr %asr30          ! disable
```

measurement period
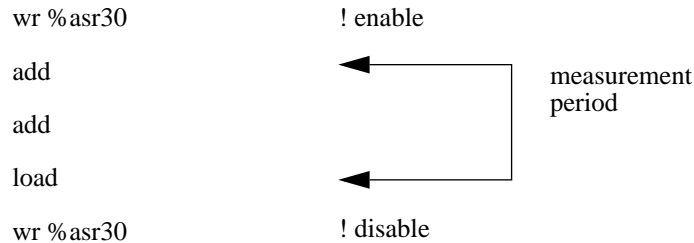
would result in 3 instructions being logged in the committed instruction counter.

When performance monitors are cleared, the value in all performance monitors will remain at 0 until they are enabled.

Q.5 Other Notes:

Q.5.1 Out-of-Order Execution

The SPARC64-III CPU implements out-of-order execution through restricted dataflow. As a result, performance monitors which are updated due to instruction execution (memory event counters, memory latency counters) will be updated out-of-order. Since execution of rd %asr30 will sync the machine, contents of performance monitors will appear to have been updated in program order when reads are initiated.

R UPA Programmer's Model

R.1 Introduction

This chapter describes the programmers model of the UC. The different registers maintained by the UC are described. Instructions which are handled by the UC are explained.

R.2 UPA PortID Register

This is a standard read only register, which is accessible by a slave read from another UPA port. This register is located at word address 0x0 in the slave physical address of the UPA port. This register cannot be read or written by ASI instruction.

FC_{16}	—	E C C	O N E	R D Q	PREQ _DQ	PREQ _RQ	UPACAP	ID
63 56	55	35 34	33	31 30	25	24 21	20 16	15 0
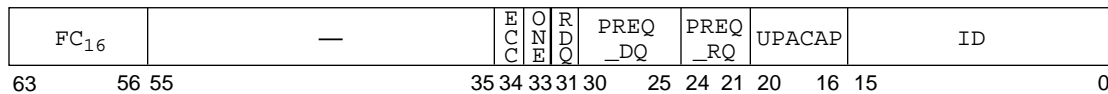
Bit[63:56]

Value=FC_{16}.

Reserved

Reserved bits, read as 0.

ECC = ECCNotValid

Indicates that this UPA port does not support ECC. Set to zero.

ONE = ONE_READ

Indicate that this UPA port supports only one outstanding slave read P_REQ transaction at a time. Set to zero.

RDQ = PINT_RDQ[1:0]

Encodes the size of the PINT_RQ and PINT_DQ queues. Specifies the number of incoming P_INT_REQ requests that the slave port can receive. Specifies the number of 64-byte interrupt datums the UPA slave port can receive. Set to one, since only one interrupt transaction can be outstanding to UC at a time.

PREQ_DQ[5:0]

Encodes the size of PREQ_DQ queue. Specifies the number of incoming quad words the UPA slave port can receive in its P_REQ write data queue. Set to zero, since incoming slave data writes are not supported by UC.

PREQ_RQ[3:0]

> Encodes the size of PREQ_RQ queue. Specifies the number of incoming P_REQ transaction request packets the UPA slave can receive. Set to one, since only one incoming P_REQ to the UC can be outstanding at a time.

UPACAP[4:0]

> Indicates the UPA module capability type:

> **UPACAP[4]**
>> Set, CPU is an interrupt handler.

> **UPACAP[3]**
>> Set, CPU is an interrupter.

> **UPACAP[2]**
>> Clear, CPU does not use UPA Slave_Int_L signal.

> **UPACAP[1]**
>> Set, CPU is a cache master.

> **UPACAP[0]**
>> Set, since CPU has a master interface.

ID[15:0]

> Module identification field:

> **ID<15:10>**
>> Manufacturer identification. These bits are set to '010000'.

> **ID<9:4>**
>> Module type

> **ID<3:0>**
>> Module revision number

R.3 UPA Config Register

This is an implementation specific read - write register. This is only accessible to the master and, not accessible for a slave read. The PCON field is readable and writable and the rest of the fields are either scanned in or hard wired.

—		UC Size	MCAP	C	—		PCON	MID	UC_CAP	
63		46	45 43	42	39	36	29 28	22 21	17 16	0
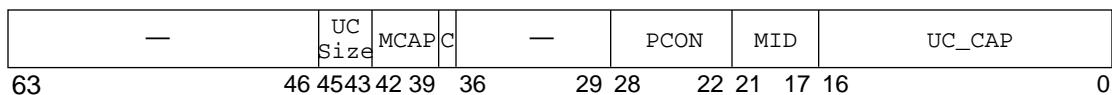
UC_Size [2:0]

Specifies the UC size. The values are scanned in.

UC_Size[2:0]	Cache Size (bytes)
000	*reserved*
001	1 Mb
010	2 Mb
011	4 Mb
100	8 Mb
101	16 Mb
110	*reserved*
111	*reserved*

MCAP

The CPU module speed; its value is scanned in.

C = CLK_MODE

Read-only field; specifies the ratio between the CPU clock and the UPA clock

CLK_MODE	Ratio
00	2:1
01	3:1
10	4:1
11	5:1

PCON

Processor Configuration.

SCIQ1[3:0] (UPA_CONFIG bits[28:25])

Size of the input request queue for master request class 1, implemented on the System Controller to which this UPA port is connected.

SCIQ1[3:0]	Queue Size
0000	1
0001	2
0010	3
0011	4
0100	5
0101	6
0110	7
0111	8
1XXX	*undefined*

SCIQ0[2:0] (UPA_CONFIG bits[24:22])

Size of the input request queue for master request class 0, implemented on the System Controller to which this UPA port is connected.

SCIQ0[2:0]	Queue Size
000	1
001	2
010	3
011	4
1XX	*undefined*

MID[4:0]

Module (Processor) ID register. Identifies the slot in which the module resides; hardwired to the slot number from the connector.

UC_CAP[16:0]

Mirrors the following fields in the UPA Port ID register:

◆ [16:15] PINT_RDQ

◆ [14:9] PREQ_DQ

◆ [8:5] PREQ_RQ

◆ [4:0] UPA_CAP

R.4 ASI Instructions for UPA Related Registers

This section describes the ASI instructions defined for UPA related registers. The following things are common to all of the ASI instructions defined in this section.

■ The opcode of the instructions should be either `ldx(a)`, `lddf(a)`, `stx(a)`, or `stdf(a)`. Otherwise, a *data_access_exception* trap with FTYPE=F_{16} (Invalid ASI) is taken.

■ No address translation is performed for the instructions.

■ VA[3:0] of all of the instructions should be 4h0. Otherwise a *mem_not_aligned* trap is taken.

■ The don't-care bits (described as '—' in the format) in VA can be any value. But it is recommended that the software should use zero for these bits.

■ The don't-care bits (described as '—' in the format) in DATA are read as zero and ignored on write.

■ The instruction operations are not affected by PSTATE.CLE. They are always treated as in a big endian mode.

■ The instructions do not cause the processor to sync.

■ The instructions are all strongly ordered regardless of load or store, and the memory model. Therefore no speculative executions are performed.

R.4.1 Read/Write UPA Configuration Register

Function: Read/Write UPA Configuration Register. See R.3, "UPA Config Register" for details.

ASI: $4A_{16}$

RW: Supervisor Read, Supervisor Write.

VA: See below.

—	0000

63   4 3   0

Data: See R.3, "UPA Config Register" for details.

Bibliography

General References

For general information, see the following:

Boney, Joel. "SPARC Version 9 Points the Way to the Next Generation RISC," *SunWorld*, October 1992, pp. 100-105.

Chien, Chen, Yizhi Lu, Anthony Wong. "Microarchitecture of HAL's Cache Subsystem." *Proceedings of Compcon*, 1995, p. 267.

Chih-Wei Chang, David, David Lyon, Charles Chen, Leon Peng, Mehran Massoumi, Matthew Hakimi, Satish Iyengar, Ellen Li, Roque Remedios. "Microarchitecture of HAL's Memory Management Unit." *Proceedings of Compcon*, 1995, p. 272.

Cohen, D., "On Holy Wars and a Plea for Peace." *Computer* 14:10, October 1981, pp. 48-54.

Comer, Douglas. "The Ubiquitous B-Tree." *ACM Computing Surveys*, Vol. 11, No. 2, June 1979.

Gwennap, Linley. "HAL Reveals Multichip SPARC Processor." *Microprocessor Report*, March 6, 1995, p. 1.

Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x, SPARC International, Inc.

Knuth, Donald. *The Art of Computer Programming, Volume 3, Searching and Sorting.* Addison-Wesley, 1974.

Patkar, Niteen, Akira Katsuno, Simon Li, Tak Maruyama, Sunil Savkar, Mike Simone, Gene Shen, Ravi Swami, DeForest Tovey. "Microarchitecture of HAL's CPU." *Proceedings of Compcon*, 1995, p. 259.

Saxena, Nirmal, Chih-Wei Chang, David, Kevin Dawallu, Jaspal Kohli, Patrick Helland. "Fault Tolerant Features in the HAL Memory Management Unit." *IEEE Transactions on Computers*, Vol. 44, No. 2, 1995, pp. 170-180.

Simone, Mike, Andrew Essen, Atsushi Ike, Anand Krishnamorty, Niteen Patkar, Murugappan Ramaswami, Viji Thirumaliswamy. *Implementation Trade-offs in Using a*

Restricted Data Flow Architecture in a High Performance RISC Microprocessor. ISCA 1995, Italy.

[Weaver, David L., editor.] *The SPARC Architecture Manual, Version 8*, Prentice-Hall, Inc., 1992.

Weaver, David L., and Tom Germond, eds. *The SPARC Architecture Manual-Version 9*, Prentice-Hall, Inc., 1994.

Wilcke, Winfried W. "Architectural Overview of HAL Systems." *Proceedings of Compcon*, 1995, p. 251.

HAL Publications

Hal Computer Systems, Inc. maintains an extensive collection of reference materials that further describe the SPARC64-III and its components. Some of these documents are available to the general public, others require the execution of a non-disclosure agreement before they are made available. Information is available in the following categories:

- High-level descriptions of SPARC64-III internal components at a conceptual level.

- Low-level descriptions of SPARC64-III internal components at an electrical and/or mechanical level.

- Application Notes describing hardware or software techniques useful or necessary for SPARC64-III.

In addition, the following specific documents and papers may be useful:

SPARC64™ Processor User's Guide, HAL Computer, #840-00003

G. Shen, N. Patkar, et al., "A 64b 4-Issue Out-of-order Execution RISC processor", *ISSCC Digest of Tech. Papers*, pp. 170-171, February 1995.

C. Asato, R. Montoye, et al., "A 14-Port 3.8ns 116-Word Read-Renaming Register File", *ISSCC Digest of Tech. Papers*, pp. 104-105, February 1995.

N. R. Saxena, et. al., "Error Detection and Handling in a Superscalar, Speculative Out-of-Order Execution Processor System", submitted to 25th annual International Symposium on Fault-Tolerant Computing, June 1995.

M. Simone, et. al., "Implementation Trade-offs in Using a Restricted Dataflow Algorithm for a High Performance RISC Processor, International Symposium on Computer Architecture, June 1995 (unpublished).

H. Li, et. al., TRACK VI - HAL Computer Systems, *COMPCON95 Digest of Papers*, pp. 251-272, March 1995

You can obtain information about the available documents using the following methods:

U. S. Mail:
 HAL Computer Systems, Inc.
 ATTN: SPARC64-III Product Marketing
 1315 Dell Avenue
 Campbell, CA 95008-6609

Voice:
 (408) 379-7000
 Ask for SPARC64-III Product Marketing

e-mail:
 marketing@hal.com

WWW:
 http://www.hal.com

Index

E

S

T