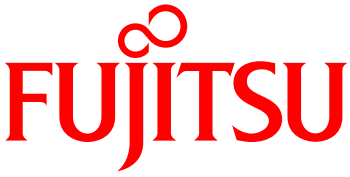


SPARC64™ VIIIfx Extensions

Fujitsu Limited

Ver 15, 26 Apr. 2010

Fujitsu Limited
4-1-1 Kamikodanaka
Nakahara-ku, Kawasaki, 211-8588
Japan

History

2009/09/08	Ver 14 released.	
2009/11/06	Added Comaptibility Note for SXAR1 instruction with non-zero s_* fields.	133
2009/11/06	Fixed typographical error in the description of exception conditions. Changed “cexc” to “aexc”.	74
2010/01/20	Fixed wrong description of SIMD load. A SIMD load does not update the basic or extended destination registers when a *data_access_error* occurs in the extended load.	82, 86, 149, 181
2010/04/13	Clarified that an `XFILL` instruction does not signal a *data_access_error* when the L1/L2 cache line contains an UE.	135
2010/04/21	Updated cache size to 6M/12way.	12, 231, 328
2009/04/26	Ver. 15 released.	

Contents

F.CHAPTER **1**

Overview

1.1 Navigating*the SPARC64™ VIIIfx Extensions*

The SPARC64 VIIIfx processor implements the instruction set architecture conforming to SPARC JPS1. The SPARC JPS1 book is organized in major sections: **Commonality**, which contains information common to all implementations, and various **Implementation Extensions**. This document defines the SPARC64 VIIIfx implementation of JPS1. As a general rule, this document does not reproduce information specified in **Commonality**.

Chapter and section headings generally match those in JPS1 **Commonality**; they describe implementation-dependent features, undefined features, or features that have been changed in SPARC64 VIIIfx. Any chapter or section not found in JPS1 **Commonality** describes additional features specific to SPARC64 VIIIfx. This document assumes the definitions provided in JPS1 **Commonality**. Please refer to the “SPARC Joint Programming Specification 1 (JPS1): Commonality” (JPS1 **Commonality**) as needed.

1.2 Fonts and Notational Conventions

This document conforms to the notational conventions specified in JPS1 **Commonality**.

Reserved Fields

Unused bits in instruction words and registers are reserved for future use. These fields are called reserved fields and are indicated by either the word “reserved” or an em dash (—).

Chapter 2 of JPS1 **Commonality** defines the following behavior for reserved fields.

- Reserved instruction fields shall read as 0. Behavior is undefined for nonzero values (Chapter 2).
- Reserved register fields should always be written by software with values of those fields previously read from that register or with zeroes; they should read as zero in hardware.

Reserved instruction fields are described in greater detail in Section 6.3.9 and Appendix I.2 of JPS1 **Commonality**.

SPARC64 VIIIfx handles reserved fields in the following manner.

- Reserved instruction fields behave as specified in this document. When behavior is not clearly specified for nonzero values, the reserved fields are ignored during instruction execution.
- Reserved register fields behave as specified in this document. When values and behavior are not specified, writes to the fields are ignored, and reads return undefined values. The behavior of writes with unspecified side effects is undefined.

Register Field Read-Write Attributes

The read-write attributes of register fields are defined below.

TABLE 1-1 Register Field Read-Write Attributes

Type	Description
	Reads return an undefined value; writes are ignored. Corresponds to a reserved register field whose value is not specified.
R	Reads to the field return the stored value; writes are ignored.
W	Reads return an undefined value; values can be written to the field.
RW	Reads to the field return the stored value; values can be written to the field.
RW1C	Reads to the field return the stored value; writing a value of 1 causes 0 to be written to the field.

F.CHAPTER **2**

Definitions

This chapter defines concepts and terminology specific to SPARC64 VIIIfx. For the definition of terms common to all implementations of JPS1, please refer to Chapter 2 of JPS1 **Commonality**.

basic floating-point registers Additional floating-point registers defined by HPC-ACE that can be used for SIMD basic operations. Registers `f[0]` – `f[254]`.

committed An instruction is said to be committed when all instructions ***executed*** prior to the instruction have ***committed*** normally and the result of the instruction is definitively known. The instruction commits and the result is reflected in software-visible resources; the previous state is discarded.

completed An instruction is said to be completed when ***execution is completed*** and the issue unit is notified that execution completed normally. The result of a completed instruction is temporarily reflected in the machine state; however, until the instruction ***commits*** the state is not permanent and the previous state can be recovered.

core A hardware structure that contains the processor pipeline and execution resources (functional units, L1 cache, etc). While a core may support one or more ***threads***, SPARC64 VIIIfx cores are single-threaded.

cycle accounting A method for analyzing the factors that are inhibiting performance.

execute To send an instruction to the execution unit and to perform the specified operation. An instruction is executing as long as it is in a ***functional unit***.

execution completion Execution is completed when the result appears on the output bus. The result on the output bus is sent to the register file as well as the other functional units.

extended floating-point registers Additional floating-point registers defined by HPC-ACE that can be used for SIMD extended operations. Registers `f[256]` – `f[512]`.

functional unit A resource that performs arithmetic operations.

HPC-ACE High Performance Computing - Arithmetic Computational Extensions. This is the general term for the set of SPARC64 VIIIfx extensions; these include the expanded register set, HPC instruction extensions, floating-point SIMD extensions, etc.

instruction dispatch To send an instruction to the execution unit. All resources required for execution of the instruction must be available.

instruction fetch To read an instruction from the instruction cache or instruction buffer and to send it to the issue unit.

instruction issue To send an instruction to a reservation station.

Memory Management Unit The address translation hardware in the processor ***core*** that translates 64-bit virtual addresses to physical addresses. The MMU includes the ***mITLB***, ***mDTLB***, ***uITLB***, ***uDTLB***, and ASI registers used to manage address translation.

mTLB Main TLB. The ***mTLB*** is split; the structures supporting instruction (I) and data (D) accesses are called the ***mITLB*** and ***mDTLB***, respectively. These supply the ***uITLB*** and ***uDTLB*** with address translations. When an address translation is not found in the ***uITLB*** or ***uDTLB***, the ***mTLB*** is searched for the missing translation. If the requested translation is found, the ***mTLB*** sends the translation to the corresponding ***uTLB***. Otherwise, an exception occurs and causes a trap. Software loads the translation into the ***mTLB***, and hardware re-executes the instruction.

out-of-order execution A microarchitecture that supports the ***execution*** of instructions out of program order. An instruction with available operands will ***execute*** ahead of an earlier instruction that is still waiting for operands.

processor module A single, physical module for processing information. A processor module is composed of one or more cores sharing an external bus.

renaming registers A buffer where execution results are temporarily stored until instructions commit and their results are written to the register file. Users cannot directly manipulate the renaming registers.

reservation station A queue (or buffer) where ***issued*** instructions are stored before being sent to the execution unit. When possible, instructions with available operands are dispatched from reservation stations to available ***functional units***. Reservation stations control ***out-of-order execution***.

(resource) release An ***execution*** resource assigned to an instruction is said to be released when it can be assigned to a subsequent instruction.

scan A method for reading and writing latches and registers inside the CPU chip. Scannable latches and registers can be read and written through a scan ring.

(SIMD) basic operation One of two operations executed by a SIMD instruction. The basic operation uses the registers indicated by the register number fields of the instruction.

(SIMD) extended operation One of two operations executed by a SIMD instruction. The extended operation uses the registers indicated by the register number fields of the instruction +256.

speculative execution Execution is said to be speculative if an instruction is ***executed*** while the direction of an older conditional branch is unknown, or while it is unknown whether an older instruction will cause an interrupt or trap to occur. An instruction that is ***executed*** using the result of a speculatively-executed instruction is also said to be speculatively executed.

stalled An instruction is said to be stalled when it is unable to issue. Depending on resource availability and program constraints, it may not be possible to issue instructions every cycle.

strong prefetch A data prefetch instruction that guarantees eventual execution. The instruction is re-executed if there are insufficient processor resources, instead of being discarded.

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

suspended A state where execution of a thread is temporarily stopped. In a suspended state no instructions are executed, but cache coherency is maintained. Suspended differs from ***sleeping***; for execution of the suspended thread to resume, an interrupt or the timer must cause a trap.

syncing instruction An instruction that causes a ***machine sync***. A **syncing** instruction ***issues*** in program order; all prior instructions must be ***committed*** before the **syncing** instruction issues. Furthermore, the following instruction does not ***issue*** until the **syncing** instruction has been ***committed***. That is, a **syncing** instruction is an instruction that ***issues***, ***executes***, and ***commits*** by itself.

thread The unit of hardware required for execution of a software instruction sequence. A thread includes software-visible resources (PC, registers, etc) and any non-visible microarchitectural resources required for instruction execution.

uTLB Micro TLB. The ***uTLB*** is split; the structures supporting instruction (I) and data (D) accesses are called the ***uITLB*** and ***uDTLB***, respectively. Hardware performs address translation using the address translations in the ***uTLB***. When a required translation is not found, the ***uTLB*** obtains the translation from the ***mTLB***.

XAR-eligible instruction An instruction that is executed using the registers specified by the combination of the bits in the `XAR` and the bits from the register number fields.

F.CHAPTER **3**

Architectural Overview

This chapter provides an overview of the SPARC64 VIIIfx processor. The section headings do not match those in JPS1 **Commonality**.

3.1 The SPARC64 VIIIfx processor

The multi-core SPARC64 VIIIfx processor integrates 8 cores, L2 cache, and memory controllers (MAC) on a single CPU chip. The processor architecture conforms to SPARC V9 but includes extensions that enhance server performance and reliability and that significantly boost performance on HPC workloads.

A High Performance Microarchitecture

SPARC64 VIIIfx is an an out-of-order, superscalar processor. Each core issues up to four instructions per cycle; the instruction fetch unit predicts the execution path, fetches instructions, and issues the instructions in-order to reservation stations. Instructions are stored in the reservation stations until they are ready to be executed. Ready instructions are dispatched to the execution unit and executed out of order. Instructions that have completed execution are committed in the original order; that is, an instruction does not commit until all prior instructions have committed. Committed instructions update the register file and/or memory, and the execution result becomes visible to the program. Out-of-order execution contributes greatly to the high performance of SPARC64 VIIIfx.

The SPARC64 VIIIfx core has a branch history buffer for predicting the execution path of branch instructions. This buffer is large enough to sustain high hit rates for large programs like DBMS and to support SPARC64 VIIIfx's sophisticated instruction fetch mechanism. The fetch mechanism minimizes the performance penalty of instruction cache misses by using the branch history buffer to predict the direction of multiple conditional branches and fetching the instructions in the predicted execution path.

SPARC64 VIIIfx processor incorporates many useful features for HPC (High Performance Computing), which include the HPC-ACE extensions to SPARC V9 and a hardware barrier for high-speed synchronization of on-chip cores. HPC-ACE expands the number of registers to 192 general-purpose and 256 floating-point registers per core, defines 7 new floating-point instructions, and supports 2-way SIMD (Single Instruction Multiple Data) execution of floating-point instructions. With SIMD execution, up to 8 floating-point operations can be executed per cycle per core. This realizes high performance on HPC workloads.

Highly-Integrated Functionality

The lowest level of the SPARC64 VIIIfx cache hierarchy is the on-chip L2 cache. Instruction and data accesses are unified, and the L2 cache is shared by all 8 cores. Having the L2 cache on chip decreases the cache access time and allows for a high associativity cache to be designed. Futhermore, it increases reliability by eliminating the need for external connections to the L2 cache.

SPARC64 VIIIfx also includes on-chip memory controllers. DIMMs are connected directly to the CPU, which significantly decreases memory access latencies.

The hardware barrier is an important feature for ensuring good performance on HPC workloads. The SPARC64 VIIIfx hardware barrier enables high-speed processing of multi-threaded jobs by minimizing thread synchronization latencies; it supports barrier synchronization of multiple cores and provides post/wait synchronization primitives for implementing the master/worker model.

High Reliability

SPARC64 VIIIfx implements the following advanced RAS features:

1. Cache RAS features

- Robust protection against cache errors
  - D1 (data level-1) cache data, U2 (unified level-2) cache data, and U2 cache tags are ECC protected.
  - I1 (instruction level-1) cache data are parity protected.
  - I1 cache tags and D1 cache tags are parity protected and duplicated.
- Automatic correction for all types of single-bit errors
  - Single-bit errors in ECC-protected data are automatically corrected.
  - I1 cache data parity errors cause I1 cache data to be invalidated and re-read.
  - I1 cache tag and D1 cache tag parity errors cause the tags to be replaced with the duplicated cache tags.
- Dynamic way reduction while maintaining cache consistency
- Marking uncorrectable errors in cacheable data
  - Hardware that first detects the uncorrectable error marks the error with a particular pattern.

- The hardware that detected the error is identified from the pattern and isolated to prevent the same error from being reported multiple times.

2. RAS features for the core

- Robust error protection
  - All data paths are parity protected.
  - Almost all software-visible registers, internal registers, and temporary registers are parity protected.
  - Execution results are checked by parity prediction or residue checks.
- Hardware instruction retry
- Support for software instruction retry (if hardware instruction retry fails)
- Error isolation for software recovery
  - The register that caused the error (suspected register) is indicated.
  - Indicates whether the instruction that caused the error can be retried.
  - Different traps are used depending on the severity of the error.

3. Enhanced software interface

- Error classification based on how severely program execution is affected
  - Urgent error (nonmaskable): Unable to continue execution without OS intervention; reported by a trap.
  - Restrainable error (maskable): OS controls whether the error is reported by a trap. The error does not directly affect program execution.
- Displaying identified errors to help determine their effect on software
- Asynchronous data error (ADE) exception for indicating additional errors
  - The exception halts execution and indicates the completion method for the instruction that signalled the exception. The completion method depends on the detected error.
  - ADE exceptions may be deferred but retryable.
  - To correctly perform error isolation and instruction retry, all simulatenously occurring errors are displayed.

3.1.1 Core Overview

The SPARC64 VIIIfx block diagram is shown in FIGURE 3-1. SPARC64 VIIIfx has 8 cores, on-chip memory controllers, and an integrated bus interface. Each core has the following components:

- Instruction control unit (IU)
- Execution unit (EU)
- Storage unit (SU)

The following component is shared by all cores:

- Secondary cache and external access unit (SXU)

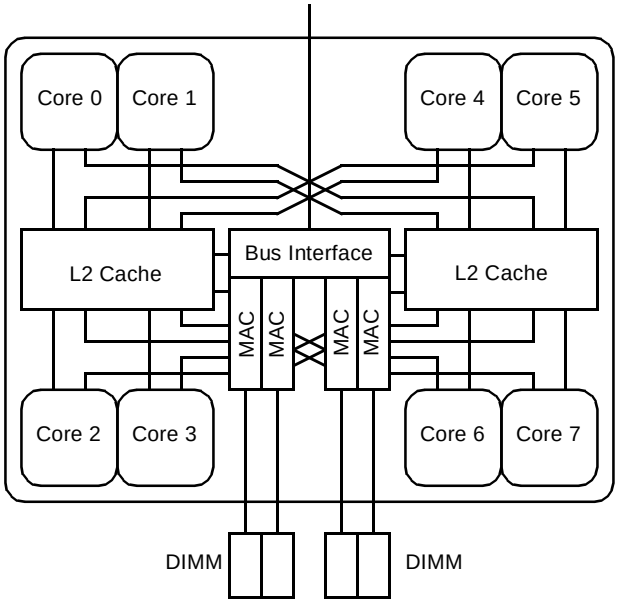


FIGURE 3-1 SPARC64 VIIIfx Block Diagram

3.1.2 Instruction Control Unit (IU)

The IU predicts the instruction execution path, fetches the predicted instructions, delivers the fetched instructions to the appropriate reservation stations, and dispatches instructions to the execution unit. Dispatched instructions are executed out of order, and the completed instructions are committed in order. The major blocks are described in TABLE 3-1.

TABLE 3-1 Major Blocks in the Instruction Control Unit

Name	Description
Instruction fetch pipeline	5-stage instruction fetch: fetch address generation, iTLB and L1 I-cache access, iTLB and L1 I-cache tag match, write to the instruction buffer, and store the result.
Branch history	A table for predicting branch target and direction.
Instruction buffer	A buffer for holding fetched instructions.

TABLE 3-1 Major Blocks in the Instruction Control Unit

Name	Description
Reservation stations	A buffer for holding instructions until they can execute. There are 5 reservation stations: RSBR for branch and other control-transfer instructions, RSA for load/store instructions, RSE for integer arithmetic instructions, and RSFA and RSFB for floating-point arithmetic instructions.
Commit stack entries	A buffer for holding information about in-flight instructions (issued but not committed).
PC, nPC, CCR, FSR	Program-visible registers for instruction execution control.

3.1.3 Execution Unit (EU)

The EU executes all integer arithmetic/logical/shift instructions, as well as all floating-point instructions and VIS instructions. TABLE 3-2 describes the major blocks in the EU.

TABLE 3-2 Major Blocks in the Execution Unit

Name	Description
GUB	General-purpose register (*gr*) renaming register file.
GPR	*Gr* architectural register file.
FUB	Floating-point registers (*fr*) renaming register file.
FPR	*Fr* architectural register file.
EU control logic	Controls the stages of instruction execution: instruction selection, register read, and execution.
Interface registers	Input/output registers to other units.
Two integer functional units (EXA, EXB)	64-bit ALU and shifters.
Two floating-point functional units (FLA, FLB)	Each floating-point functional unit can execute floating-point multiply, add/subtract, multiply-add/subtract, divide/sqrt, and graphics operations.
Two load/store functional units (EAGA, EAGB)	64-bit adders for load/store virtual address generation.

3.1.4 Storage Unit (SU)

The SU handles all read and write data for load/store instructions. Data is read from a data source and written to a data sink. TABLE 3-3 describes the major blocks in the SU.

TABLE 3-3 Major Blocks in the Storage Unit

Name	Description
Instruction level-1 cache	32-Kbyte, 2-way associative, 128-byte line. Low-latency instruction source.
Data level-1 cache	32-Kbyte, 2-way associative, 128-byte line. Low-latency load/store data source and sink.
Instruction Translation Buffer	256 entries, 2-way associative TLB (sITLB). 16 entries, fully associative TLB (fITLB).
Data Translation Buffer	512 entries, 2-way associative TLB (sDTLB). 16 entries, fully associative TLB (fDTLB).
Store Buffer and Write Buffer	Decouple store latency and the processor pipeline. Allow the pipeline to continue to operate without stalling for stores that are waiting for data. Data is eventually written into the data level-1 cache.

3.1.5 Secondary Cache and External Access Unit (SXU)

The SXU controls the operation of the unified level-2 cache and the external data access interface. TABLE 3-4 describes the major blocks in the SXU.

TABLE 3-4 Secondary Cache and External Access Unit Major Blocks

Name	Description
Unified level-2 cache	6-Mbyte, 12-way associative, 128-byte line. Write-back cache.
Move-in buffer	Caches data that is returned by the memory system in response to a cache-line read request.
Move-out buffer	Holds data for write-back to memory.

3.2 Processor Pipeline

SPARC64 VIIIfx has a 16-stage pipeline, which is shown in FIGURE 3-2 and the pipeline diagram in FIGURE 3-3.

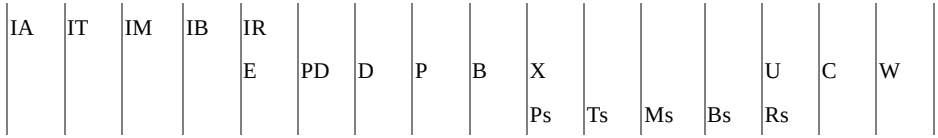


FIGURE 3-2 SPARC64 VIIIfx pipeline stages

3.2.1 Instruction Fetch Stages

- IA: Instruction address generation
- IT: Instruction TLB, instruction cache tag access
- IM: Instruction cache tag comparison
- IB: Instruction cache read to buffer
- IR: Instruction fetch result

Stages IA through IR work in concert with the cache access unit (SU) to read instructions and supply them to subsequent pipeline stages. Instructions fetched from memory or cache are stored in the Instruction Buffer (I-buffer).

SPARC64 VIIIfx has branch prediction resources called BRHIS (BRanch HIStory) and RAS (Return Address Stack). Instruction fetch stages use these resources to determine fetch addresses.

Instruction fetch stages are designed to work independently of subsequent stages as much as possible and can fetch instructions even when the execution stages stall. Instruction fetch continues until the I-Buffer is full, at which point the instruction fetch unit can send prefetch requests to move instructions into the L1 cache.

3.2.2 Issue Stages

- E: Entry
- PD: Pre-decode
- D: Decode

SPARC64 VIIIfx is an out-of-order processor. Each core has 6 functional units (two integer arithmetic/logical units, two floating-point units, and two load/store units). There are 2 reservation stations for floating-point instructions, 1 for integer arithmetic/logical

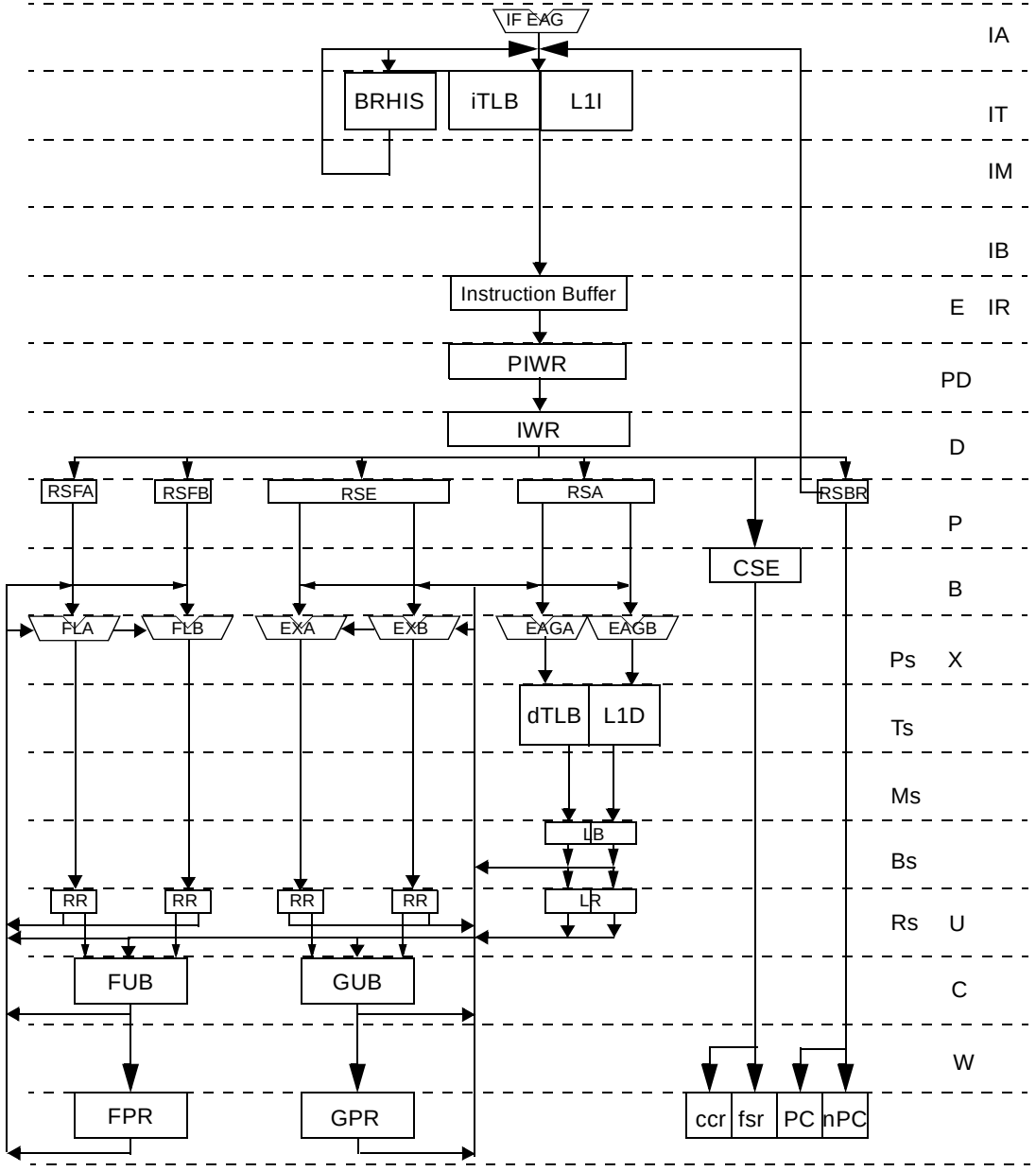


FIGURE 3-3 SPARC64 VIIIfx Pipeline Diagram

instructions, and 1 for load/store instructions. Stages E, PD, and D decode and issue instructions to the appropriate reservation station. SPARC64 VIIIfx issues up to four instructions per cycle per core.

The following resources are required for instruction execution and are assigned in the issue stages:

- Commit stack entries (CSE)
- Integer and floating-point renaming registers (GUB and FUB, respectively)
- Reservations station entries
- Memory access ports

Depending on the instruction, additional resources may be needed for execution, but all resources must be assigned in these stages. During normal execution, assigned resources are released at the last stage of the pipeline, the W-stage.[1] Instructions between the E-stage and W-stage are considered to be in flight. When an exception is signalled, all in-flight instructions and the resources assigned to them are released immediately. This allows the decoder to start issuing new instructions as quickly as possible.

3.2.3 Execution Stages

- P: Priority
- B: Buffer read
- X: Execute
- U: Update

Instructions waiting in reservation stations will be sent to fuctional units when all execution conditions are met. These conditions include knowing the values of all source data, the availability of functional units, etc. Execution latency varies from one cycle to multiple cycles, depending on the instruction.

Execution Stages for Cache Access

Memory access requests are passed to the cache access unit after the target address is calculated. Cache access stages work the same way as instruction fetch stages, except for the handling of branch prediction. See Section 3.2.1 for details. The instruction fetch stages corresponding to the cache access stages are shown below.

Instruction Fetch Stages	Cache Access
IA	Ps
IT	Ts

1. A reservation station entry is released at the X-stage.

IM	Ms
IB	Bs
IR	Rs

When an exception is signalled, memory access resources are released. The cache access pipeline continues to work to complete outgoing memory accesses. When the data is returned, it is stored in the cache.

3.2.4 Commit Stage

- W: Write

In the commit stage, instructions that were executed out of order are committed in program order. Exception handling is performed in this stage. That is, exceptions occurring in the execution stages are not handled immediately but are signalled after all prior instructions have committed.[1]

1. A RAS-related exception may be signalled before the commit stage.

F.CHAPTER **4**

Data Formats

Please refer to Chapter 4 of JPS1 **Commonality**.

F.CHAPTER 5

Registers

Chapter 5 of JPS1 **Commonality** defines three types of registers: general-purpose, ASR, and ASI registers. This chapter is divided into a section on nonprivileged registers and a section on privileged registers. While ASR and ASI registers are treated as privileged registers, this is not entirely consistent as some registers allow nonprivileged accesses. Futhermore, not all ASI registers are defined in this chapter; there are additional ASI registers defined in the Appendices.

Because the SPARC64™ VIIIfx Extensions conform to the chapter and section headings of JPS1 **Commonality** where possible, this chapter describes the implementation-dependent behavior of registers defined in Chapter 5 of JPS1 **Commonality**. For convenience, information concerning both privileged and nonprivileged ASR and ASI registers is located in Section 5.2, *"Privileged Registers"*.

Please refer to the following sections for information on additional ASI registers.

- Appendix F.10, *"Internal Registers and ASI Operations"*
- Appendix L.3.2, *"Special Memory Access ASIs"*
- Appendix L.4, *"Hardware Barrier"*
- Appendix M.3, *"Cache Control/Status Instructions"*
- Appendix N.4, *"Interrupt ASI Registers"*
- Appendix P.2.5, *"ASI_EIDR"*
- Appendix P.2.6, *"Error Detection Control (ASI_ERROR_CONTROL)"*
- Appendix P.3.1, *"ASI_STCHG_ERROR_INFO"*
- Appendix P.4.1, *"URGENT ERROR STATUS (ASI_UGESR)"*
- Appendix P.7.1, *"ASI_ASYNC_FAULT_STATUS (ASI_AFSR)"*
- Appendix R.1, *"System Config Register"*
- Appendix R.2, *"STICK Control Register"*

Appendix O.3, *"Processor State after Reset and in RED_state"*, describes register values after power-on and reset. Appendix P.8, *"Internal Register Error Handling"*, discusses register error signalling and recovery.

5.1 Nonprivileged Registers

5.1.1 General-Purpose r Registers

Registers `r[32]` – `r[63]` (`xg[0]` – `xg[31]`) are added.

There are not enough bits in the existing instruction fields to encode the new register numbers, so an additional 3 bits are stored in the `XAR.urs1`, `XAR.urs2`, `XAR.urs3`, and `XAR.urd` fields. See *"Extended Arithmetic Register (XAR) (ASR 29)"*. Since there are 32 additional registers, bits <2:1> shall be 0 for all fields. A nonzero value in bits <2:1> causes an *illegal_action* exception.

Most instructions can use the additional integer registers added by HPC-ACE. If an instruction that cannot use the HPC-ACE integer registers is executed while `XAR.v` = 1, an *illegal_action* exception is signalled.

Registers `xg[0]` – `xg[31]` are always visible regardless of the value of `PSTATE.AG`, `PSTATE.MG`, and `PSTATE.IG`.

A write to an HPC-ACE register sets `XASR.xgd` = 1.

Programming Note – When a context switch occurs, software should determine whether the HPC-ACE integer registers need to be saved.

5.1.4 Floating-Point Registers

New floating-point registers are added; all 256 double-precision floating-point registers can be used. The additional registers are numbered `f[64]` – `f[510]` (even numbers only). The `XASR` is also added; it displays the state of the additional registers. See *"Extended Arithmetic Register Status Register (XASR) (ASR 30)"* (page 33) for details.

Registers `f[0]` – `f[254]` are called the Basic Floating-Point Registers, and registers `f[256]` – `f[510]` are called the Extended Floating-Point Registers. Registers `f[0]` – `f[62]` are also called the V9 Floating-Point Registers.

Floating-Point Register Number Encoding

Double-precision register number encoding is defined in JPS1 **Commonality** under the same section heading.

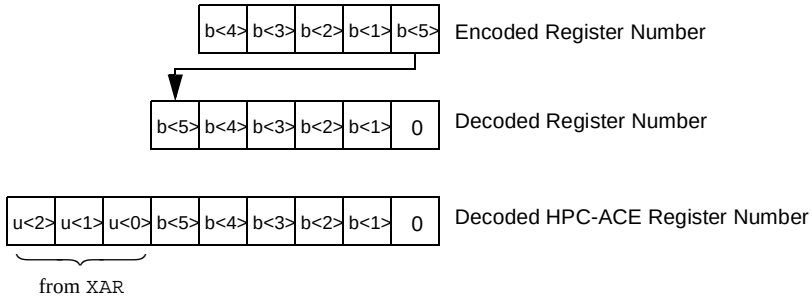


FIGURE 5-1 Double-Precision Floating-Point Register Number Encoding

There are not enough bits in the 5-bit instruction fields to specify the 256 double-precision registers defined by HPC-ACE. Instead, the upper bits of the register number are stored in the XAR, and at execution time these bits are combined. That is, the register number cannot be identified from the instruction word alone. See *"Extended Arithmetic Register (XAR) (ASR 29)"*.

A decoded HPC-ACE register number is a 9-bit number. As shown in FIGURE 5-1, the upper 3 bits from the XAR are concatenated with the decoded 6-bit register number. Since the least significant bit is always 0, all 256 even-numbered registers in `f[0]` – `f[510]` can be specified.

Using double-precision registers for single-precision operations

In SPARC64 VIIIfx, double-precision registers can be used to perform single-precision operations. This applies not only to the registers added in SPARC64 VIIIfx but also to the double-precision registers defined in SPARC V9. To use a double-precision register for a single-precision operation, it is sufficient to set `XAR.v` = 1 at execution time. Thus, a SIMD single-precision operation always uses double-precision registers.

When using a double-precision register for a single-precision operation, the following behavior differs from the SPARC V9 specification:

- The encoding of the instruction field is the same as for a double-precision register operand in TABLE 5-5 of JPS1 **Commonality**. Consequently, only even-numbered register can be used. `f[2n]` (n = 0–255)
- The upper 4 bytes of the register (the `<63:32>` operand field) are treated as a single-precision value, and the lower 4 bytes (the `<31:0>` operand field) are ignored.
- Execution results and load data are written in the upper 4 bytes, and zeroes are written in the lower 4 bytes.

Programming Note – When `XAR.v` = 1 and `XAR.urs1` = 0, the SPARC V9 double-precision register specified by `rs1` is used to perform a single-precision operation. There are similar cases for `rs2`, `rs3`, and `rd`. In these situations, the `<31:0>` operand field of the register overlaps an odd-numbered register, which will be written over with zeroes.

Endian conversion is done for each single-precision word; that is, endian conversion is done in 4-byte units.

Specifying registers for SIMD instructions

When `XAR.V` = 1 and `XAR.SIMD` = 1, the majority of instructions that use the floating-point registers become SIMD instructions. One SIMD instruction executes two floating-point operations. Registers used for SIMD instructions must be register pairs of the form `f[2n]` and `f[2n+256]` (n = 0–127). The `f[2n]` register number is specified by the instruction. An *illegal_action* exception is signalled when an unusable register is specified.

The SIMD FMADD instruction is special; f[2n+256] registers can be specified for `rs1` and `rs2`. See Appendix A.24.1, *"Floating-Point Multiply-Add/Subtract"*, for details.

Programming Note – Single-precision floating-point instructions support SIMD execution; however, double-precision registers must be used. See *"Using double-precision registers for single-precision operations"* (page 21) for details.

Of the existing floating-point instructions, the following instructions do not support SIMD execution. See TABLE A-2 for the list of instructions that do support SIMD execution.

- `FDIV`(S,D), `FSQRT`(S,D)
- VIS instructions that are not logical operations
- Instructions that reference and/or update `fcc`, `icc`, `xcc` (`FBfcc`, `FBPfcc`, `FCMP`, `FCMPE`, `FMOVcc`, etc.)
- `FMOVr`

The floating-point operation that stores its result in `f[2n]` is called the basic operation. The floating-point operation that stores its result in `f[2n+256]` is called the extended operation.

Endian conversion is performed separately for the basic and extended floating-point registers.

5.1.7 Floating-Point State Register (FSR)

FSR_nonstandard_fp (NS)

SPARC V9 defines the FSR.NS bit. When set to 1, this bit causes a SPARC V9 FPU to produce implementation-defined results that may not correspond to IEEE Std 754-1985. SPARC64 VIIIfx implements FSR.NS.

When FSR.NS = 1, a subnormal source operand or subnormal result does not cause an *fp_exception_other* exception with ftt = *unfinished_FPop*. Instead, the subnormal value is replaced with a floating-point zero value of the same sign and an *fp_exception_ieee_754* exception with fsr.cexc.nxc = 1 is signalled (maskable by FSR.TEM.NXM). See Section B.6, *"Floating-Point Nonstandard Mode"* (page 142) for details.

When FSR.NS = 0, the behavior of the FPU conforms to IEEE Std 754-1985.

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. In the initial version of SPARC64 VIIIfx, FSR.ver = 0 (impl. dep. #19). FSR.ver may have different values in future versions. Consult the SPARC64 VIIIfx Data Sheet for details.

FSR_floating-point_trap_type (*ftt*)

In SPARC64 VIIIfx, the conditions under which an *fp_exception_other* exception with FSR.ftt = *unfinished_FPop* can occur are described in Appendix B.6.1, *"fp_exception_other Exception (ftt=unfinished_FPop)"* (impl. dep. #248).

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.

The following pseudocode shows how SPARC64 VIIIfx sets the cexc bits:

```
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[1] in the CEXC field as supplied by FPU>;
else if (<FPop commits with unfinished_FPop error>)
   <no change>;
```

```
else if (<FPop commits with unimplemented_FPop error>)
   <no change>;
else
   <no change>;
```

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). SPARC64 VIIIfx chooses implementation method (1), which implements all three fields conformant to IEEE Std 754-1985. See Section 5.1.7 of JPS1 **Commonality** for the other implementation method.

Updates to cexc, aexc by SIMD Instructions

Basic and extended operations are performed simultaneously. However, because the source operands are different, either operation could cause an exception or both could cause exceptions.

When only one operation causes an exception, the same action is taken as for a non-SIMD instruction. When both operations cause exceptions, the following exceptions may be signalled by SPARC64 VIIIfx SIMD instructions; cexc and aexc are updated as shown below.

1. *fp_exception_ieee_754* exceptions are detected for both basic and extended operations.

   For the purposes of illustration, the exception caused by the basic operation is indicated in the hypothetical basic.cexc field. The exception caused by the extended operation is indicated in the hypothetical extend.cexc field. Each has bits for uf/of/dz/nx/nv.

   a. Both exceptions are masked and no exception is signalled.

      The logical OR of basic.cexc and extend.cexc is displayed in FSR.cexc. The logical OR of basic.cexc and extend.cexc is accumulated in FSR.aexc.

      ```
      FSR.cexc  ← basic.cexc | extend.cexc
      FSR.aexc  ← fsr.aexc | basic.cexc | extend.cexc
      ```

   b. Either the basic or extended operations signals an exception.

      The logical OR of basic.cexc and extend.cexc is displayed in FSR.cexc. FSR.aexc is left unchanged.

      ```
      FSR.cexc  ← basic.cexc | extend.cexc
      ```

   c. Both basic and extended operations signal exceptions.

1. For non-SIMD, 1 bit is set. Multiple bits may be set for SIMD.

The logical OR of `basic.cexc` and `extend.cexc` is displayed in `FSR.cexc`. `FSR.aexc` is left unchanged.

```
FSR.cexc  ← basic.cexc | extend.cexc
```

2. An *fp_exception_ieee_754* is detected for one operation and an *fp_exception_other* exception is detected for the other operation.

   The lower-priority *fp_exception_other* exception is signalled with `ftt` = *unfinished_FPop*. Both `FSR.aexc` and `FSR.cexc` are left unchanged.

Programming Note – When an *fp_exception_other* exceptions occurs, it is impossible for hardware to determine whether an *fp_exception_ieee_754* exception occurs simultaneously. System software must run an emulation routine to detect the second exception and update the necessary registers.

---

3. *fp_exception_other* exceptions are detected for both basic and extended operations.

   An *fp_exception_other* exception with `ftt` = *unfinished_FPop* is signalled. Both `FSR.aexc` and `FSR.cexc` are left unchanged.

Note – For a non-SIMD instruction that causes an *fp_exception_ieee_754* exception, `fsr.cexc` displays only one floating-point exception condition. For a SIMD instruction, the logical OR of the basic and extended floating-point exception conditions is displayed; that is, either one or two floating-point exception conditions may be displayed.

5.1.9 Tick (TICK) Register

SPARC64 VIIIfx implements a `TICK.counter` register with 63 bits (impl. dep. #105).

Implementation Note – In SPARC64 VIIIfx, a read of the `TICK` register returns the value displayed in `counter` when the `RDTICK` instruction *executes*, not the value when the instruction *commits* (SPARC64 VIIIfx implements out-of-order execution, so the two are clearly different). When `TICK` is read a second time, the difference between the values read from `counter` reflects the the number of processor cycles between the execution of the first and second `RDTICK` instructions. If the number of intervening instructions is large, any discrepancies between when reads were executed versus committed becomes small.

5.2 Privileged Registers

5.2.6 Trap State (TSTATE) Register

SPARC64 VIIIfx only implements bits 2:0 of the `TSTATE.CWP` field. Bits 4 and 3 read as zero, and writes to these bits are ignored.

Note – Software should not set `PSTATE.RED` = 1, as this causes an entry to `RED_state` without the required trap-related changes in the machine state.

5.2.9 Version (VER) Register

TABLE 5-1 shows the values of the `VER` register fields in SPARC64 VIIIfx.

TABLE 5-1 `VER` Register Encoding

Bits	Field	Description
63:48	`manuf`	0004_{16} (Impl. Dep. #104)
47:32	`impl`	8
31:24	`mask`	*n* (The value of n depends on the version of the processor chip.)
15:8	`maxtl`	5
4:0	`maxwin`	7

The `manuf` field displays Fujitsu's 8-bit JEDEC code; the upper 8 bits are zeroes. The values of the `manuf`, `impl`, and `mask` fields may change in future processors. The value of the `mask` field generally increases numerically with successive releases of the processor but does not necessarily increase by one for consecutive releases.

5.2.11 Ancillary State Registers (ASRs)

Please refer to Section 5.2.11 of JPS1 **Commonality** for details on the ASRs.

Performance Control Register (PCR) (ASR 16)

The SPARC64 VIIIfx specification of the PCR differs slightly from JPS1 **Commonality**. FIGURE 5-2 and TABLE 5-2 describe the SPARC64 VIIIfx implementations of JPS1 **Commonality** impl. dep. #207 and #250, as well as changes to the JPS1 **Commonality** specification of PCR.SU and PCR.SL. Bits in PCR<2:1> conform to JPS1 **Commonality**.

See Appendix Q for details on the PA Event Counters.

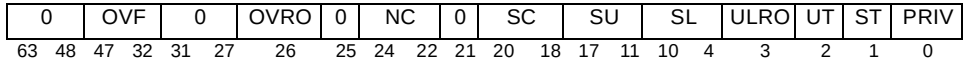


FIGURE 5-2 SPARC64 VIIIfx Performance Control Register (PCR) (ASR 16)

TABLE 5-2 PCR Bit Description

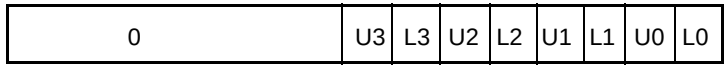
Bits	Field	Description
47:32	OVF	Overflow Clear/Set/Status. A read by RDPCR returns the overflow status of the counters, and a write by WRPCR clears or sets the overflow status bits. PCR.OVF is a SPARC64 VIIIfx implementation-dependent field (impl. dep. #207). The following figure shows the counters corresponding to the OVF bits. A write of 0 to an OVF bit clears the overflow status of the corresponding counter.  Writing a 1 via software does not cause an overflow exception.
26	OVRO	Overflow Read-Only. A write to the PCR register with write data containing a value of OVRO = 0 updates the PCR.OVF field with the OVF write data. If the write data contains a value of OVRO = 1, the OVF write data is ignored and the PCR.OVF field is not updated. Reads of the PCR.OVO field return 0. The PCR.OVRO field allows PCR to be updated without changing the overflow status. Hardware maintains the most recent state in PCR.OVF such that a subsequent read of the PCR returns the current overflow status. PCR.OVRO is a SPARC64 VIIIfx implementation-dependent field (impl. dep. #207).
24:22	NC	This read-only field indicates the number of counter pairs. In SPARC64 VIIIfx, NC has a value of 3 (indicating 4 counter pairs).
20:18	SC	PIC Pair Selection. A write updates which PIC counter pair is selected, and a read returns the current selection.

TABLE 5-2 PCR Bit Description

Bits	Field	Description
17:11	SU	This field selects the event counted by PIC<63:32>. A write updates the setting, and a read returns the current setting. The field specified in JPS1 **Commonality** is extended by 1 bit to create a 7-bit field.
10:4	SL	This field selects the event counted by PIC<63:32>. A write updates the setting, and a read returns the current setting. The field specified in JPS1 **Commonality** is extended by 1 bit to create a 7-bit field.
3	ULRO	SU/SL Read-Only. A write to the PCR register with write data containing a value of ULRO = 0 updates the PCR.SU and PCR.SL fields with the SU/SL write data. If the write data contains a value of ULRO = 1, the SU/SL write data is ignored and the PCR.SU and PCR.SL fields are not updated. Reads of the PCR.ULRO field return 0. The PCR.ULRO field allows the PIC pair selection field to be updated without changing the PCR.SU and PCR.SL settings. PCR.ULRO is a SPARC64 VIIIfx implementation-dependent field (impl. dep. #207).
2	UT	User Mode. When PSTATE.PRIV = 0, events are counted.
1	ST	System Mode. When PSTATE.PRIV = 1, events are counted. If both PCR.UT and PCR.ST are 1, all events are counted. If both PCR.UT and PCR.ST are 0, counting is disabled. PCR.UT and PCR.ST are global fields; that is, they apply to all PICs.
0	PRIV	Privileged. If PCR.PRIV = 1, executing a RDPCR, WRPCR, RDPIC, or WRPIC instruction in non-privileged mode (PSTATE.PRIV = 0) causes a *privileged_action* exception. If PCR.PRIV = 0, a non-privileged (PSTATE.PRIV = 0) attempt to update PCR.PRIV (write a value of 1) via a WRPCR instruction causes a *privileged_action* exception (impl. dep. #250).

Performance Instrumentation Counter (PIC) Register (ASR 17)

The PIC registers conform to JPS1 **Commonality**.

SPARC64 VIIIfx implements 4 PIC registers. Each is accessed by way of ASR 17, using PCR.SC as the PIC pair selection field. Read/write access to the PIC will access the PICU/PICL counter pair selected by PCR. See Appendix Q for PICU/PICL encodings of specific event counters.

On overflow, the counter wraps to 0, SOFTINT register bit 15 is set to 1, and an interrupt level-15 exception is generated. The counter overflow trap is triggered on the transition from value FFFF $FFFF_{16}$ to value 0. If multiple overflows occur simultaneously, multiple overflow status bits will be set. An overflow status bit that is already set to 1 remains unchanged.

Software clears the overflow status bits by writing zeroes to the PCR.OVF field. Software may also write ones to the overflow status bits; however, this does not cause an overflow trap.

Dispatch Control Register (DCR) (ASR 18)

SPARC64 VIIIfx does not implement the DCR register. Reads return 0, and writes are ignored. The DCR is a privileged register; an attempted access by nonprivileged (user) code generates a *privileged_opcode* exception.

Extended Arithmetic Register (XAR) (ASR 29)

The XAR is a new, non-privileged register that extends the instruction fields. It holds the upper 3 bits of an instruction's register number fields (rs1, rs2, rs3, rd) and indicates whether or not the instruction is a SIMD instruction.

The register contains fields for 2 separate instructions. There are V (valid) bits for the first and second instructions; all other fields for the given instruction are valid only when v = 1. There is no distinction made between integer and floating-point registers. The XAR can be used with either type of register.

When a trap occurs, the contents of the XAR are saved to the TXAR[TL] and all fields in the XAR are set to 0. The saved value thus corresponds to the value of the XAR just before the instruction that caused the trap was executed.

Note – If a Tcc instruction initiates a trap, the contents of the XAR just before the Tcc instruction was executed are saved.

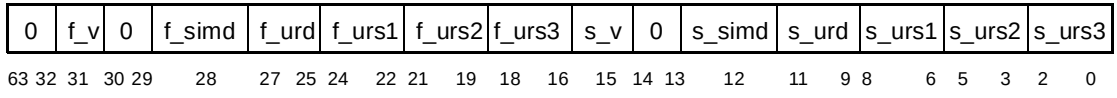


TABLE 5-3 XAR Fields

Bits	Field	Description
63:32	—	Reserved. An attempt to write a nonzero value to this field will cause an *illegal_instruction* exception.
31	f_v	This fields indicates whether the contents of fields beginning with `f_` are valid. If `f_v` = 1, the contents of the `f_` fields are applied to the instruction that executes first. After the 1st instruction completes, all `f_` fields are cleared.
30:29	—	Reserved. An attempt to write a nonzero value to this field will cause an *illegal_instruction* exception.
28	f_simd	If `f_simd` = 1, the 1st instruction is executed as a SIMD instruction. If `f_simd` = 0, execution is non-SIMD.
27:25	f_urd	Extends the `rd` field of the 1st instruction.
24:22	f_urs1	Extends the `rs1` field of the 1st instruction.
21:19	f_urs2	Extends the `rs2` field of the 1st instruction.
18:16	f_urs3	Extends the `rs3` field of the 1st instruction.
15	s_v	This fields indicates whether the contents of fields beginning with `s_` are valid. If `s_v` = 1, the contents of the `s_` fields are applied to the instruction that executes second. After the 2nd instruction completes, all `s_` fields are cleared.
14:13	—	Reserved. An attempt to write a nonzero value to this field will cause an *illegal_instruction* exception.
12	s_simd	If `s_simd` = 1, the 2nd instruction is executed as a SIMD instruction. If `s_simd` = 0, execution is non-SIMD.
11:9	s_urd	Extends the `rd` field of the 2nd instruction.
8:6	s_urs1	Extends the `rs1` field of the 2nd instruction.
5:3	s_urs2	Extends the `rs2` field of the 2nd instruction.
2:0	s_urs3	Extends the `rs3` field of the 2nd instruction.

How XAR is referred to in this specification.

The fields described in Table 5-3 have the following aliases.

- For memory access:

Alias	Field
`XAR.f_dis_hw_pf`	`XAR.f_urs3<1>`
`XAR.s_dis_hw_pf`	`XAR.s_urs3<1>`
`XAR.f_sector`	`XAR.f_urs3<0>`
`XAR.s_sector`	`XAR.s_urs3<0>`

- For SIMD FMA:

Alias	Field
XAR.f_negate_mul	XAR.f_urd<2>
XAR.s_negate_mul	XAR.s_urd<2>
XAR.f_rs1_copy	XAR.f_urs3<2>
XAR.s_rs1_copy	XAR.s_urs3<2>

- Others
  If the notation does not distinguish between the f_ and s_ fields, the values of XAR.f_v and XAR.s_v determine which field is being referenced.

Field Notation	When XAR.f_v = 1	When XAR.f_v = 0 and XAR.s_v = 1
XAR.v	XAR.f_v	XAR.s_v
XAR.urd	XAR.f_urd	XAR.s_urd
XAR.urs1	XAR.f_urs1	XAR.s_urs1
XAR.urs2	XAR.f_urs2	XAR.s_urs2
XAR.urs3	XAR.f_urs3	XAR.s_urs3
XAR.dis_hw_pf	XAR.f_dis_hw_pf	XAR.s_dis_hw_pf
XAR.sector	XAR.f_sector	XAR.s_sector
XAR.negate_mul	XAR.f_negate_mul	XAR.s_negate_mul
XAR.rs1_copy	XAR.f_rs1_copy	XAR.s_rs1_copy

XAR operation

Some instructions can reference the XAR, and some cannot.

In this document, instructions that can reference XAR are called "XAR-eligible instructions". Refer to TABLE A-2, *"Instruction Set"* (page 61) for details on which instructions are XAR eligible.

- An attempt to execute an instruction that is not XAR-eligible while XAR.v = 1 causes an *illegal_action* exception.
- XAR-eligible instructions have the following behavior.
  - If XAR.v = 1, the XAR.urs1, XAR.urs2, XAR.urs3, and XAR.urd fields are concatenated with the instruction fields rs1, rs2, rs3, and rd respectively.

Integer registers are referenced by 8-bit register numbers; the XAR fields specify the upper 3 bits, and the instruction fields specify the lower 5 bits.

Floating-point registers are referenced by 9-bit register numbers; the XAR fields specify the upper 3 bits. The double-precision encoding of the 5-bit instruction fields is decoded to generate the lower 6 bits of the register number. See *"Floating-Point Register Number Encoding"* (page 20) for details.

- If XAR.f_v = 1, the XAR.f_urs1, XAR.f_urs2, XAR.f_urs3, and XAR.f_urd fields are used.
- If XAR.f_v = 0 and XAR.s_v = 1, the XAR.s_urs1, XAR.s_urs2, XAR.s_urs3, and XAR.s_urd fields are used.

- The value of the f_ or s_ fields are only valid once. After the instruction referencing the XAR completes, the referenced fields are set to 0.
- XAR-eligible instructions cause *illegal_action* exceptions in the following cases.
  - An integer register number greater than or equal to xg[32] is specified.
  - urs1 ≠ 0 is specified for an instruction that does not use rs1.
    There are similar cases for rs2, rs3, rd.
    Specifying urs2 ≠ 0 for an instruction whose rs2 field holds an immediate value (such as simm13 or fcn) also causes an *illegal_action* exception.
  - A register number greater than or equal to f[256] is specified for the rd field of an FDIV(S,D) or FSQRT(S,D) instruction.
  - XAR.simd = 1 for an instruction that does not support SIMD execution.
  - XAR.simd = 1, and a register number greater than or equal to f[256] is specified. rs1 and rs2 of an FMADD instruction are exceptions to this rule; register numbers greater than or equal to f[256] can be specified.
    For FMADD, the XAR.urs3<2> and XAR.urd<2> bits can have values of 1. This has a different effect than specifying register numbers greater than or equal to f[256]. See *"SIMD Execution of FMA Instructions"* (page 75) for details.
  - XAR.urs3<2> ≠ 0 for a ld/st/atomic instruction.

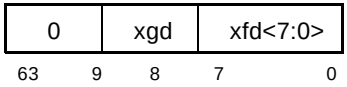
When the XAR specifies register numbers for only one instruction, either the f_ or s_ fields can be used.

Programming Note – If WRXAR is used, either XAR.f_v or XAR.s_v can be set to 1. The sxar1 instruction sets XAR.f_v to 1.

If XAR.f_v = 0, the f_simd, f_urs1, f_urs2, f_urs3, and f_urd fields are ignored even when the fields contain nonzero values. The value of each field after instruction execution is undefined. If XAR.s_v = 0, the s_simd, s_urs1, s_urs2, s_urs3, and s_urd fields are ignored even when the fields contain nonzero values. The value of each field after instruction execution is undefined.

Extended Arithmetic Register Status Register (XASR) (ASR 30)

The XASR is new, nonprivileged register.



Bits	Field	Access	Description
63:9	—	R	Reserved.
8	xgd	RW	Updating one of the xg[0] – xg[31] registers sets xgd = 1.
7:0	xfd<7:0>	RW	Updating a floating-point register sets the appropriate bit to 1.

This register is used to determine whether any of the registers added by HPC-ACE need to be saved during a context switch. Updating an HPC-ACE register sets the appropriate bit to 1.

- There is no flag indicating an update to a V9 integer register.
- Updating one of the xg[0] – xg[31] registers sets XASR.xgd = 1.
- Updating a floating-point register sets the appropriate XASR.xfd<i> = 1. The floating-point registers and corresponding xfd bits are shown below.

Floating-Point Registers	Corresponding XASR Bits
f[0] – f[62]	xfd<0>
f[64] – f[126]	xfd<1>
f[128] – f[190]	xfd<2>
f[192] – f[254]	xfd<3>
f[256] – f[318]	xfd<4>
f[320] – f[382]	xfd<5>
f[384] – f[446]	xfd<6>
f[448] – f[510]	xfd<7>

Programming Note – Updating a V9 floating-point register sets the xfd[0] bit of the XASR, and also updates the V9 FPRS. For example, updating f[15] sets both FPRS.dl = 1 and XASR.xfd<0> = 1.

Implementation Note – When MOVr, MOVcc, FMOVr, or FMOVcc is executed and a condition for moving data is not met, setting a bit to 1 in XASR is implementation dependent.

Trap XAR Registers (TXAR) (ASR 31)

The TXAR are new, privileged registers with the same fields as the XAR.

The TXAR are registers that store the value of the XAR when a trap occurs. The register field definitions are the same as for the XAR. Registers TXAR[1] – TXAR[MAXTL] are defined. When TL > 0, TXAR[TL] is visible. If TL is changed, the TXAR[TL] corresponding to the new TL can be read/written on the following instruction.

An attempt to read/write the TXAR while TL = 0 causes an *illegal_instruction* exception. Writing a nonzero value to a reserved field also causes an *illegal_instruction* exception.

5.2.12 Registers Referenced Through ASIs

This section only describes ASI registers defined in 5.2.12 of JPS1 **Commonality**. Refer to Appendix L for information on additional ASI registers.

Data Cache Unit Control Register (DCUCR)

ASI 45_{16} (ASI_DCU_CONTROL_REGISTER), VA = 00_{16}.

The DCUCR contains fields that control several memory-related hardware functions. The functions include instruction, prefetch, write and data caches, MMUs, and watchpoint setting. The SPARC64 VIIIfx implements most of the DCUCR functions described in JPS1 **Commonality**.

The DCUCR is illustrated in FIGURE 5-3 and described in TABLE 5-4.

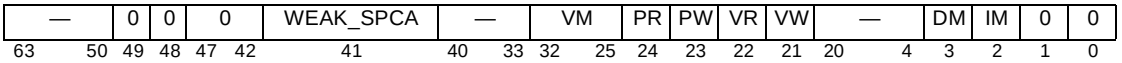


FIGURE 5-3 DCUCR (ASI 45_{16})

TABLE 5-4 DCUCR Fields

Bits	Field	Access	Description
63:50	—		Reserved
49:48	CP, CV	R	Not implemented in SPARC64 VIIIfx (impl. dep. #232). These bits read as 0, and writes to them are ignored.
47:42	impl. dep.	R	These bits read as 0, and writes to them are ignored.
41	WEAK_SPCA	RW	Disable Speculative Memory Access (impl. dep. #240). When WEAK_SPCA = 1, branch prediction is disabled; that is, the processor prefetches instructions as if branches are always predicted not taken. Loads and stores downstream of a branch are not executed until the branch direction is known. The hardware prefetch mechanism is turned off, and all prefetch instructions including strong prefetches are invalidated. Because the maximum number of bytes that can be prefetched is determined by internal CPU resources, the address to be accessed can be determined by setting weak_spca = 1.
40:33	PM<7:0>		Reserved.
32:25	VM<7:0>	RW	This field specifies the Data Watchpoint Register Mask. In SPARC64 VIIIfx, the Data Watchpoint Register is shared by the physical and virtual addresses.
24, 23	PR, PW	RW	When the value of the Data Watchpoint Register is interpreted as a physical address, a read or write access to the range of addresses specified by the VM field causes a *PA_watchpoint* exception.
22, 21	VR, VW	RW	When the value of the Data Watchpoint Register is interpreted as a virtual address, a read or write access to the range of addresses specified by the VM field causes a *VA_watchpoint* exception.
20:4	—		Reserved.
3	DM	RW	Data MMU Enable. If DM = 0, address translation for data accesses is disabled, and the virtual address is used directly as a physical address.
2	IM	RW	Instruction MMU Enable. If IM = 0, address translation for data accesses is disabled, and the virtual address is used directly as a physical address.
1	DC	R	Not implemented in SPARC64 VIIIfx (impl. dep. #253). This bit reads as 0, and writes to it are ignored.
0	IC	R	Not implemented in SPARC64 VIIIfx (impl. dep. #253). This bit reads as 0, and writes to it are ignored.

Implementation Note – When DCUCR.WEAK_SPCA = 1 and instructions downstream of a CTI instruction are prefetched, the maximum number of bytes that can be prefetched is 1KB.

Programming Note – To ensure that all speculative memory accesses are inhibited, system software should issue a membar #Sync immediately after setting DCUCR.WEAK_SPCA = 1.

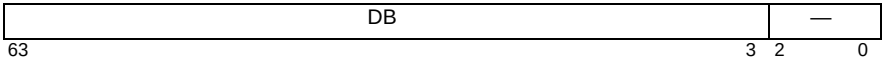
Programming Note – When the IM (IMMU enable) and DM (DMMU Enable) bits are modified in SPARC64 VIIIfx, the following instruction sequences must be executed.

```
# DCUCR.IM update
stxa DCUCR
flush

# DCUDR.DM update
stxa DCUCR
membar #sync
```

Data Watchpoint Registers

Register Name	ASI_WATCHPOINT
ASI	58_{16}
VA	38_{16}
Access Type	Supervisor Read/Write



Bits	Field	Access	Description
63:3	DB	RW	Watchpoint Address (VA or PA)

TABLE 5-18 in JPS1 **Commonality** defines the ASIs affected by watchpoint traps; these are classifed as either translating or bypass ASIs. As defined, some implementation-dependent or undefined ASIs are affected by watchpoint traps. SPARC64 VIIIfx fixes this by redefining the translating, bypass, and nontranslating ASIs. See TABLE L-1 (page 214). The ASIs affected by watchpoint traps are the translating and bypass ASIs listed in this table.

In JPS1 **Commonality**, separate virtual and physical addresses can be set for watchpoints. In SPARC64 VIIIfx, this specification is changed. Only one address is set, and matches are monitored depending on whether the address is interpreted as a virtual or physical address. `ASI_VA_WATCHPOINT` (ASI = 58_{16}, VA = 38_{16}) in JPS1 **Commonality** is renamed to `ASI_WATCHPOINT`, and `ASI_PA_WATCHPOINT` (ASI = 58_{16}, VA = 40_{16}) is deleted.

Compatibility Note – This change is not compatible with SPARC JPS1.

The method of enabling and disabling watchpoints by setting `DCUCR.VR`, `DCUCR.VW`, `DCUCR.PR`, and `DCUCR.PW` conforms to SPARC JPS1. If either `DCUCR.VR` or `DCUCR.VW` is 1, the virtual addresses of all data references are compared against the `DB` field, and a match causes a *VA_watchpoint* exception. If either `DCUCR.PR` or `DCUCR.PW` is 1, the physical addresses of all data references are compared against the `DB` field, and a match causes a *PA_watchpoint* exception. If a match occurs for both virtual and physical addresses, a *VA_watchpoint* exception is signalled.

Unimplemented ASIs defined as bypass or translating in TABLE 5-18 of JPS1 **Commonality** are not bypass or translating ASIs in SPARC64 VIIIfx and are not affected by watchpoint traps. That is, attempts to access these ASIs cause *data_access_exception* exceptions; the addresses are not compared against the contents of the watchpoint register.

When comparing the `DB` field and a physical address, bits `DB<63:41>` are ignored.

For SIMD load and SIMD store instructions, the address of both basic and extended operations are compared against the contents of the watchpoint register. If the watchpoint address and mask match the address and access length of the basic operation, the basic operation signals a *VA_watchpoint* or *PA_watchpoint* exception. If the watchpoint address and mask match the address and access length of the extended operation, the extended operation signals a *VA_watchpoint* or *PA_watchpoint* exception.

No implementation-dependent feature of SPARC64 VIIIfx reduces the reliability of data watchpoints (impl. dep. #244).

The following instructions are special cases. Refer to each instruction for details on setting watchpoints and comparing the access address against the contents of the watchpoint register.

- Appendix A.4, *"Block Load and Store Instructions (VIS I)"*
- Appendix A.30, *"Load Quadword, Atomic [Physical]"*
- Appendix A.42, *"Partial Store (VIS I)"*
- Appendix A.77, *"Store Floating-Point Register on Register Condition"*
- Appendix A.79, *"Cache Line Fill with Undetermined Values"*
- Appendix F.5.1, *"Trap Conditions for SIMD Load/Store"*

Instruction Trap Register

SPARC64 VIIIfx implements the Instruction Trap Register (impl. dep. #205).

In SPARC64 VIIIfx, the encoding of the least significant 11 bits of the displacement field of `CALL` and branch (`BPcc`, `FBPfcc`, `Bicc`, `BPr`) instructions in the instruction cache are the same as their architectural encoding (which appears in main memory) (impl. dep. #245).

5.2.13 Floating-Point Deferred-Trap Queue (FQ)

SPARC64 VIIIfx does not implement a Floating-Point Deferred-trap Queue (impl. dep. #24). An attempt to read `FQ` with an `RDPR` instruction will cause an *illegal_instruction* exception (impl. dep. #25).

5.2.14 IU Deferred-Trap Queue

SPARC64 VIIIfx does not implement an IU deferred-trap queue (impl. dep. #16)

F.CHAPTER **6**

Instructions

This chapter describes instructions specific to SPARC64 VIIIfx:

For convenience, we follow the organization of Chapter 6 in JPS1 **Commonality**. Please refer to JPS1 **Commonality** as necessary.

6.1 Instruction Execution

SPARC64 VIIIfx is an advanced, superscalar implementation of a SPARC V9 processor. Multiple instructions can be issued and executed in a single cycle. Because SPARC64 VIIIfx provides serial execution semantics, the topics described in this section are not visible to software; however, these topics are important for writing correct and efficient software.

6.1.1 Data Prefetch

The out-of-order SPARC64 VIIIfx processor speculatively executes instructions. When speculation is incorrect, the results of speculative instruction execution can be invalidated, but speculative memory accesses cannot be invalidated. Therefore, SPARC64 VIIIfx implements the following policy for speculative memory accesses.

1. When a memory operation *x* resolves to a volatile memory address (*location[x]*), SPARC64 VIIIfx does not prefetch *location[x]*. The memory address is fetched once it is certain that *x* will be executed, i.e. once *x* is *committable.*

2. When a memory operation *x* resolves to a nonvolatile memory address (*location[x]),* SPARC64 VIIIfx may prefetch *location[x],* subject to the following rules:

a. When operation *x* has store semantics and accesses a cacheable location, exclusive ownership of *location[x]* is obtained. Operations without store semantics are prefetched even if they are noncacheable.

b. Atomic operations (`CAS(X)A`, `LDSTUB`, `SWAP`) are never prefetched.

SPARC64 VIIIfx provides two mechanisms for preventing execution of speculative loads:

1. Speculative accesses to a memory page or I/O location can be disabled by setting the `E` (side-effect) bit in the corresponding `PTE`. Accesses to pages that have the `E` bit set are forced to wait until they are no longer speculative**.** See Appendix F for details.

2. Loads with `ASI_PHYS_BYPASS_WITH_EBIT[_L]` (ASI = 15_{16}, $1D_{16}$) are forced to execute in program order. These loads are not speculatively executed.

6.1.2 Instruction Prefetch

SPARC64 VIIIfx prefetches instructions to minimize the number of instances where instruction execution is stalled waiting for instructions to be delivered. Depending on the results of branch prediction, some prefetched instructions are not actually executed. In other cases, speculatively-executed instructions may access memory. Exceptions caused by instruction prefetch or speculative memory accesses are not signalled until all prior instructions have committed.[1]

6.1.3 Syncing Instructions

Executing a *syncing instruction* stalls the pipeline for a certain number of cycles. There are two types of *syncing instructions*: *pre-sync* and *post-sync*. A pre-sync instruction commits by itself after all prior instructions have committed; subsequent instructions are not executed until after the pre-sync instruction commits. A post-sync instruction prevents subsequent instructions from issuing until the post-sync instruction has committed. Some instructions have both pre-sync and post-sync effects.

In SPARC64 VIIIfx, all instructions except for stores commit in program order. Store instructions commit before their results become globally visible; that is, stores commit once the store result is written to the write-back buffer.

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

6.2 Instruction Formats and Fields

SPARC64 VIIIfx instructions are encoded in five major 32-bit formats and several minor formats. Please refer to Section 6.2 of JPS1 **Commonality** for descriptions of four of the five major formats. FIGURE 6-1 shows Format 5, which is specific to SPARC64 VIIIfx.

Format 5 (op = 2, op3 = 37_{16}): `FMADD`, `FPMADDX`, `FSELMOV`, and `FTRIMADD` (in place of `IMPDEP2A` and `IMPDEP2B`)

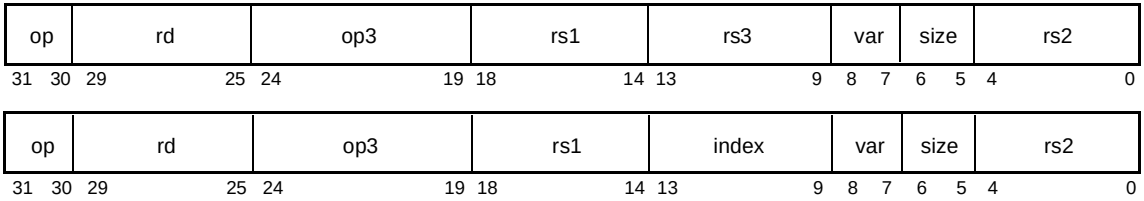


FIGURE 6-1 Summary of Instruction Formats: Format 5

Pleaser refer to Section 6.2 of JPS1 **Commonality** for a description of the instruction fields. Format 5 includes 4 additional fields, which are described in TABLE 6-1.

TABLE 6-1 Instruction Field Interpretation for Format 5

Field	Description
`rs3`	This 5-bit field specifies a floating-point register for the third source operand of a 3-operand floating-point instruction.
`var`	This 2-bit field is used to indicate the type of floating-point multiply-add/subtract instructions and to select other instructions implemented in the Impdep2 opcode space.
`size`	This 2-bit field is used to indicate the size of the operands for floating-point multiply-add/subtract instructions and to select other instructions implemented in the Impdep2 opcode space.
`index`	This field is used to indicate an entry in the `FTRIMADDd` coefficient table.

6.3 Instruction Categories

6.3.3 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`)

The SPARC64™ VIIIfx Extensions describe the `CALL` and `JMPL` instructions. Refer to JPS1 **Commonality** for the descriptions of the other control-transfer instructions.

CALL and JMPL Instructions

When `PSTATE.AM` = 0, all 64 bits of the `PC` are written into the destination register. When `PSTATE.AM` = 1, the lower 32 bits of the `PC` are written into the lower 32 bits of the destination register. Zeroes are written to the upper 32 bits (impl. dep. #125).

6.3.7 Floating-Point Operate (FPop) Instructions

The precise conditions under which an FPop causes an *fp_exception_other* exception with `FSR.ftt` = *unfinished_FPop* are defined in Appendix B.6, *"Floating-Point Nonstandard Mode"*.

6.3.8 Implementation-Dependent Instructions

SPARC64 VIIIfx defines floating-point instructions in the `IMPDEP1` and `IMPDEP2` opcode spaces. Because JPS1 **Commonality** defines the term "FPop" to refer "to those instructions encoded by `FPop1` and `FPop2` opcodes", `IMPDEP` instructions are not FPops.

Of the floating-point multiply-add/subtract instructions defined in `IMPDEP2`, quad-precision versions are defined for `FMADD`, `FMSUB`, and `FNMSUB`. Because SPARC64 VIIIfx does not support quad-precision operations, attempts to execute these instructions cause *illegal_instruction* exceptions. Only `FNMADD` does not have a quad-precision version. Quad-precision multiply-add/subtract instructions are not required SPARC V9 instructions, and system sofware is not required to emulate these operations.

Of the instructions defined in IMPDEP1 and IMPDEP2 by SPARC64 VIIIfx, the following instructions use the floating-point registers and generate *fp_disabled* exceptions if executed when PSTATE.PEF = 0 or FPRS.FEF = 0.

FCMP(GT,LE,EQ,NE,GE,LE)E(s,d), FCMP(EQ,NE)(s,d), FMAX(s,d), FMIN(s,d), FRCPA(s,d), FRSQRTA(s,d), FTRISSELd, FTRISMULd, FTRIMADDd, FSELMOV(s,d), F{N}M(ADD,SUB)(s,d), FPMADDX{HI}, ST{D}FR

Because these instructions are not FPops, an attempt to execute a reserved opcode causes an *illegal_instruction* exception as defined in JPS1 **Commonality** 6.3.9. However, other than the FPMADDX{HI} and ST{D}FR instructions, these instructions have the same FSR update behavior as all FPop instructions, as defined in JPS1 **Commonality** 6.3.7. The FTRISSELd and FSELMOV(s,d) instructions cannot generate a *fp_exception_ieee_754* exception, so they clear FSR.cexc and leaved FSR.aexc unchanged when they complete.

F.CHAPTER **7**

Traps

7.1 Processor States, Normal and Special Traps

In JPS1 **Commonality**, this section defines the CPU states and the transitions between those states. The SPARC64™ VIIIfx Extensions define these in Appendix O.1, *"Reset Types"* (page 245).

7.1.1 RED_state

See Appendix O.2.1, *"RED_state"* (page 248).

RED_state Trap Table

The `RED_state` trap vector is located at an implementation-dependent address referred to as `RSTVaddr`. The value of `RSTVaddr` is a constant within each implementation. In SPARC64 VIIIfx, the virtual address is `FFFF FFFF F000 0000`$_{16}$, which translates to the physical address `0000 01FF F000 0000`$_{16}$ (impl. dep. #114).

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.

Note – The values are overridden, not set, allowing them to be switched atomically.

SPARC64 VIIIfx has the following implementation-dependent behavior in `RED_state` (impl. dep. #115):

- While in RED_state, all address translation functions that use the ITLB are disabled. Translations that use the DTLB are disabled on entry but can be re-enabled by software while in RED_state. The TLBs can be accessed via the ASI registers.
- While the TLB (MMU) is disabled, all memory accesses are treated as noncacheable, strongly-ordered accesses.
- XIR resets are not masked and can cause exceptions.

Note – When RED_state is entered because of component failures, the handler should attempt to recover from potentially catastrophic error conditions or to disable the failing components. When RED_state is entered after a reset, the software should create the environment necessary to restore the system to a running state.

7.1.2 error_state

The processor enters error_state when a trap occurs while the processor is already at its maximum supported trap level, that is, when TL = MAXTL (impl. dep. #39).

The CPU, upon entering error_state, automatically generates a *watchdog_reset* (WDR) to exit error_state; however, the OPSR register can be configured to suppress the WDR and allow the CPU to remain in error_state (impl. dep #40, #254).

7.2 Trap Categories

7.2.2 Deferred Traps

In SPARC64 VIIIfx, certain error conditions are signalled by a deferred trap (impl. dep. #32). Please refer to Appendix P.2.2, *"Summary of Behavior During Error Detection"*, as well as Appendix P.4.3, *"Instruction Execution when an ADE Trap Occurs"*.

7.2.4 Reset Traps

When a SPARC64 VIIIfx core does not commit any instructions for a period of 6.7 seconds, a watchdog reset (WDR) occurs.

7.2.5 Uses of the Trap Categories

In SPARC64 VIIIfx, all exceptions that occur as the result of program execution are precise (impl. dep. #33).

An exception caused after the initial access of a multiple-access load or store instruction (`LDD(A)`, `STD(A)`, `LDSTUB`, `CASA`, `CASXA`, or `SWAP`) that causes a catastrophic error is precise.

7.3 Trap Control

7.3.1 PIL Control

When a SPARC64 VIIIfx core receives an interrupt request from the system, an *interrupt_vector_trap* (`TT` = 60_{16}) is generated. The trap handler reads the interrupt data and schedules SPARC V9 interrupts for processing. SPARC V9 interrupts are scheduled by writing the `SOFTINT` register. Please refer to Section 5.2.11 of JPS1 **Commonality** for details.

The `PIL` register is checked when SPARC V9 interrupts are received. If the interrupt request is not masked by the `PIL`, SPARC64 VIIIfx stops issuing new instructions, cancels all uncommitted instructions, and traps to privileged software. This action is not taken if there is a higher-priority trap that is being executed.

SPARC64 VIIIfx treats an interrupt request as a disrupting trap.

7.4 Trap-Table Entry Addresses

7.4.2 Trap Type (TT)

SPARC64 VIIIfx implements all mandatory SPARC V9 and SPARC JPS1 exceptions, as described in Chapter 7 of JPS1 **Commonality**, plus the following SPARC64 VIIIfx implementation-dependent exceptions (impl. dep. #35; impl. dep. #36).

- *async_data_error*
- *illegal_action*
- *SIMD_load_across_pages*

Traps defined in JPS1 **Commonality** are shown in TABLE 7-1 and TABLE 7-2. Shaded sections in TABLE 7-1 indicate traps that do not occur in SPARC64 VIIIfx.

TABLE 7-1 Exception and Interrupt Requests, by TT Value *(1 of 2)*

SPARC V9 M/O	JPS1 M/O	Exception or Interrupt Request	TT	Global Register Set	Priority
●	●	*Reserved*	000_{16}	*-NA-*	*-NA-*
●	●	*power_on_reset*	001_{16}	AG	0
❍	●	*watchdog_reset*	002_{16}	AG	1
❍	●	*externally_initiated_reset*	003_{16}	AG	1
●	●	*software_initiated_reset*	004_{16}	AG	1
●	●	*RED_state_exception*	005_{16}	AG	1
●	●	*Reserved*	006_{16}–007_{16}	*-NA-*	*-NA-*
●	●	*instruction_access_exception*	008_{16}	MG	5
❍	❍	*instruction_access_MMU_miss*	009_{16}	MG (*impl. dep.)*	2
❍	●	*instruction_access_error*	$00A_{16}$	AG	3
●	●	*Reserved*	$00B_{16}$–$00F_{16}$	*-NA-*	*-NA-*
●	●	*illegal_instruction*	010_{16}	AG	7
●	●	*privileged_opcode*	011_{16}	AG	6
❍	❍	*unimplemented_LDD*	012_{16}	AG	6
❍	❍	*unimplemented_STD*	013_{16}	AG	6
●	●	*Reserved*	014_{16}–$01F_{16}$	*-NA-*	*-NA-*
●	●	*fp_disabled*	020_{16}	AG	8
❍	●	*fp_exception_ieee_754*	021_{16}	AG	11
❍	●	*fp_exception_other*	022_{16}	AG	11
		(when ftt = *unimplemented_FPop*)	022_{16}	AG	8.2
●	●	*tag_overflow*	023_{16}	AG	14
❍	●	*clean_window*	024_{16}–027_{16}	AG	10
●	●	*division_by_zero*	028_{16}	AG	15
❍	❍	*internal_processor_error*	029_{16}	*impl. dep.*	*impl. dep*
●	●	*Reserved*	$02A_{16}$–$02F_{16}$	*-NA-*	*-NA-*
●	●	*data_access_exception*	030_{16}	MG	12
❍	❍	*data_access_MMU_miss*	031_{16}	MG (*impl. dep.)*	12
❍	●	*data_access_error*	032_{16}	AG	12
❍	❍	*data_access_protection*	033_{16}	MG (*impl. dep.)*	12
●	●	*mem_address_not_aligned*	034_{16}	AG	10

TABLE 7-1 Exception and Interrupt Requests, by TT Value *(2 of 2)*

SPARC V9 M/O	JPS1 M/O	Exception or Interrupt Request	TT	Global Register Set	Priority
❍	●	*LDDF_mem_address_not_aligned* (impl. dep. #109)	035_{16}	AG	10
❍	●	*STDF_mem_address_not_aligned* (impl. dep. #110)	036_{16}	AG	10
●	●	*privileged_action*	037_{16}	AG	11
❍	❍	*LDQF_mem_address_not_aligned* (impl. dep. #111)	038_{16}	AG	10
❍	❍	*STQF_mem_address_not_aligned* (impl. dep. #112)	039_{16}	AG	10
●	●	*Reserved*	$03A_{16}$–$03F_{16}$	*-NA-*	*-NA-*
❍	❍	*async_data_error*	040_{16}	AG	2
●	●	*interrupt_level_n* (*n* = 1–15)	041_{16}–$04F_{16}$	AG	32-*n*
●	●	*Reserved*	050_{16}–$05F_{16}$	*-NA-*	*-NA-*
❍	●	*interrupt_vector*	060_{16}	IG	16
❍	●	*PA_watchpoint*	061_{16}	AG	12
❍	●	*VA_watchpoint*	062_{16}	AG	11
❍	●	*ECC_error*	063_{16}	AG	33
❍	●	*fast_instruction_access_MMU_miss*	064_{16}–067_{16}	MG	2
❍	●	*fast_data_access_MMU_miss*	068_{16}–$06B_{16}$	MG	12
❍	●	*fast_data_access_protection*	$06C_{16}$–$06F_{16}$	MG	12
❍	❍	*implementation_dependent_exception_n* (impl. dep. #35)	070_{16}–072	*impl. dep.*	*impl. dep.*
❍	❍	*illegal_action*	073_{16}	AG	8.5
❍	❍	*implementation_dependent_exception_n* (impl. dep. #35)	074_{16}–076	*impl. dep.*	*impl. dep.*
❍	❍	*SIMD_load_across_pages*	077_{16}	AG	12
❍	❍	*implementation_dependent_exception_n* (impl. dep. #35)	078_{16}–07F	*impl. dep.*	*impl. dep.*
●	●	*spill_n_normal* (*n* = 0–7)	080_{16}–$09F_{16}$	AG	9
●	●	*spill_n_other* (*n* = 0–7)	$0A0_{16}$–$0BF_{16}$	AG	9
●	●	*fill_n_normal* (*n* = 0–7)	$0C0_{16}$–$0DF_{16}$	AG	9
●	●	*fill_n_other* (*n* = 0–7)	$0E0_{16}$–$0FF_{16}$	AG	9
●	●	*trap_instruction*	100_{16}–$17F_{16}$	AG	16
●	●	*Reserved*	180_{16}–$1FF_{16}$	*-NA-*	*-NA-*

TABLE 7-2 Exception and Interrupt Requests, by Priority (0 = Highest; larger number = lower priority) *(1 of 2)*

SPARC V9 M/O	JPS1 M/O	Exception or Interrupt Request	TT	Global Register Set	Priority
●	●	*power_on_reset* (POR)	001_{16}	AG	0
❍	●	*externally_initiated_reset* (XIR)	003_{16}	AG	1
❍	●	*watchdog_reset* (WDR)	002_{16}	AG	1
●	●	*software_initiated_reset* (SIR)	004_{16}	AG	1
●	●	*RED_state_exception*	005_{16}	AG	1
❍	❍	*async_data_error*	040_{16}	AG.	2
❍	●	*fast_instruction_access_MMU_miss*	064_{16}–067_{16}	MG	2
❍	●	*instruction_access_error*	$00A_{16}$	AG	3
●	●	*instruction_access_exception*	008_{16}	MG	5
●	●	*privileged_opcode*	011_{16}	AG	6
●	●	*illegal_instruction*	010_{16}	AG	7
●	●	*fp_disabled*	020_{16}	AG	8
❍	●	*fp_exception_other* (when ftt = *unimplemented_FPop*)	022_{16}	AG	8.2
❍	❍	*illegal_action*	073_{16}	AG	8.5
●	●	*spill_n_normal* (*n* = 0–7)	080_{16}–$09F_{16}$	AG	9
●	●	*spill_n_other* (*n* = 0–7)	$0A0_{16}$–$0BF_{16}$	AG	9
●	●	*fill_n_normal* (*n* = 0–7)	$0C0_{16}$–$0DF_{16}$	AG	9
●	●	*fill_n_other* (*n* = 0–7)	$0E0_{16}$–$0FF_{16}$	AG	9
❍	●	*clean_window*	024_{16}–027_{16}	AG	10
❍	●	*LDDF_mem_address_not_aligned* (impl. dep. #109)	035_{16}	AG	10
❍	●	*STDF_mem_address_not_aligned* (impl. dep. #110)	036_{16}	AG	10
●	●	*mem_address_not_aligned*	034_{16}	AG	10
❍	●	*fp_exception_ieee_754*	021_{16}	AG	11
❍	●	*fp_exception_other* (not ftt = *unimplemented_FPop*)	022_{16}	AG	11
●	●	*privileged_action*	037_{16}	AG	11
❍	●	*VA_watchpoint*	062_{16}	AG	11
●	●	*data_access_exception*	030_{16}	MG	12
❍	●	*fast_data_access_MMU_miss*	068_{16}–$06B_{16}$	MG	12
❍	●	*data_access_error*	032_{16}	AG	12
❍	●	*PA_watchpoint*	061_{16}	AG	12
❍	●	*fast_data_access_protection*	$06C_{16}$–$06F_{16}$	MG	12

TABLE 7-2 Exception and Interrupt Requests, by Priority (0 = Highest; larger number = lower priority) *(2 of 2)*

SPARC V9 M/O	JPS1 M/O	Exception or Interrupt Request	TT	Global Register Set	Priority
❍	❍	*SIMD_load_across_pages*	077_{16}	AG	12
●	●	*tag_overflow*	023_{16}	AG	14
●	●	*division_by_zero*	028_{16}	AG	15
●	●	*trap_instruction*	100_{16}–$17F_{16}$	AG	16
❍	●	*interrupt_vector*	060_{16}	IG	16
●	●	*interrupt_level_n* (n = 1–15)	041_{16}–$04F_{16}$	AG	32-*n*
❍	●	*ECC_error*	063_{16}	AG	33

7.4.3 Trap Priorities

In SPARC64 VIIIfx, the priority level of some traps have been changed from those defined in JPS1 **Commonality**.

- *fp_exception_other* has a priority of 11 as in JPS1 **Commonality**, but when `FSR.ftt` = 3 (*unimplemented_FPop*) the priority is 8.2 in SPARC64 VIIIfx.
- *VA_watchpoint* has a priority of 11, but a level-12 trap for a SIMD load or store instruction may take precedence depending on the situation. See Appendix F.5.1 for details.
- *illegal_action* is a SPARC64 VIIIfx-defined trap with a priority of 8.5. There are cases where it take precedence over a level-7 *illegal_instruction* trap. See Chapter 7.6.1 for details.
- Detecting a multiple hit in the TLB does not cause a TTE-dependent exception. See Appendix F.5.2, *"Behavior on TLB Error"* (page 182) for details
- *data_access_error* caused by a bus errror or timeout has the lowest priority among level-12 traps. See Appendix F.5 for details.

7.5 Trap Processing

In JPS1 **Commonality**, state changes during trap processing are described for various cases. Newly-added registers in SPARC64 VIIIfx always have the same behavior during trap processing; this behavior is explained below.

During trap processing, the values of the following registers are changed:

- The HPC-ACE state is preserved, and the trap handler begins executing from the first instruction that does not use any of the features added by HPC-ACE.

```
TXAR[TL]    ← XAR
XAR         ← 0
```

When an XAR-eligible instruction signals an exception, the value of XAR is saved to TXAR[TL] and XAR is set to 0 . In the case of a taken Tcc instruction, the value of XAR before the execution of Tcc is saved to TXAR[TL] .

Register changes for DONE, RETRY are described below.

```
XAR         ← TXAR[TL]
TXAR[TL]    not updated
```

Programming Note – When an emulation routine emulates an HPC-ACE instruction, TXAR[TL] should be cleared before executing a DONE instruction. This emulates the single-use behavior of the XAR.

7.6 Exception and Interrupt Descriptions

7.6.1 Traps Defined by SPARC V9 As Mandatory

- *illegal_instruction* [tt = 010_{16}] (Precise) — Takes priority over an *illegal_action* exception, but there are cases where a WRXAR, WRTXAR, or WRPR %pstate causes an *illegal_action* exception. See the instruction definitions for details.

7.6.2 SPARC V9 Optional Traps That Are Mandatory in SPARC JPS1

- *fp_exception_other* [tt = 022_{16}] (Precise) — In SPARC64 VIIIfx, has a priority level of 8.5 when an attempt to execute an unimplemented FPop causes an exception (FSR.ftt = 3, *unimplemented_FPop*).

7.6.4 SPARC V9 Implementation-Dependent, Optional Traps That Are Mandatory in SPARC JPS1

SPARC64 VIIIfx implements all six traps that are implementation dependent in SPARC V9 but mandatory in JPS1 (impl. dep. #35).

7.6.5 SPARC JPS1 Implementation-Dependent Traps

SPARC64 VIIIfx implements the following traps that are implementation dependent (impl. dep. #35).

- *async_data_error* [`tt` = 040_{16}] (Preemptive or disrupting) (impl. dep. #218) — SPARC64 VIIIfx implements the *async_data_error* exception for signalling an urgent error. Refer to Appendix P.4, *"Urgent Error"*, for details.
- *illegal_action* [`tt` = 073_{16}] (Precise) — Generated when executing an instruction that is not `XAR`-eligible while `XAR.v` = 1, or when executing an `XAR`-eligible instruction while `XAR` is set incorrectly. If `XAR` is set by `SXAR`, the exception occurs when the following instruction is executed. A `WRXAR`, `WRTXAR`, or `WRPR %pstate` generates an *illegal_action* exception instead of the higher-priority *illegal_instruction* exception. Refer to the instruction definitions for details.
- *SIMD_load_across_pages* [`tt` = 077_{16}] (Precise) — Generated when a SIMD load accesses multiple pages and the extended operation misses in the TLB.
  When hardware generates this exception and system software emulates the SIMD load, the basic and extended loads should be processed separately.

Note – If *SIMD_load_across_pages* updates the TLB, an infinite loop may occur if the basic and extended translations are alternately evicted from the TLB.

F.CHAPTER **8**

Memory Models

The SPARC V9 architecture is a *model* that specifies the behavior observable by software on SPARC V9 systems. Therefore, access to memory can be implemented in any manner, as long as the behavior observed by software conforms to that of the models described in Chapter 8 of JPS1 **Commonality** and defined in Appendix D, "*Formal Specification of the Memory Models*", also in JPS1 **Commonality**.

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. SPARC64 VIIIfx has the same specified behavior under all memory models**.**

8.1 Overview

Note – In the following section, the “hardware memory model” is distinguished from the “SPARC V9 memory model”. The SPARC V9 memory model is the memory model selected by `PSTATE.MM`.

SPARC64 VIIIfx only implements one hardware memory model, which supports all three SPARC V9 memory models (impl. dep. #113):

- **Total Store Order** — All loads are ordered with respect to earlier loads, and all stores are ordered with respect to earlier loads and stores. This behavior supports the TSO, PSO, and RMO memory models defined in SPARC V9. When `PSTATE.MM` selects PSO or RMO, SPARC64 VIIIfx uses this memory model. Since programs written for PSO or RMO will always work in Total Store Order, this behavior is safe but does not take advantage of the reduced restrictions in PSO or RMO.

8.4 SPARC V9 Memory Model

8.4.5 Mode Control

SPARC64 VIIIfx operates under TSO for all `PSTATE.MM` settings. Setting `PSTATE.MM` to 11_2 also selects TSO (impl. dep. #119). However, the encoding 11_2 may be assigned to a different memory model in future versions of SPARC64 VIIIfx and should not be used.

8.4.7 Synchronizing Instruction and Data Memory

SPARC64 VIIIfx guarantees data coherency between all caches in a core. Writes to the data cache invalidate any corresponding data in the instruction cache. If there is updated data in the data cache, reads of the instruction cache by the instruction fetch mechanism return the updated data.

This behavior does not mean that `FLUSH` instructions are never needed in SPARC64 VIIIfx. `FLUSH` instructions are needed if coherency between cache data and data in the pipeline is required.

SPARC64 VIIIfx does not support coherency between multiple processors, and the latency of a multiprocessor `FLUSH` instruction is undefined. The latency of a `FLUSH` instruction between on-chip cores depends on the CPU state; the minimum latency is 30 cycles (impl. dep. #122).

F.APPENDIX **A**

Instruction Definitions

This appendix describes SPARC64 VIIIfx implementation-dependent instructions, as well as instructions specific to SPARC64 VIIIfx. Instructions that conform to JPS1 **Commonality** are not described in this appendix; please refer to JPS1 **Commonality**. The section numbers in this appendix match those in JPS1 **Commonality**.

Instructions specific to SPARC64 VIIIfx are described in Section A.24 and Section A.72. All other sections describe instructions specified in JPS1 **Commonality**.

Definitions of implementation-dependent instructions contain only the required information. Definitions of SPARC64 VIIIfx-specific instructions contain the following information:

1. A table of the opcodes defined in the subsection. This contains information on the values of the field(s) that is unique to that instruction(s) and whether the instruction(s) can be used with certain HPC-ACE features.
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 processor and shall be 0 in any instance of the instruction. If a conforming SPARC V9 implementation encounters nonzero values in these fields, its behavior is undefined. See Section 1.2 for the behavior of *reserved* fields in SPARC64 VIIIfx.
3. A list of the suggested assembly language syntax; the syntax notation is described in Appendix G.
4. A description of the features, restrictions, and exception-causing conditions.
5. A list of exceptions that can occur as a consequence of attempting to execute the instruction(s). The following cases are not included in these lists:

    a. Exceptions due to an *instruction_access_error, instruction_access_exception, fast_instruction_access_MMU_miss*, *async_data_error, ECC_error*, and interrupts are not listed because they can occur on any instruction.

    a. An instruction that is not implemented in hardware generates an *illegal_instruction* exception (a floating-point instruction generates an *fp_exception_other* exception with `ftt` = *unimplemented_FPop*).

a. An instruction specified by `IIU_INST_TRAP` (ASI = 60_{16}, VA = 0) causes an *illegal_instruction* exception.

When specifying conditions that cause *illegal_action* exceptions, the notation for `XAR` fields does not distinguish between the `f_` and `s_` fields.

The following exceptions do not occur in SPARC64 VIIIfx:

- *instruction_access_MMU_miss*
- *data_access_MMU_miss*
- *data_access_protection*
- *unimplemented_LDD*
- *unimplemented_STD*
- *LDQF_mem_address_not_aligned*
- *STQF_mem_address_not_aligned*
- *internal_processor_error*
- *fp_exception_other* (`ftt` = *invalid_fp_register*)

This appendix does not contain any timing information (in either cycles or clock time).

TABLE A-2 summarizes all SPARC JPS1 instructions and SPARC64 VIIIfx-specific instructions. Within TABLE A-2 and in Appendix E, certain opcodes are marked with mnemonic superscripts. The superscripts and their meanings are defined in TABLE A-1. .

TABLE A-1 Opcode Superscripts

Superscript	Meaning
D	Deprecated instruction
P	Privileged opcode
P_{ASI}	Privileged action if bit 7 of the referenced ASI is 0
P_{ASR}	Privileged opcode if the referenced ASR register is privileged
P_{NPT}	Privileged action if `PSTATE.PRIV` = 0 and (S)`TICK.NPT` = 1
P_{PIC}	Privileged action if `PCR.PRIV` = 1
P_{PCR}	Privileged access to `PCR.PRIV` = 1

In TABLE A-2 and in the opcode tables of instruction definitions, the HPC-ACE columns indicate whether an instruction can be used with the indicated HPC-ACE feature.

- **Inst.** Instructions specific to SPARC64 VIIIfx (not defined in JPS1 **Commonality**).
- **Regs.** `XAR`-eligible instruction. The instruction can specify the HPC-ACE floating-point and integer registers; furthermore, a memory access instruction can specify the cache sector.
  For instructions with a ☆ in this column, `rd` must specify a basic floating-point register.

- **SIMD** Instruction can be specified as SIMD instructions.
  The quad-precision version of instructions with a † in this column cannot be specified as a SIMD instruction.

Instructions without a ✓ in any of these three columns is not XAR-eligible. Please refer to *"XAR operation"* (page 31) for more details on instructions that are not XAR-eligible.

TABLE A-2 Instruction Set *(1 of 7)*

		HPC-ACE Ext.			
Operation	**Name**	**Inst.**	**Regs.**	**SIMD**	**Page**
ADD (ADDcc)	Add (and modify condition codes)		✓		—
ADDC (ADDCcc)	Add with carry (and modify condition codes)		✓		—
ALIGNADDRESS{_LITTLE}	Calculate address for misaligned data				—
AND (ANDcc)	And (and modify condition codes)		✓		—
ANDN (ANDNcc)	And not (and modify condition codes)		✓		—
ARRAY(8,16,32)	3-D array addressing instructions				—
BPcc	Branch on integer condition codes with prediction				—
Bicc^{D}	Branch on integer condition codes				—
BMASK	Set the GSR.MASK field				—
BPr	Branch on contents of integer register with prediction				—
BSHUFFLE	Permute bytes as specified by GSR.MASK				—
CALL	Call and link				70
CASA$^{P_{ASI}}$	Compare and swap word in alternate space		✓		—
CASXA$^{P_{ASI}}$	Compare and swap doubleword in alternate space		✓		—
DONE^{P}	Return from trap				—
EDGE(8,16,32){L}	Edge handling instructions				—
FABS(s,d,q)	Floating-point absolute value		✓	†	—
FADD(s,d,q)	Floating-point add		✓	†	—
FALIGNDATA	Perform data alignment for misaligned data				—
FAND{S}	Logical AND operation		✓	✓	—
FANDNOT(1,2){S}	Logical AND operation with one inverted source		✓	✓	—
FBfcc^{D}	Branch on floating-point condition codes				—
FBPfcc	Branch on floating-point condition codes with prediction				—
FCMP(s,d,q)	Floating-point compare		✓		—
FCMPE(s,d,q)	Floating-point compare (exception if unordered)		✓		—
FCMP(GT,LE,NE,EQ)(16,32)	Pixel compare operations				—
FCMP(EQ,NE)(s,d)	Floating-point conditional compare to register	✓	✓	✓	116
FCMP(GT,LT,EQ,NE,GE,LE)E(s,d)	Floating-point conditional compare (exception if unordered)	✓	✓	✓	116
FDIV(s,d,q)	Floating-point divide		☆		—

TABLE A-2 Instruction Set *(2 of 7)*

Operation	Name	HPC-ACE Ext. Inst.	Regs.	SIMD	Page
FdMULq	Floating-point multiply double to quad		✓		—
FEXPAND	Pixel expansion				—
FiTO(s,d,q)	Convert integer to floating-point		✓	✟	—
FLUSH	Flush instruction memory		✓		—
FLUSHW	Flush register windows				—
FMADD(s,d)	Foating-point Multiply-and-Add	✓	✓	✓	72
FMAX(s,d)	Floating-point maximum	✓	✓	✓	118
FMIN(s,d)	Floating-point minimum	✓	✓	✓	118
FMSUB(s,d)	Foating-point Multiply-and-Subtract	✓	✓	✓	72
FMOV(s,d,q)	Floating-point move		✓	✟	—
FMOV(s,d,q)cc	Move floating-point register if condition is satisfied				—
FMOV(s,d,q)r	Move f-p reg. if integer reg. contents satisfy condition				—
FMUL(s,d,q)	Floating-point multiply		✓	✟	—
FMUL8x16	8x16 partitioned product				—
FMUL8x16(AU,AL)	8x16 upper/lower α partitioned product				—
FMUL8(SU,UL)x16	8x16 upper/lower partitioned product				—
FMULD8(SU,UL)x16	8x16 upper/lower partitioned product				—
FNAND{S}	Logical NAND operation		✓	✓	—
FNEG(s,d,q)	Floating-point negate		✓	✟	—
FNMADD(s,d)	Foating-point Multiply-and-Add and negate	✓	✓	✓	72
FNMSUB(s,d)	Foating-point Multiply-and-Subtract and negate	✓	✓	✓	72
FNOR{S}	Logical NOR operation		✓	✓	—
FNOT(1,2){S}	Copy negated source		✓	✓	—
FPACK(16,32, FIX)	Pixel packing				—
FPADD(16,32){S}	Pixel add (single) 16- or 32-bit				—
FPMADDX{HI}	Integer Multiply-and-Add	✓	✓	✓	80
FPMERGE	Pixel merge				—
FRCPA(s,d)	Floating-point reciprocal approximation	✓	✓	✓	120
FRSQRTA(s,d)	Floating-point reciprocal square root approximation	✓	✓	✓	120
FONE{S}	One fill		✓	✓	—
FOR{S}	Logical OR operation		✓	✓	—
FORNOT(1,2){S}	Logical OR operation with one inverted source		✓	✓	—
FPSUB(16,32){S}	Pixel subtract (single) 16- or 32-bit				—
FsMULd	Floating-point multiply single to double		✓	✓	—
FSQRT(s,d,q)	Floating-point square root		☆		—
FSRC(1,2){S}	Copy source		✓	✓	—

TABLE A-2 Instruction Set *(3 of 7)*

Operation	Name	HPC-ACE Ext. Inst.	Regs.	SIMD	Page
FSELMOV(s,d)	Move selected floating-point register	✓	✓	✓	124
F(s,d,q)TOi	Convert floating point to integer		✓	†	—
F(s,d,q)TO(s,d,q)	Convert between floating-point formats		✓	†	—
F(s,d,q)TOx	Convert floating point to 64-bit integer		✓	†	—
FSUB(s,d,q)	Floating-point subtract		✓	†	—
FTRIMADDd	Floating-point trigonometric function	✓	✓	✓	125
FTRIS(MUL,SEL)d	Floating-point trigonometric functions	✓	✓	✓	125
FXNOR{S}	Logical XNOR operation		✓	✓	—
FXOR{S}	Logical XOR operation		✓	✓	—
FxTO(s,d,q)	Convert 64-bit integer to floating-point		✓	†	—
FZERO{S}	Zero fill		✓	✓	—
ILLTRAP	Illegal instruction				—
JMPL	Jump and link				81
LDD^{D}	Load integer doubleword		✓		—
LDDA$^{D,\ P_{ASI}}$	Load integer doubleword from alternate space		✓		—
LDDA ASI_NUCLEUS_QUAD*	Load integer quadword, atomic		✓		—
LDDA ASI_QUAD_PHYS*	Load integer quadword, atomic (physical address)		✓		89
LDDF	Load double floating-point		✓	✓	82
LDDFA$^{P_{ASI}}$	Load double floating-point from alternate space		✓	✓	86
LDDFA ASI_BLK*	Block loads		✓		68
LDDFA ASI_FL*	Short floating point loads				—
LDF	Load floating-point		✓	✓	82
LDFA$^{P_{ASI}}$	Load floating-point from alternate space		✓	✓	86
LDFSR^{D}	Load floating-point state register lower		✓		82
LDQF	Load quad floating-point		✓		82
LDQFA$^{P_{ASI}}$	Load quad floating-point from alternate space		✓		86
LDSB	Load signed byte		✓		—
LDSBA$^{P_{ASI}}$	Load signed byte from alternate space		✓		—
LDSH	Load signed halfword		✓		—
LDSHA$^{P_{ASI}}$	Load signed halfword from alternate space		✓		—
LDSTUB	Load-store unsigned byte		✓		—
LDSTUBA$^{P_{ASI}}$	Load-store unsigned byte in alternate space		✓		—
LDSW	Load signed word		✓		—
LDSWA$^{P_{ASI}}$	Load signed word from alternate space		✓		—
LDUB	Load unsigned byte		✓		—
LDUBA$^{P_{ASI}}$	Load unsigned byte from alternate space		✓		—

TABLE A-2 Instruction Set *(4 of 7)*

Operation	Name	HPC-ACE Ext. Inst.	HPC-ACE Ext. Regs.	HPC-ACE Ext. SIMD	Page
LDUH	Load unsigned halfword		✓		—
LDUHA$^{P_{ASI}}$	Load unsigned halfword from alternate space		✓		—
LDUW	Load unsigned word		✓		—
LDUWA$^{P_{ASI}}$	Load unsigned word from alternate space		✓		—
LDX	Load extended		✓		—
LDXA$^{P_{ASI}}$	Load extended from alternate space		✓		—
LDXFSR	Load floating-point state register		✓		82
MEMBAR	Memory barrier				91
MOVcc	Move integer register if condition is satisfied		✓		—
MOVr	Move integer register on contents of integer register		✓		—
MULScc^{D}	Multiply step (and modify condition codes)		✓		—
MULX	Multiply 64-bit integers		✓		—
NOP	No operation		✓		93
OR (ORcc)	Inclusive-or (and modify condition codes)		✓		—
ORN (ORNcc)	Inclusive-or not (and modify condition codes)		✓		—
PDIST	Pixel component distance				—
POPC	Population count		✓		95
PREFETCH	Prefetch data		✓		96
PREFETCHA$^{P_{ASI}}$	Prefetch data from alternate space		✓		96
RDASI	Read ASI register		✓		98
RDASR$^{P_{ASR}}$	Read ancillary state register		✓		98
RDCCR	Read condition codes register		✓		98
RDDCR^{P}	Read dispatch control register		✓		98
RDFPRS	Read floating-point registers state register		✓		98
RDGSR	Read graphic status register		✓		98
RDPC	Read program counter		✓		98
RDPCR$^{P_{PCR}}$	Read performance control register		✓		98
RDPIC$^{P_{PIC}}$	Read performance instrumentation counters		✓		98
RDPR^{P}	Read privileged register		✓		—
RDSOFTINT^{P}	Read per-processor soft interrupt register		✓		98
RDSTICK$^{P_{NPT}}$	Read system TICK register		✓		98
RDSTICK_CMPR^{P}	Read system TICK compare register		✓		98
RDTICK$^{P_{NPT}}$	Read TICK register		✓		98
RDTICK_CMPR^{P}	Read TICK compare register		✓		98
RDTXAR^{P}	Read TXAR register	✓	✓		98
RDXASR	Read XASR register	✓	✓		98

TABLE A-2 Instruction Set *(5 of 7)*

Operation	Name	HPC-ACE Ext. Inst.	HPC-ACE Ext. Regs.	HPC-ACE Ext. SIMD	Page
RDY[D]	Read Y register		✓		98
RESTORE	Restore caller's window		✓		—
RESTORED[P]	Window has been restored				—
RETRY[P]	Return from trap and retry				—
RETURN	Return				—
SAVE	Save caller's window		✓		—
SAVED[P]	Window has been saved				—
SDIV[D] (SDIVcc[D])	32-bit signed integer divide (and modify condition codes)		✓		—
SDIVX	64-bit signed integer divide		✓		—
SETHI	Set high 22 bits of low word of integer register		✓		—
SHUTDOWN	Shut down the processor				100
SIAM	Set Interval Arithmetic Mode				—
SIR	Software-initiated reset				—
SLEEP	Sleep this thread				79
SLL	Shift left logical		✓		—
SLLX	Shift left logical, extended		✓		—
SMUL[D] (SMULcc[D])	Signed integer multiply (and modify condition codes)		✓		—
SRA	Shift right arithmetic		✓		—
SRAX	Shift right arithmetic, extended		✓		—
SRL	Shift right logical		✓		—
SRLX	Shift right logical, extended		✓		—
STB	Store byte		✓		—
STBA[P, ASI]	Store byte into alternate space		✓		—
STBAR[D]	Store barrier				115
STD[D]	Store doubleword		✓		—
STDA[D, P, ASI]	Store doubleword into alternate space		✓		—
ST(D,DF,X)A ASI_XFILL*	Cache line fill	✓	✓		135
STDF	Store double floating-point		✓	✓	101
STDFA[P, ASI]	Store double floating-point into alternate space		✓	✓	105
STDFA ASI_BLK*	Block stores		✓		68
STDFA ASI_FL*	Short floating point stores				—
STDFA ASI_PST*	Partial Store instructions				94
STDFR	Store double floating-point on register's condition	✓	✓	✓	130
STF	Store floating-point		✓	✓	101
STFA[P, ASI]	Store floating-point into alternate space		✓	✓	105
STFR	Store floating-point on register condition	✓	✓	✓	130

TABLE A-2 Instruction Set *(6 of 7)*

Operation	Name	HPC-ACE Ext. Inst.	HPC-ACE Ext. Regs.	HPC-ACE Ext. SIMD	Page
STFSR[D]	Store floating-point state register		✓		101
STH	Store halfword		✓		—
STHA[P, ASI]	Store halfword into alternate space		✓		—
STQF	Store quad floating-point		✓		101
STQFA[P, ASI]	Store quad floating-point into alternate space		✓		105
STW	Store word		✓		—
STWA[P, ASI]	Store word into alternate space		✓		—
STX	Store extended		✓		—
STXA[P, ASI]	Store extended into alternate space		✓		—
STXFSR	Store extended floating-point state register		✓		101
SUB (SUBcc)	Subtract (and modify condition codes)		✓		—
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)		✓		—
SUSPEND[P]	Suspend this thread				78
SWAP[D]	Swap integer register with memory		✓		—
SWAPA[D, P, ASI]	Swap integer register with memory in alternate space		✓		—
SXAR(1,2)	Set XAR	✓			133
TADDcc (TADDccTV[D])	Tagged add and modify condition codes (trap on overflow)		✓		—
Tcc	Trap on integer condition codes				108
TSUBcc (TSUBccTV[D])	Tagged subtract and modify condition codes (trap on overflow)		✓		—
UDIV[D] (UDIVcc[D])	Unsigned integer divide (and modify condition codes)		✓		—
UDIVX	64-bit unsigned integer divide		✓		—
UMUL[D] (UMULcc[D])	Unsigned integer multiply (and modify condition codes)		✓		—
WRASI	Write ASI register		✓		112
WRASR[P, ASR]	Write ancillary state register		✓		112
WRCCR	Write condition codes register		✓		112
WRDCR[P]	Write dispatch control register		✓		112
WRFPRS	Write floating-point registers state register		✓		112
WRGSR	Write graphic status register		✓		112
WRPCR[P, PCR]	Write performance control register		✓		112
WRPIC[P, PIC]	Write performance instrumentation counters register		✓		112
WRPR[P]	Write privileged register		✓		109
WRSOFTINT[P]	Write per-processor soft interrupt register		✓		112
WRSOFTINT_CLR[P]	Clear bits of per-processor soft interrupt register		✓		112
WRSOFTINT_SET[P]	Set bits of per-processor soft interrupt register		✓		112
WRTICK_CMPR[P]	Write TICK compare register		✓		112
WRSTICK[P]	Write System TICK register		✓		112

TABLE A-2 Instruction Set *(7 of 7)*

Operation	Name	HPC-ACE Ext. Inst.	HPC-ACE Ext. Regs.	HPC-ACE Ext. SIMD	Page
WRSTICK_CMPR[P]	Write System TICK compare register		✓		112
WRTXAR[P]	Write TXAR register	✓	✓		112
WRXAR	Write XAR register	✓	✓		112
WRXASR	Write XASR register	✓	✓		112
WRY[D]	Write Y register		✓		112
XNOR (XNORcc)	Exclusive-nor (and modify condition codes)		✓		—
XOR (XORcc)	Exclusive-or (and modify condition codes)		✓		—

A.4 Block Load and Store Instructions (VIS I)

Deprecated – In SPARC64 VIIIfx, block load/store instructions are provided for backwards compatibility only. It is recommended that new programs avoid using these instructions. For high-speed copying of data from memory, see Section A.79, *"Cache Line Fill with Undetermined Values"*.

The SPARC64 VIIIfx specification of block load/store differs from the specification used in SPARC64 V through SPARC64 VII. The new specification has stronger restrictions, and part of the new specification is incompatible with the previous specification. The differences are described below:

1. Block load/store memory accesses are not atomic; they are split into separate 8-byte load/store accesses in internal hardware. Each load/store obeys all ordering constraints imposed by `MEMBAR` instructions and atomic instructions.

2. The block load/store instructions adhere to TSO. That is, the ordering between the separate load/store accesses of a block load/store and between other load/store/atomic instructions conforms to TSO.

Compatibility Note – In the previous specification, the memory order did not conform to the SPARC V9 memory model; the separate 8-byte accesses were performed in RMO.

3. The order of register accesses is preserved in the same manner as for other instructions. That is, read-after-write and write-after-write register accesses by a block load/store and another instruction are performed in program order.

4. The cache behavior of a block load/store is the same as a normal load/store. A block load reads data from the L1 cache; if the data is not in the L1 cache, the L1 cache is updated with data from memory before being read. A block stores writes data to the L1 cache; if the data is not in the L1 cache, the L1 cache is updated with data from memory before being written.

Compatibility Note – The cache side effects of a block load/store have changed greatly. In the previous specification, a block load reads data from the cache; if the data is not in the cache, behavior is undefined. A block store writes data to a cache containing a dirty copy of the data; at the same time, copies in all higher-level caches (caches closer to the pipeline) are invalidated. If no cache contains a dirty copy or the data is not in the cache, the block store writes the data to memory.

5. In SPARC64 VIIIfx, block stores and block stores with commit have the same behavior.

Compatibility Note – The cache side effects of a block store with commit have changed greatly. In the previous specification, a block store with commit forces the data to be written to memory and invalidates copies in all caches.

6. For a block load/store instruction to a page with `TTE.E` = 0, any of the 8-byte load/store accesses may cause a *fast_data_access_MMU_miss* exception. When the exception is signalled for a block load, register values may or may not have been updated by the block load. When the exception is signalled for a block store, the memory state prior to the block store is preserved.

Programming Note – Block stores to certain noncacheable address spaces appear to complete normally, but no actual store is performed. Refer to the system specification for details.

Note – As defined in JPS1 **Commonality**, block load/store instructions do not cause *LDDF_mem_address_not_aligned* or *STDF_mem_address_not_aligned* exceptions (see Appendix L.3.3). However, a `LDDFA` instruction that specifies `ASI_BLK_COMMIT_{P,S}` is not a block load/store instruction, and an access aligned on a 4-byte boundary causes a *LDDF_mem_address_not_aligned* exception. See *"Block Load and Store ASIs"* (page 220).

Exceptions

illegal_instruction (misaligned `rd`)
fp_disabled
illegal_action (`XAR.v` = 1 and (`XAR.urs1` > 1 or
(`i` = 0 and `XAR.urs2` > 1) or
(`i` = 1 and `XAR.urs2` ≠ 0) or
`XAR.urs3`<2> ≠ 0);
`XAR.v` = 1 and `XAR.simd` = 1)
mem_address_not_aligned (see *"Block Load and Store ASIs"* (page 220))
LDDF_mem_address_not_aligned (see *"Block Load and Store ASIs"* (page 220))
VA_watchpoint (only detected on the first 8 bytes of a transfer)
fast_data_access_MMU_miss
data_access_exception (see *"Block Load and Store ASIs"* (page 220))
fast_data_access_protection
PA_watchpoint (only detected on the first 8 bytes of a transfer)
data_access_error

A.9 Call and Link

SPARC64 VIIIfx clears the more significant 32 bits of the `PC` value stored in `r[15]` when `PSTATE.AM` = 1 (impl. dep. #125). The updated value in `r[15]` is visible to the delay slot instruction.

Exceptions *illegal_action* (`XAR.v` = 1)

A.24 Implementation-Dependent Instructions

Opcode	op3	Operation
IMPDEP1	11 0110	Implementation-Dependent Instruction 1
IMPDEP2	11 0111	Implementation-Dependent Instruction 2

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 VIIIfx uses IMPDEP1 to encode the VIS, SUSPEND, SLEEP, FCMP*cond*{d,s}, FMIN{d,s}, FMAX{d,s}, FRCPA{d,s}, FRSQRTA{d,s}, FTRISSELd, and FTRISMULd instructions (impl. dep. #106). IMPDEP2A is used to encode the Integer Multiply-Add instructions (FPMADDX and FPMADDXHI), FTRIMADDd, and FSELMOV{d,s}; IMPDEP2B is used to encode the Floating-Point Multiply-Add/Subtract instructions (impl. dep. #106).

For information on adding new instructions to the SPARC V9 architecture using the implementation-dependent instructions, see Section I.1.2, *"Implementation-Dependent and Reserved Opcodes"*, in JPS1 **Commonality**.

Compatibility Note – These instructions replace the CPop*n* instructions in SPARC V8.

New IMPDEP1 and IMPDEP2 instructions added in SPARC64 VIIIfx are not described in Section A.24; instead, these instructions are located after Section A.71 with the other new instructions.

Exceptions Implementation-dependent.

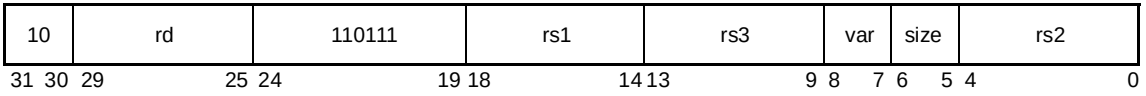
A.24.1 Floating-Point Multiply-Add/Subtract

SPARC64 VIIIfx uses the `IMPDEP2B` opcode space to implement the Floating-Point Multiply-Add/Subtract (FMA) instructions. FMA instructions support SIMD execution, which is an HPC-ACE feature. This section first describes the behavior of non-SIMD FMA instructions, then explains the use of FMA instructions with HPC-ACE features.

HPC-ACE Ext.					
Regs.	**SIMD**	**Opcode**	**Var**	**Size**[1] [2]	**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
✓	✓	`FNMSUBs`	10	01	Negative Multiply-Subtract Single
✓	✓	`FNMSUBd`	10	10	Negative Multiply-Subtract Double
✓	✓	`FNMADDs`	11	01	Negative Multiply-Add Single
✓	✓	`FNMADDd`	11	10	Negative Multiply-Add Double

1. See Section A.24.4, Section A.75, and Section A.76 for instructions with size = 00.
2. size = 11 is reserved for quad precision instructions. However, this encoding is partly used in Section A.75, *"Move Selected Floating-Point Register on Floating-Point Register's Condition"*.

Format (5)



Operation 処理	**Implementation 演算**
Multiply-add	$rd \leftarrow rs1 \times rs2 + rs3$
Multiply-Subtract	$rd \leftarrow rs1 \times rs2 - rs3$
Negative multiply-subtract	$rd \leftarrow -rs1 \times rs2 + rs3$
Negative multiply-add	$rd \leftarrow -rs1 \times rs2 - rs3$

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 `FMADD` instruction multiplies the floating-point registers specified by `rs1` and `rs2`, adds the product to the floating-point register specified by `rs3`, and writes the result into the floating-point register specified by `rd`.

The `FMSUB` instruction multiplies the floating-point registers specified by `rs1` and `rs2`, subtracts the product from the floating-point register specified by `rs3`, and writes the result into the floating-point register specified by `rd`.

The `FNMADD` instruction multiplies the floating-point registers specified by `rs1` and `rs2`, *negates* the product, *subtracts* this value from the floating-point register specified by `rs3`, and writes the result into the floating-point register specified by `rd`.

The `FNMSUB` instruction multiplies the floating-point registers specified by `rs1` and `rs2`, *negates* the product, *adds* this value from the floating-point register specified by `rs3`, and writes the result into the floating-point register specified by `rd`.

An FMA instruction is processed as a fused multiply-add/subtract operation. That is, the result of the multiply operation is not rounded and has infinite precision; the add/subtract operation is performed with a rounding step. Thus, at most one rounding error can occur.

In SPARC64 V, multiply and add/subtract were performed as separate operations. That is, the result of the multiply operation was rounded (as if it were a separate multiply operation). The add/subtract operation then performed a second rounding step. Thus, up to two rounding errors could occur.
Additionally, the behavior of `FNMADD` and `FNMSUB` differs when `rs1` or `rs2` is a `NaN` operand. SPARC64 VIIIfx outputs one of the `NaN` operands as the result; SPARC64 V inverts the sign bit of one of the `NaN` operands before outputting that value as the result.
TABLE A-3 summarizes how SPARC64 VIIIfx handles traps caused by FMA instructions. If the multiply causes an invalid (NV) exception that traps, or a denormal source operand is detected while `FSR.NS` = 1, execution is halted and the instruction generates a trap. The exception condition is indicated in `FSR.cexc`, and `FSR.aexc` is not updated. The add/subtract is only executed when the multiply does not cause an invalid exception that traps.

If the add/subtract generates a IEEE754 exception condition tha traps, FSR.cexc only indicates the trapping exception condition, and FSR.aexc is not updated. If there are no trapping IEEE754 exception conditions, FSR.cexc indicates the nontrapping exception conditions. FSR.aexc is updated with the logical OR of FSR.cexc and FSR.aexc. The *unfinished_FPop* exception conditions for rs1 and rs2 (multiply) are the same as for FMUL; the conditions for the product and rs3 (add/subtract) are the same as for FADD.

TABLE A-3 IEEE754 Exceptions for Floating-Point Multiply-Add/Subtract Instructions

FMUL	IEEE754 trap (NV or NX only)	No trap	No trap
FADD	—	IEEE754 trap	No trap
	Exception condition for FMUL	Exception condition for FADD	Nontrapping exception conditions for FADD
cexc			
aexc	Not updated	Not updated	Logical OR of cexc (above) and aexc

The values indicated in aexc depend on the exception conditions, which are summarized in TABLE A-4 and TABLE A-5. The following terminology is used for nontrapping IEEE exception conditions: uf, of, nv, and nx. These correspond to underflow (uf), overflow (of), invalid (nv), and inexact (nx) exception conditions.

TABLE A-4 Values of aexc for Nontrapping Exception Conditions, FSR.NS = 0

		FADD			
		none	nx	of nx	nv
FMUL	none	none	nx	of nx	nv
	nv	nv	—	—	nv

TABLE A-5 Values of aexc for Nontrapping Exception Conditions, FSR.NS = 1

		FADD				
		none	nx	of nx	uf nx	nv
FMUL	none	none	nx	of nx	uf nx	nv
	nv	nv	—	—	—	nv
	nx	nx	nx	of nx	uf nx	nv nx

In these tables, cases indicated by an "—" do not exist.

Programming Note – The Floating-Point Multiply-Add/Subtract instructions are implemented using the SPARC V9 IMPDEP2 opcode space. These instructions are specific to SPARC64 VIIIfx and cannot be used in any programs that will be executed on another SPARC V9 processor.

SIMD Execution of FMA Instructions

In SPARC64 VIIIfx, the basic and extended operations of a SIMD instruction are executed independently. Because the basic operation uses registers in the range `f[0]` – `f[254]`, the operation always sets the most significant bit of `rs1`, `rs2`, `rs3`, and `rd` to 0 (page 22). This restriction is relaxed for SIMD FMA instructions, such that operations between basic and extended registers can be executed.

Note – The above limitation for SIMD instructions only applies when `XAR.simd` = 1. When `XAR.simd` = 0, `rs1`, `rs2`, `rs3`, and `rd` can use any of the floating point registers.

For a SIMD FMA instruction, `rs1` and `rs2` can specify any of the floating-point registers `f[2n]` (n=0–255). When the basic operation specifies an extended register, the extended operation uses the corresponding basic register. That is, the basic operation uses registers `f[2n]` (n=0…255), and the extended operation uses `f[(2n+256)` mod `512]` (n=0…255).

On the other hand, the limitations for `rs3` and `rd` are the same as for other SIMD instructions. The basic operation must use registers `f[0]` – `f[254]`, and the extended operation must use `f[256]` – `f[510]`. That is, `urs3`<2> and `urd`<2> are never used to specify registers. SIMD FMA instructions use these bits to specify additional execution options; these bits should be 0 for all other SIMD instructions. When `urs3`<2> = 1, the register specified by `rs1` is used for both basic and extended operations. When `urd`<2> = 1, the sign of the product for the extended operation is reversed.

The meanings of `XAR.urs1`, `XAR.urs2`, `XAR.urs3`, and `XAR.urd` for a SIMD FMA instruction is summarized below:

- `XAR.urs1`<2> `rs1`<8> for the basic operation, ¬`rs1`<8> for the extended operation
- `XAR.urs2`<2> `rs2`<8> for the basic operation, ¬`rs2`<8> for the extended operation
- `XAR.urs3`<2> specifies whether the extended operation uses `rs1`<8> or ¬`rs1`<8>
- `XAR.urd`<2> specifies whether the sign of the product is reversed for the extended operation

The `rs1`<8> bit described above is a bit in the decoded HPC-ACE register number for a double precision register. See FIGURE 5-1 (page 21) for details.

	frs1: urs1<2:0>, rs1<5:0>	$frs1_i$: ¬urs1<2>, urs1<1:0>, rs1<5:0>
	frs2: urs2<2:0>, rs2<5:0>	$frs2_i$: ¬urs2<2>, urs2<1:0>, rs2<5:0>
	$frs3_b$: 1'b0, urs3<1:0>, rs3<5:0>	$frs3_e$: 1'b1, urs3<1:0>, rs3<5:0>
	frd_b: 1'b0, urd<1:0>, rd<5:0>	$frs1_i$: 1'b1, urd<1:0>, rd<5:0>
	c: urs3<2>	
	n: urd<2>	

Instruction	Basic operation	Extended operation
fmadd	$frd_b \leftarrow frs1 \times frs2 + frs3_b$	$frd_e \leftarrow (-1)^n \times (c\ ?\ frs1 : frs1_i) \times frs2_i + frs3_e$
fmsub	$frd_b \leftarrow frs1 \times frs2 - frs3_b$	$frd_e \leftarrow (-1)^n \times (c\ ?\ frs1 : frs1_i) \times frs2_i - frs3_e$
fnmsub	$frd_b \leftarrow -frs1 \times frs2 + frs3_b$	$frd_e \leftarrow -(-1)^n \times (c\ ?\ frs1 : frs1_i) \times frs2_i + frs3_e$
fnmadd	$frd_b \leftarrow -frs1 \times frs2 - frs3_b$	$frd_e \leftarrow -(-1)^n \times (c\ ?\ frs1 : frs1_i) \times frs2_i - frs3_e$

Example 1: Multiplication of complex numbers

$(a_1 + ib_1)\ (a_2 + ib_2) = (a_1a_2 - b_1b_2) + i(a_1b_2 + a_2b_1)$

```
        /*
         * X: location of source complex number
         * Y: location of source complex number
         * Z: location for destination complex number
         */

        /* setup registers */
        sxar2
        ldd,s           [X], %f0  /* %f0: a1, %f256: b1 */
        ldd,s           [Y], %f2  /* %f2: a2, %f258: b2 */
        sxar1
        fzero,s         %f4       /* clear destination registers */

        /* perform calculations */
        sxar2
        fnmaddd,snc     %f256, %f258, %f4, %f4
                             /* %f4    := -%f256 * %f258 - %f4 */
                             /* %f260  := %f256 * %f2 - %f260 */
        fmaddd,sc       %f0, %f2, %f4, %f4
                             /* %f4    := %f0 * %f2 + %f4 */
                             /* %f260  := %f0 * %f258 + %f260 */
        /* store results */
        sxar1
        std,s           %f4, [Z]
```

Example 2: 2x2 matrix multiplication

```
/*
 * A: location of source matrix: a11, a12, a21, a22
 * B: location of source matrix: b11, b12, b21, b22
 * C: location for destination matrix: c11, c12, c21, c22
 */

/* setup registers */
sxar2
ldd,s         [A], %f0  /* %f0: a11, %f256: a12 */
ldd,s         [A+16], %f2/* %f2: a21, %f258: a22 */
sxar2
ldd,s         [B], %f4  /* %f4: b11, %f260: b12 */
ldd,s         [B+16], %f6/* %f6: b21, %f262: b22 */
sxar2
fzero,s       %f8       /* %f8: c11, %f264: c12 */
fzero,s       %f10      /* %f10: c21, %f266: c22 */

/* perform calculations */
sxar2
fmaddd,sc     %f0, %f4, %f8, %f8
                   /* %f8    := %f0 * %f4 + %f8 */
                   /* %f264  := %f0 * %f260 + %f264 */
fmaddd,sc     %f256, %f6, %f8, %f8
                   /* %f8    := %f256 * %f6 + %f8 */
                   /* %f264  := %f256 * %f262 + %f264 */
sxar2
fmaddd,sc     %f2, %f4, %f10, %f10
                   /* %f10   := %f2 * %f4 + %f10 */
                   /* %f266  := %f2 * %f260 + %f266 */
fmaddd,sc     %f258, %f6, %f10, %f10
                   /* %f10   := %f258 * %f6 + %f10 */
                   /* %f266  := %f258 * %f262 + %f266 */
/* store results */
sxar2
std,s         %f8, [Z]
std,s         %f10, [Z+16]
```

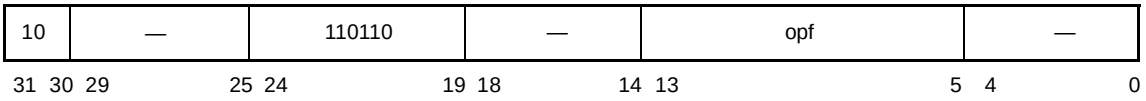
Exceptions

illegal_instruction (size = 11_2 and var ≠ 11_2)
(in this case, *fp_disabled* is not checked)
fp_disabled
fp_exception_ieee_754 (NV, NX, OF, UF)
fp_exception_other (`FSR.ftt` = *unfinished_FPop*)

A.24.2 Suspend

HPC-ACE Ext. Regs.	SIMD	opcode	opf	operation
		SUSPEND^{P}	0 1000 0010	Suspend the thread

Format (3)



Assembly Language Syntax
suspend

Description The SUSPEND instruction sets PSTATE.IE = 1 and causes the hardware thread that executed the instruction to enter SUSPENDED state. The following conditions cause the thread to exit SUSPENDED state and return to execute state:

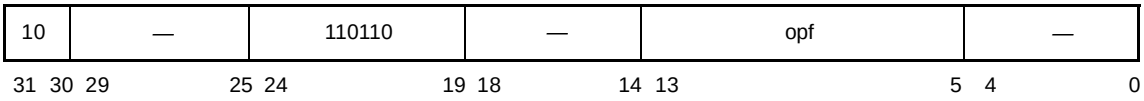
- POR, WDR, XIR
- *interrupt_vector*
- *interrupt_level_n*

Exceptions *privileged_opcode*
illegal_action (XAR.v = 1)

A.24.3 Sleep

HPC-ACE Ext. Regs.	SIMD	opcode	opf	operation
		SLEEP	0 1000 0011	Put the thread to sleep

Format (3)



Assembly Language Syntax
sleep

Description

The SLEEP instruction puts the hardware thread that executed the instruction to sleep. The following conditions wake the thread:

- POR, WDR, XIR
- *interrupt_vector*
- *interrupt_level_n*
- A specified period of time, which is implementation dependent.
  In SPARC64 VIIIfx, this is about 1.6 microseconds and is counted by STICK.
- An update of a LBSY that is assigned to one of the window ASIs.
  An update of a LBSY that is *not* assigned to a window ASI does not wake the thread.

Note – If the SLEEP instruction is executed while PSTATE.IE = 0, then an interrupt does not wake the thread.

Programming Note – If a LBSY used by the thread is updated while the thread is not sleeping, then the next SLEEP instruction may not put the thread to sleep.

Exceptions

illegal_action (XAR.v = 1)

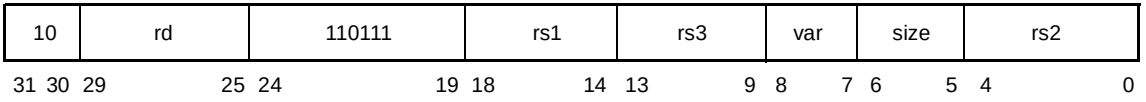
A.24.4 Integer Multiply-Add

SPARC64 VIIIfx uses the IMPDEP2A opcode space to implement the Integer Multiply-Add instructions.

HPC-ACE Ext. Regs.	SIMD	Opcode	Var[1]	Size	Operation
✓	✓	FPMADDX	00	00	Lower 8 bytes of unsigned integer multiply-add
✓	✓	FPMADDXHI	01	00	Upper 8 bytes of unsigned integer multiply-add

1.Refer to Section A.76 for var = 10 and Section A.75 for var = 11.

Format (5)



Assembly Language Syntax	
fpmaddx	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$
fpmaddxhi	$freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$

Description

The Integer Multiply-Add instruction performs a fused multiply-add operation on the unsigned 8-byte integer data stored in the floating-point registers.

FPMADDX multiplies the double-precision registers specified by rs1 and rs2, adds the product to the double-precision register specified by rs3, and writes the lower 8-bytes of the result into the double-precision register specified rd. The floating-point registers specified by rs1, rs2, and rs3 are treated as unsigned 8-byte integer data.

FPMADDXHI multiplies the double-precision registers specified by rs1 and rs2, adds the product to the double-precision register specified by rs3, and writes the upper 8 bytes of the result into the double-precision register specified by rd. The floating-point registers specified by rs1, rs2, and rs3 are treated as unsigned 8-byte integer data.

FPMADDX and FPMADDXHI do not update any bits in the FSR.

Exceptions

fp_disabled
illegal_action (XAR.v = 1 and XAR.simd = 1 and
(XAR.urs1<2> ≠ 0 or XAR.urs2<2> ≠ 0
or XAR.urs3<2> ≠ 0 or XAR.urd<2> ≠ 0))

A.25 Jump and Link

SPARC64 VIIIfx clears the more significant 32 bits of the PC value stored in r[rd] when PSTATE.AM = 1 (impl. dep. #125). The updated value in r[rd] is visible to the delay slot instruction.

When either of the 2 lowest bits of the target address is not 0, a *mem_address_not_aligned* exception occurs. DSFSR and DSFAR are not updated (impl. dep. #237).

Exceptions *illegal_action* (XAR.v = 1)

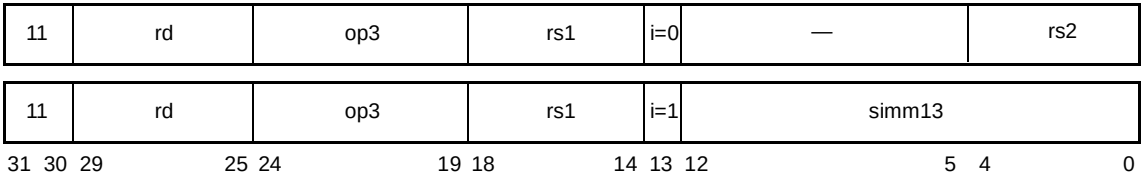
A.26 Load Floating-Point

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	rd	urd	Operation
		LDF	10 0000	0–31	—¶	Load Floating-Point Register
✓	✓	LDF	10 0000	†	0-7	Load Floating-Point Register
✓	✓	LDDF	10 0011	†	0-7	Load Double Floating-Point Register
✓		LDQF	10 0010	†	0-7	Load Quad Floating-Point Register
✓		LDFSR^D	10 0001	0	—	(see A.71.4 of JPS1 **Commonality**)
✓		LDXFSR	10 0001	1	—	Load Floating-Point State Register
		—	10 0001	2–31	—	*Reserved*

† Encoded floating-point register value, as described in Section 5.1.4 of JPS1 **Commonality**.

¶ When XAR.v = 0.

Format (3)



Assembly Language Syntax	
ld	[*address*], $freg_{rd}$
ldd	[*address*], $freg_{rd}$
ldq	[*address*], $freg_{rd}$
ldx	[*address*], %fsr

Description First, non-SIMD behavior is explained.

The load single floating-point instruction (LDF) copies a word from memory into f[rd].

The load doubleword floating-point instruction (LDDF) copied 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.

The load floating-point state register instruction (LDXFSR) waits for all FPop instructions that have not finished execution to complete and then loads a doubleword from memory into the FSR.

Load floating-point instructions access the primary address space (ASI = 80_{16}). The effective address for these instructions is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

LDF causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned. LDXFSR causes a *mem_address_not_aligned* exception if the address is not doubleword aligned. If the floating-point unit is not enabled (per FPRS.FEF and PSTATE.PEF), then a load floating-point instruction causes an *fp_disabled* exception.

In SPARC64 VIIIfx, a non-SIMD LDDF address that is aligned on a 4-byte boundary but not an 8-byte boundary causes an *LDDF_mem_address_not_aligned* exception. System software must emulate the instruction (impl.dep. #109(1)).

Because SPARC64 VIIIfx does not implement LDQF, an attempt to execute the instruction causes a *illegal_instruction* exception. *fp_disabled* is not detected. System software must emulate LDQF (impl.dep. #111(1)).

Programming Note – In SPARC V8, some compilers issued sequences of single-precision loads when they could not determine that doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned loads is expected to be fast, we recommend that compilers issue sets of single-precision loads only when they can determine that doubleword or quadword operands are *not* properly aligned.

In SPARC64 VIIIfx, when there is an access error for a non-SIMD floating-point load, the destination register remains unchanged (impl.dep. #44(1)). See the following subsection for SIMD behavior.

Programming Note – When the address fields (rs1, rs2) of the single-precision floating-point load instruction LDF specify any of the integer registers added by HPC-ACE, the destination register must be a double-precision register. This restriction is a consequence of how rd is decoded when XAR.v = 1 (page 21). A SPARC V9 single-precision register (odd-numbered register) cannot be specified for rd if rs1 or rs2 specifies a HPC-ACE integer register.

SIMD

In SPARC64 VIIIfx, a floating-point load instruction can be executed as a SIMD instruction. A SIMD load instruction simultaneously executes basic and extended loads from the effective address, for either single-precision or double-precision data. See "*Specifying registers for SIMD instructions*" (page 22) for details on how to specify the registers.

A single-precision SIMD load instruction loads 2 single-precision data aligned on a 4-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

A double-precision SIMD load instruction loads 2 double-precision data aligned on an 8-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

Note – A double-precision SIMD load that accesses data aligned on a 4-byte boundary but not an 8-byte boundary does not cause an *LDDF_mem_address_not_aligned* exception.

For both single-precision and double-precision SIMD loads, data for the basic and extended loads may be located on different memory pages. If the TLB search for the basic load succeeds and the TLB search for the extended load fails, then SPARC64 VIIIfx generates a *SIMD_load_across_pages* exception.

A SIMD load can only be used to access cacheable address spaces. An attempt to access a noncacheable address space or a nontranslating ASI using a SIMD load causes a *data_access_exception* exception. The bypass ASIs that can be accessed using a SIMD load instruction are `ASI_PHYS_USE_EC{_LITTTLE}`; a page size of 8 KB is assumed. See Appendix F.11, *"MMU Bypass"*, for details.

Like non-SIMD load instructions, memory access semantics for SIMD load instructions adhere to TSO. A SIMD load simultaneously executes basic and extended loads; however, the ordering between the basic and extended loads conforms to TSO.

In SPARC64 VIIIfx, when there is an access error for a SIMD floating-point load, the destination registers are not changed (impl.dep. #44(1)).

For a SIMD load instruction, endian conversion is done separately for the basic and extended loads. When the basic and extended data are located on different pages with different endianness, conversion is only done for one of the loads.

A watchpoint can be detected in both the basic and extended loads of a SIMD load.

Note – When `PSTATE.AM` = 1, the extended load of a single-precision SIMD load to VA = FFFF FFFF FFFF $FFFC_{16}$ or a double-precision SIMD load to VA = FFFF FFFF FFFF $FFF8_{16}$ accesses VA = 0_{16}.

For information on trap conditions and trap priorities for SIMD load exceptions, refer to Appendix F.5.1, *"Trap Conditions for SIMD Load/Store"* (page 181).

Exceptions

illegal_instruction (`LDQF`;
`LDXFSR` with `rd` = 2–31)
fp_disabled
illegal_action (`LDF`, `LDDF` with `XAR.v` = 1 and (`XAR.urs1` > 1 or
(`i` = 0 and `XAR.urs2` > 1) or
(`i` = 1 and `XAR.urs2` ≠ 0) or
`XAR.urs3`<2> ≠ 0);
`LDF`, `LDDF` with `XAR.v` = 1 and `XAR.simd` = 1 and `XAR.urd`<2> ≠ 0;

LDXFSR with XAR.v = 1 and (XAR.urs1 > 1 or
(i = 0 and XAR.urs2 > 1) or
(i = 1 and XAR.urs2 ≠ 0) or
XAR.urs3<2> ≠ 0 or
XAR.urd ≠ 0 or
XAR.simd = 1))

LDDF_mem_address_not_aligned (LDDF and (XAR.v = 0 or XAR.simd = 0))
mem_address_not_aligned
VA_watchpoint
fast_data_access_MMU_miss
SIMD_load_across_pages
data_access_exception
PA_watchpoint
data_access_error
fast_data_access_protection

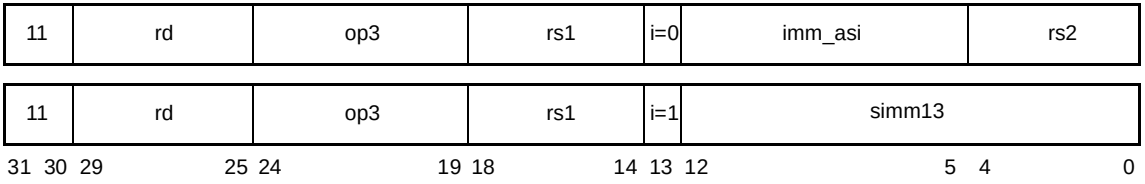
A.27 Load Floating-Point from Alternate Space

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	rd	urd	Operation
		LDFA$^{P_{ASI}}$	11 0000	0–31	—[¶]	Load Floating-Point Register from Alternate Space
✓	✓	LDFA	11 0000	[†]	0-7	Load Floating-Point Register from Alternate Space
✓	✓	LDDFA$^{P_{ASI}}$	11 0011	[†]	0-7	Load Double Floating-Point Register from Alternate Space
✓		LDQFA$^{P_{ASI}}$	11 0010	[†]	0-7	Load Quad Floating-Point Register from Alternate Space

[†] Encoded floating-point register value, as described in Section 5.1.4 of JPS1 **Commonality**.

[¶] When XAR.v = 0.

Format (3)



Assembly Language Syntax	
lda	[*regaddr*] *imm_asi*, *freg$_{rd}$*
lda	[*reg_plus_imm*] %asi, *freg$_{rd}$*
ldda	[*regaddr*] *imm_asi*, *freg$_{rd}$*
ldda	[*reg_plus_imm*] %asi, *freg$_{rd}$*
ldqa	[*regaddr*] *imm_asi*, *freg$_{rd}$*
ldqa	[*reg_plus_imm*] %asi, *freg$_{rd}$*

Description First, non-SIMD behavior is explained.

The load single floating-point from alternate space instruction (LDFA) copies a word from memory into f[rd].

The load double floating-point from alternate space instruction (LDDFA) copies a word-aligned doubleword from memory into a double-precision floating-point register.

The load quad floating-point from alternate space instruction (LDQFA) copies a word-aligned quadword from memory into a quad-precision floating-point register.

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 0; otherwise, it is not privileged. The effective address for these instructions is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

LDFA causes a *mem_address_not_aligned* exception if the effective memory address is not aligned on a 4-byte boundary. If the floating-point unit is not enabled (per FPRS.FEF and PSTATE.PEF), then load floating-point from alternate space instructions cause an *fp_disabled* exception.

In SPARC64 VIIIfx, a non-SIMD LDDFA address that is aligned on a 4-byte boundary but not an 8-byte boundary causes a *LDDF_mem_address_not_aligned* exception. System software must emulate the instruction (impl.dep. #109(2)). Because SPARC64 VIIIfx does not implement LDQFA, an attempt to execute the instruction causes a *illegal_instruction* exception. *fp_disabled* is not detected. System software must emulate LDQFA (impl.dep. #111(2).

Depending on the ASI number, memory accesses that are not 8-byte accesses are defined. Refer to other sections in Appendix A.

Implementation Note – LDFA and LDDFA cause a *privileged_action* exception if PSTATE.PRIV = 0 and bit 7 of the ASI is 0.

Programming Note – In SPARC V8, some compilers issued sequences of single-precision loads when they could not determine that doubleword or quadword operands were properly aligned. For SPARC V9, since emulation of misaligned loads is expected to be fast, compilers should issue sets of single-precision loads only when they can determine that doubleword or quadword operands are not properly aligned.

In SPARC64 VIIIfx, when a non-SIMD floating-point load causes an access error, the destination register is not changed (impl. dep. #44(2)).

Programming Note – When the address fields (rs1, rs2) of the single-precision floating-point load instruction LDFA reference any of the integer registers added by HPC-ACE, the destination register must be a double-precision register. This restriction is a consequence of how rd is decoded when XAR.v = 1 (page 21). A SPARC V9 single-precision register (odd-numbered register) cannot be specified for rd if rs1 or rs2 specifies a HPC-ACE integer register.

SIMD Refer to the SIMD subsection of Section A.26, *"Load Floating-Point"*.

Exceptions

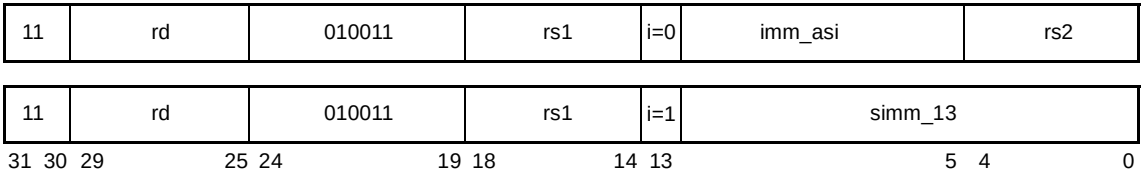
illegal_instruction (LDQFA only)
fp_disabled
illegal_action (LDFA, LDDFA with XAR.v = 1 and (XAR.urs1 > 1 or
(i = 0 and XAR.urs2 > 1) or
(i = 1 and XAR.urs2 ≠ 0) or
XAR.urs3<2> ≠ 0);
LDFA, LDDFA with XAR.v = 1 and XAR.simd = 1 and XAR.urd<2> ≠ 0)
LDDF_mem_address_not_aligned (LDDFA and (XAR.v = 0 or XAR.simd = 0))
mem_address_not_aligned
privileged_action
VA_watchpoint
fast_data_access_MMU_miss
SIMD_load_across_pages
data_access_exception
fast_data_access_protection
PA_watchpoint
data_access_error

A.30 Load Quadword, Atomic [Physical]

The Load Quadword ASIs in this section are specific to SPARC64 VIIIfx.

HPC-ACE Ext. Regs.	SIMD	Opcode	imm_asi	ASI value	Operation
✓		LDDA	ASI_QUAD_LDD_PHYS	34_{16}	128-bit atomic load, physically addressed
✓		LDDA	ASI_QUAD_LDD_PHYS_L	$3C_{16}$	128-bit atomic load, little-endian, physically addressed

Format (3) LDDA



Assembly Language Syntax	
ldda	[*reg_addr*] *imm_asi*, reg_{rd}
ldda	[*reg_plus_imm*] %asi, reg_{rd}

Description

ASIs 34_{16} and $3C_{16}$ are used with the LDDA instruction to atomically read a 128-bit data, physically-addressed data item. The data are placed in an even/odd pair of 64-bit registers. The lower-addressed 64 bits are placed in the even-numbered register; the higher-addressed 64 bits are placed in the odd-numbered register.

ASIs 34_{16} and $3C_{16}$ are specific to SPARC64 VIIIfx. These ASIs are for physically-addressed data; the ASIs for virtually-addressed data are ASIs 24_{16} and $2C_{16}$. An access that is not aligned on a 16-byte boundary causes a *mem_address_not_aligned* exception.

A memory access using ASI_QUAD_LDD_PHYS{_L} behaves as if TTE bits were set to the following:

- TTE.NFO = 0
- TTE.CP = 1
- TTE.CV = 0
- TTE.E = 0
- TTE.P = 1
- TTE.W = 0

Note – The value of TTE.IE depends on the endianness of the ASI. TTE.IE = 0 for ASI 034_{16}, and TTE.IE = 1 for ASI $03C_{16}$.

For this reason, these ASIs can only be used with accesses to cacheable address spaces. Semantically, ASI_QUAD_LDD_PHYS{_L}is equivalent to the combination of ASI_NUCLEUS_QUAD_LDD and ASI_PHYS_USE_EC.

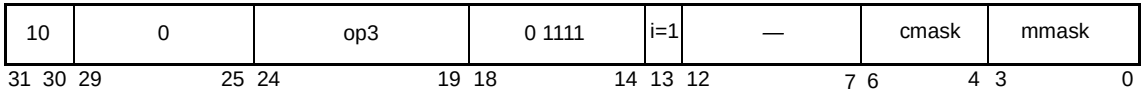
Endian translation is performed separately for the upper-addressed 64 bits and and lower-addressed 64 bits before writing the destination registers.

Exceptions

illegal_instruction (misaligned rd)
illegal_action (XAR.v = 1 and (XAR.urs1 > 1 or
(i = 0 and XAR.urs2 > 1) or
(i = 1 and XAR.urs2 ≠ 0) or
XAR.urs3<2> ≠ 0
XAR.urd > 1);
XAR.v = 1 and XAR.simd = 1)
privileged_action
mem_address_not_aligned
fast_data_access_MMU_miss
data_access_exception
fast_data_access_protection
PA_watchpoint (recognized on only the first 8 bytes of a transfer)
data_access_error

A.35 Memory Barrier

Format (3)



Assembly Language Syntax	
`membar`	*membar_mask*

Description

The memory barrier instruction, `MEMBAR`, has two complementary functions: to express order constraints between memory references and to provide explicit control of memory-reference completion. The `membar_mask` field in the suggested assembly language is the concatenation of the `cmask` and `mmask` instruction fields.

The `mmask` field is encoded in bits 3 through 0 of the instruction. TABLE A-6 specifies the order constraint that each bit of `mmask` (selected when set to 1) imposes on memory references appearing before and after the `MEMBAR`. From zero to four mask bits can be selected in the `mmask` field.

TABLE A-6 Ordering Constraints Specified by `mmask` Bits

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. In SPARC64 VIIIfx, this bit has no effect because all stores are performed in program order.
`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. In SPARC64 VIIIfx, all stores are performed in program order, and the ordering between a load and a store is guaranteed. This bit has no effect.
`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. In SPARC64 VIIIfx, this bit has no effect because all loads are performed in program order.

The cmask field is encoded in bits 6 through 4 of the instruction. Bits in the cmask field, described in TABLE A-7, specify additional constraints on the order of memory references and the processing of instructions. If cmask is zero, then MEMBAR enforces the partial ordering specified by the mmask field; if cmask is nonzero, then completion and partial order constraints are applied.

TABLE A-7 cmask Bits

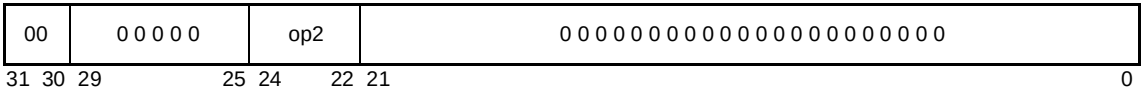
Mask Bit	Function	Name	Description
cmask<2>	Synchronization barrier	#Sync	All operations (including nonmemory reference operations) appearing prior to the MEMBAR must have been performed and the effects of any exceptions become visible before any instruction after the MEMBAR may be initiated.
cmask<1>	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. Equivalent to #Sync in SPARC64 VIIIfx.
cmask<0>	Lookaside barrier	#Lookaside	A store appearing before the MEMBAR must complete before any load following the MEMBAR referencing the same address can be initiated. Equivalent to #Sync in SPARC64 VIIIfx.

Exceptions *illegal_action* (XAR.v = 1)

A.41 No Operation

HPC-ACE Ext. Regs.	SIMD	Opcode	op2	Operation
✓		NOP	100	No Operation

Format (2)



Assembly Language Syntax
nop

Description NOP is a special case of the SETHI instruction, with imm22 = 0 and rd = 0.

The NOP instruction changes no program-visible state, except that of the PC and nPC registers. However, a NOP that is executed while xar.urd = 1 is interpreted as a SETHI instruction whose rd specifies r[32], which is updated.

Exceptions *illegal_action* (XAR.v = 1 and
(XAR.simd = 1 or XAR.urs1 ≠ 0 or XAR.urs2 ≠ 0 or
XAR.urs3 ≠ 0 or XAR.urd > 1))

A.42 Partial Store (VIS I)

Watchpoint detection for partial store instructions is conservative in SPARC64 VIIIfx. The DCUCR Data Watchpoint masks are only checked for a nonzero value (watchpoint enabled). The byte store mask in r[rs2] of the partial store instruction is ignored, and a watchpoint exception can occur even if the mask is zero (that is, when no store occurs) (impl. dep. #249).

Implementation Note – When the byte store mask for a partial store instruction to a noncacheable address space is 0, SPARC64 VIIIfx generates a bus transaction with a zero-byte mask.

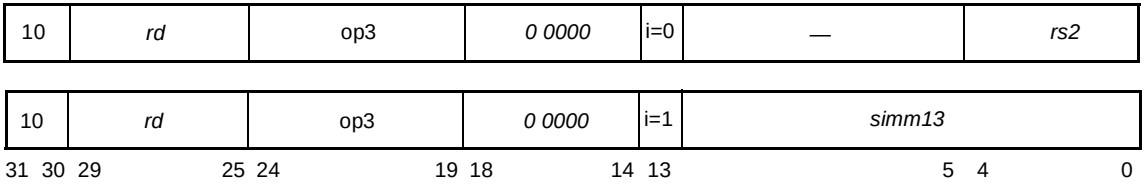
Exceptions

illegal_instruction (i = 1)
fp_disabled
illegal_action (XAR.v = 1)
LDDF_mem_address_not_aligned (see *"Partial Store ASIs"* (page 221))
mem_address_not_aligned (see *"Partial Store ASIs"* (page 221))
VA_watchpoint
fast_data_access_MMU_miss
data_access_exception (see *"Partial Store ASIs"* (page 221))
fast_data_access_protection
PA_watchpoint
data_access_error

A.48 Population Count

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	Operation
✓		POPC	10 1110	Population Count

Format (3)



Assembly Language Syntax	
popc	*reg_or_imm*, *reg*$_{rd}$

Description POPC counts the number of one bits in r[rs2] if $i = 0$, or the number of one bits in sign_ext(*simm13*) if $i = 1$, and stores the count in r[rd]. This instruction does not modify the condition codes.

Note – Unlike SPARC64 V, SPARC64 VIIIfx implements this instruction in hardware.

Exceptions

illegal_instruction (instruction<18:14> ≠ 0)
illegal_action (XAR.v = 1 and (XAR.urs1 ≠ 0 or
(i = 0 and XAR.urs2 > 1) or
(i = 1 and XAR.urs2 ≠ 0) or
XAR.urs3 ≠ 0 or
XAR.urd > 1 or
XAR.simd = 1)

A.49 Prefetch Data

In SPARC64 VIIIfx, the PREFETCHA instruction is valid for the following ASIs:

- ASI_PRIMARY (080_{16}), ASI_PRIMARY_LITTLE (088_{16})
- ASI_SECONDARY (081_{16}), ASI_SECONDARY_LITTLE (089_{16})
- ASI_NUCLEUS (04_{16}), ASI_NUCLEUS_LITTLE ($0C_{16}$)
- ASI_PRIMARY_AS_IF_USER (010_{16}), ASI_PRIMARY_AS_IF_USER_LITTLE (018_{16})
- ASI_SECONDARY_AS_IF_USER (011_{16}), ASI_SECONDARY_AS_IF_USER_LITTLE (019_{16})

If any other ASI is specified, PREFETCHA executes as a NOP.

In SPARC64 VIIIfx, the size of a data block is 128 bytes and the alignment is a 128-byte boundary (impl. dep. #103(3)). For the PREFETCH/PREFETCHA instructions, specifying any address in a data block causes the entire data block to be prefetched. There are no alignment restrictions on the address specified.

Address spaces with TTE.CP = 0 are nonprefetchable, and a prefetch to these address spaces executes as a NOP.

TABLE A-8 describes the prefetch variants implemented in SPARC64 VIIIfx.

TABLE A-8 Prefetch Variants

fcn	Which cache to move data to	Cache state	Description
0	L1D	S,E	
1	L2	S,E	
2	L1D	M,E	
3	L2	M,E	
4	—	—	NOP
5-15	reserved (SPARC V9)		*illegal_instruction* exception is signalled
16-19	*implementation dependent*		NOP
20	L1D	S,E	Strong Prefetch
21	L2	S,E	Strong Prefetch
22	L1D	M,E	Strong Prefetch
23	L2	M,E	Strong Prefetch
24-31	*implementation dependent*		NOP

Strong Prefetch

A prefetch instruction with fcn = 20, 21, 22 or 23 is a Strong Prefetch. In SPARC64 VIIIfx, a strong prefetch is guaranteed to execute, except when a TLB miss occurs and `DCUCR.weak_spca` = 1.

Programming Note – If there is a lack of CPU resources, prefetches may not be executed; however, a strong prefetch will execute. This may negatively affect the execution of subsequent loads and stores; unnecessary use of strong prefetched should be avoided.

SPARC64 VIIIfx does not cause a *fast_data_access_MMU_miss* exception when fcn = 20, 21, 22, or 23 (impl. dep. #103(2)).

Hardware Prefetch

Enabling/disabling hardware prefetch does not affect the execution of `PREFETCH` and `PREFETCHA` instructions. The value of `XAR.dis_hw_pf` is ignored.

Exceptions

illegal_instruction (`fcn` = 5–15)
illegal_action (`XAR.v` = 1 and (`XAR.simd` = 1 or
`XAR.urs1` > 1 or
(`i` = `0` and `XAR.urs2` > 1) or
(`i` = `1` and `XAR.urs2` ≠ 0) or
`XAR.urs3`<2> ≠ 0 or
`XAR.urd` ≠ 0))

A.51 Read State Register

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	rs1	Operation
✓		RDY[D]	10 1000	0	Read Y Register; deprecated (see A.71.9 in JPS1 **Commonality**)
		—	10 1000	1	*Reserved*
✓		RDCCR	10 1000	2	Read Condition Codes Register
✓		RDASI	10 1000	3	Read ASI Register
✓		RDTICK[P$_{NPT}$]	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	STBAR, MEMBAR, or *Reserved*; see Appendix A.51, *"Read State Register"*, in JPS1 **Commonality**
		RDASR	10 1000	16-31	Read non-SPARC V9 ASRs
✓		RDPCR[P$_{PCR}$]		16	Read Performance Control Registers (PCR)
✓		RDPIC[P$_{PIC}$]		17	Read Performance Instrumentation Counters (PIC)
✓		RDDCR[P]		18	Read Dispatch Control Register (DCR)
✓		RDGSR		19	Read Graphic Status Register (GSR)
		—		20–21	Implementation dependent (impl. dep. #8, 9)
✓		RDSOFTINT[P]		22	Read per-processor Soft Interrupt Register
✓		RDTICK_CMPR[P]		23	Read Tick Compare Register
✓		RDSTICK[P$_{NPT}$]		24	Read System TICK Register
✓		RDSTICK_CMPR[P]		25	Read System TICK Compare Register
		—		26-29	Reserved
✓		RDXASR		30	Read XASR
✓		RDTXAR[P]		31	Read TXAR

For more information about the shaded areas in the table above, see Section A.51, *"Read State Register"*, in JPS1 **Commonality**.

In SPARC64 VIIIfx, if PSTATE.PRIV = 0 and PCR.PRIV = 1, a read of the PCR register by the RDPCR instruction causes a *privileged_action* exception. If PSTATE.PRIV = 0 and PCR.PRIV = 0, a read of the PCR register by the RDPCR instruction does not cause an exception (impl. dep. #250).

When PSTATE.PRIV = 0, a RDTXAR causes a *privileged_opcode* exception.

Exceptions

privileged_opcode (RDDCR, RDSOFTINT, RDTICK_CMPR, RDSTICK, RDSTICK_CMPR, and RDTXAR)

illegal_instruction (RDASR with rs1 = 1 or 7–14;
RDASR with rs1 = 15 and rd ≠ 0;
RDASR with rs1 = 20–21, 26–29;
RDTXAR with TL = 0)

fp_disabled (RDGSR with PSTATE.PEF = 0 or FPRS.FEF = 0)

illegal_action (XAR.v = 1 and
(XAR.simd = 1 or XAR.urs1 ≠ 0 or XAR.urs2 ≠ 0 or XAR.urs3 ≠ 0 or XAR.urd > 1))

privileged_action (RDTICK with PSTATE.PRIV = 0 and TICK.NPT = 1;
RDPIC with PSTATE.PRIV = 0 and PCR.PRIV = 1;
RDSTICK with PSTATE.PRIV = 0 and STICK.NPT = 1;
RDPCR with PSTATE.PRIV = 0 and PCR.PRIV = 1)

A.59 SHUTDOWN (VIS I)

In SPARC64 VIIIfx, `SHUTDOWN` acts as `NOP` in privileged mode (impl. dep. #206).

Exceptions *privileged_opcode*
illegal_action (`XAR.v` = 1)

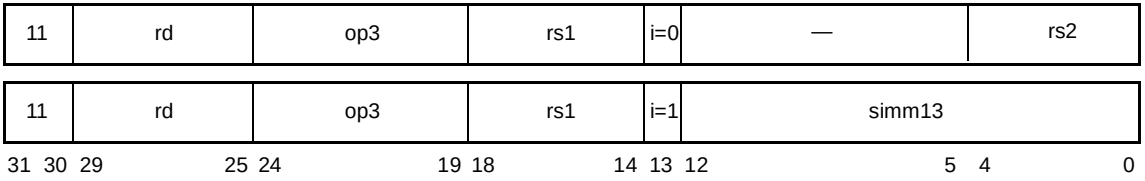
A.61 Store Floating-Point

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	rd	urd	Operation
		STF	10 0100	0–31	—¶	Store Floating-Point Register
✓	✓	STF	10 0100	†	0-7	Store Floating-Point Register
✓	✓	STDF	10 0111	†	0-7	Store Double Floating-Point Register
✓		STQF	10 0110	†	0-7	Store Quad Floating-Point Register
✓		STFSR^{D}	10 0101	0	—	(see A.71.11 in JPS1 **Commonality**)
✓		STXFSR	10 0101	1	—	Store Floating-Point State Register
		–	10 0101	2–31	0	*Reserved*

† Encoded floating-point register value, as describe in Section 5.1.4 of JPS1 **Commonality**.

¶ When XAR.v = 0.

Format (3)



Assembly Language Syntax	
st	$freg_{rd}$, [*address*]
std	$freg_{rd}$, [*address*]
stq	$freg_{rd}$, [*address*]
stx	%fsr, [*address*]

Description First, non-SIMD behavior is described.

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.

The store floating-point state register instruction (STXFSR) waits for any currently executing FPop instructions to complete, and then it writes all 64 bits of the FSR into memory.

STXFSR zeroes FSR.ftt after writing the FSR to memory.

Implementation Note – FSR.ftt should not be zeroed until it is known that the store will not cause a precise trap.

The effective address for these instructions is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

STF causes a *mem_address_not_aligned* exception if the effective memory address is not word 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), then a store floating-point instruction causes an *fp_disabled* exception.

In SPARC64 VIIIfx, a non-SIMD STDF address that is aligned on a 4-byte boundary but not an 8-byte boundary causes an *STDF_mem_address_not_aligned* exception. System software must emulate the instruction (impl.dep. #110(1)).

Because SPARC64 VIIIfx does not implement STQF, an attempt to execute the instruction causes a *illegal_instruction* exception. *fp_disabled* is not detected. System software must emulate STQF (impl.dep. #112(1)).

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.

Programming Note – When the address fields (rs1, rs2) of the single-precision floating-point store instruction STF reference any of the integer registers added by HPC-ACE, the destination register must be a double-precision register. This restriction is a consequence of how rd is decoded when XAR.v = 1 (page 21). A SPARC V9 single-precision register (odd-numbered register) cannot be specified for rd if rs1 or rs2 specifies an HPC-ACE integer register.

SIMD

In SPARC64 VIIIfx, a floating-point store instruction can be executed as a SIMD instruction. A SIMD store instruction simultaneously executes basic and extended stores to the effective address, for either single-precision or double-precision data. See *"Specifying registers for SIMD instructions"* (page 22) for details on specifying the registers.

A single-precision SIMD store instruction stores 2 single-precision data aligned on an 8-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

A double-precision SIMD store instruction stores 2 double-precision data aligned on a 16-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

Note – A double-precision SIMD store that accesses data aligned on a 4-byte boundary but not an 8-byte boundary does not cause a *STDF_mem_address_not_aligned* exception. Unlike a double-precision SIMD load, a double-precision SIMD store aligned on an 8-byte boundary causes a *mem_address_not_aligned* exception.

A SIMD store can only be used to access cacheable address spaces. An attempt to access a noncacheable address space or a nontranslating ASI using a SIMD store causes a *data_access_exception*. The bypass ASIs that can be accessed using a SIMD load instruction are `ASI_PHYS_USE_EC{_LITTTLE}`.

Like non-SIMD store instructions, memory access semantics for SIMD load instructions adhere to TSO. A SIMD store simultaneously executes basic and extended stores; however, the ordering between the basic and extended stores conforms to TSO.

A watchpoint can be detected in both the basic and extended stores of a SIMD store.

For more information regarding SIMD store exception conditions and instruction priority, see Appendix F.5.1, *"Trap Conditions for SIMD Load/Store"* (page 181).

Exceptions

illegal_instruction (`STXFSR` with `rd` = 2–31)
fp_disabled
illegal_action (`STF`, `STDF` with `XAR.v` = 1 and (`XAR.urs1` > 1 or
(`i` = `0` and `XAR.urs2` > 1) or
(`i` = `1` and `XAR.urs2` ≠ 0) or
`XAR.urs3`<2> ≠ 0);
`STF`, `STDF` with `XAR.v` = 1 and `XAR.simd` = 1 and `XAR.urd`<2> ≠ 0;
`STXFSR` with `XAR.v` = 1 and (`XAR.urs1` > 1 or
(`i` = `0` and `XAR.urs2` > 1) or
(`i` = `1` and `XAR.urs2` ≠ 0) or
`XAR.urs3`<2> ≠ 0 or
`XAR.urd` ≠ 0 or
`XAR.simd` = 1))
mem_address_not_aligned
STDF_mem_address_not_aligned (`STDF` and (`XAR.v` = 0 or `XAR.simd` = 0))
VA_watchpoint

fast_data_access_MMU_miss
data_access_exception
fast_data_access_protection
PA_watchpoint
data_access_error

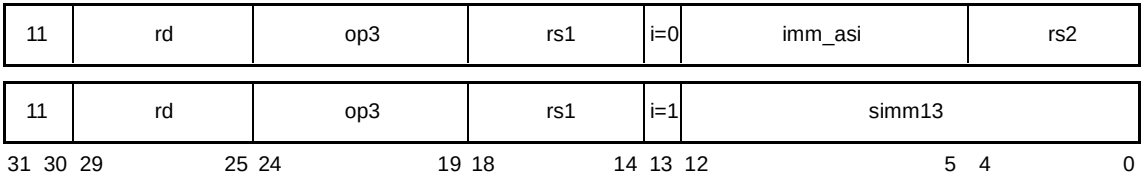
A.62 Store Floating-Point into Alternate Space

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	rd	urd	Operation
		STFA^PASI	11 0100	0–31	—¶	Store Floating-Point Register to Alternate Space
✓	✓	STFA^PASI	11 0100	†	0-7	Store Floating-Point Register to Alternate Space
✓	✓	STDFA^PASI	11 0111	†	0-7	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 Section 5.1.4 of JPS1 **Commonality**.

¶ When XAR.v = 0.

Format (3)



Assembly Language Syntax	
sta	*freg$_{rd}$, [regaddr] imm_asi*
sta	*freg$_{rd}$, [reg_plus_imm]* %asi
stda	*freg$_{rd}$, [regaddr] imm_asi*
stda	*freg$_{rd}$, [reg_plus_imm]* %asi
stqa	*freg$_{rd}$, [regaddr] imm_asi*
stqa	*freg$_{rd}$, [reg_plus_imm]* %asi

Description First, non-SIMD behavior is explained.

The store single floating-point into alternate space instruction (STFA) copies f[rd] into memory.

The store double floating-point into alternate space instruction (STDFA) copies a doubleword from a double floating-point register into a word-aligned doubleword in memory.

The store quad floating-point into alternate space instruction (STQFA) copies the contents of a quad floating-point register into a word-aligned quadword in memory.

Store floating-point into alternate space instructions contain the address space identifier (ASI) to be used for the store in the imm_asi field if i = 0 or in the ASI register if i = 1. The access is privileged if bit 7 of the ASI is 0; otherwise, it is not privileged. The effective address for these instructions is "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign_ext(simm13)" if i = 1.

STFA causes a *mem_address_not_aligned* exception if the effective memory address is not word aligned. If the floating-point unit is not enabled for the source register rd (per FPRS.FEF and PSTATE.PEF), store floating-point into alternate space instructions cause an *fp_disabled* exception.

Implementation Note – STFA and STDFA cause a *privileged_action* exception if PSTATE.PRIV = 0 and bit 7 of the ASI is 0. This check is not performed for STQFA.

Depending on the ASI, memory accesses that are not 8-byte accesses are defined. Refer to other sections in Appendix A.

In SPARC64 VIIIfx, a non-SIMD STDFA address that is aligned on a 4-byte boundary but not an 8-byte boundary causes an *STDF_mem_address_not_aligned* exception. System software must emulate the instruction (impl.dep. #110(2)).

Because SPARC64 VIIIfx does not implement STQFA, an attempt to execute the instruction causes a *illegal_instruction* exception. *fp_disabled* is not detected. System software must emulate STQFA (impl.dep. #112(2)).

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.

Programming Note – When the address fields (rs1, rs2) of the single-precision floating-point store instruction STFA reference any of the integer registers added by HPC-ACE, the destination register must be a double-precision register. This restriction is a consequence of how rd is decoded when XAR.v = 1 (page 21). A SPARC V9 single-precision register (odd-numbered register) cannot be specified for rd if rs1 or rs2 specifies a HPC-ACE integer register.

SIMD Refer to the SIMD subsection in Section A.61, *"Store Floating-Point"*.

Exceptions

fp_disabled

illegal_action (STFA, STDFA with XAR.v = 1 and (XAR.urs1 > 1 or
(i = 0 and XAR.urs2 > 1) or
(i = 1 and XAR.urs2 ≠ 0) or
XAR.urs3<2> ≠ 0);
STFA, STDFA with XAR.v = 1 and XAR.simd = 1 and XAR.urd<2> ≠ 0)

mem_address_not_aligned

STDF_mem_address_not_aligned (STDFA and (XAR.v = 0 or XAR.simd = 0))

privileged_action

VA_watchpoint

fast_data_access_MMU_miss

data_access_exception

fast_data_access_protection

PA_watchpoint

data_access_error

A.68 Trap on Integer Condition Codes (Tcc)

The `Tcc` instruction does not depend on the value of `XAR` and behaves as defined in JPS1 **Commonality**. An *illegal_action* exception does not occur.

When an exception occurs and *trap_instruction* is signalled, the contents of the `XAR` immediately prior to the execution of the `Tcc` instruction are copied to the `TXAR`. When an exception does not occur, if `XAR.f_v` = 1 then the contents of `XAR.f_*` are set to 0, and if `XAR.f_v` = 0 and `XAR.s_v` = 1 then the contents of `XAR.s_*` are set to 0. See *"XAR operation"* (page 31) for details.

Programming Note – Because `Tcc` always ignores the value of `XAR`, the `Tcc` instruction can be inserted at any location. This is useful for implementing breakpoints for a debugger.

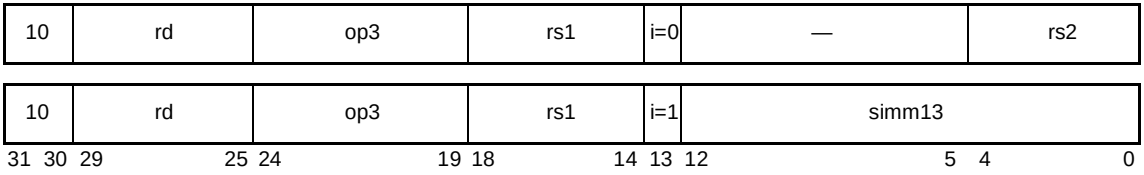
Exceptions

illegal_instruction (`cc1` ▯ `cc0` = 01_2 or 11_2, or reserved fields nonzero)
trap_instruction

A.69 Write Privileged Register

HPC-ACE Ext.				
Regs.	**SIMD**	**Opcode**	**op3**	**Operation**
✓		WRPR[P]	11 0010	Write Privileged Register

Format (3)



rd	Privileged Register
0	TPC
1	TNPC
2	TSTATE
3	TT
4	TICK
5	TBA
6	PSTATE
7	TL
8	PIL
9	CWP
10	CANSAVE
11	CANRESTORE
12	CLEANWIN
13	OTHERWIN
14	WSTATE
15–31	*Reserved*

Assembly Language Syntax	
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %tpc
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %tnpc
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %tstate
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %tt
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %tick
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %tba
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %pstate
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %tl
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %pil
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %cwp
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %cansave
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %canrestore
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %cleanwin
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %otherwin
wrpr	*reg*$_{rs1}$, *reg_or_imm*, %wstate

Description

This instruction stores the value "r[rs1] **xor** r[rs2]" if i = 0, or "r[rs1] **xor** sign_ext(simm13)" if i = 1 to the writable fields of the specified privileged state register. **Note:** The operation is 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, and 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, TT, and TSTATE for any trap level; however, take care that traps do not occur while the TL register is modified.

The WRPR instruction is a *non*-delayed-write instruction. The instruction immediately following the WRPR observes any changes made to processor state made by the WRPR.

WRPR instructions with rd in the range 15–31 are reserved for future versions of the architecture; executing a WRPR instruction with rd in that range causes an *illegal_instruction* exception.

A WRPR to PSTATE that specifies a reserved combination of AG, IG, and MG bits causes an *illegal_instruction* exception; however, this exception has a lower priority than a *llegal_action* exception.

Exceptions

privileged_opcode
illegal_instruction ((rd = 15–31) or ((rd ≤ 3) and (TL = 0));
(rd = 6 and reserved combination of AG, IG, and MG))
illegal_action (XAR.v = 1 and (XAR.simd = 1 or
XAR.urs1 > 1 or
(i = 0 and XAR.urs2 > 1) or
(i = 1 and XAR.urs2 ≠ 0) or
XAR.urs3 ≠ 0 or
XAR.urd ≠ 0))

A.70 Write State Register

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	rd	Operation
✓		WRY[D]	11 0000	0	Write Y register; deprecated (see A.71.18 of JPS1 **Commonality**)
		—	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	*Reserved*
		—	11 0000	15	Software-initiated reset (see A.60 of JPS1 **Commonality**)
		WRASR	11 0000	16–31	Write non-SPARC V9 ASRs
✓		WRPCR[P PCR]		16	Write Performance Control Registers (PCR)
✓		WRPIC[P PIC]		17	Write Performance Instrumentation Counters (PIC)
✓		WRDCR[P]		18	Write Dispatch Control Register (DCR)
✓		WRGSR		19	Write Graphic Status Register (GSR)
✓		WRSOFTINT_SET[P]		20	Set bits of per-processor Soft Interrupt Register
✓		WRSOFTINT_CLR[P]		21	Clear bits of per-processor Soft Interrupt Register
✓		WRSOFTINT[P]		22	Write per-processor Soft Interrupt Register
✓		WRTICK_CMPR[P]		23	Write Tick Compare Register
✓		WRSTICK[P]		24	Write System TICK Register
✓		WRSTICK_CMPR[P]		25	Write System TICK Compare Register
		—		26-28	Reserved
✓		WRXAR		29	Write XAR
✓		WRXASR		30	Write XASR
✓		WRTXAR[P]		31	Write TXAR

For more information about the shaded areas in the table above, see Section A.70, *"Write State Register"*, in JPS1 **Commonality**.

In SPARC64 VIIIfx, if PSTATE.PRIV = 0 and PCR.PRIV = 1, a read of the PCR register by the WRPCR instruction causes a *privileged_action* exception. If PSTATE.PRIV = 0 and PCR.PRIV = 0, a read of the PCR register by the WRPCR instruction does not cause an exception. (impl. dep. #250).

A WRXAR or WRTXAR that attempts to write a nonzero value to a *reserved* field in the XAR causes an *illegal_instruction* exception. However, if both *illegal_instruction* and *illegal_action* exceptions are generated, the *illegal_action* exception takes priority and is signalled.

Note – Executing a WRTXAR instruction while TL = 0 causes an *illegal_instruction* exception, regardless of the value of the XAR.

When WRXAR writes XAR.v = 0 or WRTXAR writes TXAR.v = 0, the value of the corresopnding fields are undefined, regardless of the values written to them. That is,

- When XAR.f_v = 0 is written, the values of XAR.f_urs1, XAR.f_urs2, XAR.f_urs3, XAR.f_urd, and XAR.f_simd are undefined, regardless of the values written to them.
- When XAR.s_v = 0 is written, the values of XAR.s_urs1, XAR.s_urs2, XAR.s_urs3, XAR.s_urd, and XAR.s_simd are undefined, regardless of the values written to them.
- When TXAR.f_v = 0 is written, the values of TXAR.f_urs1, TXAR.f_urs2, TXAR.f_urs3, TXAR.f_urd, and TXAR.f_simd are undefined, regardless of the values written to them.
- When TXAR.s_v = 0 is written, the values of TXAR.s_urs1, TXAR.s_urs2, TXAR.s_urs3, TXAR.s_urd, and TXAR.s_simd are undefined, regardless of the values written to them.

Implementation Note – When XAR.v = 0 is written, an implemention can choose to set the corresponding fields to 0.

Exceptions

software_initiated_reset (rd = 15, rs1 = 0, and i = 1 only)

privileged_opcode (WRDCR, WRSOFTINT_SET, WRSOFTINT_CLR, WRSOFTINT, WRTICK_CMPR, WRSTICK, WRSTICK_CMPR, and WRTXAR)

illegal_instruction (WRASR with rd = 1, 4, 5, 7–14, 26-28;
WRASR with rd = 15 and rs1 ≠ 0 or i ≠ 1,
WRTXAR with TL = 0;
WRXAR with reserved fields to nonzero)

fp_disabled (WRGSR with PSTATE.PEF = 0 or FPRS.FEF = 0)

illegal_action (XAR.v = 1 and (XAR.simd = 1 or
XAR.urs1 > 1 or
(i = 0 and XAR.urs2 > 1) or
(i = 1 and XAR.urs2 ≠ 0) or
XAR.urs3 ≠ 0 or
XAR.urd ≠ 0))

privileged_action (WRPIC with PSTATE.PRIV = 0 and PCR.PRIV = 1,

WRPCR with PSTATE.PRIV = 0 and PCR.PRIV = 1;
WRPCR to modify PCR.PRIV
with PSTATE.PRIV = 0 and PCR.PRIV = 0)

A.71 Deprecated Instructions

The deprecated instructions in Appendix A.71 of JPS1 **Commonality** are provided only for compatibility with previous versions of the architecture. They should not be used in new software.

A.71.10 Store Barrier

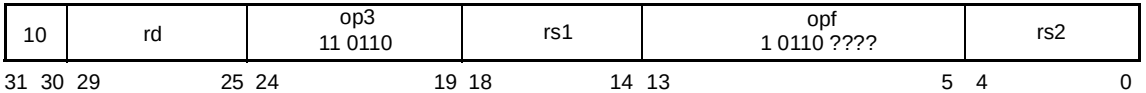
In SPARC64 VIIIfx, `STBAR` behaves as `NOP` since the hardware memory model always enforces the semantics of this instruction for all memory accesses.

Exceptions *illegal_action* (`XAR.v` = 1)

A.72 Floating-Point Conditional Compare to Register

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	opf	Operation	Register Contents Test
✓	✓	FCMPEQd	11 0110	1 0110 0000	Compare Double Equal	f[rs1] = f[rs2]
✓	✓	FCMPEQEd	11 0110	1 0110 0010	Compare Double Equal, Exception if Unordered	f[rs1] = f[rs2]
✓	✓	FCMPLEEd	11 0110	1 0110 0100	Compare Double Less Than or Equal, Exception if Unordered	f[rs1] ≤ f[rs2]
✓	✓	FCMPLTEd	11 0110	1 0110 0110	Compare Double Less Than, Exception if Unordered	f[rs1] < f[rs2]
✓	✓	FCMPNEd	11 0110	1 0110 1000	Compare Double Not Equal	f[rs1] ≠ f[rs2]
✓	✓	FCMPNEEd	11 0110	1 0110 1010	Compare Double Not Equal, Exception if Unordered	f[rs1] ≠ f[rs2]
✓	✓	FCMPGTEd	11 0110	1 0110 1100	Compare Double Greater Than, Exception if Unordered	f[rs1] > f[rs2]
✓	✓	FCMPGEEd	11 0110	1 0110 1110	Compare Double Greater Than or Equal, Exception if Unordered	f[rs1] ≥ f[rs2]
✓	✓	FCMPEQs	11 0110	1 0110 0001	Compare Single Equal	f[rs1] = f[rs2]
✓	✓	FCMPEQEs	11 0110	1 0110 0011	Compare Single Equal, Exception if Unordered	f[rs1] = f[rs2]
✓	✓	FCMPLEEs	11 0110	1 0110 0101	Compare Single Less Than or Equal, Exception if Unordered	f[rs1] ≤ f[rs2]
✓	✓	FCMPLTEs	11 0110	1 0110 0111	Compare Single Less Than, Exception if Unordered	f[rs1] < f[rs2]
✓	✓	FCMPNEs	11 0110	1 0110 1001	Compare Single Not Equal	f[rs1] ≠ f[rs2]
✓	✓	FCMPNEEs	11 0110	1 0110 1011	Compare Single Not Equal, Exception if Unordered	f[rs1] ≠ f[rs2]
✓	✓	FCMPGTEs	11 0110	1 0110 1101	Compare Single Greater Than, Exception if Unordered	f[rs1] > f[rs2]
✓	✓	FCMPGEEs	11 0110	1 0110 1111	Compare Single Greater Than or Equal, Exception if Unordered	f[rs1] ≥ f[rs2]

Format (3)



Assembly Language Syntax	
`fcmpgte{s,d}`	*fregrs1*, *fregrs2*, *fregrd*
`fcmplte{s,d}`	*fregrs1*, *fregrs2*, *fregrd*
`fcmpeqe{s,d}`	*fregrs1*, *fregrs2*, *fregrd*
`fcmpnee{s,d}`	*fregrs1*, *fregrs2*, *fregrd*
`fcmpgee{s,d}`	*fregrs1*, *fregrs2*, *fregrd*
`fcmplee{s,d}`	*fregrs1*, *fregrs2*, *fregrd*
`fcmpeq{s,d}`	*fregrs1*, *fregrs2*, *fregrd*
`fcmpne{s,d}`	*fregrs1*, *fregrs2*, *fregrd*

Description

The above instructions compare the values in the floating-point registers specified by `rs1` and `rs2`. If the condition specified by the instruction is met, then the floating-point register specified by `rd` is written entirely with ones. If the condition is not met, then `rd` is written entirely with zeroes.

When the source operands are SNaN or QNaN, generated exceptions and instruction results are described below. The “exception” column indicates the value set in `FSR.cexc` when an *fp_exception_ieee_754* exception occurs. The “rd” column indicates the value stored in `rd` when no exception occurs.

	SNan		QNan	
Instructions	**Exception**	**rd**	**Exception**	**rd**
`FCMPGTE{s,d}`, `FCMPLTE{s,d}`, `FCMPGEE{s,d}`, `FCMPLEE{s,d}`	NV	all0	NV	all0
`FCMPEQE{s,d}`	NV	all0	NV	all0
`FCMPNEE{s,d}`	NV	all1	NV	all1
`FCMPEQ{s,d}`	NV	all0	—	all0
`FCMPNE{s,d}`	NV	all1	—	all1

Programming Note – These instruction can be efficiently used with `FSELMOV{s,d}`, `STFR`, `STDFR`, and the VIS logical instructions.

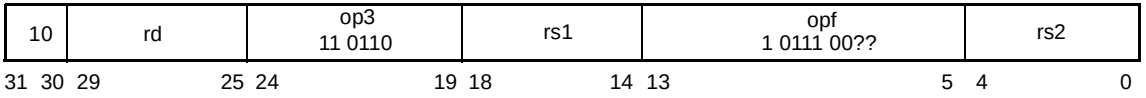
Exceptions

fp_disabled
illegal_action (`XAR.v` = 1 and `XAR.urs3` ≠ 0;
`XAR.v` = 1 and `XAR.simd` = 1 and
(`XAR.urs1`<2> ≠ 0 or `XAR.urs2`<2> ≠ 0 or `XAR.urd`<2> ≠ 0))
fp_exception_ieee_754 (`NV` if unordered)

A.73 Floating-Point Minimum and Maximum

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	opf	Operation
✓	✓	FMAXd	11 0110	1 0111 0000	Select Maximum Double
✓	✓	FMAXs	11 0110	1 0111 0001	Select Maximum Single
✓	✓	FMINd	11 0110	1 0111 0010	Select Minimum Double
✓	✓	FMINs	11 0110	1 0111 0011	Select Minimum Single

Format (3)



Assembly Language Syntax	
fmax{s,d}	*fregrs1*, *fregrs2*, *fregrd*
fmin{s,d}	*fregrs1*, *fregrs2*, *fregrd*

Description

FMAX{s, d} compares the values in the floating-point registers specified by rs1 and rs2. If f[rs1] > f[rs2], then rs1 is written to the floating-point register specified by rd. Otherwise, rs2 is written to rd.

FMIN{s, d} compares the values in the floating-point registers specified by rs1 and rs2. If f[rs1] < f[rs2], then rs1 is written to the floating-point register specified by rd. Otherwise, rs2 is written to rd.

FMIN and FMAX ignore the sign of a zero value. When the value of f[rs1] is +0 or -0 and the value of f[rs2] is +0, -0, the value of f[rs2] is written to the destination register.

When one of the source operand is QNaN and the other operand is neither QNaN nor SNaN, the value of the source that is not QNaN is written to the destination register. Unlike other instructions, FMIN and FMAX do not propagate QNaN. When one of the source operand is

SNaN, or both operands are QNaN, the value defined by TABLE B-1 of JPS1 **Commonality** is stored in `rd`. Furthermore, when one of the source operand is QNaN or SNaN, SPARC64 VIIIfx detects an *fp_exception_ieee_754* exception.

TABLE A-9 Operands and the result of `FMIN` and `FMAX`

rs1	rs2	rd	Exception
not NaN	not NaN	min(`rs1`, `rs2`), or max(`rs1`, `rs2`)	—
not NaN	QNaN	`rs1`	NV
not NaN	SNaN	QSNaN2	NV
QNaN	not NaN	`rs2`	NV
QNaN	QNaN	`rs2` (QNaN)	NV
QNaN	SNaN	QSNaN2	NV
SNaN	not NaN	QSNaN1	NV
SNaN	QNaN	QSNaN1	NV
SNaN	SNaN	QSNaN2	NV

Exceptions

fp_disabled

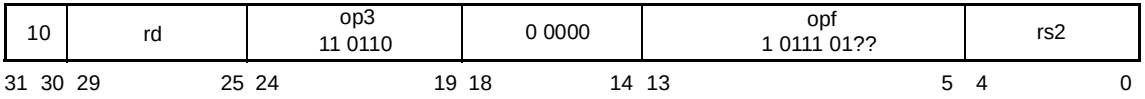
illegal_action (`XAR.v` = 1 and `XAR.urs3` ≠ 0;
`XAR.v` = 1 and `XAR.simd` = 1 and
(`XAR.urs1`<2> ≠ 0 or `XAR.urs2`<2> ≠ 0 or `XAR.urd`<2> ≠ 0))

fp_exception_ieee_754 (`NV` if unordered)

A.74 Floating-Point Reciprocal Approximation

HPC-ACE Ext. Regs.	SIMD	Opcode	op3	opf	Operation
✓	✓	FRCPAd	11 0110	1 0111 0100	Reciprocal Approximation Double
✓	✓	FRCPAs	11 0110	1 0111 0101	Reciprocal Approximation Single
✓	✓	FRSQRTAd	11 0110	1 0111 0110	Reciprocal Approximation of Square Root, Double
✓	✓	FRSQRTAs	11 0110	1 0111 0111	Reciprocal Approximation of Square Root, Single

Format (3)



Assembly Language Syntax	
`frcpa{s,d}`	*freg*$_{rs2}$, *freg*$_{rd}$
`frsqrta{s,d}`	*freg*$_{rs2}$, *freg*$_{rd}$

Description `FRCPA{s,d}` calculates the reciprocal approximation of the value in the floating-point register specified by `rs2` and stores the result in the floating-point register specified by `rd`. Although the result is approximate, the calculation ignores `FSR.RD`. The resulting rounding error is less than 1/256. In other words,

$$\left|\frac{frcpa(x) - 1/x}{1/x}\right| < \frac{1}{256}$$

Results and exception conditions for FRCPA{s,d} are shown in TABLE A-10. The upper row in each entry indicates the type(s) of exception if an exception is signalled, and the lower row in each entry indicates the result when an exception is not signalled. For more information on the causes of a *fp_exception_ieee_754* exception, refer to Appendix B in this document and in JPS1 **Commonality**.

TABLE A-10 FRCPA{s,d} Results

	Exceptions and Results	
op2	**FSR.NS = 0**	**FSR.NS = 1**
+∞	— 0	— 0
+N ($N \geq 2^{126}$ for single, $N \geq 2^{1022}$ for double)	UF approximation of +1/N (denormalized)[1]	UF, NX +0
+N (+Nmin ≤ $N < 2^{126}$ for single, +Nmin ≤ $N < 2^{1022}$ for double)	— approximation of +1/N	— approximation of +1/N
+D	*unfinished_FPop* —	DZ +∞
+0	DZ +∞	DZ +∞
-0	DZ −∞	DZ −∞
-D	*unfinished_FPop* —	DZ −∞
-N (+Nmin ≤ $N < 2^{126}$ for single, +Nmin ≤ $N < 2^{1022}$ for double)	— approximation of -1/N	— approximation of -1/N
-N ($N \geq 2^{126}$ for single, $N \geq 2^{1022}$ for double)	UF approximation of -1/N (denormalized)[1]	UF, NX -0
−∞	— -0	— -0
SNaN	NV QSNaN2	NV QSNaN2
QNaN	— op2	— op2

1.When the result is denormal, the rounding error may be larger than 1/256.

N	Positive normalized number (not zero, NaN, infinity)
D	Positive denormalized number.
Nmin	Minimum value when rounding a normalized number.
dNaN	Sign of QNaN is 0 and all bits of the exponent and significand are 1.
QSNaN2	See TABLE B-1 in JPS1 **Commonality**.

FRSQRTA{s, d} calculates the reciprocal approximation of the square root of the value in the floating-point register specified by rs2 and stores the result in the floating-point register specified by rd. Although the result is approximate, the calculation ignores FSR.RD. The resulting rounding error is less than 1/256. In other words,

$$\left|\frac{frsqrta(x) - 1/(\sqrt{x})}{1/(\sqrt{x})}\right| < \frac{1}{256}$$

Results and exception conditions for FRSQRTA{s, d} are shown in TABLE A-11. The upper row in each entry indicates the type(s) of exception if an exception is signalled, and the lower row in each entry indicates the result when an exception is not signalled. For more information on the causes of a *fp_exception_ieee_754* exception, refer to Appendix B in this document and in JPS1 **Commonality**.

TABLE A-11 FRSQRTA{s,d} Results

	Exceptions and Results	
op2	**FSR.NS = 0**	**FSR.NS = 1**
+∞	— 0	— 0
+N	— $+1/(\sqrt{N})$	— $+1/(\sqrt{N})$
+D	*unfinished_FPop* —	DZ +0
+0	DZ +0	DZ +0
-0	DZ +0	DZ +0
-D	NV dNaN	NV dNaN
-N	NV dNaN	NV dNaN
−∞	NV dNaN	NV dNaN

TABLE A-11 FRSQRTA{s,d} Results

	Exceptions and Results	
op2	**FSR.NS = 0**	**FSR.NS = 1**
SNaN	NV QSNaN2	NV QSNaN2
QNaN	— op2	— op2

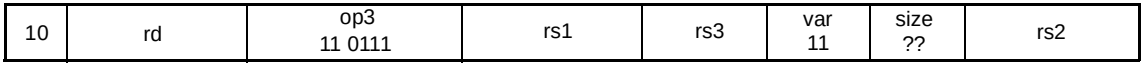
Exceptions

illegal_instruction (instruction<18:14> ≠ 0)
fp_disabled
illegal_action (XAR.v = 1 and (XAR.urs1 ≠ 0 or XAR.urs3 ≠ 0);
XAR.v = 1 and XAR.simd = 1 and
(XAR.urs2<2> ≠ 0 or XAR.urd<2> ≠ 0))
fp_exception_ieee_754 (NV, DZ, UF, NX for FRCPA{s, d};
NV, DZ for FRSQRTA{s, d})
fp_exception_other (ftt = *unfinished_FPop*)

A.75 Move Selected Floating-Point Register on Floating-Point Register's Condition

HPC-ACE Ext.						
Regs.	**SIMD**	**Opcode**	**op3**	**var**	**size**	**Operation**
✓	✓	FSELMOVd	11 0111	11	00	Select and Move Double
✓	✓	FSELMOVs	11 0111	11	11	Select and Move Single

Format (5)



Assembly Language Syntax	
`fselmov{s,d}`	*fregrs1*, *fregrs2*, *fregrs3*, *fregrd*

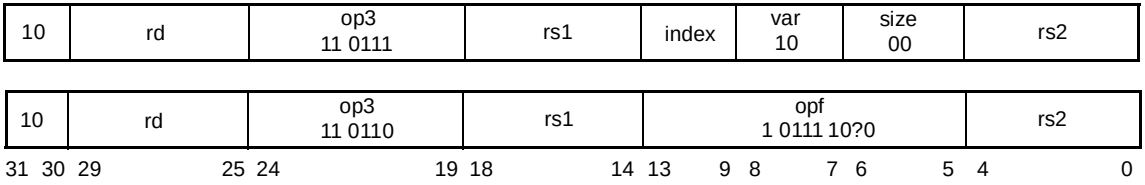
Description `FSELMOV{s, d}` selects `rs1` or `rs2` according to the most significant bit of the floating-point register specified by `rs3`. The value of the selected register is then stored in the floating-point register specified by `rd`. If bit 63 of the register specified by `rs3` is 1, then `rs1` is selected. If the bit is 0, then `rs2` is selected.

Exceptions *fp_disabled*
illegal_action (`XAR.v` = 1 and `XAR.simd` = 1 and
(`XAR.urs1`<2> ≠ 0 or `XAR.urs2`<2> ≠ 0 or
`XAR.urs3`<2> ≠ 0 or `XAR.urd`<2> ≠ 0))

A.76 Floating-Point Trigonometric Functions

HPC-ACE Ext.					
Regs.	**SIMD**	**Opcode**	**op3**	**opf**	**Operation**
✓	✓	FTRIMADDd	11 0111	—	Trigonometric Multiply-Add Double
✓	✓	FTRISMULd	11 0110	1 0111 1010	Calculate starting value for FTRIMADDd
✓	✓	FTRISSELd	11 0110	1 0111 1000	Select coefficient for final calculation in Taylor series approximation

Format (5 and 3)



Assembly Language Syntax	
ftrimaddd	*fregrs1, fregrs2, index, fregrd*
ftrismuld	*fregrs1, fregrs2, fregrd*
ftrisseld	*fregrs1, fregrs2, fregrd*

Operation	Implementation
FTRIMADDd	rd ← rs1 × abs(rs2) + T[rs2<63>][index]
FTRISMULd	rd ← (rs2<0> << 63) ^ (rs1 × rs1)
FTRISSELd	rd ← (rs2<1> << 63) ^ (rs2<0> ? 1.0 : rs1)

Description These instructions accelerate the calculation of the Taylor series approximation of the sine function; that is, sin(x) can be calculated for any arbitrary value using the FTRIMADDd, FTRISMULd, and FTRISSELd instructions. All three instructions are defined as double-precision instructions only. FTRIMADDd calculates series terms for either sin(x) or cos(x), where the argument is adjusted to be in the range $-\pi/4 < x \leq \pi/4$. These series terms are used

$$\sin x \cong x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \frac{1}{9!}x^9 - \frac{1}{11!}x^{11} + \frac{1}{13!}x^{13} - \frac{1}{15!}x^{15}$$

$$= x\left(1 - \frac{1}{3!}x^2 + \frac{1}{5!}x^4 - \frac{1}{7!}x^6 + \frac{1}{9!}x^8 - \frac{1}{11!}x^{10} + \frac{1}{13!}x^{12} - \frac{1}{15!}x^{14}\right)$$

$$= x \cdot \underbrace{\left(\left(\left(\left(\left(\left(\left(\left(0 \cdot x^2 - \frac{1}{15!}\right)x^2 + \frac{1}{13!}\right)x^2 - \frac{1}{11!}\right)x^2 + \frac{1}{9!}\right)x^2 - \frac{1}{7!}\right)x^2 + \frac{1}{5!}\right)x^2 - \frac{1}{3!}\right)x^2 + 1\right)}_{\texttt{FTRIMADDd}}$$

$$\cos x \cong 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \frac{1}{8!}x^8 - \frac{1}{10!}x^{10} + \frac{1}{12!}x^{12} - \frac{1}{14!}x^{14}$$

$$= 1 \cdot \underbrace{\left(\left(\left(\left(\left(\left(\left(\left(0 \cdot x^2 - \frac{1}{14!}\right)x^2 + \frac{1}{12!}\right)x^2 - \frac{1}{10!}\right)x^2 + \frac{1}{8!}\right)x^2 - \frac{1}{6!}\right)x^2 + \frac{1}{4!}\right)x^2 - \frac{1}{2!}\right)x^2 + 1\right)}_{\texttt{FTRIMADDd}}$$

FIGURE A-1 Supporting Operations Performed by SPARC64 VIIIfx Trignometric Functions

to perform the supporting operations shown in FIGURE A-1. See the example at the end of this section for a full description of how sin(x) can be calculated for an arbitrary “x” using these support operations.

`FTRIMADDd` multiplies the values in the double-precision registers specified by `rs1` and `rs2` and adds the product to the double-precision number obtained from a table built into the functional unit. This double-precision number is specified by the `index` field. The result is stored in the double-precision register specified by `rd`. `FTRIMADDd` is used to calculate series terms in the Taylor series of sin(x) or cos(x), where $-\pi/4 < x \leq \pi/4$.

`FTRISMULd` squares the value in the double-precision register specified by `rs1`. The sign of the squared value is selected according to bit 0 of the double-precision register specified by `rs2`. The result is written to the double-precision register specified by `rd`. `FTRISMULd` is used to calculate the starting value of `FTRIMADDd`.

`FTRISSELd` checks bit 0 of the double-precision register specified by `rs2`. Based on this bit, either the double-precision register specified by `rs1` or the value 1.0 is selected. Bit 1 of `rs2` indicates the sign; the exclusive OR of this bit and the selected value is written to the double-precision register specified by `rd`. `FTRISSELd` is used to select the coefficient for calculating the last step in the Taylor series approximation.

To calculate the series terms of sin(x) and cos(x), the initial source operands of `FTRIMADDd` are zero for `f[rs1]` and x^2 for `f[rs2]`, where $-\pi/4 < x \leq \pi/4$. `FTRIMADDd` is executed 8 times; this calculates the sum of 8 series terms, which gives the resulting number sufficient precision for a double-precision floating-point number. As show in TABLE A-5, the coefficients of the series terms are different for sin(x) and cos(x). `FTRIMADDd` uses the sign of `rs2` to determine which set of coefficients to use.

- When `f[rs2]`<63> = 0, the coefficient table for sin(R) is used.
- When `f[rs2]`<63> = 1, the coefficient table of cos(R) is used.

The expected usage for `FTRIMADDd` is shown in the example below. Coefficients are chosen to minimize the loss of precision; these differ slightly from the exact mathematical values. TABLE A-12 and TABLE A-13 show the coefficient tables for `FTRIMADDd`.

TABLE A-12 Coefficient Table for sin(x) (`f[rs2]`<63> = 0)

Coefficient used for the operation			Exact value of the coefficient
Index	**Hexadecimal representation**	**Decimal representation**	
0	$3ff0\ 0000\ 0000\ 0000_{16}$	1.0	= 1/1!
1	$bfc5\ 5555\ 5555\ 5543_{16}$	-0.1666666666666661	> -1/3!
2	$3f81\ 1111\ 1110\ f30c_{16}$	0.8333333333320002e-02	< 1/5!
3	$bf2a\ 01a0\ 19b9\ 2fc6_{16}$	-0.1984126982840213e-03	> -1/7!
4	$3ec7\ 1de3\ 51f3\ d22b_{16}$	0.2755731329901505e-05	< 1/9!
5	$be5a\ e5e2\ b60f\ 7b91_{16}$	-0.2505070584637887e-07	> -1/11!
6	$3de5\ d840\ 8868\ 552f_{16}$	0.1589413637195215e-09	< 1/13!
7	$0000\ 0000\ 0000\ 0000_{16}$	0	> -1/15!

TABLE A-13 Coefficient Table for cos(x) (`f[rs2]`<63> = 1)

Coefficient used for the operation			Exact value of the coefficient
Index	**Hexadecimal representation**	**Decimal representation**	
0	$3ff0\ 0000\ 0000\ 0000_{16}$	1.0	= 1/0!
1	$bfe0\ 0000\ 0000\ 0000_{16}$	-0.5000000000000000	= -1/2!
2	$3fa5\ 5555\ 5555\ 5536_{16}$	0.4166666666666645e-01	< 1/4!
3	$bf56\ c16c\ 16c1\ 3a0b_{16}$	-0.1388888888886111e-02	> -1/6!
4	$3efa\ 01a0\ 19b1\ e8d8_{16}$	0.2480158728388683e-04	< 1/8!
5	$be92\ 7e4f\ 7282\ f468_{16}$	-0.2755731309913950e-06	> -1/10!
6	$3e21\ ee96\ d264\ 1b13_{16}$	0.2087558253975872e-08	< 1/12!
7	$bda8\ f763\ 80fb\ b401_{16}$	-0.1135338700720054e-10	> -1/14!

The initial value in `f[rs2]` of `FTRIMADDd` is calculated using `FTRISMULd`, which is executed with `f[rs1]` set to x, where $-\pi/4 < x \leq \pi/4$, and `f[rs2]` set to Q, as defined in FIGURE A-2. `FTRISMULd` returns x^2 as the result, where the sign bit specifies which set of coefficients to use to calculate the series terms. Q is an integer, not a floating-point number. `f[rs2]`<63:1> are not used. An exception is not detected if `f[rs2]` is NaN.

The final step in the calculation of the Taylor series is the multiplication of the `FTRIMADDd` result and the coefficient selected by `FTRISSELd`. This coefficient is selected by executing `FTRISSELd` with `f[rs1]` set to x, where $-\pi/4 < x \leq \pi/4$, and `f[rs2]` set to Q, as defined in FIGURE A-2; either x or 1.0 is selected, and the appropriate sign is affixed to the result. Q is an integer, not a floating-point number. `f[rs2]`<63:2> are not used. An exception is not detected if `f[rs2]` is NaN.

$$q\text{: } (2q-1)\cdot\frac{\pi}{4} < x \leq (2q+1)\cdot\frac{\pi}{4}$$

Q: q **mod** 4

$$R\text{: } x - q\cdot\frac{\pi}{2} \quad \left(-\frac{\pi}{4} < R \leq \frac{\pi}{4}\right)$$

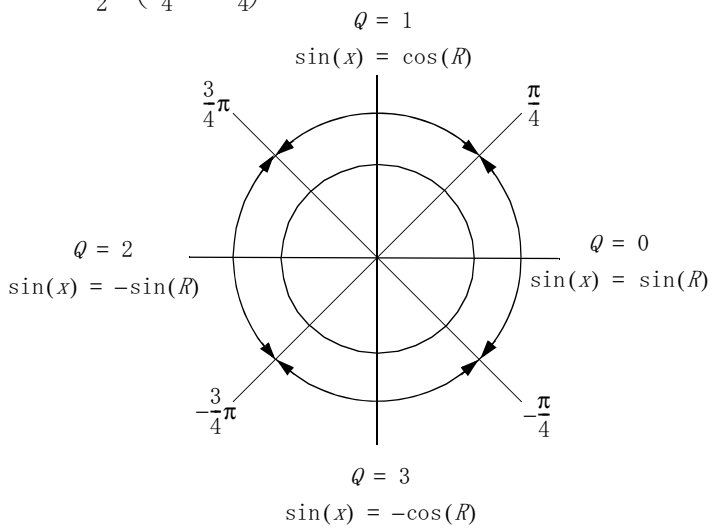


FIGURE A-2 Relationships for Calculating sin(x)

Example: calculating sin(x)

```
/*
 * Input value: x
 * q: where (2q-1)* π /4 < x <= (2q+1)* π /4
 * Q: q%4
 * R: x - q *  π /2
 */

ftrismuld      R, Q, M
```

```
ftrisseld      R, Q, N

/*
 *  M ← R2[63]=table_type, R2[62:0]=R2
 *          Because R2 is always positive, the sign bit (bit <63>) is always 0.
 *          This sign bit is used to indicate the table_type for ftrimaddd.
 *  N ← coefficient used in the final step; the value is (1.0 or R) * sign.
 *  S ← 0
 */

ftrimaddd      S, M, 7, S
ftrimaddd      S, M, 6, S
ftrimaddd      S, M, 5, S
ftrimaddd      S, M, 4, S
ftrimaddd      S, M, 3, S
ftrimaddd      S, M, 2, S
ftrimaddd      S, M, 1, S
ftrimaddd      S, M, 0, S
fmuld          S, N, S

/*
 *  S ← Result
 */
```

Exceptions

illegal_instruction (FTRIMADDd with index > 7)
fp_disabled
illegal_action (XAR.v = 1 and XAR.urs3 ≠ 0;
XAR.v = 1 and XAR.simd = 1 and
(XAR.urs1<2> ≠ 0 or XAR.urs2<2> ≠ 0 or XAR.urd<2> ≠ 0))
fp_exception_ieee_754 (FTRIMADDd with NV, NX, OF, UF;
FTRISMULd with NX, OF, UF;
FTRISMULd with NV (rs1 only))
fp_exception_other (FTRIMADDd, FTRISMULd with ftt = *unfinished_FPop*)

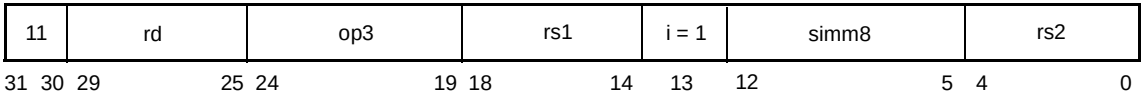
A.77 Store Floating-Point Register on Register Condition

HPC-ACE Ext.						
Regs.	**SIMD**	**Opcode**	**op3**	**rd**	**urd**	**Operation**
		STFR	10 1100	0–31	¶	Store Floating-Point Register on Register Condition
✓	✓	STFR	10 1100	†	0-7	Store Floating-Point Register on Register Condition
✓	✓	STDFR	10 1111	†	0-7	Store Double Floating-Point Register on Register Condition

† Encoded floating-point register value, as described in *Floating-Point Register Number Encoding* in Section 5.1.4 of JPS1 **Commonality**.

¶ When XAR.v = 0.

Format (3)



Assembly Language Syntax	
stfr	$freg_{rd}$, $freg_{rs2}$, [*address*]
stdfr	$freg_{rd}$, $freg_{rs2}$, [*address*]

Description

When the most significant bit of f[rs2] is 1, STFR writes the contents of the single-precision floating-point register f[rd] to the write address, which must be aligned on a 4-byte boundary.

When the most significant bit of f[rs2] is 1, STDFR writes the contents of the double-precision floating-point register f[rd] to the write address, which must be aligned on an 8-byte boundary.

The write address is calculated as "r[rs1] + sign_ext(simm8 << 2)".

STFR causes a *mem_address_not_aligned* exception when the access address is not aligned on a 4-byte boundary.

STDFR causes a *mem_address_not_aligned* exception when the access address is not aligned on an 8-byte boundary.

A non-SIMD STDFR that is aligned on a 4-byte boundary but not an 8-byte boundary causes a *STDF_mem_address_not_aligned* exception.

STFR and STDFR cause *fp_disabled* exceptions when the floating-point unit cannot be used, which depends on the setting of FPRS.FEF and PSTATE.PEF.

When a watchpoint is detected for a STFR or STDFR instruction, an exception is generated regardless of whether the store is actually performed.

Programming Note – When the address fields (rs1, rs2) of the single-precision floating-point store instruction STFR reference any of the integer registers added by HPC-ACE, the destination register must be a double-precision register. This restriction is a consequence of how rd is decoded when XAR.v = 1 (page 21). A SPARC V9 single-precision register (odd-numbered register) cannot be specified for rd if rs1 or rs2 specifies a HPC-ACE integer register.

SIMD

In SPARC64 VIIIfx, STFR and STDFR can be executed as SIMD instruction. A SIMD STFR or SIMD STDFR instruction simultaneously executes basic and extended stores to the effective address, for either single-precision or double-precision data. See *"Specifying registers for SIMD instructions"* (page 22) for details on specifying the registers.

A SIMD STFR instruction stores two single-precision data aligned on an 8-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

A SIMD STDFR instruction stores two double-precision data aligned on a 16-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception. A SIMD STDFR that is aligned on a 4-byte boundary does not cause a *STDF_mem_address_not_aligned* exception.

SIMD STFR and SIMD STDFR can only be used to access cacheable address spaces. An attempt to access a noncacheable address space using a SIMD STFR or SIMD STDFR causes a *data_access_exception* exception. The bypass ASIs that can be accessed using a SIMD store are ASI_PHYS_USE_EC{_LITTTLE}.

Like non-SIMD store instructions, memory access semantics for SIMD STFR and SIMD STDFR instructions adhere to TSO. SIMD STFR and SIMD STDFR instructions simultaneously executes basic and extended loads; however, the ordering between the basic and extended loads conforms to TSO.

A watchpoint can be detected in both the basic and extended stores of a SIMD STFR or SIMD STDFR.

For more information regarding SIMD STFR and SIMD STDFR exception conditions and instruction priority, see Appendix F.5.1, *"Trap Conditions for SIMD Load/Store"* (page 181).

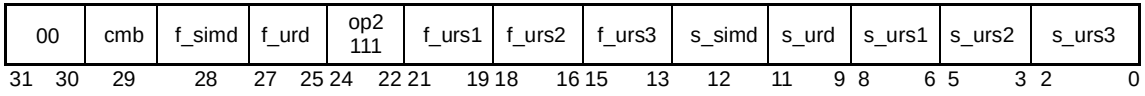
Exceptions

illegal_instruction (`i` = 0)
fp_disabled
illegal_action (`XAR.v` = 1 and (`XAR.urs1` > 1 or
`XAR.urs3`<2> ≠ 0);
`XAR.v` = 1 and `XAR.simd` = 1 and
(`XAR.urs2`<2> ≠ 0 or `XAR.urd`<2> ≠ 0))
mem_address_not_aligned
STDF_mem_address_not_aligned (`STDFR` and (`XAR.v` = 0 or `XAR.simd` = 0))
VA_watchpoint
fast_data_access_MMU_miss
data_access_exception
fast_data_access_protection
PA_watchpoint
data_access_error

A.78 Set XAR (SXAR)

HPC-ACE Ext. Regs.	SIMD	Opcode	op2	cmb	Operation
		SXAR1	111	0	Set XAR for the following instruction
		SXAR2	111	1	Set XAR for the following two instructions

Format (2)



Assembly Language Syntax
sxar1
sxar2

Description

The SXAR instructions update the XAR. The XAR holds value for up to 2 instructions. SXAR1 sets values for 1 instruction, and SXAR2 sets values for 2 instructions. Fields that start with f_ are used by the instruction that immediately follows SXAR, and fields that start with s_are used by the second instruction that follows SXAR. For SXAR1, the s_* fields are ignored.

Compatibility Note – Although an *illegal_instruction* exception is *not* signalled for an SXAR1 with non-zero s_* fileds, use of such an SXAR1 instruction is strongly discouraged for compatibility reasons.

SXAR instructions are used when up to 2 instructions that follow an SXAR instruction specify the integer or floating-point registers added in SPARC64 VIIIfx, or when SIMD instructions are specified.

Implementation Note – Hardware may be implemented to enable high-speed execution of consecutive instructions.

When an SXAR instruction and the following instruction are not consecutive in memory, such as when an SXAR instruction is placed in a delay slot, a Tcc instruction is inserted between the two instructions. This may cause a decrease in performance.

There are cases where IIU_INST_TRAP cannot be detected during SXAR execution.The SXAR instruction itself is not an XAR-eligible instruction, and an attempt to execute SXAR while XAR.v = 1 causes an *illegal_action* exception.

Compatibility Note – op = 00_2 and op2 = 111_2 are reserved in SPARC V9, but SPARC V8 defines the FBcc instruction in these opcodes. When running a SPARC V8 application on SPARC64 VIIIfx, there is the possibility of different behavior.

Programming Note – The SXAR instruction word contains the value to be set in XAR, but this value is not shown by the assembly syntax. HPC-ACE behavior is indicated by mnenomic suffixes appended to the following instruction(s), and the assembler sets this information in the SXAR instruction word.

```
sxar1
faddd,s   %f0, %f2, %f4          /* SIMD */
```

Exceptions *illegal_action* (XAR.v = 1)

A.79 Cache Line Fill with Undetermined Values

Regs.	HPC-ACE Ext. SIMD	Opcode	imm_asi	ASI Value	Operation
✓		STXA STDA[D] STDFA	ASI_XFILL_AIUP	72_{16}	Accesses the cache at the specified address in the primary ASI and fills the cache line with undetermined values.
✓		STXA STDA[D] STDFA	ASI_XFILL_AIUS	73_{16}	Accesses the cache at the specified address in the secondary ASI and fills the cache line with undetermined values.
✓		STXA STDA[D] STDFA	ASI_XFILL_P	$f2_{16}$	Accesses the cache at the specified address in the primary ASI and fills the cache line with undetermined values.
✓		STXA STDA[D] STDFA	ASI_XFILL_S	$f3_{16}$	Accesses the cache at the specified address in the secondary ASI and fills the cache line with undetermined values.

Description

When STXA, STDA, and STDFA instructions specify the any of the above ASIs, the cache line corresponding to the specified address is secured for a write to the cache, and the cache line is filled with undetermined values. Data is not transferred to the CPU from memory. As long as the address specified by the instruction is a virtual address aligned on an 8-byte boundary, any address in the cache line can be specified.

A STXA or STDA address that is not aligned on an 8-byte boundary causes a *mem_address_not_aligned* exception.

A STDFA address that is aligned on a 4-byte boundary but not an 8-byte boundary causes a *STDF_mem_address_not_aligned* exception. An address that is not aligned on an 8-byte boundary nor a 4-byte boundary causes a *mem_address_not_aligned* exception.

The XFILL_{AIUP,AIUS,S,P} ASIs are not affected by the hardware prefetch setting. The value of XAR.dis_hw_pf is ignored.

The ordering between XFILL_{AIUP,AIUS,S,P} and the following memory access conforms to TSO.

An attempt to access a page with TTE.CP = 0 using XFILL_{AIUP,AIUS,S,P} is detected as a watchpoint, alignment, or protection violation, and the cache line fill is not performed.

An *ECC_error* exception caused by a bus error or bus timeout is not signalled for XFILL_{AIUP,AIUS,S,P}. Also, a *data_access_error* is not signalled when the address specified by the instruction exists in the L1 or L2 caches and there is an UE in that cache line.

A watchpoint is detected if all 128 bytes of XFILL_{AIUP,AIUS,S,P} are matched.

If a subsequent access to the same cache line occurs while the cache line is being filled, the access is delayed until the cache line fill commits.

Programming Notes – A MEMBAR is not needed between XFILL and the following access.

Because the following access is delayed, performance can be negatively affected. When performance is required, it is important to execute XFILL well ahead of the actual store. The time required to commit XFILL depends on the system; thus, there may be cases where XFILL is executed sufficiently early on one system, but not sufficiently early for a future version of the processor.

The XFILL_{AIUP,AIUS,S,P} ASIs were implemented to accelerate the memset() and memcpy() functions. Sample code for memset()/memcpy() is shown below. HPC-ACE mnenomic suffixes are used. See Appendix G.4, *"HPC-ACE Notation"* (page 206) for details

Note that both pieces of sample code assume that infrequently reused data is stored in sector 0. The actual usage of sector 0 and sector 1 depends on the application; thus, if sector 1 is used to cache frequently reused data, using the following sample code "as is" may cause a reduction in performance.

```
[memset(0) pseudo-code]
 /*
  *  %i0: dst
  */
 ahead = 4 * 128;! adjust as needed
 for (i = 0 ; i < size; i += 128) {
    stxa      %g0, [%i0+ahead] #ASI_XFILL

    sxar2
    stx,d     %g0, [%i0]
    stx,d     %g0, [%i0+8]
    sxar2
    stx,d     %g0, [%i0+16]
    stx,d     %g0, [%i0+24]
    sxar2
    stx,d     %g0, [%i0+32]
    stx,d     %g0, [%i0+40]
    sxar2
```

```
   stx,d     %g0, [%i0+48]
   stx,d     %g0, [%i0+56]
   sxar2
   stx,d     %g0, [%i0+64]
   stx,d     %g0, [%i0+72]
   sxar2
   stx,d     %g0, [%i0+80]
   stx,d     %g0, [%i0+88]
   sxar2
   stx,d     %g0, [%i0+96]
   stx,d     %g0, [%i0+104]
   sxar2
   stx,d     %g0, [%i0+112]
   stx,d     %g0, [%i0+120]

   add       %i0, 128, %i0
}

[memcpy() pseudo-code]
/*
 *  %i0: dst
 *  %i1: src
 */
ahead = 4 * 128;! adjust as needed
for (i = 0 ; i < size; i += 128) {
   prefetch  [%i1+128], #n_reads
   ldx       [%i1], %l2
   ldx       [%i1+8], %l3
   ldx       [%i1+16], %l4
   ldx       [%i1+24], %l5
   ldx       [%i1+32], %l6
   ldx       [%i1+40], %l7
   ldx       [%i1+48], %o0
   ldx       [%i1+56], %o1
   ldx       [%i1+64], %o2
   ldx       [%i1+72], %o3
   ldx       [%i1+80], %o4
   ldx       [%i1+88], %o5
   ldx       [%i1+96], %o6
   ldx       [%i1+104], %o7
   ldx       [%i1+112], %i6
   ldx       [%i1+120], %i7

   stxa      %g0, [%i0+ahead] #ASI_XFILL

   prefetch  [%i0+128], #n_writes
   sxar2
```

```
        stx,d     %l2, [%i0]
        stx,d     %l3, [%i0+8]
        sxar2
        stx,d     %l4, [%i0+16]
        stx,d     %l5, [%i0+24]
        sxar2
        stx,d     %l6, [%i0+32]
        stx,d     %l7, [%i0+40]
        sxar2
        stx,d     %o0, [%i0+48]
        stx,d     %o1, [%i0+56]
        sxar2
        stx,d     %o2, [%i0+64]
        stx,d     %o3, [%i0+72]
        sxar2
        stx,d     %o4, [%i0+80]
        stx,d     %o5, [%i0+88]
        sxar2
        stx,d     %o6, [%i0+96]
        stx,d     %o7, [%i0+104]
        sxar2
        stx,d     %i6, [%i0+112]
        stx,d     %i7, [%i0+120]

        add       %i1, 128, %i1
        add       %i0, 128, %i0
    }
```

Exceptions

*fp_disabled (*STDFA*)*
*illegal_action (*STXA, STDA *with* XAR.v *= 1 and (*XAR.urs1 *> 1 or*
*(*i *= 0 and* XAR.urs2 *> 1) or*
*(*i *= 1 and* XAR.urs2 *≠ 0) or*
XAR.urs3<*2*> *≠ 0 or*
XAR.urd *> 1);*
STDFA *with* XAR.v *= 1 and(*XAR.urs1 *> 1 or*
*(*i *= 0 and* XAR.urs2 *> 1) or*
*(*i *= 1 and* XAR.urs2 *≠ 0) or*
XAR.urs3<*2*> *≠ 0);*
XAR.v *= 1 and* XAR.simd *= 1)*
mem_address_not_aligned
STDF_mem_address_not_aligned
*privileged_action (*ASI_XFILL_AIUP, ASI_XFILL_AIUS*)*
VA_watchpoint
fast_data_access_MMU_miss
data_access_exception
fast_data_access_protection

PA_watchpoint
data_access_error

F.APPENDIX **B**

IEEE Std. 754-1985 Requirements for SPARC-V9

The IEEE Std. 754-1985 floating-point standard contains a number of implementation dependencies. Appendix B of JPS1 **Commonality** specifies choices for these implementation dependencies, to ensure that SPARC V9 implementations are as consistent as possible. Please refer to JPS1 **Commonality** for details.

This appendix describes the following:

- Conditions under which an unfinished_FPop can occur
- *Floating-Point Nonstandard Mode* on page 142

The first item describes the implementation dependencies defined in the subsection *"FSR_floating-point_trap_type (ftt)"* of Section 5.1.7 in JPS1 **Commonality**. For convenience, this document describes that information in this appendix.

B.1 Traps Inhibiting Results

Please refer to Section B.1 in JPS1 **Commonality**.

The SPARC64 VIIIfx hardware, in conjunction with system software, produces the results described in this section.

B.6 Floating-Point Nonstandard Mode

This section descibes the behavior of SPARC64 VIIIfx in nonstandard mode, which deviates from IEEE 754-1985. For the reader's convenience, this section also describes the conditions under which an *fp_exception_other* exception with `FSR.ftt` = *unfinished_FPop* can occur, even though this exception only occurs in standard mode (`FSR.NS` = 0).

SPARC64 VIIIfx floating-point hardware only handles numbers in a specific range. If the values of the source operands or the intermediate result predict that the final result will not be in the specified range, SPARC64 VIIIfx generates an *fp_exception_other* exception with `FSR.ftt` = 02_{16} (*unfinished_FPop*). Subsequent processing is handled by software; an emulation routine completes the operation in accordance with IEEE 754-1985 (impl. dep. #3)。

SPARC64 VIIIfx implements a nonstandard mode, which is enabled when `FSR.NS` = 1. See *"FSR_nonstandard_fp (NS)"* (page 23). The floating-point behavior of SPARC64 VIIIfx depends on the value of `FSR.NS`.

B.6.1 *fp_exception_other* Exception (ftt=*unfinished_FPop)*

Almost all SPARC64 VIIIfx floating-point arithmetic operations can cause an *fp_exception_other* exception with `FSR.ftt` = *unfinished_FPop* (see specific instruction definitions for details). Conditions under which this exception occurs are described below.

1. When one operand is denormal and all other operands are normal (not zero, infinity, NaN), an *fp_exception_other* exception with *unfinished_FPop* occurs. The exception does not occur when the result is a zero or an overflow.
2. When all operands are denormal and the result is not a zero or an overflow, an *fp_exception_other* exception with *unfinished_FPop* occurs.
3. When all operands are normal, the result before rounding is denormal, `TEM.UFM` = 0, and the result is not a zero, an *fp_exception_other* exception with *unfinished_FPop* occurs.

When the result is expected to be a constant, such as zero or infinity, and the calculation is simple enough for hardware, SPARC64 VIIIfx performs the operation. An *unfinished_FPop* does not occur.

Implementation Note – To detect these conditions precisely requires a large amount of hardware. To avoid this hardware cost, SPARC64 VIIIfx detects approximate conditions by calculating the exponent of the intermediate result (that is, the exponent before rounding) from the source operands. Since detection is approximate and conservative, an *unfinished_FPop* may be generated even when the actual result is a zero or an overflow.

TABLE B-1 describes the formulae used to estimate the result exponent for detecting *unfinished_FPop* conditions. Here, *Er* is an approximation of the biased result exponent before the significand is aligned and before rounding; it is calculated using only the source exponents (*esrc1*, *esrc2*).

TABLE B-1 Result Exponent Approximation for Detecting *unfinished_FPop* Exceptions

Operation	Formula
`fmuls`	*Er* = *esrc1* + *esrc2* – 126
`fmuld`	*Er* = *esrc1* + *esrc2* – 1022
`fdivs`	*Er* = *esrc1* - *esrc2* + 126
`fdivd`	*Er* = *esrc1* - *esrc2* + 1022

esrc1 and *esrc2* are the biased exponents of the source operands. When an source operand is a denormalized number, the corresponding exponent is 0.

Once *Er* is calculated, *eres* can be obtained. *eres* is the biased result exponent after the significand is aligned and before rounding. That is, the significand is left-shifted or right-shifted so that an implicit 1 is immediately to the left of the binary point. *eres* is the value obtained from adding or subtracting the amount shifted to *Er.*

TABLE B-2 describes the conditions under which each floating-point instruction generates an *unfinished_FPop* exception.

TABLE B-2 *unfinished_FPop* Detection Conditions

Operation	Detection Condition
`FdTOs`	–25 < *eres* < 1 and `TEM.UFM` = 0.
`FsTOd`	The second operand (`rs2`) is denormal.
`FADDs`, `FSUBs`, `FADDd`, `FSUBd`	1. One operand is denormal, and the other operand is normal (not zero, infinity, NaN). [1] 2. Both operands are denormal. 3. Both operands are normal (not zero, infinity, NaN), *eres* < 1, and `TEM.UFM` = 0.

TABLE B-2 *unfinished_FPop* Detection Conditions *(Continued) (Continued)*

Operation	Detection Condition
FMULs, FMULd	1. One operands is denormal, the other operand is normal (not zero, infinity, NaN), and single precision: -25 < *Er* double precision: -54 < *Er* 2. Both operands are normal (not zero, infinity, NaN), TEM.UFM = 0, and single precision: –25 < *eres* < 1 double precision: –54 < *eres* < 1
FsMULd	1. One operand is denormal, and the other operand is normal (not zero, infinity, NaN). 2. Both operands are denormal.
FDIVs, FDIVd	1. The dividend (rs1) is normal (not zero, infinity, NaN), the divisor (rs2) is denormal, and single precision: *Er* < 255 double precision: *Er* < 2047 2. The dividend (rs1) is denormal, the divisor (rs2) is normal (not zero, infinity, NaN), and single precision: –25 < *Er* double precision: –54 < *Er* 3. Both operands are denormal. 4. Both operands are normal (not zero, infinity, NaN), TEM.UFM = 0, and single precision: –25 < *eres* < 1 double precision: –54 < *eres* < 1
FSQRTs, FSQRTd	The source operand (rs2) is positive, nonzero, and denormal.
FMADD{s,d}, FMSUB{s,d}, FNMADD{s,d}, FNMSUB{s,d}	Same conditions as FMUL{s,d} for the multiplication, and same conditions as FADD{s,d} for the add.
FTRIMADDd	Same conditions as FMUL{s,d} for the multiplication. An add does not occur.
FTRISMULd	1. When rs1 is normal (not zero, infinity, NaN) and TEM.UFM = 0, and double-precision: –54 < eres < 1
FRCPA{s,d}	When the operands are denormal.
FRSQRTA{s,d}	When the operands are positive, nonzero, and denormal.

1.When the source operand is zero and denormal, the generated result conforms to IEEE754-1985.

Conditions for a Zero Result

When a condition listed in TABLE B-3 is `true`, SPARC64 VIIIfx generates a zero result; that is, the result is a denormalized minimum or a zero, depending on the rounding mode (`FSR.RD`).

TABLE B-3 Conditions for a Zero Result

Operations	Conditions		
	One operand is denormal[1]	Both are denormal	Both are normal[2]
`FdTOs`	always	—	*eres* ≤ -25
`FMULs`, `FMULd`	single precision: *Er* ≤ –25 double precision: *Er* ≤ –54	always	single precision: *eres* ≤ –25 double precision: *eres* ≤ –54
`FDIVs`, `FDIVd`	single precision: *Er* ≤ –25 double precision: *Er* ≤ –54	never	single precision: *eres* ≤ –25 double precision: *eres* ≤ –54

1.Except when both operands are zero, NaN, or infinity.

2.And when neither operand is NaN or infinity. If both operands are zero, *eres* is never less than zero.

Conditions for an Overflow Result

If a condition listed in TABLE B-4 is `true`, SPARC64 VIIIfx assumes the operation causes an overflow.

TABLE B-4 Conditions for an Overflow Result

Operations	Conditions
`FDIVs`	The divisor (`rs2`) is denormal and *Er* ≥ 255.
`FDIVd`	The divisor (`rs2`) is denormal and *Er* ≥ 2047.

B.6.2 Behavior when FSR.NS = 1

When `FSR.NS` = 1 (nonstandard mode), SPARC64 VIIIfx replaces all denormal source operands and denormal results with zeroes. This behavior is described below in greater detail:

- When one operand is denormal and none of the operands is zero, infinity, or NaN, the denormal operand is replaced with a zero of the same sign, and the operation is performed. After the operation, `cexc.nxc` is set to 1 unless one of the following conditions occurs; in which case, `cexc.nxc` = 0.
  - A *division_by_zero* or an *invalid_operation* is detected for a `FDIV{s,d}`.
  - An *invalid_operation* is detected for a `FSQRT{s,d}`.
  - The operation is a `FRPCA{s,d}` or a `FRSQRTA{s,d}`.

  When `cexc.nxc` = 1 and `TEM.NXM` = 1 in `FSR`, a *fp_exception_ieee_754* exception occurs.

- When the result before rounding is denormal, the result is replaced with a zero of the same sign.

  If `TEM.UFM` = 1 in `FSR`, then `cexc.ufc` = 1; if `TEM.UFM` = 0 and `TEM.NXM` = 1, then `cexc.nxc` = 1. In both cases, a *fp_exception_ieee_754* exception occurs. When `TEM.UFM` = 0 and `TEM.NXM` = 0, both `cexc.nxc` and `cexc.ufc` are set to 1.

When `FSR.NS` = 1, SPARC64 VIIIfx does not generate *unfinished_FPop* exceptions or return denormalized numbers as results.

TABLE B-5 summarizes the exceptions generated by the floating-point arithmetic instructions[1] listed in TABLE B-2. All possible exceptions and masked exceptions are listed in the "Result" column. The generated exception depends on the value of `FSR.NS`, the source operand type, the result type, and the value of `FSR.TEM`; it can be found by tracing the conditions from left to right. If `FSR.NS` = 1 and the source operands are denormal, refer to TABLE B-6. In TABLE B-5, the shaded areas in the "Result" column conform to IEEE754-1985.

Note – In Table B-5 and TABLE B-6, lowercase exceptional conditions (nx, uf, of, dv, nv) do not signal IEEE 754 exceptions. Uppercase exceptional conditions (NX, UF, OF, DZ, NV) do signal IEEE 754 exceptions.

TABLE B-5 Floating-Point Exception Conditions and Results *(1 of 2)*

FSR.NS	Source Denormal[1]	Result Denormal[2]	Zero Result	Overflow Result	UFM	OFM	NXM	Result
0	No	Yes	Yes	—	1	—	—	UF
					0	—	1	NX
							0	uf + nx, a signed zero, or a signed Dmin[3]
			No	—	1	—	—	UF
					0	—	—	*unfinished_FPop*[4]
		No	—	—	—	—	—	Conforms to IEEE754-1985
	Yes	—	Yes	—	1	—	—	UF
					0	—	1	NX
							0	uf + nx, a signed zero, or a signed Dmin
			No	Yes	—	1	—	OF
					—	0	1	NX
							0	of + nx, a signed infinity, or a signed Nmax[5]
				No	—	—	—	*unfinished_FPop*

1. rs2 for `FTRISmuld` is not a floating-point number and cannot be denormal.

TABLE B-5 Floating-Point Exception Conditions and Results *(Continued) (2 of 2)*

FSR.NS	Source Denormal[1]	Result Denormal[2]	Zero Result	Overflow Result	UFM	OFM	NXM	Result
1	No	Yes	—	—	1	—	—	UF
					0	—	1	NX
							0	uf + nx, a signed zero
		No	—	—	—	—	—	Conforms to IEEE754-1985
	Yes	—	—	—	—	—	—	TABLE B-6

1.One operand is denormal, and the other operands are normal (not zero, infinity, NaN) or denormal.

2.The result before rounding turns out to be denormal.

3.Dmin = denormalized minimum.

4.If the operation is `FADD{s,d}` or `FSUB{s,d}` and the source operands are zero and denormal, SPARC64 VIIIfx does not generate an *unfinished_FPop*; instead, the operation is performed conformant to IEEE754-1985.

5.Nmax = normalized maximum.

TABLE B-6 describes SPARC64 VIIIfx behavior when `FSR.NS` = 1 (nonstandard mode). Shaded areas in the "Result" column conform to IEEE754-1985.

TABLE B-6 Operations with Denormal Source Operands when `FSR.NS` = 1 *(1 of 2)*

Instruction	Source Operand			FSR.TEM				Result
	op1	op2	op3	UFM	NXM	DVM	NVM	
`FsTOd`	—	Denorm	—	—	1	—	—	NX
					0	—	—	nx, a signed zero
`FdTOs`	—	Denorm	—	1	—	—	—	UF
				0	1	—	—	NX
					0	—	—	uf + nx, a signed zero
`FADD{s,d}` `FSUB{s,d}`	Denorm	Normal	—	—	1	—	—	NX
					0	—	—	nx, op2
	Normal	Denorm	—		1	—	—	NX
					0	—	—	nx, op1
	Denorm	Denorm	—		1	—	—	NX
					0	—	—	nx, a signed zero
`FMUL{s,d}` `FsMULd`	Denorm	—	—	—	1	—	—	NX
					0	—	—	nx, a signed zero
	—	Denorm	—		1	—	—	NX
					0	—	—	nx, a signed zero

TABLE B-6 Operations with Denormal Source Operands when FSR.NS = 1 *(2 of 2)*

Instruction	Source Operand			FSR.TEM				Result
	op1	op2	op3	UFM	NXM	DVM	NVM	
FDIV{s,d}	Denorm	Normal	—	—	1	—	—	NX
					0	—	—	nx, a signed zero
	Normal	Denorm	—		—	1	—	DZ
						0	—	dz, a signed infinity
	Denorm	Denorm	—		—	—	1	NV
							0	nv, dNaN[1]
FSQRT{s,d}	—	Denorm and op2 > 0	—	—	1	—	—	NX
					0	—	—	nx, zero
		Denorm and op2 < 0	—		—	—	1	NV
							0	nv, dNaN[1]
FMADD{s,d} FMSUB{s,d} FNMADD{s,d} FNMSUB{s,d} FTRIMADDd[2]	Denorm	—	Normal	—	1	—	—	NX
					0	—	—	nx, op3
			Denorm		1	—	—	NX
					0	—	—	nx, zero with same sign as the result before rounding
	—	Denorm	Normal	—	1	—	—	NX
					0	—	—	nx, op3
			Denorm		1	—	—	NX
					0	—	—	nx, zero with same sign as the result before rounding
	Normal	Normal	Denorm	—	1	—	—	NX
					0	—	—	nx, op1 × op2[3]
FTRISMULd	Denorm	—	—	—	1	—	—	NX
					0	—	—	nx, zero whose sign bit is op2<0>
FRCPA{s,d}	—	Denorm	—	—	—	1	—	DZ
						0	—	dz, infinity with same sign as the result before rounding
FRSQRTA{s,d}	—	Denorm	—	—	—	1	—	DZ
						0	—	dz, infinity with same sign as the result before rounding

1.A single-precision dNaN is $7FFF.FFFF_{16}$, and a double-precision dNaN is $7FFF.FFFF.FFFF.FFFF_{16}$.

2.op3 is obtained from a table in the functional unit and is always normal.

3.When op1 × op2 is denormal, op1 × op2 becomes a zero with the same sign.

F.APPENDIX **C**

Implementation Dependencies

This appendix summarizes how implementation dependencies defined in JPS1 **Commonality** are implemented in SPARC64 VIIIfx. In SPARC V9 and SPARC JPS1, the notation "**IMPL. DEP. #*nn*:**" identifies the definition of an implementation dependency; the notation "(impl. dep. #*nn*)" identifies a reference to an implementation dependency. These dependencies are described by their number *nn* in TABLE C-1.

Note – SPARC International maintains a document, *Implementation Characteristics of Current SPARC-V9-based Products, Revision 9.x*, that describes the implementation-dependent design features of all SPARC V9-compliant implementations. Contact SPARC International for this document at:

home page: `www.sparc.org`
email: info@sparc.org

C.4 List of Implementation Dependencies

TABLE C-1 summaries how JPS1 implementation dependencies are implemented in SPARC64 VIIIfx.

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(1 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
1	**Software emulation of instructions** The operating system emulates all quad-precision instructions that generate an *illegal_instruction* or *unimplemented_FPop* exception.	—

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(2 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
2	**Number of IU registers** SPARC64 VIIIfx supports eight register windows (NWINDOWS = 8). SPARC64 VIIIfx also supports two additional global register sets (Interrupt globals and MMU globals) and registers added by HPC-ACE. There are a total of 160 integer registers.	—
3	**Incorrect IEEE Std. 754-1985 results** See Section B.6, *"Floating-Point Nonstandard Mode"*, for details.	142
4–5	Reserved.	
6	**I/O registers privileged status** This item is out of the scope of this document. Refer to the SPARC64 VIIIfx System Specification.	—
7	**I/O register definitions** This item is out of the scope of this document. Refer to the SPARC64 VIIIfx System Specification.	—
8	RDASR/WRASR **target registers** In SPARC64 VIIIfx, the XAR, XASR, and TXAR can be read by RDASR, and the XASR and TXAR can be written by WRASR.	98, 112
9	RDASR/WRASR **privileged status** In SPARC64 VIIIfx, the TXAR is a privileged register.	98, 112
10–12	Reserved.	
13	**VER.impl** VER.impl = 8 for the SPARC64 VIIIfx processor.	26
14–15	Reserved.	—
16	**IU deferred-trap queue** SPARC64 VIIIfx does not implement an IU deferred-trap queue.	38
17	Reserved.	—
18	**Nonstandard IEEE 754-1985 results** When FSR.NS = 1, a denormal result is replaced with zeroes in SPARC64 VIIIfx. See Section B.6, *"Floating-Point Nonstandard Mode"*, for details.	142
19	**FPU version,** FSR.ver FSR.ver = 0 in SPARC64 VIIIfx.	23
20–21	Reserved.	
22	**FPU** TEM**,** cexc**, and** aexc SPARC64 VIIIfx hardware implements all bits in the TEM, cexc, and aexc fields.	23
23	**Floating-point traps** In SPARC64 VIIIfx, floating-point traps are always precise. A FQ is not needed.	38
24	**FPU deferred-trap queue (FQ)** SPARC64 VIIIfx does not implement a floating-point deferred-trap queue.	38

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(3 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
25	`RDPR` **of FQ with nonexistent FQ** Attempting to execute an `RDPR` of the `FQ` causes an *illegal_instruction* exception.	38
26–28	Reserved.	—
29	**Address space identifier (ASI) definitions** The ASIs that are supported by SPARC64 VIIIfx are defined in Appendix L.	213
30	**ASI address decoding** SPARC64 VIIIfx decodes all 8 bits of the ASI specifier.	—
31	**Catastrophic error exceptions** SPARC64 VIIIfx implements a watchdog timer. If no instructions are committed for a specified number of cycles, the CPU tries to cause an *async_data_error* trap. After 6.7 seconds, the processor enters `error_state`. The processor can be configured to recover from `error_state` by generating a WDR on entry to `error_state`.	246
32	**Deferred traps** In SPARC64 VIIIfx, severe errors are reported by deferred traps. SPARC64 VIIIfx does not implement a deferred trap queue.	46, 255
33	**Trap precision** The only deferred traps are traps that report severe errors. In SPARC64 VIIIfx, all traps that occur as the result of instruction execution are precise.	46
34	**Interrupt clearing** See Appendix N for details on interrupt handling.	239
35	**Implementation-dependent traps** SPARC64 VIIIfx supports the following implementation-dependent traps: • *interrupt_vector_trap* (`tt` = 060_{16}) • *PA_watchpoint* (`tt` = 061_{16}) • *VA_watchpoint* (`tt` = 062_{16}) • *ECC_error* (`tt` = 063_{16}) • *fast_instruction_access_MMU_miss* (`tt` = 064_{16}–067_{16}) • *fast_data_access_MMU_miss* (`tt` = 068_{16}–$06B_{16}$) • *fast_data_access_protection* (`tt` = $06C_{16}$–$06F_{16}$) • *async_data_error* (`tt` = 040_{16})	53
36	**Trap priorities** SPARC64 VIIIfx implementation-dependent traps have the following priorities: • *interrupt_vector_trap* (priority=16) • *PA_watchpoint* (priority=12) • *VA_watchpoint* (priority=1) • *ECC_error* (priority=33) • *fast_instruction_access_MMU_miss* (priority = 2) • *fast_data_access_MMU_miss* (priority = 12) • *fast_data_access_protection* (priority = 12) • *async_data_error* (priority = 2)	51
37	**Reset trap** SPARC64 VIIIfx implements power-on resets (POR) and the watchdog reset.	46

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(4 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
38	**Effect of reset trap on implementation-dependent registers** See Section O.2, *"RED_state and error_state"*.	247
39	**Entering** `error_state` **on implementation-dependent errors** The processor enters `error_state` after 6.7 seconds have elapsed in a watchdog timeout, or when a normal trap or SIR occurs while `TL = MAXTL`.	46
40	`Error_state` **processor state** After entering `error_state`, SPARC64 VIIIfx can generate a watchdog reset. The states of almost all error-logging registers are preserved (also see impl. dep. #254).	46
41	Reserved.	
42	`FLUSH` **instruction** SPARC64 VIIIfx implements the `FLUSH` instruction in hardware.	—
43	Reserved.	
44	**Data access FPU trap** The destination register(s) are unchanged if an access error occurs.	82
45–46	Reserved.	
47	**`RDASR`** The `XAR`, `XASR`, and `TXAR` can be read in SPARC64 VIIIfx using `rd` = 29–31. At this time, • Bits <18:0> of the instruction field are handled in the same way as for other `RDASR`. That is, <18:14> is `rs1` and <13> is `i`. When `i=0`, <12:5> is *reserved* and <4:0> is `rs2`. When `i=1`, <12:0> is `simm13`. • Only `TXAR` is a privileged register. A nonzero reserved field does not cause an *illegal_instruction* exception.	98
48	**`WRASR`** The `XAR`, `XASR`, and `TXAR` can be written in SPARC64 VIIIfx using `rd` = 29–31. At this time, • Bits <18:0> of the instruction field are handled in the same way as for other `WRASR`. That is, <18:14> is `rs1` and <13> is `i`. When `i=0`, <12:5> is *reserved* and <4:0> is `rs2`. When `i=1`, <12:0> is `simm13`. • The operation `rs1` **xor** `rs2` or `rs1` **xor** `simm13` is performed. • Only `TXAR` is a privileged register. A nonzero reserved field does not cause an *illegal_instruction* exception.	112
49–54	Reserved.	
55	**Floating-point underflow detection** As specified in JPS1, SPARC64 VIIIfx detects underflow conditions before rounding.	—
56–100	Reserved.	
101	**Maximum trap level** `MAXTL` = 5.	26

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(5 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
102	**Clean windows trap** SPARC64 VIIIfx generates a *clean_window* traps; register windows are cleaned by software.	—
103	**Prefetch instructions** SPARC64 VIIIfx implements `PREFETCH fcn` 0–3 and 20–23 with the following implementation-dependent behavior: • The `PREFETCH` instruction has observable effects in privileged mode. • The `PREFETCH` instruction never causes a *fast_data_access_MMU_miss* trap. • The block of memory prefetched is one 128-byte cache line; that is, its size is 128 bytes and its alignment is 128 bytes. • See Section A.49, *"Prefetch Data"*, for descriptions of the prefetch variants and their characteristics. • Prefetches to the following ASIs are valid: `ASI_PRIMARY`, `ASI_SECONDARY`, or `ASI_NUCLEUS`, `ASI_PRIMARY_AS_IF_USER`, `ASI_SECONDARY_AS_IF_USER`, and the corresponding little-endian ASIs.	96
104	**`VER.manuf`** `VER.manuf` = 0004_{16}. The lower 8 bits display Fujitsu's JEDEC manufacturing code.	26
105	**TICK register** SPARC64 VIIIfx implements all 63 bits in `TICK.counter`; the counter is incremented every clock cycle.	25
106	**`IMPDEP`*n* instructions** In addition to VIS1 and VIS2 instructions, SPARC64 VIIIfx implements a large number of SPARC64 VIIIfx-specific instructions.	71
107	**Unimplemented `LDD` trap** SPARC64 VIIIfx implements `LDD` in hardware.	—
108	**Unimplemented `STD` trap** SPARC64 VIIIfx implements `STD` in hardware.	—
109	***LDDF_mem_address_not_aligned*** In SPARC64 VIIIfx, a non-SIMD `LDDF` address that is aligned on a 4-byte boundary but not an 8-byte boundary causes a *LDDF_mem_address_not_aligned* exception. System software emulates the instruction. A SIMD `LDDF`, however, causes a *mem_address_not_aligned* exception instead.	82, 86
110	***STDF_mem_address_not_aligned*** In SPARC64 VIIIfx, a non-SIMD `STDF` address that is aligned on a 4-byte boundary but not an 8-byte boundary causes a *STDF_mem_address_not_aligned* exception. System software emulates the instruction. A SIMD `STDF`, however, causes a *mem_address_not_aligned* exception instead.	101, 105
111	***LDQF_mem_address_not_aligned*** SPARC64 VIIIfx does not implement `LDQF`, and an attempt to execute `LDQF` causes an *illegal_instruction* exception. The processor does not check *fp_disabled*. System software emulates `LDQF`.	82, 86

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(6 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
112	***STQF_mem_address_not_aligned*** SPARC64 VIIIfx does not implement `STQF`, and an attempt to execute `STQF` causes an *illegal_instruction* exception. The processor does not detected an *fp_disabled* exception. System software emulates `STQF`.	101, 105
113	**Implemented memory models** SPARC64 VIIIfx implements Total Store Order (TSO) for all memory models specified in `PSTATE.MM`. See Chapter 8 for details.	55
114	**`RED_state` trap vector address (`RSTVaddr`)** `RSTVaddr` is a constant in SPARC64 VIIIfx, with the following value: VA=FFFF FFFF F000 0000_{16} PA=01FF F000 0000_{16}	45
115	**`RED_state` processor state** See Section 7.1.1 for details on behavior while in `RED_state`.	45
116	**`SIR_enable` control flag** As specified in JPS1, the `SIR_enable` control flag does not exist in SPARC64 VIIIfx. The `SIR` instruction behaves like a `NOP` in nonprivileged mode.	—
117	**MMU disabled prefetch behavior** In SPARC64 VIIIfx, `PREFETCH` commits without accessing memory when the DMMU is disabled. As specified in Section F.5 of JPS1 **Commonality**, a nonfaulting load causes a *data_access_exception* exception.	183
118	**Identifying I/O locations** TThis item is out of the scope of this document. Refer to the SPARC64 VIIIfx System Specification.	—
119	**Unimplemented values for `PSTATE.MM`** Writing 11_2 into `PSTATE.MM` causes the machine to use the TSO memory model. However, the encoding 11_2 should not be used because future versions of SPARC64 VIIIfx may assign this encoding to a different memory model.	56
120	**Coherence and atomicity of memory operations** This item is out of the scope of this document. Refer to the SPARC64 VIIIfx System Specification.	—
121	**Implementation-dependent memory model** Accesses to a page with the `E` bit set (that is, to a volatile page) are processed in program order.	—
122	**`FLUSH` latency** Since the `FLUSH` instruction synchronizes cache states between all on-chip cores, the execution latency depends on the processor state. Assuming that all prior instructions have committed, the latency of a `FLUSH` is 30 processor cycles.	56
123	**Input/output (I/O) semantics** This item is out of the scope of this document. Refer to the SPARC64 VIIIfx System Specification.	—

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(7 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
124	**Implicit ASI when `TL` > 0** As specified in JPS1, when `TL` > 0, `ASI_NUCLEUS` or `ASI_NUCLEUS_LITTLE` are used depending on the value of `PSTATE.CLE`.	—
125	**Address masking** When `PSTATE.AM` = 1, SPARC64 VIIIfx masks the high-order 32 bits of the `PC` transmitted to the specified destination register(s).	42, 70, 81
126	**Register Windows State Registers width** In SPARC64 VIIIfx, `NWINDOWS` is 8. Thus, only 3 bits in the `CWP`, `CANSAVE`, `CANRESTORE`, and `OTHERWIN` registers are valid. On an attempt to write a value greater than `NWINDOWS` – 1 to any of these registers, only the lower 3 bits are written; the upper bits are ignored. The `CLEANWIN` register contains 3 bits.	—
127–201	Reserved.	
202	*fast_ECC_error* **trap** SPARC64 VIIIfx does not implement the *fast_ECC_error* trap.	—
203	**Dispatch Control Register bits 13:6 and 1** SPARC64 VIIIfx does not implement `DCR`.	29
204	**`DCR` bits 5:3 and 0** SPARC64 VIIIfx does not implement `DCR`.	29
205	**Instruction Trap Register** SPARC64 VIIIfx implements the Instruction Trap Register as defined in JPS1.	37
206	**`SHUTDOWN` instruction** In privileged mode, SPARC64 VIIIfx executes the `SHUTDOWN` instruction as a `NOP`.	100
207	**`PCR` register bits 47:32, 26:17, and bit 3** SPARC64 VIIIfx uses these bits to implement the following features: • Bits 47:32 – set/clear/show overflow status (`OVF`) • Bit 26 – set `OVF` field read-only (`OVRO`) • Bits 24:22 – indicate the number of counter pairs (`NC`) • Bits 20:18 – select the counter pair (`SC`) • Bit 3 – set `SU`/`SL` field read-only (`ULRO`) Other implementation-dependent bits are read as 0 and writes to these bits are ignored.	27
208	**Ordering of errors captured in instruction execution** SPARC64 VIIIfx signals errors in program order.	255
209	**Software intervention after instruction-induced error** In SPARC64 VIIIfx, an error synchronous to instruction execution is signalled as a precise exception.	—
210	**ERROR output signal** This item is beyond the scope of this document. Refer to the SPARC64 VIIIfx System Specification.	—

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(8 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
211	**Error logging registers' information** In SPARC64 VIIIfx, the cause of a fatal error is not displayed in the ASI_STCHG_ERR_INFO register.	272
212	**Trap with fatal error** In SPARC64 VIIIfx, a fatal error does not cause a trap.	272
213	**AFSR.PRIV** SPARC64 VIIIfx does not implement the AFSR.PRIV bit.	285
214	**Enable/disable control for deferred traps** SPARC64 VIIIfx does not provide an enable/disable control feature for deferred traps.	—
215	**Error barrier** —	—
216	***data_access_error* trap precision** In SPARC64 VIIIfx, a *data_access_error* trap is always precise.	—
217	***instruction_access_error* trap precision** In SPARC64 VIIIfx, an *instruction_access_error* trap is always precise.	—
218	***async_data_error*** SPARC64 VIIIfx generates the *async_data_error* trap with TT = 40_{16}.	47, 255
219	**Asynchronous Fault Address Register (AFAR) allocation** SPARC64 VIIIfx does not implement the AFAR.	—
220	**Addition of logging and control registers for error handling** SPARC64 VIIIfx implements various RAS features for ensuring high reliability. See Appendix P for details.	255
221	**Special/signalling ECCs** —	—
222	**TLB organization** SPARC64 VIIIfx has the following TLB organization: • Level-1 micro ITLB (uITLB), fully associative • Level-1 micro DTLB (uDTLB), fully associative • Level-2 IMMU-TLB, which consists of the sITLB (set-associative Instruction TLB) and fITLB (fully-associative Instruction TLB). • Level-2 DMMU-TLB, which consists of the sDTLB (set-associative Data TLB) and fDTLB (fully-associative Data TLB).	175
223	**TLB multiple-hit detection** In SPARC64 VIIIfx, a multiple hit is detected only when the fTLB is accessed on a micro-TLB miss.	176
224	**MMU physical address width** In SPARC64 VIIIfx, the MMU supports a physical address width of 41 bits. The PA field of the TTE holds a 41-bit physical address. Bits <46:41> always read as 0, and writes to these bits are ignored.	178

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(9 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
225	**TLB locking of entries** When a TTE with the lock bit set is written into the TLB via the Data In register, SPARC64 VIIIfx writes this entry to the appropriate fTLB and locks the entry. Otherwise, the TTE is written into the appropriate sTLB or fTLB, depending on the page size.	178
226	**`TTE` support for `CV` bit** SPARC64 VIIIfx does not support the `CV` bit in TTE. Since I1 and D1 are virtually indexed caches, SPARC64 VIIIfx supports hardware unaliasing. Also see impl. dep. #232.	178
227	**TSB number of entries** The SPARC64 VIIIfx specification does not support a TSB; this implementation dependency is not applicable.	—
228	**`TSB_Hash` supplied from TSB or context-ID register** The SPARC64 VIIIfx specification does not support a TSB; this implementation dependency is not applicable.	—
229	**`TSB_Base` address generation** The SPARC64 VIIIfx specification does not support a TSB; this implementation dependency is not applicable.	—
230	***data_access_exception* trap** SPARC64 VIIIfx generates a *data_access_exception* only for the causes listed in Appendix F.5 of JPS1 **Commonality**.	179
231	**MMU physical address variability** In SPARC64 VIIIfx, the width of the physical address is 41 bits.	183
232	**DCU Control Register `CP` and `CV` bits** SPARC64 VIIIfx does not implement the `CP` and `CV` bits in the DCU Control Register. Also see impl. dep. #226.	34, 183
233	**`TSB_Hash` field** The SPARC64 VIIIfx specification does not support a TSB; this implementation dependency is not applicable.	184
234	**TLB replacement algorithm** fTLB is pseudo-LRU. sTLB is LRU.	192
235	**TLB data access address assignment** See Appendix F.10.4.	192
236	**`TSB_Size` field width** In SPARC64 VIIIfx, `TSB_Size` is the 4-bit field in bits <3:0>. The value written in `TSB_Size` is returned on a read. SPARC64 VIIIfx preserves this value, but does not use it.	194
237	**`DSFAR`/`DSFSR` for `JMPL`/`RETURN` *mem_address_not_aligned*** A *mem_address_not_aligned* exception that occurs during a `JMPL` or `RETURN` instruction does not update either the `D-SFAR` or `D-SFSR`.	81, 180, 195

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(10 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
238	**TLB page offset for large page sizes** In SPARC64 VIIIfx, page offset data is discarded on a TLB write, and undefined data is returned on a read.	178
239	**Register access by ASIs 55_{16} and $5D_{16}$** In SPARC64 VIIIfx, VA<63:18> of IMMU ASI 55_{16} and DMMU ASI $5D_{16}$ are ignored.	184
240	**DCU Control Register bits 47:41** SPARC64 VIIIfx uses bit <41> to implement `WEAK_SPCA`, which enables/disables speculative memory access.	34
241	**Address Masking and `DSFAR`** When `PSTATE.AM` = 1, SPARC64 VIIIfx writes zeroes to the more-significant 32 bits of `DSFAR`.	?
242	**TLB lock bit** In SPARC64 VIIIfx, only the fITLB and the fDTLB support the lock bit. In sITLB and sDTLB, the lock bit is read as 0 and writes to the bit are ignored.	178
243	**Interrupt Vector Dispatch Status Register BUSY/NACK pairs** In SPARC64 VIIIfx, 8 BUSY/NACK bit pairs are implemented.	242
244	**Data Watchpoint Reliability** No implementation-dependent feature in SPARC64 VIIIfx reduces the reliability of data watchpoints.	36
245	**Call/Branch displacement encoding in I-Cache** In SPARC64 VIIIfx, the least significant 11 bits (bits 10:0) of a `CALL` or branch (`BPcc`, `FBPfcc`, `Bicc`, `BPr`) instruction in an instruction cache are identical to the architectural encoding (which appears in main memory).	?
246	**`VA`<38:29> for Interrupt Vector Dispatch Register Access** SPARC64 VIIIfx ignores all 10 bits of `VA`<38:29> when the Interrupt Vector Dispatch Register is written.	242
247	**Interrupt Vector Receive Register SID fields** `SID_H` and `SID_L` values are undefined.	243
248	**Conditions for *fp_exception_other* with *unfinished_FPop*** SPARC64 VIIIfx generates a *fp_exception_other* with floating-point trap type of *unfinished_FPop* for the conditions described in Section 5.1.7 of JPS1 **Commonality**.	23
249	**Data watchpoint for Partial Store instruction** In SPARC64 VIIIfx, watchpoint detection is conservative for a Partial Store instruction. The `DCUCR` Data Watchpoint masks are only checked for a nonzero value (watchpoint enabled). The byte store mask in `r[rs2]` of the Partial Store instruction is ignored, and a watchpoint exception can occur even if the mask is zero (that is, when no store occurs).	94

TABLE C-1 SPARC64 VIIIfx Implementation of JPS1 Implementation Dependencies *(11 of 11)*

Nbr	SPARC64 VIIIfx Implementation Notes	Page
250	**`PCR` accessibility when `PSTATE.PRIV` = 0** In SPARC64 VIIIfx, the accessibility of the `PCR` when `PSTATE.PRIV` = 0 is determined by `PCR.PRIV`. When `PSTATE.PRIV` = 0 and `PCR.PRIV` = 1, an attempt to execute either `RDPCR` or `WRPCR` will cause a *privileged_action* exception. When `PSTATE.PRIV` = 0 and `PCR.PRIV` = 0, `RDPCR` is executed normally, and `WRPCR` only generates a *privileged_action* exception when an attempt is made to change (that is, write a 1 to) `PCR.PRIV`.	27, 28, 98
251	Reserved.	—
252	**`DCUCR.DC` (Data Cache Enable)** SPARC64 VIIIfx does not implement `DCUCR.DC`.	34
253	**`DCUCR.IC` (Instruction Cache Enable)** SPARC64 VIIIfx does not implement `DCUCR.IC`.	34
254	**Means of exiting `error_state`** Normally, the SPARC64 VIIIfx processor, upon entering `error_state`, generates a *watchdog_reset* (WDR) and resets itself. However, `OPSR` can be set so that an entry to `error_state` does not generate a *watchdog_reset* and the processor remains halted in `error_state`.	46, 253
255	**`LDDFA` with ASI $E0_{16}$ or $E1_{16}$ and misaligned destination register number** A misaligned destination register number does not cause an exception.	220
256	**`LDDFA` with ASI $E0_{16}$ or $E1_{16}$ and misaligned memory address** SPARC64 VIIIfx has the following behavior: • If aligned on an 8-byte boundary, causes a *data_access_exception* exception. Does not cause an address alignment exception. • If aligned on a 4-byte boundary, causes a *LDDF_mem_address_not_aligned* exception. • Otherwise, causes a *mem_address_not_aligned* exception.	220
257	**`LDDFA` with ASI $C0_{16}$–$C5_{16}$ or $C8_{16}$–CD_{16} and misaligned memory address** SPARC64 VIIIfx has the following behavior: • If aligned on an 8-byte boundary, causes a *data_access_exception* exception. Does not cause an address alignment exception. • If aligned on a 4-byte boundary, causes a *LDDF_mem_address_not_aligned* exception. • Otherwise, causes a *mem_address_not_aligned* exception.	220
258	**`ASI_SERIAL_ID`** SPARC64 VIIIfx provides an identification code for each processor.	220

F.APPENDIX **D**

Formal Specification of the Memory Models

Please refer to Appendix D in JPS1 **Commonality**.

F.APPENDIX **E**

Opcode Maps

Appendix E contains the instruction opcode maps for all SPARC JPS1 instructions and instructions added by HPC-ACE.

Opcodes marked with a dash (—) are reserved; an attempt to execute a reserved opcode shall cause a trap unless the opcode is an implementation-specific extension to the instruction set. See Section 6.3.9, *Reserved Opcodes and Instruction Fields*, in JPS1 **Commonality** for more information.

In this appendix and in Appendix A, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE A-1 (page 60). For deprecated opcodes, see Section A.71, *Deprecated Instructions*, in JPS1 **Commonality**.

In the tables in this appendix, *reserved* (—) and shaded entries indicate opcodes that are not implemented in SPARC64 VIIIfx processors.

TABLE E-1 op<1:0>

op <1:0>			
0	**1**	**2**	**3**
Branches and SETHI *See* TABLE E-2.	CALL	Arithmetic & Miscellaneous *See* TABLE E-3.	Loads/Stores *See* TABLE E-4.

TABLE E-2 op2<2:0> (op = 0)

op2 <2:0>							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ILLTRAP	BPcc – *See* TABLE E-7	Bicc[D] – *See* TABLE E-7	BPr – *See* TABLE E-8	SETHI NOP[†]	FBPfcc – *See* TABLE E-7	FBfcc[D] – *See* TABLE E-7	SXAR

†rd = 0, imm22 = 0

The ILLTRAP encoding generates an *illegal_instruction* trap.

TABLE E-3 op3<5:0> (op = 2)

op3<3:0>	op3 <5:4>			
	0	**1**	**2**	**3**
0	ADD	ADDcc	TADDcc	WRY[D] (rd = 0) — (rd = 1) WRCCR (rd = 2) WRASI (rd = 3) — (rd = 4, 5) WRFPRS (rd = 6) WRPCR[PPCR] (rd = 16) WRPIC[PPIC] (rd = 17) WRDCR[P] (rd = 18) WRGSR (rd = 19) WRSOFTINT_SET[P] (rd = 20) WRSOFTINT_CLR[P] (rd = 21) WRSOFTINT[P] (rd = 22) WRTICK_CMPR[P] (rd = 23) WRSTICK[P] (rd = 24) WRSTICK_CMPR[P] (rd = 25) WRXAR (rd = 29) WRXASR (rd = 30) WRTXAR[P] (rd = 31) SIR (rd = 15, rs1 = 1, 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 E-5
5	ANDN	ANDNcc	SLL (x = 0), SLLX (x = 1)	FPop2 – See TABLE E-6
6	ORN	ORNcc	SRL (x = 0), SRLX (x = 1)	IMPDEP1 (VIS) – See TABLE E-12 and TABLE E-13
7	XNOR	XNORcc	SRA (x = 0), SRAX (x = 1)	IMPDEP2 (FMADD/SUB, etc.) – See TABLE E-14

TABLE E-3 op3<5:0> (op = 2)

op3<3:0>	op3 <5:4>			
	0	**1**	**2**	**3**
8	ADDC	ADDCcc	RDY[D] (rs1 = 0) — (rs1 = 1) RDCCR (rs1 = 2) RDASI (rs1 = 3) RDTICKP[NPT] (rs1 = 4) RDPC (rs1 = 5) RDFPRS (rs1 = 6) RDPCR[PPCR] (rs1 = 16) RDPIC[PPIC] (rs1 = 17) RDDCR[P] (rs1 = 18) RDGSR (rs1 = 19) RDSOFTINT[P] (rs1 = 22) RDTICK_CMPR[P] (rs1 = 23) RDSTICK[PNPT] (rs1 = 24) RDSTICK_CMPR[P] (rs1 = 25) RDXASR (rs1 = 30) RDTXAR[P] (rs1 = 31) 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 E-7
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 E-8	–

TABLE E-4 op3<5:0> (op = 3)

op3<3:0>	op3 <5:4>			
	0	**1**	**2**	**3**
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]

TABLE E-4 op3<5:0> (op = 3) *(Continued)*

op3<3:0>	op3 <5:4>			
	0	**1**	**2**	**3**
4	STW	STWA$^{P_{ASI}}$	STF	STFA$^{P_{ASI}}$
5	STB	STBA$^{P_{ASI}}$	STFSR^{D}, STXFSR	—
6	STH	STHA$^{P_{ASI}}$	STQF	STQFA$^{P_{ASI}}$
7	STD^{D}	STDA$^{P_{ASI}}$	STDF	STDFA$^{P_{ASI}}$
8	LDSW	LDSWA$^{P_{ASI}}$	—	—
9	LDSB	LDSBA$^{P_{ASI}}$	—	—
A	LDSH	LDSHA$^{P_{ASI}}$	—	—
B	LDX	LDXA$^{P_{ASI}}$	—	—
C	—	—	STFR	CASA$^{P_{ASI}}$
D	LDSTUB	LDSTUBA$^{P_{ASI}}$	PREFETCH	PREFETCHA$^{P_{ASI}}$
E	STX	STXA$^{P_{ASI}}$	—	CASXA$^{P_{ASI}}$
F	SWAP^{D}	SWAPA$^{D, P_{ASI}}$	STDFR	—

LDQF, LDQFA, STQF, STQFA, and the *reserved* (—) opcodes cause an *illegal_instruction* trap on a SPARC64 VIIIfx processor.

TABLE E-5 opf<8:0> (op = 2, op3 = 34_{16} = FPop1)

opf<8:3>	opf<2:0>							
	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

TABLE E-5 opf<8:0> (op = 2, op3 = 34_{16} = FPop1) *(Continued)*

opf<8:3>	opf<2:0>							
	0	1	2	3	4	5	6	7
$0A_{16}$	—	—	—	—	—	—	—	—
$0B_{16}$	—	—	—	—	—	—	—	—
$0C_{16}$	—	—	—	—	—	—	—	—
$0D_{16}$	—	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 and *reserved* (—) opcodes cause an *fp_exception_other* trap with ftt = *unimplemented_FPop* on a SPARC64 VIIIfx processor.

TABLE E-6 opf<8:0> (op = 2, op3 = 35_{16} = FPop2)

opf<8:4>	opf<3:0>								8–F
	0	1	2	3	4	5	6	7	
00_{16}	—	FMOVs (fcc0)	FMOVd (fcc0)	FMOVq (fcc0)	—	†	†	†	—
01_{16}	—	—	—	—	—	—	—	—	—
02_{16}	—	—	—	—	—	FMOVsZ	FMOVdZ	FMOVqZ	—
03_{16}	—	—	—	—	—	—	—	—	—
04_{16}	—	FMOVs (fcc1)	FMOVd (fcc1)	FMOVq (fcc1)	—	FMOVsLEZ	FMOVdLEZ	FMOVqLEZ	—

TABLE E-6 `opf<8:0>` (`op` = 2, `op3` = 35_{16} = `FPop2`) *(Continued)*

opf<8:4>	opf<3:0>								
	0	1	2	3	4	5	6	7	8–F
05_{16}	—	FCMPs	FCMPd	FCMPq	—	FCMPEs	FCMPEd	FCMPEq	—
06_{16}	—	—	—	—	—	FMOVsLZ	FMOVdLZ	FMOVqLZ	—
07_{16}	—	—	—	—	—	—	—	—	—
08_{16}	—	FMOVs (fcc2)	FMOVd (fcc2)	FMOVq (fcc2)	—	†	†	†	—
09_{16}	—	—	—	—	—	—	—	—	—
$0A_{16}$	—	—	—	—	—	FMOVsNZ	FMOVdNZ	FMOVqNZ	—
$0B_{16}$	—	—	—	—	—	—	—	—	—
$0C_{16}$	—	FMOVs (fcc3)	FMOVd (fcc3)	FMOVq (fcc3)	—	FMOVsGZ	FMOVdGZ	FMOVqGZ	—
$0D_{16}$	—	—	—	—	—	—	—	—	—
$0E_{16}$	—	—	—	—	—	FMOVsGEZ	FMOVdGEZ	FMOVqGEZ	—
$0F_{16}$	—	—	—	—	—	—	—	—	—
10_{16}	—	FMOVs (icc)	FMOVd (icc)	FMOVq (icc)	—	—	—	—	—
11_{16}–17_{16}	—	—	—	—	—	—	—	—	—
18_{16}	—	FMOVs (xcc)	FMOVd (xcc)	FMOVq (xcc)	—	—	—	—	—
19_{16}–$1F_{16}$	—	—	—	—	—	—	—	—	—

†Reserved variation of `FMOVR`

Shaded and *reserved* (—) opcodes cause an *fp_exception_other* trap with `ftt` = *unimplemented_FPop* on a SPARC64 VIIIfx processor.

TABLE E-7 `cond<3:0>`

cond<3:0>	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 = $3A_{16}$
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

TABLE E-7 cond<3:0>

cond<3:0>	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 = $3A_{16}$
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 E-8 Encoding of rcond<2:0> Instruction Field

		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 E-9 cc / opf_cc Fields (MOVcc and FMOVcc)

opf_cc			Condition Code Selected
cc2	cc1	cc0	
0	0	0	fcc0
0	0	1	fcc1
0	1	0	fcc2
0	1	1	fcc3
1	0	0	icc
1	0	1	—
1	1	0	xcc
1	1	1	—

TABLE E-10 cc Fields (FBPfcc, FCMP, and FCMPE)

cc1	cc0	Condition Code Selected
0	0	fcc0
0	1	fcc1
1	0	fcc2
1	1	fcc3

TABLE E-11 cc Fields (BPcc and Tcc)

cc1	cc0	Condition Code Selected
0	0	icc
0	1	—
1	0	xcc
1	1	—

TABLE E-12 IMPDEP1: opf<8:0> for VIS opcodes (op = 2, op3 = 36_{16}), where $0 \le$ opf<8:4> ≤ 7

	opf<8:4>							
opf<3:0>	**00_{16}**	**01_{16}**	**02_{16}**	**03_{16}**	**04_{16}**	**05_{16}**	**06_{16}**	**07_{16}**
0_{16}	EDGE8	ARRAY8	FCMPLE16	—	—	FPADD16	FZERO	FAND
1_{16}	EDGE8N	—	—	FMUL 8x16	—	FPADD16S	FZEROS	FANDS
2_{16}	EDGE8L	ARRAY16	FCMPNE16	—	—	FPADD32	FNOR	FXNOR
3_{16}	EDGE8LN	—	—	FMUL 8x16AU	—	FPADD32S	FNORS	FXNORS
4_{16}	EDGE16	ARRAY32	FCMPLE32	—	—	FPSUB16	FANDNOT2	FSRC1
5_{16}	EDGE16N	—	—	FMUL 8x16AL	—	FPSUB16S	FANDNOT2S	FSRC1S
6_{16}	EDGE16L	—	FCMPNE32	FMUL 8SUx16	—	FPSUB32	FNOT2	FORNOT2
7_{16}	EDGE16LN	—	—	FMUL 8ULx16	—	FPSUB32S	FNOT2S	FORNOT2S

TABLE E-12 IMPDEP1: opf<8:0> for VIS opcodes (op = 2, op3 = 36_{16}), where $0 \le$ opf<8:4> ≤ 7

opf<3:0>	opf<8:4>							
	00_{16}	01_{16}	02_{16}	03_{16}	04_{16}	05_{16}	06_{16}	07_{16}
8_{16}	EDGE32	ALIGN ADDRESS	FCMPGT16	FMULD 8SUx16	FALIGNDATA	—	FANDNOT1	FSRC2
9_{16}	EDGE32N	BMASK	—	FMULD 8ULx16	—	—	FANDNOT1S	FSRC2S
A_{16}	EDGE32L	ALIGN ADDRESS _LITTLE	FCMPEQ16	FPACK32	—	—	FNOT1	FORNOT1
B_{16}	EDGE32LN	—	—	FPACK16	FPMERGE	—	FNOT1S	FORNOR1S
C_{16}	—	—	FCMPGT32	—	BSHUFFLE	—	FXOR	FOR
D_{16}	—	—	—	FPACKFIX	FEXPAND	—	FXORS	FORS
E_{16}	—	—	FCMPEQ32	PDIST	—	—	FNAND	FONE
F_{16}	—	—	—	—	—	—	FNANDS	FONES

TABLE E-13 IMPDEP1: opf<8:0> for VIS opcodes (op = 2, op3 = 36_{16}), where $08_{16} \le$ opf<8:4> $\le 1F_{16}$

opf<3:0>	opf<8:4>				
	08_{16}	09_{16}–15_{16}	16_{16}	17_{16}	18_{16}–$1F_{16}$
0_{16}	SHUTDOWN	—	FCMPEQd	FMAXd	—
1_{16}	SIAM	—	FCMPEQs	FMAXs	—
2_{16}	SUSPEND^{P}	—	FCMPEQEd	FMINd	—
3_{16}	SLEEP	—	FCMPEQEs	FMINs	—
4_{16}	—	—	FCMPLEEd	FRCPAd	—
5_{16}	—	—	FCMPLEEs	FRCPAs	—
6_{16}	—	—	FCMPLTEd	FRSQRTAd	—
7_{16}	—	—	FCMPLTEs	FRSQRTAs	—
8_{16}	—	—	FCMPNEd	FTRISSELd	—
9_{16}	—	—	FCMPNEs	—	—
A_{16}	—	—	FCMPNEEd	FTRISMULd	—
B_{16}	—	—	FCMPNEEs	—	—
C_{16}	—	—	FCMPGTEd	—	—
D_{16}	—	—	FCMPGTEs	—	—

TABLE E-13 IMPDEP1: opf<8:0> for VIS opcodes (op = 2, op3 = 36_{16}), where $08_{16} \le$ opf<8:4> $\le 1F_{16}$

opf<3:0>	opf<8:4>				
	08_{16}	09_{16}–15_{16}	16_{16}	17_{16}	18_{16}–$1F_{16}$
E_{16}	—	—	FCMPGEEd	—	—
F_{16}	—	—	FCMPGEEs	—	—

TABLE E-14 IMPDEP2 (op = 2, op3 = 37_{16})

size	var			
	00_{02}	01_{02}	10_{02}	11_{02}
00_{02}	FPMADDX	FPMADDXHI	FTRIMADDd	FSELMOVd
01_{02}	FMADDs	FMSUBs	FNMSUBs	FNMADDs
10_{02}	FMADDd	FMSUBd	FNMSUBd	FNMADDd
11_{02}	(reserved for quad operations)			FSELMOVs

F.APPENDIX **F**

Memory Management Unit

This appendix defines the implementation-dependent features of the SPARC64 VIIIfx MMU and also describes features added in SPARC64 VIIIfx. Parts of the SPARC64 VIIIfx MMU are not JPS1-compatible. Refer to the following sections for details:

- Section F.4, *"Hardware Support for TSB Access"*
- Section F.10, *"Internal Registers and ASI Operations"*

F.1 Virtual Address Translation

IMPL. DEP. #222: TLB organization is JPS1 implementation dependent.

SPARC64 VIIIfx has the following 2-level TLB organization:

- Level-1 micro-ITLB (uITLB), fully associative
- Level-1 micro-DTLB (uDTLB), fully associative
- Level-2 IMMU-TLB, whichy consists of the sITLB (set-associative Instruction TLB) and fITLB (fully-associative Instruction TLB).
- Level-2 DMMU-TLB, which consists of the sDTLB (set-associative Data TLB) and fDTLB (fully-associative Data TLB).

TABLE F-1 describes the structure of SPARC64 VIIIfx TLBs.

The micro-ITLB and micro-DTLB are used as temporary memory by the corresponding main TLBs, that is, the IMMU-TLB and DMMU-TLB. The contents of the micro-TLBs are a subset of the contents of the main TLBs, and hardware maintains coherency between the micro-TLBs and main TLBs.

The micro-TLBs cannot be managed directly by software and do not affect the behavior of software, except in the case of TLB multiple-hit detection. This behavior is described below; micro-TLBs are not discussed further in this document.

TABLE F-1 Structure of SPARC64 VIIIfx TLBs

Feature	sITLB and sDTLB	fITLB and fDTLB
Entries	256 (sITLB), 512 (sDTLB)	16
Associativity	2-way set associative	Fully associative
Locked entries	Not supported	Supported
Page size	2 page sizes	All page sizes

IMPL. DEP. #223: Whether TLB multiple-hit detection is supported in a JPS1 processor is implementation dependent.

The SPARC64 VIIIfx MMU supports TLB multiple-hit detection when a multiple hit occur in the fTLB of a main TLB. A multiple hit in an fTLB is not detected if a hit occurs in the corresponding micro-TLB. See Appendix F.5.2 for details.

F.2 Translation Table Entry (TTE)

The Translation Table Entry (TTE) holds the virtual-to-physical mapping for a single page, as well as the attributes of that page. The TTE is divided into two 64-bit data representing the tag and data of the translation. When the translation tag is matched, the translation data is used to perform the address translation.

In SPARC JPS1, a TTE is an entry of the TSB. Additionally, both the TLB Data In Register and Data Out Register use the TTE format. SPARC64 VIIIfx does not provide hardware support for TSB access but does use the TTE format for TLB entries. The JPS1 definitions of the TTE are shown in FIGURE F-1 and TABLE F-2.

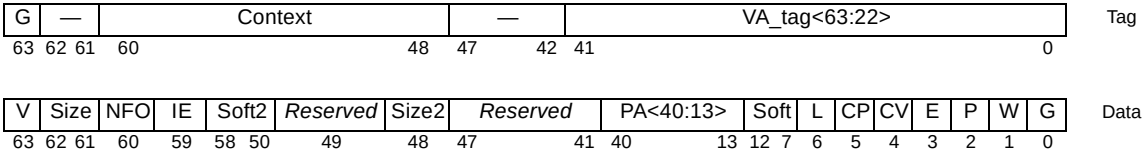


FIGURE F-1 Translation Table Entry (TTE)

TABLE F-2 TTE Bit Description *(1 of 3)*

Bits	Field	Description
Tag – 63	G	Global. If the Global bit is set, the Context field of the TLB entry is ignored during hit detection. This behavior allows any page to be shared among all (user or supervisor) contexts running in the same processor. The Global bit is duplicated in the TTE tag and data to optimize the software miss handler.
Tag – 60:48	Context	The 13-bit context identifier associated with the TTE.
Tag – 41:0	VA_tag	Virtual Address Tag. The virtual page number.
Data – 63	V	Valid. If the Valid bit is set, then the remaining fields of the TTE are meaningful. Note that the explicit Valid bit is redundant with the software convention of encoding an invalid TTE with an unused context. The encoding of the context field is necessary to cause a failure in the TTE tag comparison, and the explicit Valid bit in the TTE data simplifies the TLB miss handler.
Data – 62:61	Size	The 3-bit value formed by the concatenation of size2 and size encodes the page size. **Size2 Size<1:0> Page Size** 000_2 8 Kbyte 001_2 64 Kbyte 010_2 512 Kbyte 011_2 4 Mbyte 100_2 32 Mbyte 101_2 256 Mbyte 110_2 2 Gbyte
Data – 60	NFO	No Fault Only. If the no-fault-only bit is set, loads with ASI_PRIMARY_NO_FAULT, ASI_SECONDARY_NO_FAULT, and their *_LITTLE variations are translated. Any other access will trap with a *data_access_exception* trap (FT = 10_{16}). The NFO bit in the IMMU is read as 0 and ignored when written. The ITLB-miss handler should generate an error if this bit is set before the TTE is loaded into the TLB.
Data – 59	IE	Invert Endianness. If this bit is set for a page, accesses to the page are processed with inverse endianness from that specified by the instruction (big for little, little for big). See Section F.7 of JPS1 **Commonality** for details. The IE bit in the IMMU is read as 0 and ignored when written. **Note:** This bit is intended to be set primarily for noncacheable accesses. The performance of cacheable accesses will be degraded as if the access missed the D-cache.
Data - 58:50	Soft2	Software-defined field, provided for use by the operating system. Hardware is not required to maintain this field in the TLB, so when it is read from the TLB, it may read as zero.
Data – 49	*Reserved*	*Reserved*, read as 0.
Data – 48	Size2	See the description of the size field.

TABLE F-2 TTE Bit Description *(2 of 3)*

Bits	Field	Description
Data – 47:41	*Reserved*	*Reserved*, read as 0.
Data – 40:13	PA	The physical page number. Page offset bits for larger page sizes (such as PA<15:13>, PA<18:13>, and PA<21:13> for 64-Kbyte, 512-Kbyte, and 4-Mbyte pages, respectively) are ignored during normal translation. SPARC64 VIIIfx supports a physical address width of 41 bits. This differs from JPS1 **Commonality**.(*impl.dep.#224*) When an entry is read from the TLB, the value returned for the PA page offset bits is undefined. The value returned for the VA page offset bits is undefined for pages larger than 8KB. (*impl.dep.#238*)
Data – 12:7	Soft	Software-defined field, provided for use by the operating system. Hardware is not required to maintain this field in the TLB, so when it is read from the TLB, it may read as zero.
Data – 6	L	Lock. If the lock bit is set, then the TTE entry will be “locked down” when it is loaded into the TLB; that is, if this entry is valid, it will not be replaced by the automatic replacement algorithm invoked by an ASI store to the Data In Register. The lock bit has no meaning for an invalid entry. Software must ensure that at least one entry is not locked when replacing a TLB entry. When a write occurs via TLB Data In, SPARC64 VIIIfx automatically determines whether the entry is locked. If TTE.L = 1, the fTLB is written. If TTE.L = 0, either the fTLB or the sTLB is written depending on the page size. (*impl.dep.#225*)In SPARC64 VIIIfx, both the fITLB and fDTLB implement the lock bit. The sITLB and sDTLB do not implement the lock bit; writes to the field are ignored, and reads return 0. (*impl.dep.#242*)
Data – 5 Data – 4	CP, CV	The cacheable-in-physically-indexed-cache and cacheable-in-virtually-indexed-cache bits indicate whether the page is cacheable. When CP = 1, data is cached in the I1, D1, and U2 caches. None of the SPARC64 VIIIfx TLBs implement the CV bit. SPARC64 VIIIfx supports hardware unaliasing for the caches. Writes to the CV bit are ignored, and reads return 0. (*impl.dep.#226*)
Data – 3	E	Side effect. If the side-effect bit is set, nonfaulting loads will trap for addresses within the page, noncacheable memory accesses other than block loads and stores are strongly ordered against other E-bit accesses, and noncacheable stores are not merged. This bit should be set for pages that map I/O devices having side effects. The E bit in the IMMU is read as 0 and ignored when written. **Note:** The E bit does not force a noncacheable access. It is expected, but not required, that the CP bit will be set to 0 when the E bit is set. If both the CP bit and the E bit are set to 1, the result is undefined. **Note:** The E bit and the NFO bit are mutually exclusive; both bits should never be set.

TABLE F-2 TTE Bit Description *(3 of 3)*

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

F.4 Hardware Support for TSB Access

In JPS1 **Commonality**, the TSB is managed by software. On a TLB miss, hardware computes the pointer to the TSB entry that is thought to contain the missing VA. However, the formation of TSB Pointers can be easily performed using simple integer instructions. Furthermore, JPS1 **Commonality** only provides TSB hardware support for 8KB and 64KB pages; no support is provided for larger page sizes. For these reasons, SPARC64 VIIIfx does not implement hardware support for TSB access.

SPARC64 VIIIfx does implement the TSB Base Register. On a TLB miss, system software can obtain the base address of the TSB from the TSB Base Register instead of from memory. Thus, the only overhead on a TLB miss are the few instructions required to compute the TSB pointer; performance should be relatively unchanged compared to previous processors. Refer to Section F.10.6 for details on the TSB Base Register.

F.5 Faults and Traps

IMPL. DEP. #230: The cause of a *data_access_exception* trap is implementation dependent in JPS1, but there are several mandatory causes of a *data_access_exception* trap.

SPARC64 VIIIfx signals a *data_access_exception* for the conditions defined in Section F.5 of JPS1 **Commonality**. However, caution is needed when dealing with an invalid ASI. See Section F.10.9, *"I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR)"*, for details.

IMPL. DEP. #237: Whether the fault status and/or address (DSFSR/DSFAR) are captured when *mem_address_not_aligned* is generated during a JMPL or RETURN instruction is implementation dependent.

On SPARC64 VIIIfx, the fault status and address (DSFSR/DSFAR) are not captured when a *mem_address_not_aligned* exception is generated during a JMPL or RETURN instruction.

In SPARC64 VIIIfx, additional traps are recorded by the MMU: *instruction_access_error*, *data_access_error*, and *SIMD_load_across_pages*. TABLE F-3 reproduces TABLE F-2 of JPS1 **Commonality** and adds information on these additional MMU traps.

TABLE F-3 MMU Trap Types, Causes, and Stored State Register Update Policy

Ref #	Trap Name	Trap Cause	Registers Updated (Stored State in MMU) I-SFSR	I-MMU Tag Access	D-SFSR, SFAR	D-MMU Tag Access[1]	Trap Type
1.	*fast_instruction_access_MMU_miss*	I-TLB miss	X[2]	X			64_{16}–67_{16}
2.	*instruction_access_exception*	Several (see below)	X[2]	X			08_{16}
3.	*fast_data_access_MMU_miss*	D-TLB miss			X[3]	X	68_{16}–$6B_{16}$
4.	*data_access_exception*	Several (see below)			X[3]	X[4]	30_{16}
5.	*fast_data_access_protection*	Protection violation			X[3]	X	$6C_{16}$-$6F_{16}$
6.	*privileged_action*	Use of privileged ASI			X[3]		37_{16}
7.	watchpoint	Watchpoint hit			X[3]		61_{16}–62_{16}
8.	*mem_address_not_aligned*, **_mem_address_not_aligned*	Misaligned memory operation			(impl. dep #237)		35_{16}, 36_{16}, 38_{16}, 39_{16}
9.	*instruction_access_error*	Several (see below)	X[2]				$0A_{16}$
10	*data_access_error*	Several (see below)			X[3]		32_{16}
11	*SIMD_load_across_pages*	D-TLB miss on extended portion of SIMD load			X[3]		77_{16}

1.Includes TAG_ACCESS_EXT_REG.

2.See Section F.10.9 for deatils on I-SFSR.

3.See Section F.10.9 for details on D-SFSR and D-SFAR

4.After a *data_access_exception* is signalled, the context field of the D-MMU Tag Access Register is undefined.

A *data_access_error* trap caused by a bus error or bus timeout has the lowest priority of all level-12 traps.

Ref #1~8 in TABLE F-3 conform to the definitions in Section F.5 of JPS1 **Commonality**. Ref #9, #10, and #11 are described below.

Ref 9: instruction_access_error — Signalled upon detection of at least one of the following exceptional conditions.

- An uncorrectable error is detected on an instruction fetch.
- A bus error is generated by an instruction fetch memory reference.
- A fITLB multiple hit is detected.

Ref 10: data_access_error — Signalled upon the detection of at least one of the following exceptional conditions.

- An uncorrectable error is detected on a data access.
- A bus timeout is generated by a data access memory reference.
- A fDTLB multiple hit is detected.

Note – SPARC64 VIIIfx implements a store buffer, so there are cases where a *data_access_error* is not signalled for a read from a given address. See Section P.7.1 for details.

Ref 11: SIMD_load_across_pages — Signalled when the extended operation of a SIMD load causes a TLB miss. The DSFAR displays the address of the extended operation.

Programming Note – When *SIMD_load_across_pages* is signalled, system software should emulate the operation instead of updating the TLB. Because the TLB does not need to be updated, the TAG_ACCESS_REG is not updated. See Section 7.6.5.

F.5.1 Trap Conditions for SIMD Load/Store

The priority of SIMD load/store exceptions are specified in TABLE 7-2. Priorities are assigned such that when exceptions are signalled, it appears as if the basic operation is processed before the extended operation. The DSFSR and DSFAR display information on whichever operation caused the exception.

Note – The *SIMD_load_across_pages* exception is caused by the extended operation.

In some cases, a *VA_watchpoint* exception caused by the extended operation takes priority over any level-12 exceptions (*fast_data_MMU_miss*, *data_access_exception*, *fast_data_access_protection*, *data_access_error*, *data_access_protection*) caused by the basic operation.

F.5.2 Behavior on TLB Error

SPARC64 VIIIfx signals a *data_access_error* exception when a multiple hit is detected in the fTLB. Software is not notified of a multiple hit in the sTLB; instead, the entries are invalidated. When a parity error is discovered while the TLB is being searched, the entry is invalidated (sTLB) or automatically corrected (fTLB); software is not notified. All traps must occur in program order, but invalidation and automatic correction occur when the error is detected; that is, these actions are also performed when errors are detected during speculative execution of memory accesses.

TABLE F-4 shows the behavior of SPARC64 VIIIfx when a parity error or multiple hit occurs in the TLB.

TABLE F-4 Behavior on Detection of a Parity Error or a Multiple Hit

Parity Error		Multiple Hit		
sTLB	**fTLB**	**sTLB**	**fTLB**	**Behavior**
✓				Entry is invalidated, and a *fast_instruction_access_MMU_miss* or *fast_data_access_MMU_miss* is signalled.
	✓			Automatic correction.[1] Not visible to software.
✓	✓			The fTLB entry is automatically corrected[1], and the sTLB entry is invalidated.
		✓		Entries are invalidated, and a *fast_instruction_access_MMU_miss* or *fast_data_access_MMU_miss* is signalled.
			✓	An *instruction_access_error* or *data_access_error* is signalled.[2]
		✓	✓	The multiple hit is not detected and the contents of the sTLB are used.[3]
✓		✓		All entries where a multiple hit or parity error occur are invalidated.
	✓	✓		The fTLB entry is automatically corrected,[1] and the sTLB entries are invalidated.
✓			✓	The sTLB entry is invalidated, and the multiple hit[2] in the fTLB causes an *instruction_access_error* or *data_access_error.*
	✓		✓	The entry containing the parity error is automatically corrected, and the multiple hit causes a *instruction_access_error* or *data_access_error.*

1.The fTLB is duplicated, so the error is correctable. If it cannot be corrected, the error is fatal.

2.There are cases where a multiple hit in the fTLB is not detected.

3.When a multiple hit occurs between the sTLB and fTLB.

When a parity error or multiple hit occurs for a sTLB entry, the entry is invalidated. Software is not notified of this action. For a SIMD load, however, the sTLB entry needed by the extended load may be invalidated during the search of the TLB by the basic load due to a parity error or multiple hit. In this case, an exception of the form *SIMD_load_across_pages* is signalled.

A parity error or multiple hit can be detected at the same time as any of the exceptions listed in TABLE F-3; invalidating a TLB entry does not affect whether other exceptions are detected. That is, when a parity error or multiple hit caused by speculative execution is detected, that entry is invalidated.

Note – When a multiple hit is detected, it is impossible determine which TTE is the correct one. No TTE-dependent exceptions (*data_access_exception*, *PA_watchpoint*, *fast_data_access_protection*, *SIMD_load_across_pages*) are detected.

F.8 Reset, Disable, and RED_state Behavior

IMPL. DEP. #231: The variability of the width of physical address is implementation dependent in JPS1, and if variable, the initial width of the physical address after reset is also implementation dependent in JPS1.

See the description of the PA field in the Data section of TABLE F-2. The width of physical address in SPARC64 VIIIfx is 41 bits.

IMPL. DEP. #232: Whether CP and CV bits exist in the DCU Control Register is implementation dependent in JPS1.

SPARC64 VIIIfx does not implement the DCU Control Register. CP and CV bits do not exist.

When the DMMU is disabled, the MMU behaves as if TTE bits were set to the following:

- TTE.IE ← 0
- TTE.P ← 0
- TTE.W ← 1
- TTE.NFO ← 0
- TTE.CV ← 0
- TTE.CP ← 0
- TTE.E ← 1

IMPL. DEP. #117: Whether prefetch and nonfaulting loads always succeed when the MMU is disabled is implementation dependent.

When the DMMU is disabled in SPARC64 VIIIfx, the PREFETCH instruction completes without performing a memory access; a nonfaulting load causes a *data_access_exception* exception, as defined in Section F.5 of JPS1 **Commonality**.

F.10 Internal Registers and ASI Operations

The SPARC64 VIIIfx specification does not implement TSB hardware support. For this reason, the following registers that are defined in JPS1 **Commonality** are not implemented in SPARC64 VIIIfx.

TABLE F-5 Invalid MMU Registers in SPARC64 VIIIfx

IMMU ASI	DMMU ASI	VA	Register Name
50_{16}	58_{16}	48_{16}	Instruction/Data TSB Primary Extension Registers
—	58_{16}	50_{16}	DATA TSB Secondary Extension Register
50_{16}	58_{16}	58_{16}	I/D TSB Nucleus Extension Registers
51_{16}	59_{16}	00_{16}	I/D TSB 8KB Pointer Registers
52_{16}	$5A_{16}$	00_{16}	I/D TSB 64KB Pointer Registers
—	$5B_{16}$	00_{16}	DATA TSB Direct Pointer Register

Accesses to these ASIs and VAs cause *data_access_exception* exceptions.

F.10.1 Accessing MMU Registers

IMPL. DEP. #233: Whether the TSB_Hash field is implemented in I/D Primary/Secondary/Nucleus TSB Extension Register is implementation dependent in JPS1.

Since SPARC64 VIIIfx does not define the TSB Extension register, the above implementation dependency has no meaning.

IMPL. DEP. #239: The register(s) accessed by IMMU ASI 55_{16} and DMMU ASI $5D_{16}$ at virtual addresses 40000_{16} to $60FF8_{16}$ are implementation dependent.

See Impl. Dep. #235 in *"I/D TLB Data In, Data Access, and Tag Read Registers"* (page 192).

In addition to the registers listed in TABLE F-9 of JPS1 **Commonality**, SPARC64 VIIIfx assigns MMU functions to ASI_DCUCR (page 34) and ASI_MCNTL (page 184)

ASI_MCNTL (Memory Control Register)

Register Name	ASI_MCNTL
ASI	45_{16}
VA	08_{16}
Access Type	Supervisor read/write

Bit	Field	Access	Description
63:20	Reserved		
19		RW	
18:17	hpf	RW	Sets the hardware prefetch mode. 00_2: Hardware prefetch generates strong prefetches. 01_2: Hardware prefetches are not generated. 10_2: Hardware prefetch generates weak prefetches. 11_2: reserved When $11_2$is set, the behavior of hardware prefetch is undefined.
16	NC_Cache	RW	Force instruction caching for instructions in noncacheable address spaces. If NC_Cache is set to 1, the CPU performs a 16-byte noncacheable access 8 times, which writes a total of 128 bytes to the I1 cache. This does not affect the behavior of data accesses. NC_Cache is provided to improve the execution speed of OBP functions, and OBP should set NC_Cache to 0 when turning over control to the OS. Otherwise, noncacheable instructions may be left in the I1 cache.
15	fw_fITLB	RW	Force write to fITLB on an ITLB update. If fw_fITLB is set to 1, a TLB write using the ITLB Data In Register always writes fITLB. fw_fITLB is provided for use by OBP functions.

Bit	Field	Access	Description
14	fw_fDTLB	RW	Force write to fDTLB on a DTLB update. If fw_fDTLB is set to 1, a TLB write using the DTLB Data In Register always writes fDTLB. fw_fDTLB is provided for use by OBP functions.
13:12	RMD	R	The value of this field is always 2. This field is read-only, and writes to this field are ignored.
11:8	Reserved		
7	mpg_sITLB[1]	RW	This bit enables the multiple page size function in the sITLB. If mpg_sITLB is set to 1, the sITLB can store TTEs of a different page size per context. If mpg_sITLB is set to 0, the page size information in the context register and IMMU_TAG_ACCESS_EXT are ignored, and the default page sizes (8K for the 1st sITLB, 4M for the 2nd sITLB) are used.
6	mpg_sDTLB[1]	RW	This bit enables the multiple page size function in the sDTLB. If mpg_sDTLB is set to 1, the sDTLB can store TTEs of a different page size per context. If mpg_sDTLB is set to 0, the page size information in the context register and DMMU_TAG_ACCESS_EXT are ignored, and the default page sizes (8K for the 1st sDTLB, 4M for the 2nd sDTLB) are used.
5:0	Reserved		

1. Setting mpg_sITLB = 1 and mpg_sDTLB = 0 is not allowed. The behavior of SPARC64 VIIIfx is undefined for this combination.

F.10.2 Context Registers

sTLBs are composed of two separate 2-way set-associative TLBs. The 1st and 2nd sTLBs in the sITLB hold 128 entries each, and the sTLBs in the sDTLB hold 256 entries each. By default, the 1st sTLB only stores 8-KB page TTEs and the 2nd sTLB only stores 4-MB page TTEs. By setting `MCNTL.mpg_sITLB` and `MCNTL.mpg_sDTLB` to 1, TTEs of any one page size (8 KB, 64 KB, 512 KB, 4 MB, 32MB, 256MB, 2GB) can be stored for each context. The page sizes for the 1st and 2nd sTLBs can be set separately; both sTLBs can also be set to the same page size settings.

Page sizes are set by the fields of the Context Registers. `ASI_PRIMARY_CONTEXT_REG` fields set the page sizes for the sITLB and sDTLB; sDTLB page sizes can also be set by the `ASI_SECONDARY_CONTEXT_REG` fields. If the 1st and 2nd sTLBs have the same page size settings, the entire sTLB behaves like a single 4-way set-associative TLB.

Page sizes have the following encoding:

- 000_{02} = 8 KB
- 001_{02} = 64 KB
- 010_{02} = 512 KB
- 011_{02} = 4 MB
- 100_{02} = 32 MB
- 101_{02} = 256 MB
- 110_{02} = 2 GB

Note – When the encoding 111_{02} is specified, SPARC64 VIIIfx behavior is undefined.

In addition to the Context Registers defined in JPS1 **Commonality**, SPARC64 VIIIfx defines the Shared Context Register. The shared context is a virtual address space shared by two or more processes and can be used to hold instructions or shared data. Like the secondary context, the shared context enables access to another context from the current context, with the following differences:

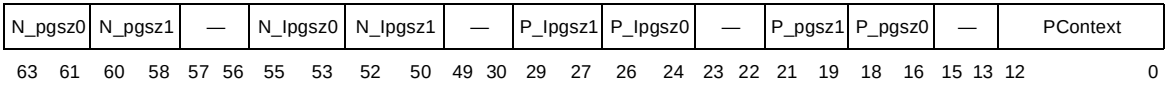
- To access the secondary context address space, an explicit ASI load/store instruction must be used. The shared context address space can be accessed implicitly, like an access to the primary context address space.
- The secondary context can only be used for data access; the shared context can be used for both instruction fetch and data access.

In the following descriptions, the term “effective context” is used. Because there are multiple context registers, the instruction and processor state determine which context register is being used; the context identifier of that context register is called the effective context.

- The effective context of an access with `TL` = 0 by instruction fetch or an implicit ASI load/store instruction is the value of `ASI_PRIMARY_CONTEXT`.
- The effective context of an access with `TL` > 0 by instruction fetch or an implicit ASI load/store instruction is the value of `ASI_NUCLEUS_CONTEXT`.
- The effective context of an explicit ASI load/store instruction is determined from the ASI.

ASI_PRIMARY_CONTEXT

Register Name	ASI_PRIMARY_CONTEXT
ASI	58_{16}
VA	08_{16}
Access Type	Supervisor read/write

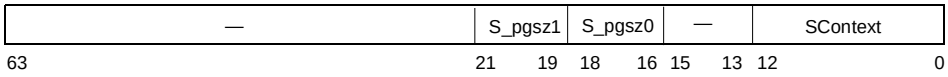


Bit	Field	Access	Description
63:61	N_pgsz0	RW	Nucleus context, page size of the 1st sDTLB.
60:58	N_pgsz1	RW	Nucleus context, page size of the 2nd sDTLB.
55:53	N_Ipgsz0	RW	Nucleus context, page size of the 1st sITLB.
52:50	N_Ipgsz1	RW	Nucleus context, page size of the 2nd sITLB.
29:27	P_Ipgsz1	RW	Primary context, page size of the 2nd sITLB.
26:24	P_Ipgsz0	RW	Primary context, page size of the 1st sITLB.
21:19	P_pgsz1	RW	Primary context, page size of the 2nd sDTLB.
18:16	P_pgsz0	RW	Primary context, page size of the 1st sDTLB.
12:0	PContext	RW	Primary context identifier.

Values written to the page size fields can always be read, regardless of the settings of ASI_MCNTL.mpg_sITLB and ASI_MCNTL.mpg_sDTLB.

ASI_SECONDARY_CONTEXT

Register Name	ASI_SECONDARY_CONTEXT
ASI	58_{16}
VA	10_{16}
Access Type	Supervisor read/write

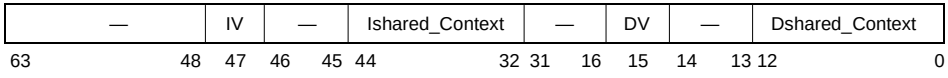


Bit	Field	Access	Description
21:19	S_pgsz1	RW	Secondary context, page size of the 2nd sDTLB.
18:16	S_pgsz0	RW	Secondary context, page size of the 1st sDTLB.
12:0	SContext	RW	Secondary context identifier.

Values written to the page size fields can always be read, regardless of the settings of ASI_MCNTL.mpg_sITLB and ASI_MCNTL.mpg_sDTLB.

ASI_SHARED_CONTEXT

Register Name	ASI_SHARED_CONTEXT
ASI	58_{16}
VA	68_{16}
Access Type	Supervisor read/write



Bit	Field	Access	Description
47	IV	RW	Ishared_Context Valid. When IV = 1 and Ishared_Context is not 0, the values of both the effective context and Ishared_Context are used in MMU translation of instruction fetches. When IV = 0 or Ishared_Context is 0, only the effective context is used.
44:32	Ishared_Context	RW	Context identifier used for instruction fetches to the shared context.
15	DV	RW	Dshared_Context Valid. When DV = 1 and Dshared_Context is not 0, the values of both the effective context and Dshared_Context are used in MMU translation of data accesses. When DV = 0 or Dshared_Context is 0, only the effective context is used.
12:0	Dshared_Context	RW	Context identifier used for data accesses to the shared context.

The ASI_SHARED_CONTEXT register indicates whether an MMU translation should be performed using both the effective context and the shared context; that is, whether the TLB is searched for entries that match either the shared context or effective context. The register also indicates the current context identifier for the shared context. When IV or DV is set to 1 and the context identifier is not 0, the register is valid. When the effective context is 0, the shared context is not used, regardless of the setting of IV or DV. For example, a load instruction to ASI_AS_IF_USER_SECONDARY while TL > 0 has an effective context of SContext. Thus, whether the shared context is used or not depends on whether or not SContext is 0.

The shared context has the same features as the effective context, except for page size settings. SPARC64 VIIIfx has two sITLBs and two sDTLBs; TTE page size settings can be set for each sTLB and for each context. However, the shared context does not have its own page size settings; page size settings for the effective context are used. When ASI_MCNTL.mpg_sI/DTLB = 0, the page size setting is 8 KB for the 1st sTLB and 4 MB for the 2nd sTLB. When ASI_MCNTL.mpg_sI/DTLB = 1, the page size setting is P_pgsz0/S_pgsz0/P_Ipgsz0 for the 1st sTLB and P_pgsz0/S_pgsz0/P_Ipgsz0 for the 2nd sTLB.

Note – N_pgsz0/1 are never used because the shared context is not valid when the effective context is 0.

Programming Note – To efficiently use the sTLBs with the shared context, set `P_pgsz(0,1)/P_Ipgsz(0,1)/S_pgsz(0,1)` to the same page size settings for all contexts that are used with the shared context.

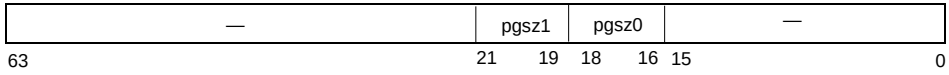
F.10.3 Instruction/Data MMU TLB Tag Access Registers

When a MMU miss or access violation causes an exception and the shared context is valid, the I/D TLB Tag Access Registers display the context ID of the effective context.

Programming Note – To store a shared context TTE in the TLB, the context ID of the shared context needs to be set in the I/D TLB Tag Access Registers before writing the I/D TLB Data In/Data Access Registers.

ASI_I/DMMU_TAG_ACCESS_EXT

Register Name	`ASI_IMMU_TAG_ACCESS_EXT`, `ASI_DMMU_TAG_ACCESS_EXT`
ASI	50_{16} (IMMU), 58_{16} (IMMU)
VA	60_{16}
Access Type	Supervisor read/write



When a MMU exception causes a trap, hardware saves the VA and context that caused the exception to the Tag Access Registers (`ASI_I/DMMU_TAG_ACCESS`), depending on the trap type. See TABLE F-3 (page 180) for details. To simplify the calculation of the sTLB index when a TTE is written to the TLB using the I/DTLB Data In Registers, SPARC64 VIIIfx saves the page size information (for the effective context) that is missing from the Tag Access Registers to the `ASI_I/DMMU_TAG_ACCESS_EXT` registers.

Note – When the page size of the TTE being written is different than the value of `ASI_I/DMMU_TAG_ACCESS_EXT.pgsz0/1`, the TTE is written into the fTLB instead of the sTLB.

When *instruction_access_exception* and *data_access_exception* exceptions are generated, the ASI_I/DMMU_TAG_ACCESS_EXT registers are not valid and the values are undefined. Also, when ASI_MCNTL.mpg_sITLB = 0, ASI_I/DMMU_TAG_ACCESS_EXT is not valid and the value is undefined. When ASI_MCNTL.mpg_sDTLB = 0, ASI_I/DMMU_TAG_ACCESS_EXT is not valid and the value is undefined

F.10.4 I/D TLB Data In, Data Access, and Tag Read Registers

IMPL. DEP. #234: The replacement algorithm of a TLB entry is implementation dependent in JPS1.

The replacement algorithm is pseudo-LRU for the fTLB and LRU for the sTLB.

IMPL. DEP. #235: The MMU TLB data access address assignment and the purpose of the address are implementation dependent in JPS1.

The MMU TLB data access address assignment and the purpose of the address in SPARC64 VIIIfx are shown in TABLE F-6.

TABLE F-6 MMU TLB Data Access Address Assignment

Bit	Field	Access	Description
17:16	TLB#	RW	Specifies the accessed TLB. 00_{02}: fTLB (16 entries) 01_{02}: reserved 10_{02}: sTLB(256 entries for IMMU, 512 for DMMU) 11_{02}: reserved
15	Reserved		
13:3	TLB index	RW	TLB index number. • For the fTLB, the lower 4 bits are the index number and the upper 7 bits are ignored. The relationship between the value of the lower 4 bits and the TLB index is as follows: 0-15: fTLB index number • For the sITLB, bits <13:12> indicate the way and bits <8:3> indicate the index. Bits <11:9> are ignored. The relationships between the value of the field and the TLB index is as follows: 0-63: 1st sITLB, way 0 index number 512-575: 1st sITLB, way 1 index number 1024-1087: 2nd sITLB, way 0 index number 1536-1599: 2nd sITLB, way 1 index number • For the sDTLB, bits <13:12> indicate the way and bits <9:3> indicate the index. Bits <11:10> are ignored. The relationships between the value of the field and the TLB index is as follows: 0-127: 1st sDTLB, way 0 index number 512-639: 1st sDTLB, way 1 index number 1024-1151: 2nd sDTLB, way 0 index number 1536-1663: 2nd sDTLB, way 1 index number

Note – For a TLB write using the I/D Data In Registers, entries with `TTE.G` = 1 are always written to the fTLB.

I/D MMU TLB Tag Read Register

IMPL. DEP. #238: When read, an implementation will return either 0 or the value previously written to them.

See the description of the PA field in TABLE F-2 (page 177).

The VA format for the TLB Tag Read Registers is the same as the VA format for the TLB Data Access Registers. See TABLE F-6 for details.

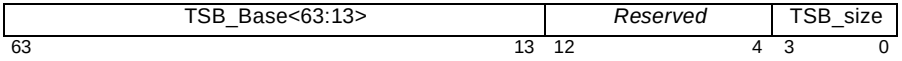
I/D MMU TLB Tag Access Register

When a TTE is written to the TLB using the I/D TLB Data Access Registers or I/D TLB Data In Registers, hardware checks that the information in the I/D TLB Tag Access Register is consistent. If the information is not consistent, the TLB is not updated.

However, when an entry with `TTE.V` = 0 is written using the I/D TLB Data Access Registers, the entry is written without checking for consistency. This allows specific TLB entires to be removed. This feature can be used to erase errors in TLB entries caused by software.

Implementation Note – Reading an entry with `TTE.V` = 0 returns all zeroes.

F.10.6 I/D TSB Base Registers



SPARC64 VIIIfx does not provide hardware support for the TSB. However, the TSB Base registers, which can be managed by system software, are implemented. JPS1 **Commonality** defines the following fields in the TSB Base Registers:

- `TSB_Base`
- `Split`
- `TSB_Size`

The SPARC64 VIIIfx TSB Base Registers implement the `TSB_Base` and `TSB_Size` fields; the `Split` field is *reserved.*

`TSB_Size` is a 4-bit field in bits <3:0> (impl.dep. #236). Values written in `TSB_Size` are returned on reads. Hardware preserves this value and but does not use it.

F.10.7 I/D TSB Extension Registers

SPARC64 VIIIfx does not support the TSB Extension Registers. An attempt to read or write these registers causes a *data_access_exception* exception.

F.10.8 I/D TSB 8-Kbyte and 64-Kbyte Pointer and Direct Pointer Registers

SPARC64 VIIIfx does not support these registers. Attempts to read or write these registers cause *data_access_exception* exceptions.

F.10.9 I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR)

IMPL. DEP. (FIGURE F-15, TABLE F-12 in Commonality): Bits <63:25> in the I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR) are implementation-dependent.

The SPARC64 VIIIfx implementation of I/D-SFSR is shown in FIGURE F-2.

FIGURE F-2 MMU I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR)

Bits <24:0> conform to JPS1 **Commonality**. The I-SFSR bits are described in TABLE F-7 and the D-SFSR bits are described in TABLE F-10.

TABLE F-7 I-SFSR Bit Description *(1 of 2)*

Bit	Field	Access	Description
63:62	TLB#	RW	Indicates that an error occured in the mITLB. In SPARC64 VIIIfx, the field always displays the value 00_{02}.
59:49	index	RW	Indicates the index number when an error occurs in the mITLB. When multiple errors occur, only one of the index numbers is shown.
46	MK	RW	Marked Uncorrectable Error. In SPARC64 VIIIfx, all uncorrectable errors are marked before being reported. When ISFSR.UE = 1, MK is always set to 1. See Appendix P.2.4 for details.
45:32	EID	RW	Error Marking ID. This field is valid when MK is 1. See Appendix P.2.4 for details.
31	UE	RW	Uncorrectable Error (UE). Setting UE = 1 indicates that there is an uncorrectable error in instruction fetch data. This bit is only valid for *instruction_access_error* exceptions.
30	BERR	RW	Indicates that the instruction fetch returned a memory bus error. This bit is only valid for *instruction_access_error* exceptions.

TABLE F-7 I-SFSR Bit Description *(2 of 2)*

Bit	Field	Access	Description
29	BRTO	RW	Indicates that the instruction fetch returned a bus timeout. This bit is only valid for *instruction_access_error* exceptions.
27:26	mITLB<1:0>	RW	mITLB Error Status. When a multiple hit is detected during a search of the mITLB, mITLB<1> is set to 1. mITLB<0> is always 0. This field is only valid for *instruction_access_error* exceptions.
25	NC	RW	Indicates that a noncacheable address space is referenced. This bit is only valid for *instruction_access_error* exceptions caused by an uncorrectable error, bus error, or bus timeout. Otherwise, the value of this bit is undefined.
23:16	ASI<7:0>	RW	Indicates the ASI number used by the access that caused the exception. This field is only valid when ISFSR.FV is set to 1. When TL = 0, the ASI displayed in this field is 80_{16}(ASI_PRIMARY). When TL > 0, the ASI is 04_{16} (ASI_NUCLEUS).
15	TM	RW	Indicates that a TLB miss occurred during the instruction fetch.
13:7	FT<6:0>	RW	Specifies the exact condition that caused the exception. SeeTABLE F-8 for the encoding of this field. This field is only valid for *instruction_access_exception* exceptions. It always reads as 0 for *fast_instruction_access_MMU_miss* exceptions and reads as 01_{16} for *instruction_access_exception* exceptions.
5:4	CT<1:0>	RW	Indicates the Context Register selection of the instruction fetch that caused the exception, as described below. The context is set to 11_{02} when the access ASI is not a translating ASI, or is an invalid ASI. 00_{02}: Primary 01_{02}: Reserved 10_{02}: Nucleus 11_{02}: Reserved Note that an encoding for the Shared Context is not defined. When a multiple hit involving a shared context is detected, information on the effective context is displayed.
3	PR	RW	Indicates that the faulting instruction fetch occurred while in privileged mode. This field is only valid when ISFSR.FV = 1.
1	OW	RW	Indicates that the exception was detected while ISFSR.FV= 1. This bit is set to 1 when ISFSR.FV = 1 and 0 when ISFSR.FV = 0.
0	FV	RW	Fault Valid. This bit is set to 1 when an exception other than a TLB miss exception occurs in the IMMU. When this bit is 0, the values of the other fields in the ISFSR have no meaning, except in the case of a MMU miss.

TABLE F-8 describes the encoding of the ISFSR.FT field.

TABLE F-8 Instruction Synchronous Fault Status Register FT (Fault Type) Field

FT<6:0>	Fault Type
01_{16}	Privilege violation. Indicates that TTE.P = 1 and PSTATE.PRIV = 0 for the instruction fetch. A privilege violation is signalled by an *instruction_access_exception* exception.
02_{16}	Reserved

TABLE F-8 Instruction Synchronous Fault Status Register FT (Fault Type) Field

FT<6:0>	Fault Type
04_{16}	Reserved
08_{16}	Reserved
10_{16}	Reserved
20_{16}	Reserved
40_{16}	Reserved

I-SFSR is updated when a *fast_instruction_access_MMU_miss*, *instruction_access_exception*, or *instruction_access_error* exception occurs. TABLE F-9 shows which fields are updated by each exception.

TABLE F-9 I-SFSR Update Policy

Field		TLB#, index	FV	OW	PR, CT[1]	FT	TM	ASI	UE, BERR, BRTO, mITLB, NC[2]
When I-SFSR.OW = 0, 0: 0 is set. 1: 1 is set. V: A valid value is set. —: Invalid field.									
Miss:	*fast_instruction_access_MMU_miss*	—	0	0	V	—	1	—	—
Exception:	*instruction_access_exception*	—	1	0	V	V	0	V	—
Error:	*instruction_access_error*	V[3]	1	0	V	—	0	V	V
When I-SFSR.OW = 1, 0: 0 is set. 1: 1 is set. K: Original value is preserved. U: Updated.									
Error on exception		U[3]	1	1	U	K	K	U	U
Exception on error		K	1	1	U	U	K	U	K
Error on miss		U[3]	1	K	U	K	1	U	U
Exception on miss		K	1	K	U	U	1	U	K
Miss on exception/error		K	1	K	K	K	1	K	K
Miss on miss		K	K	K	U	K	1	K	K

1.The value of ISFSR.CT is 11_{02} when the ASI is not a translating ASI, or is an invalid ASI.

2.Only valid for an *instruction_access_error* caused by an uncorrectable error, a bus error, or a bus timeout.

3.Only when there is a multiple hit in the TLB.

TABLE F-10 D-SFSR Bit Description *(1 of 2)*

Bit	Field	Access	Description
63:62	TLB#	RW	Indicates that an error occured in the mDTLB. In SPARC64 VIIIfx, the field always displays the value 00_{02}.
59:49	index	RW	Indicates the index number when an error occurs in the mDTLB. When multiple errors occur, only one of the index numbers is shown.
46	MK	RW	Marked Uncorrectable Error. In SPARC64 VIIIfx, all uncorrectable errors are marked before being reported. When DSFSR.UE = 1, MK is always set to 1. See Appendix P.2.4 for details.
45:32	EID	RW	Error Marking ID. This field is valid when MK is 1. See Appendix P.2.4 for details.
31	UE	RW	Uncorrectable Error (UE). Setting UE = 1 indicates that there is an uncorrectable error in the access data. This bit i only valid for *data_access_error* exceptions.
30	BERR	RW	Indicates that the data access returned a memory bus error. This bit is only valid for *data_access_error* exceptions.
29	BRTO	RW	Indicates that the data access returned a bus timeout. This bit is only valid for *data_access_error* exceptions.
27:26	mDTLB<1:0>	RW	mDTLB Error Status. When a multiple hit is detected during a search of the mDTLB, mDTLB<1> is set to 1. mDTLB<0> is always 0. This field is only valid for *data_access_error* exceptions.
25	NC	RW	Indicates that a noncacheable address space is referenced. This bit is only valid for *data_access_error* exceptions caused by an uncorrectable error, bus error, or bus timeout. Otherwise, the value of this bit is undefined.
24	NF	RW	Indicates that a nonfaulting load instruction caused the exception.
23:16	ASI<7:0>	RW	Indicates the ASI number used by the access that caused the exception. This field is only valid when DSFSR.FV is set to 1. If the data access does not explicitly specify the ASI used, an implicit ASIs is used; this field is set to one of the following values: TL = 0, PSTATE.CLE = 0 80_{16} (ASI_PRIMARY) TL = 0, PSTATE.CLE = 1 88_{16} (ASI_PRIMARY_LITTLE) TL > 0, PSTATE.CLE = 0 04_{16} (ASI_NUCLEUS) TL > 0, PSTATE.CLE = 1 $0C_{16}$ (ASI_NUCLEUS_LITTLE)
15	TM	RW	Indicates that a TLB miss occurred during the data access.
13:7	FT<6:0>	RW	Specifies the exact condition that caused the exception. See TABLE F-11 for the encoding of this field.
6	E	RW	Indicates an access to a page with side effects. E is set to 1 when an exception is caused by an access to a page with TTE.E = 1 or by an access to ASI 15_{16} or $1D_{16}$. This bit is only valid for *data_access_error* exceptions caused by an uncorrectable error, bus error, or bus timeout. Otherwise, the value of this bit is undefined.

TABLE F-10 D-SFSR Bit Description *(2 of 2)*

Bit	Field	Access	Description
5:4	CT<1:0>	RW	Indicates the Context Register selection of the data access that caused the exception, as described below. The context is set to 11_{02} when the access ASI is not a translating ASI, or is an invalid ASI. 00_{02}:Primary 01_{02}: Secondary 10_{02}: Nucleus 11_{02}: Reserved When a *data_access_exception* trap is caused by an invalid ASI and instruction combination (i.e., atomic quad load, block load/store, block commit store, partial store, short floating-point load/store, and xfill ASIs that can only be used with specified memory access instructions), CT indicates the context of the ASI specified by the instruction. Note that an encoding for the Shared Context is not defined. When a multiple hit involving a shared context is detected, information on the effective context is displayed.
3	PR	RW	Indicates the faulting data access occurred while in privileged mode. This field is only valid when FV = 1.
2	W	RW	Indicates that a write instruction (store or atomic load/store instruction) caused the exception.
1	OW	RW	Indicates that the exception was detected while DSFSR.FV = 1. This bit is set to 1 when DSFSR.FV = 1 and 0 when DSFSR.FV = 0.
0	FV	RW	Fault Valid. This is set when an exception other than a TLB miss occurs in the DMMU. When this bits is 0, the values of the other fields in the DSFSR have no meaning, except in the case of a MMU miss.

TABLE F-11 defines the encoding of the DSFSR.FT field.

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

FT<6:0>	Fault Type
01_{16}	Privilege violation. Indicates an attempt to access a page with TTE.P = 1 while PSTATE.PRIV = 0 or using ASI_PRIMARY/ SECONDARY_AS_IF_USER{_LITTLE}. A privilege violation is signalled by a *data_access_exception* exception.
02_{16}	FT<1> is set to 1 when a nonfaulting load accesses a page with TTE.E = 1.
04_{16}	FT<2> is set to 1 when an atomic instruction (CASA, CASXA, SWAP, SWAPA, LDSTUB, LDSTUBA), an atomic quad load instruction (LDDA with ASI = 024_{16}, $02C_{16}$, 034_{16}, or $03C_{16}$), or a SIMD load/store accesses a page with TTE.CP = 0.
08_{16}	FT<3> is set to 1 when an access specifies an invalid ASI, an invalid VA, or an improper access type (read/write). An invalid ASI check is performed prior to the search of the TLB for the TTE; if any of the above conditions hold true, a *data_access_exception* exception is signalled. That is, when FT<3> = 1, the values of the other bits in FT are undefined because the conditions that set those bits require information in the TTE. An instruction that specifies an access of invalid length causes the appropriate *mem_address_not_aligned* or **mem_address_not_aligned* exception; the value of FT is undefined. See Appendix L.3.3 for details.

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

FT<6:0>	Fault Type
10_{16}	FT<4> is set to 1 when a data access other than a nonfaulting load accesses a page with TTE.NFO = 1.
20_{16}	Reserved.
40_{16}	Reserved.

If multiple conditions caused the exception, multiple bits in the DSFSR.FT field may be set.

D-SFSR is updated when a *fast_data_access_MMU_miss*, *data_access_exception*, *fast_data_access_protection*, *VA_watchpoint*, *PA_watchpoint*, *privileged_action*, *mem_address_not_aligned*, or *data_access_error* exception occurs. TABLE F-12 shows which fields are updated by each field.

TABLE F-12 D-SFSR Update Policy

Field		TLB#, index	FV	OW	W, PR, NF, CT[1]	FT	TM	ASI	UE, BERR, BRTO, mDTLB, NC[2] , E[2]	DSFAR
When D-SFSR.OW = 0, 0: 0 is set. 1: 1 is set. V: A valid value is set. —: Invalid field.										
Miss:	*fast_data_access_MMU_miss*	—	0	0	V	—	1	—	—	V
Exception:	*data_access_exception*	—	1	0	V	V	0	V	—	V
Faults:	*fast_data_access_protection*	—	1	0	V	—	0	V	—	V
	PA_watchpoint	—	1	0	V	—	0	V	—	V
	VA watchpoint	—	1	0	V	—	0	V	—	V
	privileged_action[3]	—	1	0	V	—	0	V	—	V
	mem_address_not_aligned, **_mem_address_not_aligned*	—	1	0	V	—	0	V	—	V
	data_access_error	V[4]	1	0	V	—	0	V	V	V
	SIMD_load_across_pages	—	1	0	V	—	0	V	—	V
When D-SFSR.OW = 1, 0: 0 is set. 1: 1 is set. K: Original value is preserved. U: Updated.										
Fault on exception		U[4]	1	1	U	K	K	U	U	U
Exception on fault		K	1	1	U	U	K	U	K	U
Fault on miss[5]		U[4]	1	K	U	K	1	U	U	U
Exception on miss[5]		K	1	K	U	U	1	U	K	U

TABLE F-12 D-SFSR Update Policy

Field	TLB#, index	FV	OW	W, PR, NF, CT[1]	FT	TM	ASI	UE, BERR, BRTO, mDTLB, NC[2] , E[2]	DSFAR
Miss on fault/exception	K	1	K	K	K	1	K	K	K
Miss on miss	K	K	K	U	K	1	K	K	K

1.The value of DSFSR.CT is 11_{02} when the ASI is not a translating ASI, or is an invalid ASI.

2.Only valid for a *data_access_error* exception caused by an uncorrectable error, bus error, or bus timeout.

3.Memory access instruction only.

4.Only when there is a multiple hit in the TLB.

5.Fault/exception on miss describes the state where a miss occurs, then a fault/exception occurs before software can clear the DSFSR.

F.10.10 Synchronous Fault Addresses

When a *VA_watchpoint* or *PA_watchpoint* exception occurs, D-SFAR displays the address specified by the instruction that caused the exception.

For a SIMD load/store instruction, however, the address of the extended operation is displayed when a watchpoint exception is detected for the extended operation only. That is, the displayed address is the address of the instruction plus 4 for a single-precision operation, or the address of the instruction plus 8 for a double-precision operation.

F.10.11 I/D MMU Demap

When Demap is used to remove an entry from a sTLB, the page size used to calculate the index is derived from the context field of the ASI_I/DMMU_DEMAP access address in the same way as a normal TLB access. That is, when ASI_MCNTL.mpg_sI/DTLB are 0, the page size setting is 8 KB for the 1st sTLB and 4 MB for the 2nd sTLB. When ASI_MCNTL.mpg_sI/DTLB are 1, the page size settings of the Context Register specified by the context field are used.

The page size is also used to select TTEs removed by a Demap Page or Demap Context operation. That is, if the page size does not match the page size of a TLB entry, that entry is not removed.

Note – A Demap Page or Demap Context operation should specify a valid context ID. When 01_2 or 11_2 is specified for the IMMU or 11_2 is specified for the DMMU, unrelated sTLB entries may be removed.

All sTLB entries are removed by a Demap All operation, regardless of the page size.

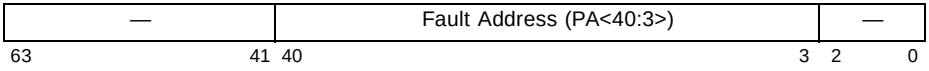
The shared context cannot be specified for a demap operation.

Programming Note – Shared context TTEs can be removed by temporarily changing the entries to specify the secondary context register.

F.10.12 Synchronous Fault Physical Addresses

JPS1 **Commonality** defines registers that store the virtual address when a IMMU or DMMU exception occurs. In addition to these registers, SPARC64 VIIIfx defines the IMMU and DMMU Synchronous Fault Physical Address Registers (`I/D-SFPAR`), which store the physical addresses.

Register Name	`ASI_IMMU_SFPAR`, `ASI_DMMU_SFPAR`
ASI	50_{16} (IMMU), 58_{16} (DMMU)
VA	78_{16}
Access Type	Supervisor read/write



The `I/D-SFPAR` display the physical address of the memory access that caused the exception. When *instruction/data_access_error* exceptions occur and one or more of the `MK`, `UE`, `BERR`, and `BRTO` fields of the `I/D-SFSR` are set to 1, these registers are updated.

F.11 MMU Bypass

In SPARC64 VIIIfx, the following two DMMU Bypass ASIs are defined:

- `ASI_ATOMIC_QUAD_LDD_PHYS` (ASI 34_{16})
- `ASI_ATOMIC_QUAD_LDD_PHYS_LITTLE` (ASI $3C_{16}$)

The physical page attribute bits are set as shown in TABLE F-13. The first four rows are the same as the page attribute bits defined in TABLE F-15 of JPS1 **Commonality**.

TABLE F-13 Bypass Attribute Bits in SPARC64 VIIIfx

ASI Name	ASI Value	Attribute Bits CP	IE	CV	E	P	W	NFO	Size
ASI_PHYS_USE_EC	14_{16}	1	0	0	0	0	1	0	8 Kbytes
ASI_PHYS_USE_EC_LITTLE	$1C_{16}$								
ASI_PHYS_BYPASS_EC_WITH_EBIT	15_{16}	0	0	0	1	0	1	0	8 Kbytes
ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE	$1D_{16}$								
ASI_ATOMIC_QUAD_LDD_PHYS	34_{16}	1	0	0	0	0	0	0	8 Kbytes
ASI_ATOMIC_QUAD_LDD_PHYS_LITTLE	$3C_{16}$								

F.12 Translation Lookaside Buffer Hardware

F.12.2 TLB Replacement Policy

Automatic TLB Replacement

On a write to the TLB via the I/D MMU Data In Registers, hardware selects which entry in which TLB to replace. Replacement occurs according to the following rules:

1. If all of the following conditions are satisfied, then the replacement occurs in the sTLB. Otherwise, the replacement occurs in the fTLB.
   - Entry to be written is `TTE.L = 0` and `TTE.G = 0`.
   - When `ASI_MCNTL.mpg_sITLB/mpg_sDTLB` = 0, page size is either 8KB or 4MB. When `ASI_MCNTL.mpg_sITLB/mpg_sDTLB` = 1, page size matches the page size of the `I/DTLB_TAG_ACCESS_EXT` context register.
   - `ASI_MCNTL.fw_fITLB/fDTLB` = 0.
2. When the sTLB is selected, the virtual page number corresponding to the page size is obtained from the VA of the TLB Tag Access and used as the index. The replacement policy for entries at this index is LRU.
3. When the fTLB is selected, the entry to be replaced is determined using the following procedure:
   a. Starting from entry 0, the first entry found that is empty is replaced. If there are no empty entries, then

b. starting from entry 0, the first entry that is unlocked and whose used[1] bit is 0 is replaced. If there are no unused, unlocked entries, then

c. all used bits are set to 0, and step b is repeated.

If all entries are locked, then the TLB is not written and no exception is signalled.

4. Writes to the fTLB are checked for a multiple hit; that is, the TTE already in the fTLB is compared with the TTE that is to be written. When a multiple hit occurs, the new TTE is not written.

Restrictions on Direct Replacement of sTLB Entries

There are no restrictions for a TTE being written to the sTLB via the I/D MMU Data Access Registers. SPARC64 VIIIfx does not check that the TTE page size and sTLB page size match.

1. Internal TLB flag. Not visible to software.

F.APPENDIX **G**

Assembly Language Syntax

G.1 Notation Used

G.1.5 Other Operand Syntax

The syntax for software traps has been changed from JPS1 **Commonality**. The updated syntax is shown below.

software_trap_number

Can be any of the following:

reg_{rs1}	(equivalent to reg_{rs1} + %g0)
reg_{rs1} + *imm7*	
reg_{rs1} – *imm7*	
imm7	(equivalent to %g0 + *imm7*)
imm7 + reg_{rs1}	(equivalent to reg_{rs1} + *imm7*)
reg_{rs1} + reg_{rs2}	

Here, *imm7* is an unsigned immediate constant that can be represented in 7 bits. The resulting operand value (software trap number) must be in the range 0–127, inclusive.

G.4 HPC-ACE Notation

When an instruction is executed, the value of the XAR determines whether the instruction uses any of the features added by the HPC-ACE extensions. Generally, these features are specified by combining an arithmetic instruction with SXAR. This section defines the assembly language syntax for specifying HPC-ACE features.

HPC-ACE extends the instruction definitions to support the use of HPC-ACE registers, SIMD execution, sector cache, and hardware prefetch enable/disable. While SXAR fully specifies these features, the following notation is defined to facilitate easy reading of the assembly language:

1. SXAR is written as sxar1 or sxar2. These instructions have no arguments.
2. The HPC-ACE registers are indicated directly in the arguments of the instruction.
3. Other HPC-ACE features are indicated by appending suffixes to the instruction mnemonic.
4. The features for an instruction are always specified by the closest SXAR that precedes the instruction. SXARs in instruction sequences that branch to a point between the corresponding SXAR and the instruction never specify features for that instruction.

A SXAR must be placed 1 or 2 instructions prior to any instruction that uses the notation described in items 2 and 3. There are cases where the assembler cannot automatically determine that a SXAR needs to be inserted; thus, SXAR cannot be ommitted.

Whether a label can be inserted between the corresponding SXAR and the instruction is not defined, as item 4 clearly defines which SXAR specifies the HPC-ACE feature(s).

G.4.1 Suffixes for HPC-ACE Extensions

A comma (,) is placed after the instruction mnemonic, and the alphanumeric character(s) that immediately follow the comma specify various HPC-ACE features. These suffixes are shown in TABLE G-1.

TABLE G-1 Suffixes for HPC-ACE Extensions

XAR Notation	Suffix	Remarks
XAR.simd	s	
XAR.dis_hw_pf	d	

TABLE G-1 Suffixes for HPC-ACE Extensions

XAR Notation	Suffix	Remarks
`XAR.sector`	1	'0' indicates sector 0 (default sector)
`XAR.negate_mul`	n	
`XAR.rs1_copy`	c	

Suffixes are not case-sensitive. When two or more suffixes are specified, the suffixes may be specified in any order.

Example 1: SIMD instruction, using HPC-ACE registers

```
sxar2
fmaddd    %f0, %f2, %f510    /*HPC-ACE register used, non-SIMD */
fmaddd,s  %f0, %f2, %f4      /*SIMD, extended operation uses HPC-ACE
                               registers  */
```

Example 2: SIMD load from sector 1

```
sxar1
ldd,s1    [%xg24], %f0       /* Suffix 'ls' is also acceptable */
```

F.APPENDIX **H**

Software Considerations

Please refer to Appendix H in JPS1 **Commonality**.

F.APPENDIX **I**

Extending the SPARC V9 Architecture

Please refer to Appendix I in JPS1 **Commonality**.

F.APPENDIX **J**

Changes from SPARC V8 to SPARC V9

Please refer to Appendix J in JPS1 **Commonality**.

F.APPENDIX **K**

Programming with the Memory Models

Please refer to Appendix K in JPS1 **Commonality**.

F.APPENDIX **L**

Address Space Identifiers

This appendix lists all ASIs supported by SPARC64 VIIIfx and describes the ASIs specific to SPARC64 VIIIfx.

L.2 ASI Values

The SPARC V9 address space identifier (ASI) is evenly divided into restricted and unrestricted halves. ASIs in the range 00_{16}–$7F_{16}$ are restricted. ASIs in the range 80_{16}–FF_{16} are unrestricted. An attempt by nonprivileged software to access a restricted ASI causes a *privileged_action* trap.

ASIs are also divided into translating, bypass, and nontranslating types. Translating ASIs are translated by the MMU. Bypass ASIs are not translated by the MMU; instead, they pass through their virtual addresses as physical addresses. Nontranslating ASIs access internal CPU resources. TABLE L-1 shows the ASI types as defined in SPARC64 VIIIfx.

Compatibility Note – In JPS1 **Commonality**, the 3 ASI types include implementation-dependent and undefined ASIs. SPARC64 VIIIfx redefines the 3 ASI types to only include defined ASIs.

TABLE L-1 SPARC64 VIIIfx ASI Types

ASI Type	ASI Range
Translating ASIs	Restricted 04_{16}, $0C_{16}$, 10_{16}, 11_{16}, 18_{16}, 19_{16}, 24_{16}, $2C_{16}$, 70_{16}–73_{16}, 78_{16}, 79_{16} Unrestricted 80_{16}–83_{16}, 88_{16}–$8B_{16}$, $C0_{16}$–$C5_{16}$, $C8_{16}$–CD_{16}, $D0_{16}$–$D3_{16}$, $D8_{16}$–DB_{16}, $E0_{16}$, $E1_{16}$, $F0_{16}$–$F3_{16}$, $F8_{16}$, $F9_{16}$
Bypass ASIs	Restricted 14_{16}, 15_{16}, $1C_{16}$, $1D_{16}$, 34_{16}, $3C_{16}$
Nontranslating ASIs	Restricted 45_{16}, 48_{16}–$4C_{16}$, $4F_{16}$, 50_{16}, 53_{16}–58_{16}, $5C_{16}$–60_{16}, 67_{16}, $6D_{16}$–$6F_{16}$, 74_{16}, 77_{16}, $7F_{16}$ Unrestricted $E7_{16}$, EF_{16}

The ASI types are related to data watchpoints. Refer to *"Data Watchpoint Registers"* in this document, as well as in JPS1 **Commonality**.

L.3 SPARC64 VIIIfx ASI Assignments

Every load or store address in a SPARC V9 processor has an 8-bit Address Space Identifier (ASI) appended to the virtual address (VA). Together, the VA and the ASI fully specify the address. For instruction fetches and memory access instructions that do not specify the ASI, an implicit ASI generated by the hardware is used. When a load from alternate space or store into alternate space 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 used not only to access address spaces but also to access internal registers, such as MMU and hardware barrier registers.

Section L.3.1 includes information on all ASIs defined in JPS1 **Commonality**, as well as the ASIs added in SPARC64 VIIIfx.

L.3.1 Supported ASIs

TABLE L-2 lists the SPARC V9 architecture-defined ASIs, ASIs that were not defined in SPARC V9 but are required for JPS1 processors, and ASIs defined by SPARC64 VIIIfx. The shaded portions indicate ASIs that were defined in SPARC V9 or JPS1 but are not defined in SPARC64 VIIIfx.

ASIs marked with a closed bullet (●) are SPARC V9 architecture-defined ASIs. All operand sizes are supported when accessing one of these ASIs.

ASIs marked with an open bullet (❍) were not defined in SPARC V9 but are required to be implemented in all JPS1 processors. These ASIs can be used only with LDXA, STXA, LDDFA, or STDFA instructions, unless otherwise noted.

ASIs marked with a star (★) are ASIs defined by SPARC64 VIIIfx. These ASIs can be used only with LDXA, STXA, LDDFA, or STDFA instructions, unless otherwise noted.

The "VA", "Effective Bits", and "Aligment" columns in TABLE L-2 specify which virtual addresses are valid for the ASIs.

- The "VA" column indicates the virtual address. An "—" indicates that any address can be specified. If "encode" is shown, refer to the description of that ASI.
- The "Effective Bits" column indicates which bits in the VA are valid. Invalid bits are ignored.
  - "full" indicates all 64 bits are valid.
  - "physical" indicates bits up to the physical address width are valid.
  - bit<a:b> indicates that bits in the range a to b are valid
- The "Aligment" column indicates memory alignment restrictions, if any. An "—" indicates that there are no alignment restrictions. Refer to the descriptions of individual ASIs for information on the exceptions generated by improperly aligned addresses.

See Appendix L.3.3 for information on the exceptions generated by an access to an undefined ASI or an invalid combination of an ASI and a memory access instruction.

TABLE L-2 SPARC64 VIIIfx ASIs *(1 of 5)*

ASI	VA	Effective bits	Alignment	ASI Name (and Abbreviation)	Access	Page
● 04_{16}	—	full	—	ASI_NUCLEUS (ASI_N)	RW	
● $0C_{16}$	—	full	—	ASI_NUCLEUS_LITTLE (ASI_NL)	RW	
● 10_{16}	—	full	—	ASI_AS_IF_USER_PRIMARY (ASI_AIUP)	RW	
● 11_{16}	—	full	—	ASI_AS_IF_USER_SECONDARY (ASI_AIUS)	RW	
❍ 14_{16}	—	physical	—	ASI_PHYS_USE_EC	RW	
❍ 15_{16}	—	physical	—	ASI_PHYS_BYPASS_EC_WITH_EBIT	RW	
● 18_{16}	—	full	—	ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)	RW	
● 19_{16}	—	full	—	ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)	RW	
❍ $1C_{16}$	—	physical	—	ASI_PHYS_USE_EC_LITTLE (ASI_PHYS_USE_EC_L)	RW	
❍ $1D_{16}$	—	physical	—	ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE (ASI_PHYS_BYPASS_EC_WITH_EBIT_L)	RW	
❍ 24_{16}	—	full	16byte	ASI_NUCLEUS_QUAD_LDD	R	

TABLE L-2 SPARC64 VIIIfx ASIs *(2 of 5)*

	ASI	VA	Effective bits	Alignment	ASI Name (and Abbreviation)	Access	Page
❍	$2C_{16}$	—	full	16byte	ASI_NUCLEUS_QUAD_LDD_LITTLE (ASI_NUCLEUS_QUAD_LDD_L)	R	
★	34_{16}	—	physical	16byte	ASI_ATOMIC_QUAD_LDD_PHYS	R	89
★	$3C_{16}$	—	physical	16byte	ASI_ATOMIC_QUAD_LDD_PHYS_LITTLE	R	89
❍	45_{16}	00_{16}	bit<7:0>	8byte	ASI_DCU_CONTROL_REGISTER (ASI_DCUCR)	RW	34
❍	45_{16}	08_{16}	bit<7:0>	8byte	ASI_MEMORY_CONTROL_REG (ASI_MCNTL)	RW	185
★	46_{16}	00_{16}	bit<7:0>	8byte		R	
★	47_{16}	00_{16}	bit<7:0>	8byte		R	
❍	48_{16}	00_{16}	bit<7:0>	8byte	ASI_INTR_DISPATCH_STATUS (ASI_MONDO_SEND_CTRL)	R	242
❍	49_{16}	00_{16}	bit<7:0>	8byte	ASI_INTR_RECEIVE (ASI_MONDO_RECEIVE_CTRL)	RW	243
★	$4A_{16}$	—	bit<7:0>	8byte	ASI_SYS_CONFIG	R	323
★	$4B_{16}$	00_{16}	bit<7:0>	8byte	ASI_STICK_CNTL	RW	324
❍	$4C_{16}$	00_{16}	bit<7:0>	8byte	ASI_ASYNC_FAULT_STATUS (ASI_AFSR)	RW	285
★	$4C_{16}$	08_{16}	bit<7:0>	8byte	ASI_URGENT_ERROR_STATUS (ASI_UGESR)	R	275
★	$4C_{16}$	10_{16}	bit<7:0>	8byte	ASI_ERROR_CONTROL (ASI_ECR)	RW	270
★	$4C_{16}$	18_{16}	bit<7:0>	8byte	ASI_STATE_CHANGE_ERROR_INFO (ASI_STCHG_ERR_INFO)	RW	272
❍	$4D_{16}$	00_{16}			ASI_ASYNC_FAULT_ADDR (ASI_AFAR)	R	
★	$4F_{16}$	00_{16}	bit<7:0>	8byte	ASI_SCRATCH_REG0	RW	220
★	$4F_{16}$	08_{16}	bit<7:0>	8byte	ASI_SCRATCH_REG1	RW	220
★	$4F_{16}$	10_{16}	bit<7:0>	8byte	ASI_SCRATCH_REG2	RW	220
★	$4F_{16}$	18_{16}	bit<7:0>	8byte	ASI_SCRATCH_REG3	RW	220
★	$4F_{16}$	20_{16}	bit<7:0>	8byte	ASI_SCRATCH_REG4	RW	220
★	$4F_{16}$	28_{16}	bit<7:0>	8byte	ASI_SCRATCH_REG5	RW	220
★	$4F_{16}$	30_{16}	bit<7:0>	8byte	ASI_SCRATCH_REG6	RW	220
★	$4F_{16}$	38_{16}	bit<7:0>	8byte	ASI_SCRATCH_REG7	RW	220
❍	50_{16}	00_{16}	bit<7:0>	8byte	ASI_IMMU_TAG_TARGET	R	
❍	50_{16}	18_{16}	bit<7:0>	8byte	ASI_IMMU_SFSR	RW	195
❍	50_{16}	28_{16}	bit<7:0>	8byte	ASI_IMMU_TSB_BASE	RW	194
❍	50_{16}	30_{16}	bit<7:0>	8byte	ASI_IMMU_TAG_ACCESS	RW	194
❍	50_{16}	48_{16}			ASI_IMMU_TSB_PEXT_REG	RW	
❍	50_{16}	58_{16}			ASI_IMMU_TSB_NEXT_REG	RW	
★	50_{16}	60_{16}	bit<7:0>	8byte	ASI_IMMU_TAG_ACCESS_EXT	RW	191
★	50_{16}	78_{16}	bit<7:0>	8byte	ASI_IMMU_SFPAR	RW	202
❍	51_{16}	00_{16}			ASI_IMMU_TSB_8KB_PTR_REG	R	
❍	52_{16}	00_{16}			ASI_IMMU_TSB_64KB_PTR_REG	R	

TABLE L-2 SPARC64 VIIIfx ASIs *(3 of 5)*

	ASI	VA	Effective bits	Alignment	ASI Name (and Abbreviation)	Access	Page
★	53_{16}	—	bit<7:0>	8byte	ASI_SERIAL_ID	R	220
❍	54_{16}	—	bit<7:0>	8byte	ASI_ITLB_DATA_IN_REG	W	192
❍	55_{16}	encode	bit<17:0>	8byte	ASI_ITLB_DATA_ACCESS_REG	RW	192
❍	56_{16}	encode	bit<17:0>	8byte	ASI_ITLB_TAG_READ_REG	R	193
❍	57_{16}	encode	full	8byte	ASI_IMMU_DEMAP	W	201
❍	58_{16}	00_{16}	bit<7:0>	8byte	ASI_DMMU_TAG_TARGET_REG	R	
❍	58_{16}	08_{16}	bit<7:0>	8byte	ASI_PRIMARY_CONTEXT_REG	RW	188
❍	58_{16}	10_{16}	bit<7:0>	8byte	ASI_SECONDARY_CONTEXT_REG	RW	188
❍	58_{16}	18_{16}	bit<7:0>	8byte	ASI_DMMU_SFSR	RW	195
❍	58_{16}	20_{16}	bit<7:0>	8byte	ASI_DMMU_SFAR	RW	
❍	58_{16}	28_{16}	bit<7:0>	8byte	ASI_DMMU_TSB_BASE	RW	194
❍	58_{16}	30_{16}	bit<7:0>	8byte	ASI_DMMU_TAG_ACCESS	RW	194
❍	58_{16}	38_{16}	bit<7:0>	8byte	ASI_DMMU_WATCHPOINT_REG	RW	36
❍	58_{16}	40_{16}			ASI_DMMU_PA_WATCHPOINT_REG	RW	
❍	58_{16}	48_{16}			ASI_DMMU_TSB_PEXT_REG	RW	
❍	58_{16}	50_{16}			ASI_DMMU_TSB_SEXT_REG	RW	
❍	58_{16}	58_{16}			ASI_DMMU_TSB_NEXT_REG	RW	
★	58_{16}	60_{16}	bit<7:0>	8byte	ASI_DMMU_TAG_ACCESS_EXT	RW	191
★	58_{16}	68_{16}	bit<7:0>	8byte	ASI_SHARED_CONTEXT_REG	RW	189
★	58_{16}	78_{16}	bit<7:0>	8byte	ASI_DMMU_SFPAR	RW	202
❍	59_{16}	00_{16}			ASI_DMMU_TSB_8KB_PTR_REG	R	
❍	$5A_{16}$	00_{16}			ASI_DMMU_TSB_64KB_PTR_REG	R	
❍	$5B_{16}$	00_{16}			ASI_DMMU_TSB_DIRECT_PTR_REG	R	
❍	$5C_{16}$	—	bit<7:0>	8byte	ASI_DTLB_DATA_IN_REG	W	192
❍	$5D_{16}$	encode	bit<17:0>	8byte	ASI_DTLB_DATA_ACCESS_REG	RW	192
❍	$5E_{16}$	encode	bit<17:0>	8byte	ASI_DTLB_TAG_READ_REG	R	193
❍	$5F_{16}$	encode	full	8byte	ASI_DMMU_DEMAP	W	201
❍	60_{16}	—	bit<7:0>	8byte	ASI_IIU_INST_TRAP	RW	37
★	67_{16}	—	bit<7:0>	8byte	ASI_FLUSH_L1I	W	233
★	$6D_{16}$	00_{16}–58_{16}	bit<7:0>	8byte	ASI_BARRIER_INIT	RW	224
★	$6E_{16}$	00_{16}	bit<7:0>	8byte	ASI_ERROR_IDENT (ASI_EIDR)	RW	270
★	$6F_{16}$	00_{16}–58_{16}	bit<7:0>	8byte	ASI_BARRIER_ASSIGN	RW	226
❍	70_{16}	—	full	64byte	ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)	RW	
❍	71_{16}	—	full	64byte	ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS)	RW	
★	72_{16}	—	full	8byte	ASI_XFILL_AIUP	W	135
★	73_{16}	—	full	8byte	ASI_XFILL_AIUS	W	135

TABLE L-2 SPARC64 VIIIfx ASIs *(4 of 5)*

	ASI	VA	Effective bits	Alignment	ASI Name (and Abbreviation)	Access	Page
★	74_{16}	—	physical	8byte	ASI_CACHE_INV	W	233
❍	77_{16}	40_{16}	bit<7:0>	8byte	ASI_INTR_DATA0_W	W	242
❍	77_{16}	48_{16}	bit<7:0>	8byte	ASI_INTR_DATA1_W	W	242
❍	77_{16}	50_{16}	bit<7:0>	8byte	ASI_INTR_DATA2_W	W	242
❍	77_{16}	58_{16}			ASI_INTR_DATA3_W	W	
❍	77_{16}	60_{16}			ASI_INTR_DATA4_W	W	
❍	77_{16}	68_{16}			ASI_INTR_DATA5_W	W	
❍	77_{16}	80_{16}			ASI_INTR_DATA6_W	W	
❍	77_{16}	88_{16}			ASI_INTR_DATA7_W	W	
❍	77_{16}	encode|70_{16}	bit<26:24>, bit<16:14>, bit<13:0>	8byte	ASI_INTR_DISPATCH_W	W	242
❍	78_{16}	—	full	64byte	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUPL)	RW	
❍	79_{16}	—	full	64byte	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL)	RW	
❍	$7F_{16}$	40_{16}	bit<7:0>	8byte	ASI_INTR_DATA0_R	R	242
❍	$7F_{16}$	48_{16}	bit<7:0>	8byte	ASI_INTR_DATA1_R	R	242
❍	$7F_{16}$	50_{16}	bit<7:0>	8byte	ASI_INTR_DATA2_R	R	242
❍	$7F_{16}$	58_{16}			ASI_INTR_DATA3_R	R	
❍	$7F_{16}$	60_{16}			ASI_INTR_DATA4_R	R	
❍	$7F_{16}$	68_{16}			ASI_INTR_DATA5_R	R	
❍	$7F_{16}$	80_{16}			ASI_INTR_DATA6_R	R	
❍	$7F_{16}$	88_{16}			ASI_INTR_DATA7_R	R	
●	80_{16}	—	full	—	ASI_PRIMARY (ASI_P)	RW	
●	81_{16}	—	full	—	ASI_SECONDARY (ASI_S)	RW	
●	82_{16}	—	full	—	ASI_PRIMARY_NO_FAULT (ASI_PNF)	R	
●	83_{16}	—	full	—	ASI_SECONDARY_NO_FAULT (ASI_SNF)	R	
●	88_{16}	—	full	—	ASI_PRIMARY_LITTLE (ASI_PL)	RW	
●	89_{16}	—	full	—	ASI_SECONDARY_LITTLE (ASI_SL)	RW	
●	$8A_{16}$	—	full	—	ASI_PRIMARY_NO_FAULT_LITTLE (ASI_PNFL)	R	
●	$8B_{16}$	—	full	—	ASI_SECONDARY_NO_FAULT_LITTLE (ASI_SNFL)	R	
❍	$C0_{16}$	—	full	—	ASI_PST8_PRIMARY (ASI_PST8_P)	W	221
❍	$C1_{16}$	—	full	—	ASI_PST8_SECONDARY (ASI_PST8_S)	W	221
❍	$C2_{16}$	—	full	—	ASI_PST16_PRIMARY (ASI_PST16_P)	W	221
❍	$C3_{16}$	—	full	—	ASI_PST16_SECONDARY (ASI_PST16_S)	W	221
❍	$C4_{16}$	—	full	—	ASI_PST32_PRIMARY (ASI_PST32_P)	W	221

TABLE L-2 SPARC64 VIIIfx ASIs *(5 of 5)*

ASI	VA	Effective bits	Alignment	ASI Name (and Abbreviation)	Access	Page
❍ $C5_{16}$	—	full	—	ASI_PST32_SECONDARY (ASI_PST32_S)	W	221
❍ $C8_{16}$	—	full	—	ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL)	W	221
❍ $C9_{16}$	—	full	—	ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL)	W	221
❍ CA_{16}	—	full	—	ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL)	W	221
❍ CB_{16}	—	full	—	ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL)	W	221
❍ CC_{16}	—	full	—	ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL)	W	221
❍ CD_{16}	—	full	—	ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL)	W	221
❍ $D0_{16}$	—	full	—	ASI_FL8_PRIMARY (ASI_FL8_P)	RW	
❍ $D1_{16}$	—	full	—	ASI_FL8_SECONDARY (ASI_FL8_S)	RW	
❍ $D2_{16}$	—	full	—	ASI_FL16_PRIMARY (ASI_FL16_P)	RW	
❍ $D3_{16}$	—	full	—	ASI_FL16_SECONDARY (ASI_FL16_S)	RW	
❍ $D8_{16}$	—	full	—	ASI_FL8_PRIMARY_LITTLE (ASI_FL8_PL)	RW	
❍ $D9_{16}$	—	full	—	ASI_FL8_SECONDARY_LITTLE (ASI_FL8_SL)	RW	
❍ DA_{16}	—	full	—	ASI_FL16_PRIMARY_LITTLE (ASI_FL16_PL)	RW	
❍ DB_{16}	—	full	—	ASI_FL16_SECONDARY_LITTLE (ASI_FL16_SL)	RW	
❍ $E0_{16}$	—	full	—	ASI_BLOCK_COMMIT_PRIMARY (ASI_BLK_COMMIT_P)	W	220
❍ $E1_{16}$	—	full	—	ASI_BLOCK_COMMIT_SECONDARY (ASI_BLK_COMMIT_S)	W	220
★ $E7_{16}$	00_{16}	bit<7:0>	8byte	ASI_SCCR	RW	234
★ EF_{16}	00_{16}–58_{16}	bit<7:0>	8byte	ASI_LBSY, ASI_BST	RW	227
❍ $F0_{16}$	—	full	64byte	ASI_BLOCK_PRIMARY (ASI_BLK_P)	RW	
❍ $F1_{16}$	—	full	64byte	ASI_BLOCK_SECONDARY (ASI_BLK_S)	RW	
★ $F2_{16}$	—	full	8byte	ASI_XFILL_P	W	135
★ $F3_{16}$	—	full	8byte	ASI_XFILL_S	W	135
❍ $F8_{16}$	—	full	64byte	ASI_BLOCK_PRIMARY_LITTLE (ASI_BLK_PL)	RW	
❍ $F9_{16}$	—	full	64byte	ASI_BLOCK_SECONDARY_LITTLE (ASI_BLK_SL)	RW	

L.3.2 Special Memory Access ASIs

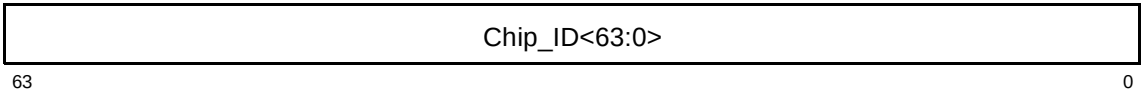
Please refer to Section L.3.2 in JPS1 **Commonality**.

In addition to the ASIs described in JPS1 **Commonality**, SPARC64 VIIIfx supports the ASIs described below.

ASI 53_{16} (ASI_SERIAL_ID)

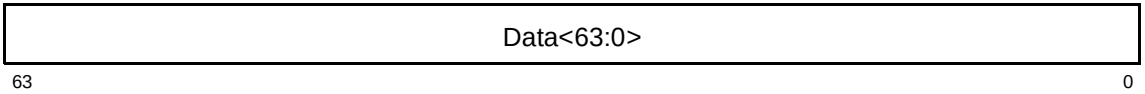
SPARC64 VIIIfx provides an unique ID code for each CPU chip. Using this ID code and the information in the Version Register (page 26), a completely unique CPU ID can be generated.

This register is read-only. A write to this register causes a *data_access_error* exception.



ASI $4F_{16}$ (ASI_SCRATCH_REGx)

SPARC64 VIIIfx provides eight 64-bit registers that can be used by supervisor software.



Register Name	`ASI_SCRATCH_REG`*x* (*x* = 0–7)
ASI	$4F_{16}$
VA	VA<5:3> = register number The other VA bits must be zero.
Access Type	Supervisor read/write

Block Load and Store ASIs

As describe in the definition of the Block Store with Commit instruction (see *"Block Load and Store Instructions (VIS I)"* (page 68)), ASIs $E0_{16}$ and $E1_{16}$ can only be used with `STDFA` instructions. These ASIs cannot be used with `LDDFA`. If either ASI is specified, `LDDFA` has the following behavior:

1. No exception is generated due to a misaligned `rd` (impl. dep. #255).
2. Depending on the memory address alignment, the following exceptions are generated (impl. dep. #256).
   - If aligned on an 8-byte boundary, causes a *data_access_exception* exception with `DSFSR.FTYPE` = 08_{16} (invalid ASI).

- If aligned on an 4-byte boundary, causes a *LDDF_mem_address_not_aligned* exception.
- Otherwise, causes a *mem_address_not_aligned* exception.

Partial Store ASIs

As described in the definition of the Partial Store instruction (see *"Partial Store (VIS I)"* (page 94)), ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–CD_{16} can only be used with `STDFA` instructions. These ASIs cannot be used with `LDDFA`. If either ASI is specified, `LDDFA` has the following behavior:

- Depending on the memory address alignment, the following exceptions are generated (impl. dep. #257).
  - If aligned on an 8-byte boundary, causes a *data_access_exception* exception with `DSFSR.FTYPE` = 08_{16} (invalid ASI).
  - If aligned on an 4-byte boundary, causes a *LDDF_mem_address_not_aligned* exception.
  - Otherwise, causes a *mem_address_not_aligned* exception.

L.3.3 Trap Priority for ASI and Instruction Combinations

In SPARC64 VIIIfx, the behavior of exceptions generated by an undefined ASI or an invalid instruction and ASI combination differs slightly from JPS1 **Commonality**. This section describes these exceptions as defined in SPARC64 VIIIfx, listed in order of priority.

1. There are cases where a Block Load/Store or Partial Store instructions causes an *illegal_instruction* exception. See the description of the specific instruction for details. If the `rd` field of `LDDA` or `STDA` specifies an odd-number register, an *illegal_instruction* exception is signalled.

2. The memory alignment restriction specified for the instruction is checked; an improperly aligned address causes a *mem_address_not_aligned* or **_mem_address_not_aligned* exception.

   a. Data for block load/store instructions must be aligned on 64-byte boundaries. An improperly aligned address causes a *mem_address_not_aligned* exception. *LDDF_mem_address_not_aligned* and *STDF_mem_address_not_aligned* exceptions are not signalled.

      A `LDDFA` instructions that specifies a block store with commit ASI is not a block load/store instruction. This specification does not apply.

   b. Data for 16-bit short load/store instructions must be aligned on 2-byte boundaries. An improperly aligned address causes a *mem_address_not_aligned* exception. *LDDF_mem_address_not_aligned* and *STDF_mem_address_not_aligned* exceptions are not signalled.

c. Data for 8-bit short load/store instructions must be aligned on 1-byte boundaries; the address is never improperly aligned.

d. Data for partial store instructions must be aligned on 8-byte boundaries. An improperly aligned address causes a *mem_address_not_aligned* exception. *LDDF_mem_address_not_aligned* and *STDF_mem_address_not_aligned* exceptions are not signalled.

   A `LDDFA` instructions that specifies a partial store ASI is not a partial store instruction. This specification does not apply.

e. For `LDDFA` and `STDFA` instructions used with any ASI that is not specified above, accesses aligned on 4-byte boundaries cause *LDDF_mem_address_not_aligned* and *STDF_mem_address_not_aligned* exceptions, respectively.

f. Any improperly aligned address that is not described above causes a *mem_address_not_aligned* exception.

For items e and f, whether the ASI access is defined or undefined takes priority over whether the ASI and instruction combination is valid. A *data_access_exception* (`FT` = 08_{16}) exception is not signalled.

3. If the ASI and instruction combination is not valid, a *data_access_exception* exception is signalled.

   However, `PREFETCHA` does not cause a *data_access_exception* exception; the instruction is processed as a nop.

L.3.4 Timing for Writes to Internal Registers

In SPARC64 VIIIfx, almost all nontranslating ASIs map to CPU internal registers. Most of these internal registers, which include MMU and hardware barrier registers, have side effects; however, the ordering of nontranslating ASI accesses is not guaranteed. Software should perform an explicit `membar #Sync` after updating an internal register in order to guarantee that the results (side-effects) are visible to subsequent instructions.

L.4 Hardware Barrier

SPARC64 VIIIfx provides a hardware barrier mechanism that facilitates high speed synchronization in a CPU chip. The on-chip barrier mechanism is shared by all of the cores. FIGURE L-1 shows the barrier resources.

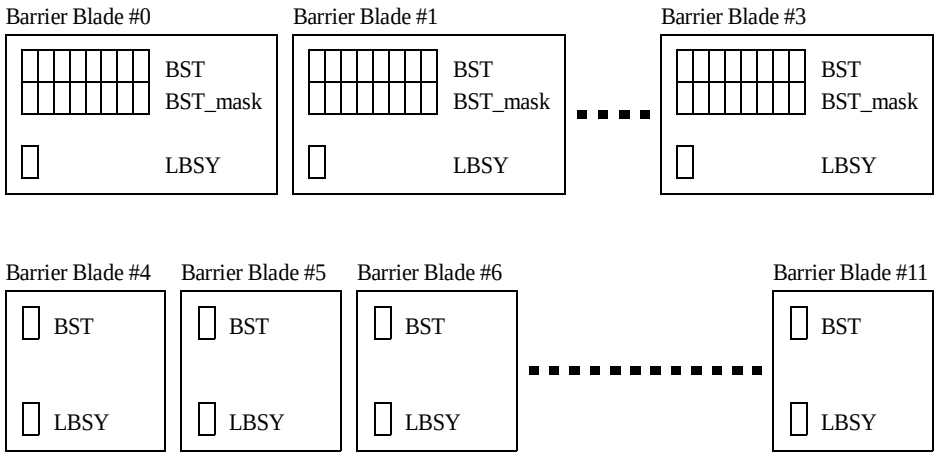


FIGURE L-1 SPARC64 VIIIfx Barrier Resources

SPARC64 VIIIfx has twelve Barrier Blades, which are the primary barrier resources. Each Barrier Blade contains a number of BST (Barrier Status) bits and a mask that selects bits in the BST, as well as a LBSY (Last Barrier Synchronization) bit that stores the synchronization value last used in that Barrier Blade. Four of the Barrier Blades have 8-bit BSTs and BST_masks, which correspond to the on-chip cores. The other eight Barrier Blades have 1-bit BSTs and no BST_masks. The first four are intended to be used for implementing barrier synchronization of multiple threads, and the other eight for implementing post-wait synchronization of thread pairs.

Barrier synchronization is established once all BST bits selected by the BST_mask are set to the same value, either 0 or 1. This synchronization value (0 or 1) is then copied to the LBSY. Update of the LBSY is done atomically, such that a read before modifying the BST always returns the old value and a read after modifying the BST always returns the new value.

Consequently, when a software thread reaches the barrier, the thread reads the LBSY, writes the appropriate BST bit, then waits for the value of LBSY to be updated; this update indicates to the thread that synchronization has been established. The value of LBSY after each BST update can be checked using a spin loop; however, because multiple cores/threads share certain resources, spin loops are inefficient and cause contention with other cores/threads. In SPARC64 VIIIfx, the `SLEEP` instruction can be used to put waiting cores/threads to sleep. An update to LBSY wakes these sleeping cores/threads and returns them to execute state. This achieves high-speed synchronization and efficient use of CPU resources.

Since the LBSY stores the last synchronization value used in the Barrier Blade, software can easily determine the value that should be used to set BST bits when the Barrier Blade is next used. That is, if a read of the LSBY returns 0, then a software thread should write a 1 to the appropriate BST bit. Similarly, if LBSY is 1, then a 0 should be written.

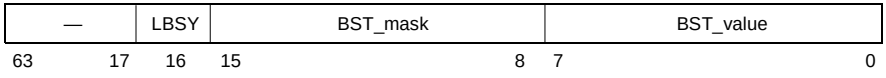
Each core/thread has 12 window ASIs that correspond to the 12 Barrier Blades. User software should access barrier resources through window ASIs; barrier resources should not be accessed directly. The use of window ASIs simplifies hardware barrier operation, hides the actual BST bits, and minimizes the possibility of corrupting the current barrier status.

The memory model for barrier resources conforms to TSO, as defined in Section 8 of JPS1 **Commonality**. That is, accesses to Barrier Blades and memory are performed in program order, except when a store is followed by a load. When a store to a window ASI is followed by a load or a `LBSY` read, a `membar #storeload` must be inserted between the two accesses.

Note – SPARC64 VIIIfx does not support barrier synchronization between CPU chips.

L.4.1 Initialization and Status of Barrier Resources

Register Name	`ASI_BARRIER_INIT`
ASI	$6D_{16}$
VA	00_{16}, 08_{16}, 10_{16}, 18_{16}, 20_{16}, 28_{16}, 30_{16}, 38_{16}, 40_{16}, 48_{16}, 50_{16}, 58_{16}
Access Type	Supervisor read/write



ASI_BARRIER_INIT is used to initialize the Barrier Blade specified by the VA, as well as to obtain the current status. Reads return the current status, and writes set new values.

The BST_mask and BST_value fields indicate the barrier group and the barrier status, respectively. Each bit in these fields corresponds to a core. For BST_mask, a 1 indicates that the corresponding core uses the Barrier Blade. A 0 indicates that the core does not use the Barrier Blade.

Bit	Field	Access	Description
63:17	reserved		
16	LBSY	RW	Last BST synchronization value.
15:8	BST_mask	RW	BST mask. Each bit corresponds to an on-chip core: • BB#0–BB#3 BST_mask<0> core 0 BST_mask<1> core 1 BST_mask<2> core 2 BST_mask<3> core 3 BST_mask<4> core 4 BST_mask<5> core 5 BST_mask<6> core 6 BST_mask<7> core 7 • The BST_mask field does not exist in BB#4–BB#11.
7:0	BST_value	RW	BST value. Each bit corresponds to an on-chip core: • BB#0–BB#3 BST_value<0> core 0 BST_value<1> core 1 BST_value<2> core 2 BST_value<3> core 3 BST_value<4> core 4 BST_value<5> core 5 BST_value<6> core 6 BST_value<7> core 7 • BB#4–BB#11 BST_value<0> core 0–7

On a read, the values of the BST_value, BST_mask, and LBSY fields of the Barrier Blade specified by the VA are returned.

For BB#0–#3, each bit in the BST_mask and BST_value fields corresponds to a specific core. If a BST_mask bit is 0, the value that is read from the corresponding BST_value bit is undefined.

For post/wait Barrier Blades, only the LBSY and BST_value<0> bits are meaningful. The other bits read as 0.

On a write, the BST_value, BST_mask, and LBSY fields of the Barrier Blade specified by the VA are updated.For BB#0–#3, each bit in the BST_mask and BST_value fields corresponds to a specific core. If a BST_mask bit is 0, whether or not an attempt to write a 1 in the corresponding BST_value bit succeeds is undefined.

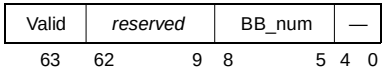
For post/wait Barrier Blades, only the LBSY and BST_value<0> bits are meaningful. Writes to other bits are ignored.

After a write is completed, hardware checks whether synchronization has been established, then updates the LBSY field accordingly. For example, when BST_value and BST_mask are all ones and LBSY is zero, LBSY is immediately updated to 1.

When BST_mask = 0, the current value of LBSY is preserved. Hardware does not check whether synchronization has been established.

L.4.2 Assignment of Barrier Resources

Register Name	ASI_BARRIER_ASSIGN
ASI	$6F_{16}$
VA	00_{16}, 08_{16}, 10_{16}, 18_{16}, 20_{16}, 28_{16}, 30_{16}, 38_{16}, 40_{16}, 48_{16}, 50_{16}, 58_{16}
Access Type	Supervisor read/write



ASI_BARRIER_ASSIGN is used to obtain the current assignment of the window ASI (ASI_BST/ASI_LBSY) specified by the VA, as well as to change this assignment. BB_num specifies the Barrier Blade that is assigned to the window ASI specified by the VA.

Bit	Field	Access	Description
63	Valid	RW	
62:9	reserved		
8:5	BB_num	RW	Indicates the Barrier Blade assigned to the window ASI.
4:0	reserved		

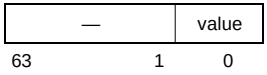
- A read returns the Barrier Blade assignment. When the window ASI specified by the VA is assigned to a Barrier Blade, valid = 1 and the assignment is indicated in BB_num. When the window ASI specified by the VA is not assigned to a Barrier Blade, valid = 0 and the value of BB_num is undefined.
- On a write,
  - When valid = 1, LBSY and BST of the Barrier Blade indicated by BB_num are assigned to the window ASI specified by the VA. After the write completes, user software can write BST using ASI_BST and read LBSY using ASI_LBSY.
  - When valid = 0, the assignment is released. After the write completes, a write to ASI_BST is ignored, and a read of ASI_LBSY returns an undefined value.
  - The value of BB_num is valid for the range 0–11. Writes that attempt to specify a value of 12 or greater are ignored.

When settings for ASI_BARRIER_INIT and ASI_BARRIER_ASSIGN are inconsistent, behavior is undefined. Hardware does not detect these inconsistencies; software is responsible for ensuring these situations do not occur. Synchronization is not guaranteed for cases where a write to ASI_BARRIER_INIT occurs while a Barrier Blade is in use, a BST<i> is assigned to a window ASI while BST_mask<i> = 0, etc.

Programming Note – System software should only assign a Barrier Blade after it has been initialized. Assignment of a non-initialized Barrier Blade may cause unexpected results.

L.4.3 Window ASI for Barrier Resources

Register Name	ASI_LBSY (read), ASI_BST (write)
ASI	EF_{16}
VA	00_{16}, 08_{16}, 10_{16}, 18_{16}, 20_{16}, 28_{16}, 30_{16}, 38_{16}, 40_{16}, 48_{16}, 50_{16}, 58_{16}
Access Type	Read/Write



ASI_LBSY/ASI_BST are window ASIs through which user programs can access barrier resources. There are 12 window ASIs, which are specified by the VA.

Bit	Field	Access	Description
63:1	reserved		
0	Value	RW	A read returns LBSY of the Barrier Blade assigned to the window ASI. A write updates the BST bit.

A read to an unassigned window ASI returns an undefined value. A write to an unassigned window is ignored; no exception is generated.

Sample Code for Barrier Synchronization

```
/*
 * %r1: VA of a window ASI
 * %r2, %r3: work registers
 */
```

```
    ldxa      [%r1]ASI_LBSY, %r2 !  read current LBSY
    not       %r2                !  flip LBSY bit
    and       %r2, 1, %r2        !  mask reserved bits
    stxa      %r2, [%r1]ASI_BST  !  update BST
    membar    #storeload         !  wait for stxa to complete

loop:
    ldxa      [%r1]ASI_LBSY, %r3 !  read LBSY
    and       %r3, 1, %r3        !  mask reserved bits
    subcc     %r3, %r2, %g0      !  check if status changed
    bne,a     loop
    sleep                        !  if not changed, sleep for a while
```

F.APPENDIX **M**

Cache Organization

M.1 Cache Types

SPARC64 VIIIfx has two levels of on-chip cache, with the following characteristics:

- Split level-1 instruction and data caches; the level-2 cache is unified.
- Level-1 caches are virtually indexed, physically tagged (VIPT); the level-2 cache is physically indexed, physically tagged (PIPT).
- Cache line size for both level-1 and level-2 caches is 128 bytes.
- All lines in the level-1 caches are included in the level-2 cache.
- Hardware maintains cache coherency between level-1 caches and between level-1 caches and the level-2 cache. That is,
  - When a cache line in the level-2 cache is invalidated and that data is present in level-1 cache(s), those cache line(s) are also invalidated.
  - When a self-modifying instruction stream updates data in a level-1 data cache, the corresponding instruction sequence in the level-1 instruction cache is invalidated.
- The level-2 cache is shared by all the cores in a processor module.

M.1.1 Level-1 Instruction Cache (L1I Cache)

The characteristics of a level-1 instruction cache are shown below.

Feature	Value
Size	32 Kbytes
Associativity	2-way
Line Size	128-byte
Indexing	Virtually indexed, physically tagged (VIPT)
Tag Protection	Parity and duplication
Data Protection	Parity
Misc. Features	—

Although L1I caches are VIPT, the `TTE.CV` bit is meaningless because SPARC64 VIIIfx implements hardware unaliasing.

Instructions fetched from noncacheable address spaces are not cached in L1I caches. Noncacheable accesses occur in the following 3 cases:

- `PSTATE.RED` = 1
- `DCUCR.IM` = 0
- `TTE.CP` = 0

When `MCNTL.NC_CACHE` = 1, SPARC64 VIIIfx treats all instructions as instructions in cacheable address spaces, regardless of the conditions listed above. See *"ASI_MCNTL (Memory Control Register)"* (page 185) for details.

Programming Note – This feature is intended to be used by the OBP to facilitate diagnostics procedures. When the OBP uses this feature, it must clear `MCNTL.NC_CACHE` and invalidate all L1I data via `ASI_FLUSH_L1I` before exiting.

M.1.2 Level-1 Data Cache (L1D Cache)

Level-1 data caches are writeback caches. Their characteristics are shown below.

Feature	Value
Size	32 Kbytes
Associativity	2-way
Line Size	128-byte
Indexing	Virtually indexed, physically tagged (VIPT)
Tag Protection	Parity and duplication
Data Protection	ECC
Misc. Features	Sector Cache

Although L1D caches are VIPT, the `TTE.CV` bit is meaningless because SPARC64 VIIIfx implements hardware unaliasing.

Data accessed from noncacheable address spaces are not cached in L1D caches. Noncacheable accesses occur in the following 3 cases:

- Accesses via `ASI_PHYS_BYPASS_EC_WITH_E_BIT` (15_{16}) or `ASI_PHYS_BYPASS_EC_WITH_E_BIT_LITTLE` ($1D_{16}$).
- `DCUCR.DM` = 0
- `TTE.CP` = 0

Data in noncacheable address spaces are not cached in L1D caches, regardless of the value of `MCNTL.NC_CACHE`.

M.1.3 Level-2 Unified Cache (L2 Cache)

The level-2 unified cache is a writeback cache. Its characteristics are shown below.

Feature	Value
Size	6Mbytes
Associativity	12-way
Line Size	128-byte
Indexing	Physically indexed, physically tagged (PIPT)
Tag Protection	ECC
Data Protection	ECC
Misc. Features	Index Hash, Sector Cache

Data in noncacheable address spaces are not cached in the L2 cache, regardless of the value of `MCNTL.NC_CACHE`.

Index Hash

In SPARC64 VIIIfx, L2 cache indexes are generated using the following hash function:

- index<11:9> = PA<33:31> **xor** PA<30:28> **xor** PA<27:25> **xor** PA<24:22> **xor** PA<21:19> **xor** PA<18:16>
- index<8:0> = PA<15:7>

M.2 Cache Coherency Protocols

Note – SPARC64 VIIIfx does not support multiprocessor configurations. This section has been deleted.

M.3 Cache Control/Status Instructions

M.3.1 Flush Level-1 Instruction Cache L1 (ASI_FLUSH_L1I)

Register Name	ASI_FLUSH_L1I
ASI	67_{16}
VA	Any 8-byte aligned VA
Access Type	Supervisor write only

ASI_FLUSH_L1I invalidates all contents of the level-1 instruction cache in the core that executed the ASI store. A write to this ASI with any 8-byte aligned VA and any data invalidates the L1I cache.

ASI_FLUSH_L1I is write-only. An attempt to read the register causes a *data_access_exception* exception.

M.3.2 Cache invalidation (ASI_CACHE_INV)

Register Name	ASI_CACHE_INV
ASI	74_{16}
VA	Physical Address
Access Type	Supervisor write only

ASI_CACHE_INV writes the specified cache line to memory, then invalidates the copies in the L1 caches of all on-chip cores and in the L2 cache. Cache lines are specified by the physical address indicated in the VA field.

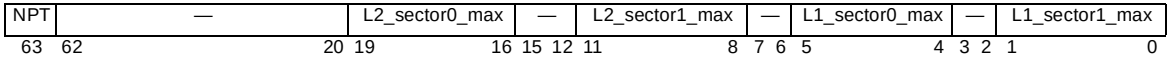
ASI_CACHE_INV is write-only. An attempt to read the register causes a *data_access_exception* exception.

Note – If DCUCR.WEAK_SPCA = 0, cache lines invalidated by ASI_CACHE_INV may immediately reenter the cache due to speculative execution and/or hardware prefetches. To guarantee that the cache does not contain the specified data, DCUCR.WEAK_SPCA should be set to 1 before executing ASI_CACHE_INV.

M.3.3 Sector Cache Configuration Register (SCCR)

Register Name	ASI_SCCR
ASI	$E7_{16}$
VA	00_{16}
Access Type	User read/write (with restrictions)

The ASI_SCCR controls the settings for the sector cache. There is only one SCCR for the entire CPU; it is shared by all of the cores.



Bit	Field	Access	Description
63	NPT	RW	Privileged access. When NPT = 1 and PSTATE.priv = 0, an attempted access to the SCCR causes a *privileged_action* exception. When NPT = 0, user software can set NPT to 1.
62:	—		reserved.
19:16	L2_sector0_max	RW	Maximum number of ways in the L2 cache that can be used by sector 0.
15:12	—		reserved.
11:8	L2_sector1_max	RW	Maximum number of ways in the L2 cache that can be used by sector 1.
7:6	—		reserved.
5:4	L1_sector0_max	RW	Maximum number of ways in the L1 cache that can be used by sector 0. If one core updates this field, the L1 cache settings for all cores are updated.
3:2	—		reserved.
1:0	L1_sector1_max	RW	Maximum number of ways in the L1 cache that can be used by sector 1. If one core updates this field, the L1 cache settings for all cores are updated.

Warning – Because the entire chip shares the SCCR, if a core is currently using the sector cache and another core sets SCCR.NPT to 1, the first core can no longer access the SCCR.

SPARC64 VIIIfx introduces a mechanism for splitting caches into two “sectors” that can be managed separately. This organization is called a sector cache. Sectors are specified by memory access instructions; the accessed data is stored in the specified sector. In SPARC64 VIIIfx, sector caches are implemented for both the L1 and L2 caches. L1 and L2 sector cache mechanisms can be enabled and disabled independently.

The size of a sector specifies the maximum number of cache ways per index that can be used by a sector. In a set-associative cache, a single index corresponds to multiple ways; for a given index, the sector sizes specify the maximum number of ways used by sector 0 and the maximum number of ways used by sector 1. All indexes have the same sector sizes; that is, sector sizes cannot be specified individually for each index.

For the sector cache to be valid, the sector sizes for sectors 0 and 1 must be at least 1 cache way. If a sector size larger than the number of cache ways is specified, the sector size is assumed to be the number of ways. The sum of the sector sizes does not need to equal the number of ways in the cache. When the number of ways of either sector is 0, the sector cache is not valid.

The sector cache mechanism affects the replacement of cache data. When the sector cache is not valid, evicted entries are selected from all cache ways. When the sector cache is valid, evicted entries are selected such that each sector does not exceed its specified sector size. That is, if the number of entries at that index for that sector is less than the sector size, the evicted entry is selected from cache ways that are not part of the sector. If the number of entries at that index is greater than or equal to the sector size, the evicted entry is selected from that sector.

Regardless of whether the sector cache is valid or whether there is an access to data in the cache, software can always access data in all cache ways. If an access specifies a different sector than the sector of the data being accessed, the sector of the data being accessed is updated.

Notes – Sector information is updated for data reads and prefetches.
Sector information is specified for each cache line. Accesses to different data in a cache line may specify different sectors, but the sector specified for the entire cache line is the sector specified by the last access.

Memory access instructions (load/store/atomic/prefetch) specify the cache sector using `XAR.sector` (`XAR.urs3`<0>). If `XAR.sector` = 0, then sector 0 is specified; if `XAR.sector` = 1, then sector 1 is specified.

Sector information and the sector cache mechanism are distinct concepts. Sector information describes an attribute of the data; the sector cache mechanism describes the cache replacement policy. Even if the sector cache mechanism is disabled, sector information is always preserved. For example, if the L1 sector cache mechanism is disabled while the L2 sector cache mechanism is enabled, L1 write-back data is updated in the the L2 cache based on the sector information of that data.

Implementation Note – The method and timing for communicating changes in the sector information of an L1 cache to the L2 cache is implementation dependent.

The maximum number of ways for each sector is used to determine how cache data should be updated. When these numbers are set, however, the number of ways currently allocated to a sector may exceed the new maximum; these cache ways are not forcefully invalidated. For example, when sector 0 uses 5 ways and the maximum number of ways for sector 0 is set to 2, SPARC64 VIIIfx does not instantly invalidate 3 of these ways. It could be said that the maximum number of ways is in fact the target number of ways that should be allocated to a given cache sector.

This document does not specify how each sector should be used.

The algorithm for sector cache operation is explained below. Because this algorithm is the same for the L1 and L2 caches, the `L1_` and `L2_` prefixes are dropped in the following subsections. The number of ways in the cache is written as *nway* .

Setting the SCCR value

- When `sector0_max` > 0 and `sector1_max` > 0, the sector cache is valid.
- When `sector0_max` = 0 or `sector1_max` = 0, the sector cache is not valid.
- It is not necessary that `sector0_max` + `sector1_max` = nway.

Managing the Sector Cache

The number of cache ways used by sector 0 is described by `sector0_use`, and the number of ways used by sector 1 is described by `sector1_use`. The following are always true:

`sector0_use` + `sector1_use` ≤ *nway*
0 ≤ `sector0_use` ≤ *nway*, 0 ≤ `sector1_use` ≤ *nway*

Behavior when a memory access to sector number *S* is requested:

- When a cache hit occurs in a way that is assigned to a different sector than *S*, the number of ways used by each sector is adjusted.

  `sector`*S*`_use++`, `sector`*T*`_use--` (where sector *T* is the other sector)

  This may cause `sector`*S*`_use` > `sector`*S*`_max` (when `sector`*S*`_max` < *nway*).

- When there is a cache miss
  - If there is an empty way, that way is assigned to sector *S*.

    `sector`*S*`_use++`

    This may cause the value of `sector`*S*`_use` to be larger than the value of `sector`*S*`_max`.

- If sector*S*_use < min(*nway*, sector*S*_max), the oldest way in sector *T* is replaced and assigned to sector *S*.

  sector*S*_use++, sector*T*_use--

- If sector*S*_use ≥ min(*nway*, sector*S*_max), the oldest way in sector *S* is replaced and assigned to sector *S*.

  sector*S*_use and sector*T*_use are unchanged

  Even if sector*S*_use > min(*nway*, sector*S*_max), the value of sector*S*_use does not decrease. It is necessary to access sector *T* to move the value of sector*S*_use closer to the value of min(*nway*, sector*S*_max).

Behavior when the Sector Cache is Not Valid

- When a cache miss occurs and all cache ways are occupied, the oldest way is selected to be replaced. sector*S*_use and sector*T*_use are not used.

  Even when the sector cache is not valid, sector information is preserved.

Note – Because SPARC64 VIIIfx processes memory accesses out of order, sector information may not be updated according to the intentions of the user program.

XAR.sector can be specified for all XAR-eligible memory access instructions, but this is only meaningful when the access is to an address space with TTE.CP = 1. When the access is to an address space with TTE.CP = 0 or to a nontranslating ASI, the value of XAR.sccs is ignored; no exception is signalled.

M.4 Hardware Prefetch

SPARC64 VIIIfx implements hardware that detects memory accesses to consecutive, cacheable addresses and generates prefetches.[1] The hardware prefetch mechanism monitors load and store instructions to cacheable address spaces; the PREFETCH, PREFETCHA, LDSTUB, LDSTUBA, SWAP, SWAPA, CASA, CASXA, block load/store, partial store, short load/store, and xfill instructions are not monitored.

The behavior of the hardware prefetch mechanism is described below:

1. When a ld/st instruction misses in the L1 cache (at address A), hardware starts monitoring the adjacent cache lines (A+128, A-128).

1. Here, consecutive addresses means addresses that are in consecutive cache lines (128 bytes).

2. If there is an access to a monitored address (for example, A+128), a prefetch is generated for the adjacent cache line (A+256). A the same time, that cache line (A+256) is monitored for ld/st accesses.

3. A ld/st access to A+256 generates a prefetch to A+384.

A cache miss triggers monitoring for cache accesses; a cache access to a monitored address, regardless if it hits or misses, causes a consecutive access.

Thus, if there are a large number of such consecutive accesses, distant addresses may be prefetched and/or data may be prefetched into the L1 cache (initially, data is only prefetched into the L2 cache).

Software can control the hardware prefetch mechanism in two ways:

1. `ASI_MCNTL.hpf` turns the entire hardware prefetch mechanism on/off. See *"ASI_MCNTL (Memory Control Register)"* (page 185) for details.

2. `XAR.dis_hw_pf` turns hardware prefetch on/off for individual instructions. When `XAR.dis_hw_pf` = 1 and a ld/st instruction misses in the L1 cache, adjacent addresses are not monitored for cache misses. When `XAR.dis_hw_pf` = 0 and a ld/st instruction misses in the L1 cache, adjacent addresses are monitored for cache misses (if `ASI_MCNTL.hpf` = 1).

Note – The SPARC64 VIIIfx specification does not define the type of prefetches generated by the hardware prefetch mechanism.

The `XAR.dis_hw_pf` bit can be set for all XAR-eligible memory access instructions, but this is only meaningful for load and store instructions to address spaces with `TTE.CP` = 1. The value of `XAR.dis_hw_pf` is ignored for accesses to address spaces with `TTE.CP` = 0, accesses to nontranslating ASIs, and accesses by the `PREFETCH`, `PREFETCHA`, `LDSTUB`, `LDSTUBA`, `SWAP`, `SWAPA`, `CASA`, `CASXA`, block load/store, short load/store, and xfill instructions. No exception is signalled.[1]

1. Because the partial store instruction is not `XAR`-eligible, the hardware prefetch bit cannot be set.

F.APPENDIX **N**

Interrupt Handling

N.1 Interrupt Vector Dispatch

When a processor[1] dispatches an interrupt to another processor, software first writes the interrupt data to `ASI_INTR_DATA_[0-2]W`. A subsequent write to `ASI_INTR_DISPATCH_W` triggers the interrupt delivery. The processor polls `INTR_DISPATCH_STATUS`'s BUSY and BUSY bits to determine whether the interrupt has been successfully delivered. FIGURE N-1 illustrates the steps for interrupt dispatch.

1. Here, a processor is the unit of hardware that executes instructions. It is equivalent to a SPARC64 VIIIfx core.

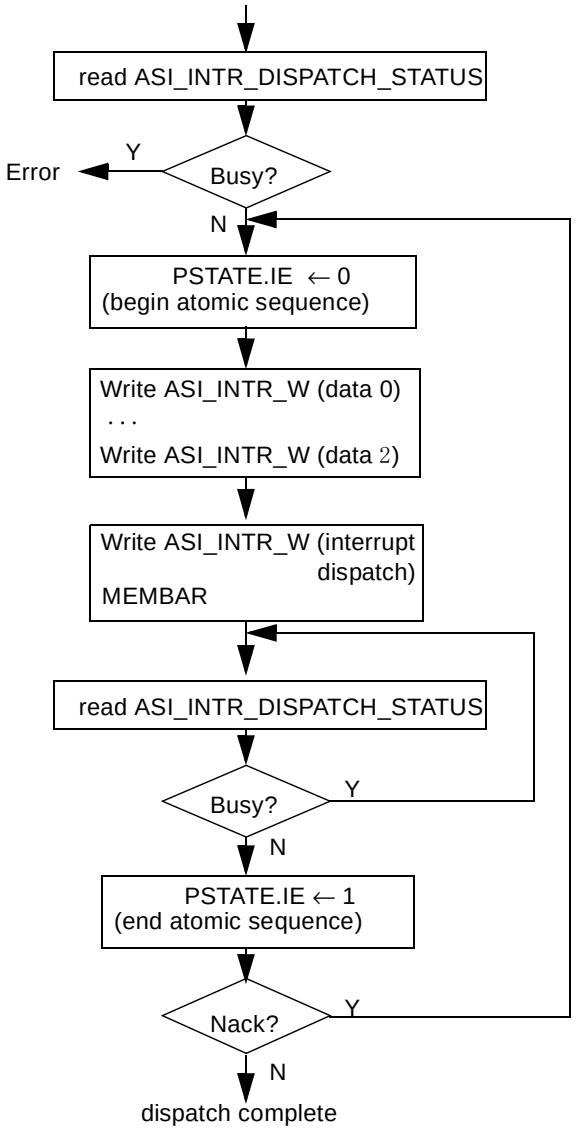


FIGURE N-1 Dispatching an Interrupt

N.2 Interrupt Vector Receive

When an interrupt packet is received, ASI_INTR_DATA_[0-2]R are updated with the incoming data in conjunction with the setting of the BUSY bit in the ASI_INTR_RECEIVE register. If interrupts are enabled (PSTATE.IE = 1), then an interrupt trap is generated. Software reads the data to determine the entry point of the appropriate trap handler. The handler may reprioritize the trap as a lower-priority interrupt in the software handler.

If an error is detected in an incoming packet, the BUSY bit in the ASI_INTR_RECEIVE register is not set. In this case, ASI_INTR_DATA_[0-2]R may also contain errors and should not be read. See Section P.8.3, *"ASI Register Error Handling"* (page 289) for details.

FIGURE N-2 illustrates the steps for interrupt receive.

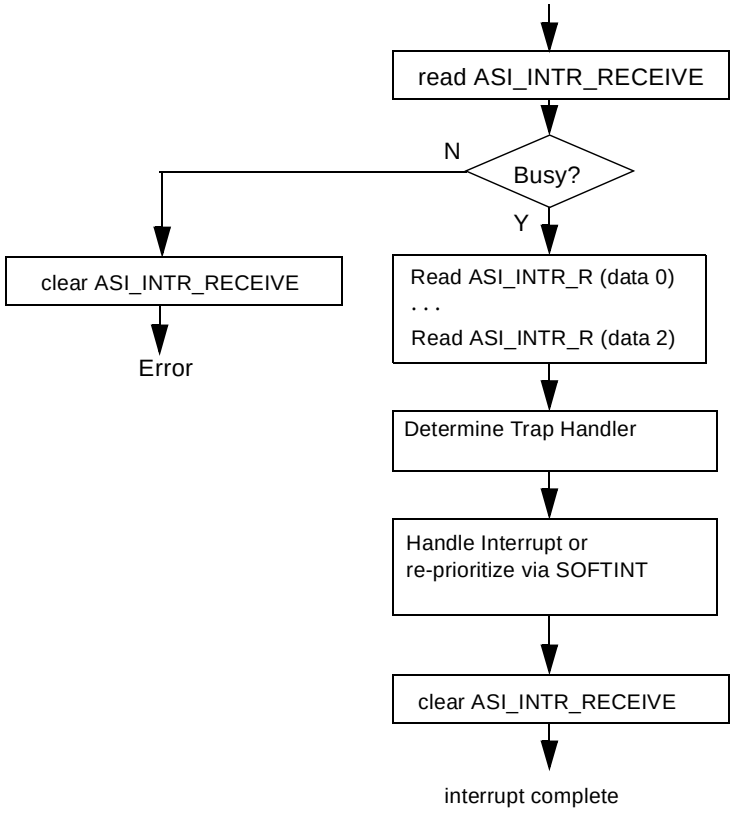


FIGURE N-2 Receiving an Interrupt

N.4 Interrupt ASI Registers

N.4.1 Outgoing Interrupt Vector Data<7:0> Register

Although JPS1 **Commonality** defines eight Outgoing Interrupt Vector Data Registers, SPARC64 VIIIfx only implements three of these registers. An attempt to write `ASI_INTR_DATA_[3-7]W` causes an undefined ASI exception.

Compatibility Note – This change is not compatible with SPARC JPS1.

N.4.2 Interrupt Vector Dispatch Register

In SPARC64 VIIIfx, all 10 `VA`<38:29> bits are ignored when the Interrupt Vector Dispatch Register is written (impl. dep. #246).

SPARC64 VIIIfx implements 8 BUSY/NACK bit pairs. When the `ASI_INTR_DISPATCH_W` register is written, bits `BN`<4:3> (= `VA`<28:27>) are disregarded.

In SPARC64 VIIIfx, bits ITID<9:3> (= `VA`<23:17>) are ignored.

N.4.3 Interrupt Vector Dispatch Status Register

In SPARC64 VIIIfx, 8 BUSY/NACK bit pairs are implemented. Up to 8 interrupts may be outstanding at one time.

Reads to bits <63:16> return 0.

N.4.4 Incoming Interrupt Vector Data Registers

Although JPS1 **Commonality** defines eight Incoming Interrupt Vector Data Registers, SPARC64 VIIIfx only implements three of these registers. An attempt to write `ASI_INTR_DATA_[3-7]R` causes an undefined ASI exception.

Compatibility Note – This change is not compatible with SPARC JPS1.

N.4.5 Interrupt Vector Receive Register

SPARC64 VIIIfx displays a 10-bit value in the SID_H and SID_L fields of the Interrupt Vector Receive Register, but the value displayed is undefined. (impl. dep. #247).

N.6 Identifying an Interrupt Target

SPARC64 VIIIfx has multiple cores in a single processor module. Thus, SPARC64 VIIIfx needs a mechanism for identifying which core should receive the interrupt. The two methods of identification are ASI_SYS_CONFIG.ITID and ASI_EIDR. Firmware intializes ASI_EIDR, which is then used to identify the thread that receives the interrupt.

For correct delivery of interrupt packets, the ASI_EIDR of each core should be initialized with a unique ASI_EIDR<2:0> value. If this value is not unique, it cannot be guaranteed that interrupt packets will be sent to the correct target.

F.APPENDIX **O**

Reset, RED_state, and error_state

This appendix describes behavior after power-on and reset. In JPS1 **Commonality**, reset behavior is described in Chapter 7.1. However, reset behavior is strongly dependent on the hardware implementation; the SPARC64™ VIIIfx Extensions describes that information in this appendix. See Chapter 7.1 for information on software-observable behavior, such as the values of registers on entry into `RED_state` and the `RED_state` trap vector.

This appendix describes the following items:

The sections in this appendix do not match those in JPS1 **Commonality**.

O.1 Reset Types

This section describes the four reset types: power-on resets (POR), externally initiated reset (XIR), watchdog reset (WDR), and software-initiated reset (SIR).

POR and XIR affect all the cores in a processor module. In other words, all the cores process the same trap. On the other hand, WDR and SIR only affect the core that caused the reset. Other cores are unaffected and continue to run.

O.1.1 Power-on Reset (POR)

For a POR to occur in SPARC64 VIIIfx, a sequence of JTAG commands must be issued to the processor by an external facility.

When the reset pin is asserted or the Power Ready signal is de-asserted, the processor halts and enters a state where only JTAG commands can be executed. Except for changes caused by the execution of JTAG commands, the processor does not update any software-visible resources and does not change the state of the memory system.

When a POR is received, the processor enters `RED_state`, causes a *power_on_reset* trap (`TT` = 1), and begins executing instructions at `RSTVaddr` + 20_{16}.

O.1.2 Watchdog Reset (WDR)

A watchdog reset (WDR) is also generated in the following cases:

- `TL` < `MAXTL`, and a second watchdog timeout is detected.
- `TL` = `MAXTL`, and a watchdog timeout is detected.
- `TL` = `MAXTL`, and a trap occurs.

When a watchdog timeout is detected while `TL` < `MAXTL`, the processor causes a *watchdog_reset* exception (`TT` = 2) and begins executing instructions at `RSTVaddr` + 40_{16}. In the other two cases, the CPU enters `error_state` without updating `TT`.

O.1.3 Externally Initiated Reset (XIR)

When an XIR request from the system is received, the processor enters `RED_state`, causes an *externally_initiated_reset* trap (`TT` = 3) and begins executing instructions at `RSTVaddr` + 60_{16}.

O.1.4 Software-Initiated Reset (SIR)

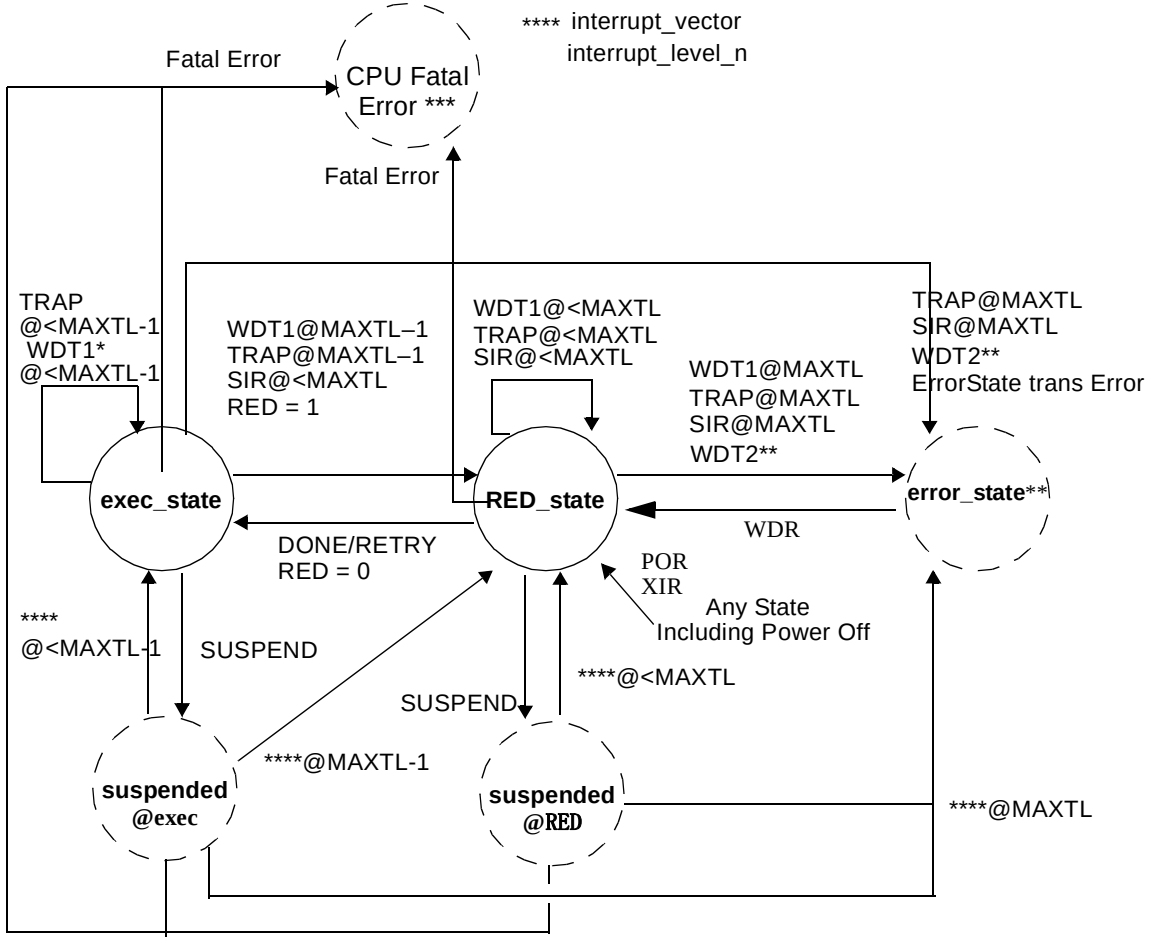
Any core in the CPU chip can initiate a software-initiated reset using an `SIR` instruction.

If an `SIR` instruction is executed while `TL` < `MAXTL` (5), the processor enters `RED_state`, causes *software_initiated_reset* trap (`TT` = 4), and begins executing instructions at `RSTVaddr` + 80_{16}.

If an `SIR` instruction is executed while `TL` = 5, the processor enters `error_state` and eventually generates a watchdog reset trap.

O.2 RED_state and error_state

In addition to the processor states defined in JPS1 **Commonality**, the CPU Fatal Error and suspended states are also defined.



* WDT1 is the initial watchdog timeout.

** WDT2 is the second watchdog timeout. WDT2 causes the CPU to enter `error_state`. Normally, `error_state` immediately generates a watchdog reset trap, and the CPU enters `RED_state`; thus, `error_state` is transient. The `OPSR` (Operation Status Register) can be set so that entry into `error_state` does not cause a watchdog reset, and the CPU remains in `error_state`.

***In `CPU_fatal_error_state`, `P_FERR` indicates that a fatal error has been detected in the CPU, and the system causes a FATAL reset. A soft POR of the CPU is initiated.

FIGURE O-1 Processor State Diagram

O.2.1 RED_state

Also see Section 7.1.1.

Once the processor enters RED_state for any reason except a power-on reset (POR), software should not attempt to return to execute_state. If software attempts a return, the state of the processor is unpredictable.

When a reset or trap causes the processor to enter RED_state, instructions are executed starting from the appropriate offset in the RED_state trap vector, which is located at RSTVaddr. In SPARC64 VIIIfx, RSTVaddr is VA = FFFF FFFF F000 0000_{16}, which is equivalent to PA = 0000 01FF F000 0000_{16}.

Setting PSTATE.RED = 1 also causes the processor to enter RED_state. In this case, the processor does not branch to the RED_state trap vector.

The following list further describes processor behavior on entry to RED_state, and behavior while in RED_state:

- When a reset or trap causes the processor to enter RED_state, hardware invalidates a number of features, and the ASI_DCUCR is updated. If needed, software should reset the values of this register.
- When a condition other than a reset or trap causes the processor to enter RED_state (such as when a WRPR sets PSTATE.RED to 1), DCUCR bits are not updated. The only effect is that the IMMU is disabled.
- While the processor is in RED_state, the IMMU is disabled. That is, the value of DCUCR.IM is ignored.
- Caches coherence is preserved while the processor is in RED_state.

O.2.2 error_state

The processor enters error_state when a trap occurs while TL = MAXTL (5) or when a second watchdog time-out occurs.

Normally, the processor immediately generates a watchdog reset trap (WDR) and enters RED_state. The OPSR (Operating Status Register) can be set such that a watchdog reset is not generated on entry to error_state, and the processor remains in error_state.

O.2.3 CPU Fatal Error state

When the processor detects a fatal error, the processor enters the CPU Fatal Error state. The processor reports the fatal error to the system and halts.

O.3 Processor State after Reset and in RED_state

TABLE O-1, TABLE O-2, and TABLE O-3 show the processor states after various resets and while in RED_state.

Programming Note – To return from error_state, SPARC64 VIIIfx may cause a WDR. In this case, software observes that the cause of the WDR was an entry to error_state; that is, the WDR corresponds to 2 transitions of the hardware state. The WDR column in TABLE O-1, TABLE O-2 and TABLE O-3 shows the state of registers before and after the WDR. This does not include the changes to the register state caused by the entry to error_state.

TABLE O-1 shows the values of the privileged and nonprivileged registers after a trap or reset causes the processor to enter RED_state. If RED_state is entered because a WRPR instruction sets the PSTATE.RED bit, the privileged and nonprivileged registers are not changed, except for the PSTATE.RED bit.

TABLE O-1 Nonprivileged and Privileged Register State after Reset and in RED_state *(1 of 2)*

Name	POR[1]	WDR[2]	XIR	SIR	RED_state
Integer registers	Unknown/Unchanged	Unchanged			
Floating-point registers	Unknown/Unchanged	Unchanged			
RSTV value	VA = FFFF FFFF F000 0000_{16} PA = 01FF F000 0000_{16}				
PC	RSTV | 20_{16}	RSTV |40_{16}	RSTV | 60_{16}	RSTV | 80_{16}	RSTV | $A0_{16}$
nPC	RSTV | 24_{16}	RSTV | 44_{16}	RSTV | 64_{16}	RSTV | 84_{16}	RSTV | $A4_{16}$
PSTATE					
AG	1 (Alternate globals)				
MG	0 (MMU globals not selected)				
IG	0 (Interrupt globals not selected)				
IE	0 (Interrupt disable)				
PRIV	1 (Privileged mode)				
AM	0 (Full 64-bit address)				
PEF	1 (FPU on)				
RED	1 (Red_state)				
MM	00_2 (TSO)				
TLE	0	Unchanged			
CLE	0	Copied from TLE			
TBA<63:15>	Unknown/Unchanged	Unchanged			
Y	Unknown/Unchanged	Unchanged			
PIL	Unknown/Unchanged	Unchanged			

TABLE O-1 Nonprivileged and Privileged Register State after Reset and in RED_state *(2 of 2)*

Name	POR[1]	WDR[2]	XIR	SIR	RED_state
CWP	Unknown/Unchanged	Unchanged except for register-window traps	Unchanged	Unchanged	Unchanged except for register-window traps
TT[TL]	1	trap type or 2	3	4	trap type
CCR	Unknown/Unchanged	Unchanged			
ASI	Unknown/Unchanged	Unchanged			
TL	MAXTL	min (TL + 1, MAXTL)			
TPC[TL]	Unknown/Unchanged	PC			
TNPC[TL]	Unknown/Unchanged	nPC			
TSTATE CCR ASI PSTATE CWP PC nPC	Unknown/Unchanged	 CCR ASI PSTATE CWP PC nPC			
TICK NPT Counter	 1 Restart at 0	 Unchanged Count	 Unchanged Restart at 0	 Unchanged Count	
CANSAVE	Unknown/Unchanged	Unchanged			
CANRESTORE	Unknown/Unchanged	Unchanged			
OTHERWIN	Unknown/Unchanged	Unchanged			
CLEARWIN	Unknown/Unchanged	Unchanged			
WSTATE OTHER NORMAL	 Unknown/Unchanged Unknown/Unchanged	 Unchanged Unchanged			
VER MANUF IMPL MASK MAXTL MAXWIN	 0004_{16} 8 Mask dependent 5_{16} 7_{16}				
FSR	0	Unchanged			
FPRS	Unknown/Unchanged	Unchanged			

1.A hard POR occurs during power-on. Soft POR occurs when the reset signal is asserted.

2.The first watchdog timeout is taken in execute_state (PSTATE.RED = 0). The following watchdog timeout or a watchdog timeout while TL = MAXTL causes the processor to enter RED_state. See Appendix O.1.2 for details.

TABLE O-2 shows the values of the ASR registers after a trap or reset causes the processor to enter RED_state. Setting PSTATE.RED with a WRPR instruction does not change the ASR registers.

TABLE O-2 ASR State after Reset and in RED_state

ASR	Name	POR[1]	WDR[2]	XIR	SIR	RED_state
16	PCR UT ST Others	 0 0 Unknown/Unchanged	Unchanged			
17	PIC	Unknown/Unchanged	Unchanged			
18	DCR	Always 0				
19	GSR IM IRND Others	 0 0 Unknown/Unchanged	 Unchanged Unchanged Unchanged			
22	SOFTINT	Unknown/Unchanged	Unchanged			
23	TICK_COMPARE INT_DIS TICK_CMPR	 1 0	 Unchanged Unchanged			
24	STICK NPT Counter	 1 Restart at 0	 Unchanged Count			
25	STICK_COMPARE INT_DIS TICK_CMPR	 1 0	 Unchanged Unchanged			
29	XAR	0	0			
30	XASR	Unknown/Unchanged	Unchanged			
31	TXAR[TL]	Unknown/Unchanged	XAR			

1.A hard POR occurs during power-on. Soft POR occurs when the reset signal is asserted.

2.The first watchdog timeout is taken in execute_state (PSTATE.RED = 0). The following watchdog timeout or a watchdog timeout while TL = MAXTL causes the processor to enter RED_state. See Appendix O.1.2 for details.

TABLE O-3 shows the values of the ASI registers after a trap or reset causes the processor to enter RED_state. Setting PSTATE.RED with a WRPR instruction does not change the ASI registers.

TABLE O-3 ASI Register State after Reset and in RED_state *(1 of 2)*

ASI	VA	Name	POR[1]	WDR[2]	XIR	SIR	RED_state
45_{16}	00_{16}	DCUCR	0	0			
45_{16}	08_{16}	MCNTL RMD Others	 2 0	 2 0			
48_{16}	00_{16}	INTR_DISPATCH_STATUS	0	Unchanged			
49_{16}	00_{16}	INTR_RECEIVE	0	Unchanged			
$4A_{16}$	—	SYS_CONFIG ITID	 System-Defined Value/ Unchanged	 Unchanged			
$4B_{16}$	00_{16}	STICK_CNTL	0	Unchanged			
$4C_{16}$	00_{16}	AFSR	Unknown/Unchanged	Unchanged			
$4C_{16}$	08_{16}	UGESR	Unknown/Unchanged	Unchanged			
$4C_{16}$	10_{16}	ERROR_CONTROL WEAK_ED Others	 1 Unknown/Unchanged	 1 Unchanged			
$4C_{16}$	18_{16}	STCHG_ERR_INFO	Unknown/Unchanged	Unchanged			
$4F_{16}$	00_{16}–38_{16}	SCRATCH_REGs	Unknown/Unchanged	Unchanged			
50_{16}	00_{16}	IMMU_TAG_TARGET	Unknown/Unchanged	Unchanged			
50_{16}	18_{16}	IMMU_SFSR	Unknown/Unchanged	Unchanged			
50_{16}	28_{16}	IMMU_TSB_BASE	Unknown/Unchanged	Unchanged			
50_{16}	30_{16}	IMMU_TAG_ACCESS	Unknown/Unchanged	Unchanged			
50_{16}	60_{16}	IMMU_TAG_ACCESS_EXT	Unknown/Unchanged	Unchanged			
50_{16}	78_{16}	IMMU_SFPAR	Unknown/Unchanged	Unchanged			
53_{16}	—	SERIAL_ID	Constant value	Constant value			
54_{16}	—	ITLB_DATA_IN	Unknown/Unchanged	Unchanged			
55_{16}	—	ITLB_DATA_ACCESS	Unknown/Unchanged	Unchanged			
56_{16}	—	ITLB_TAG_READ	Unknown/Unchanged	Unchanged			
57_{16}	—	ITLB_DEMAP	Unknown/Unchanged	Unchanged			
58_{16}	00_{16}	DMMU_TAG_TARGET	Unknown/Unchanged	Unchanged			
58_{16}	08_{16}	PRIMARY_CONTEXT	Unknown/Unchanged	Unchanged			
58_{16}	10_{16}	SECONDARY_CONTEXT	Unknown/Unchanged	Unchanged			
58_{16}	18_{16}	DMMU_SFSR	Unknown/Unchanged	Unchanged			
58_{16}	20_{16}	DMMU_SFAR	Unknown/Unchanged	Unchanged			
58_{16}	28_{16}	DMMU_TSB_BASE	Unknown/Unchanged	Unchanged			
58_{16}	30_{16}	DMMU_TAG_ACCESS	Unknown/Unchanged	Unchanged			

TABLE O-3 ASI Register State after Reset and in RED_state *(2 of 2)*

ASI	VA	Name	POR[1]	WDR[2]	XIR	SIR	RED_state
58_{16}	38_{16}	DMMU_WATCHPOINT	Unknown/Unchanged	Unchanged			
58_{16}	60_{16}	DMMU_TAG_ACCESS_EXT	Unknown/Unchanged	Unchanged			
58_{16}	68_{16}	SHARED_CONTEXT	Unknown/Unchanged	Unchanged			
58_{16}	78_{16}	DMMU_SFPAR	Unknown/Unchanged	Unchanged			
$5C_{16}$	—	DTLB_DATA_IN	Unknown/Unchanged	Unchanged			
$5D_{16}$	—	DTLB_DATA_ACCESS	Unknown/Unchanged	Unchanged			
$5E_{16}$	—	DTLB_TAG_READ	Unknown/Unchanged	Unchanged			
$5F_{16}$	—	DMMU_DEMAP	Unknown/Unchanged	Unchanged			
60_{16}	—	IIU_INST_TRAP	0	Unchanged			
$6D_{16}$	00_{16}–58_{16}	BARRIER_INIT	0	Unchanged			
$6E_{16}$	00_{16}	EIDR	0/Unchanged	Unchanged			
$6F_{16}$	00_{16}-58_{16}	BARRIER_ASSIGN	0	Unchanged			
77_{16}	40_{16}–50_{16}	INTR_DATA0:2_W	Unknown/Unchanged	Unchanged			
77_{16}	70_{16}	INTR_DISPATCH_W	Unknown/Unchanged	Unchanged			
$7F_{16}$	40_{16}–50_{16}	INTR_DATA0:2_R	Unknown/Unchanged	Unchanged			
$E7_{16}$	00_{16}	SCCR NPT Others	 1 0	Unchanged			
EF_{16}	00_{16}-58_{16}	LBSY, BST	0	Unchanged			

1.A hard POR occurs during power-on. Soft POR occurs when the reset signal is asserted.

2.The first watchdog timeout is taken in execute_state (PSTATE.RED = 0). The following watchdog timeout or a watchdog timeout while TL = MAXTL causes the processor to enter RED_state. See Appendix O.1.2 for details.

O.3.1 Operating Status Register (OPSR)

The OPSR is the control register for the CPU chip. The value of the OPSR is specified externally and cannot be changed by software. This value is set during the hardware power-on/reset sequence before the CPU starts running and can be changed later using a JTAG command.

Most of the OPSR settings are not visible to software.

F.APPENDIX **P**

Error Handling

This appendix describes the behavior of SPARC64 VIIIfx when an error occurs, as well as information on error recovery for operating system and firmware programmers. Section headings differ from those of Appendix P in JPS1 **Commonality**.

P.1 Error Types

In SPARC64 VIIIfx, errors are divided into the following 4 types:

- Fatal Errors
- Error State Transition Errors
- Urgent Errors
- Restrainable Errors

The SPARC64 VIIIfx processor has eight cores per processor module (cores are single-threaded). The method for identifying which core caused an error depends on the error type.

An error that is caused by instruction execution or that occurs in a thread-specific resource is called an error synchronous to thread execution. These errors are reported to the thread that caused the error. The *instruction_access_error* and *data_access_error* exceptions are belong to this group of errors.

An error that is not caused by instruction execution or that occurs in a resource shared by multiple threads is called an error asynchronous to thread execution. These errors are reported to all threads associated with the resource that caused the error.

Error marking is essentially asynchronous to thread execution. When an unmarked, uncorrectable error (unmarked UE) is detected in the L1$ or L2$, the error is marked by the valid core with the smallest `EIDR`. A valid core is a core that has not been degraded.

Another issue is how to log and report errors when the thread that caused the error is suspended. Except for fatal errors, the error is not reported until the thread exits the suspended state.

P.1.1 Fatal Errors

A fatal error is an error that affects the error system.

a. Data coherency of the system cannot be preserved

All errors that destroy cache coherency belong in this category.

b. Invalid system control flow is detected; validity of subsequent system behavior cannot be guaranteed

When a fatal error is detected, the CPU enters CPU Fatal Error state, reports the occurrence of the fatal error to the system, and halts. After the system receives the report of the fatal error, the system halts.

All fatal errors are asynchronous to thread execution. If a fatal error is detected in a given thread, all threads within the processor module signal a Power On Reset (POR), regardless of whether any threads are suspended.

P.1.2 Error State Transition Errors

An `error_state` transition error (EE) is a serious error that prevents the CPU from reporting the error with a trap. However, any damage caused by the error is limited to within the CPU.

When an `error_state` transition error is detected, the CPU enters `error_state`. The CPU exits `error_state` by causing a watchdog reset, enters `RED_state`, and begins executing the watchdog reset trap handler.

EE asynchronous to thread execution

The following `error_state` transition errors are asynchronous to thread execution. If an EE asynchronous to thread execution is detected in a thread, error information is stored in the `ASI_STCHG_ERROR_INFO` registers of all threads in the core. WDR exceptions are signalled (unless a thread is suspended). Threads in other cores are not affected.

- EE_TRAP_ADR_UE
- EE_OTHER

EE synchronous to thread execution

The following `error_state` transition errors are synchronous to thread execution. If an EE synchronous to thread execution is detected in a thread, error information is stored in the `ASI_STCHG_ERROR_INFO` register of that thread, and a WDR exception occurs. Other threads are not affected.

- EE_SIR_IN_MAXTL

- EE_TRAP_IN_MAXTL
- EE_WDT_IN_MAXTL
- EE_SECOND_WDT

Note – SPARC64 VIIIfx cores are not multi-threaded. The `ASI_STCHG_ERROR_INFO` of the given core stores error information for both `error_state` transition errors synchronous to thread execution and asynchronous to thread execution.

P.1.3 Urgent Errors

An urgent error (UGE) is an error that requires immediate intervention by system software. There are the following types of urgent errors:

- Errors that affect instruction excution
  - I_UGE: Instruction urgent error
  - IAE: Instruction access error
  - DAE: Data access errors
- Errors that are independent of instruction execution
  - A_UGE: Autonomous urgent error

Errors that affects instruction execution

An error that inhibits instruction execution is detected during instruction execution and prevents futher execution.

When the error is detected while `ASI_ERROR_CONTROL.WEAK_ED` = 0 (as set by privileged software for a normal program execution environment), an exception is generated. This error is nonmaskable.

When `ASI_ERROR_CONTROL.WEAK_ED` = 1 (multiple error or during POST/OBP reset processing), one of the following occurs:

- Whenever possible, the CPU writes an indeterminate value to the destination register of the inhibited instruction, and the instruction commits.
- Otherwise, an exception is generated. The inhibited instruction is executed in the same manner as when `ASI_ERROR_CONTROL.WEAK_ED` = 0.

There are three types of errors inhibit instruction execution:

- I_UGE **(instruction urgent error)** — Errors other than IAE (instruction access error) and DAE (data access error). I_UGEs are divided into two groups.
  - **An uncorrectable error in an internal software-visible register that inhibits instruction execution**

An uncorrectable error in the PSTATE, PC, NPC, CCR, ASI, FSR, or GSR register belongs to this group of errors. The first watchdog timeout also belongs to this group of I_UGEs.

- **An error in the execution unit**

  Errors in the execution unit, errors in the temporary registers, and internal bus errors belong to this group of errors.

I_UGE is equivalent to a preemptive error, which is described in Appendix P.2.2.

- IAE **(instruction access error)** — The *instruction_access_error* exception, as defined in JPS1 **Commonality**. In SPARC64 VIIIfx, when an UE is detected in the cache or main memory during instruction fetch, an IAE is generated.

  IAE is a precise exception.

- DAE **(data access error)** — The *data_access_error* exception, as defined in JPS1 **Commonality**. In SPARC64 VIIIfx, when an UE is detected in the cache or main memory during a data access, a DAE is generated.

  DAE is a precise exception.

Urgent Error Independent of Instruction Execution

- A_UGE **(Autonomous Urgent Error)** — An error that occurs independent of instruction execution and requires immediate processing.

  During normal program execution, ASI_ERROR_CONTROL.WEAK_ED = 0. In this case, an A_UGE exception is suppressed during processing of the UGE (that is, in the *async_data_error* trap handler).

  Otherwise, in cases such as a multiple error or during POST/OBP reset processing, ASI_ERROR_CONTROL.WEAK_ED = 1 is set by software. In this case, an A_UGE exception is not generated.

  There are two types of A_UGE:

  - An error that occurs in an important resource and that causes a fatal error or error_state transition error is when the resouce is used.
  - An error that occurs in an important resource and that causes an OS panic.

    OS panic occurs when the resource containing the error is used and execution cannot be continued.

  A_UGE is a disrupting error, with the following differences from SPARC V9:

  - PSTATE.IE = 0 does not mask an A_UGE trap.
  - There are cases where the instruction pointed to by TPC cannot complete precisely. The completion method for the instruction is displayed in the trap status register.

Exception Signalling for Urgent Errors

When an urgent error is detected and not masked, the error is reported to system software by one of the following exceptions:

- I_UGE, A_UGE: *async_data_error* exception
- IAE: *instruction_access_error* exception
- DAE: *data_access_error* exception

Urgent error asynchronous to thread execution

The following errors are asynchronous to thread execution. If these errors occur in a thread, the ASI_UGESR registers of all threads in the core record the error, and *async_data_error* exceptions are signalled. Suspended threads do not signal the exception. Other threads are not affected.

- IAUG_CRE
- IAUG_TSBCTXT
- IUG_TSBP
- IUG_PSTATE
- IUG_TSTATE
- IUG_%F (excluding `f[n]` parity errors)
- IUR_%R (excluding `r[n]` or `Y` parity errors)
- IUG_WDT
- IUG_DTLB
- IUG_ITLB
- IUG_COREERR

Urgent error synchronous to thread execution

The following errors are synchronous to thread execution. If these errors occur in a thread, only the ASI_UGESR register of that thread records the error. An *async_data_error* exceptionis signalled, unless the thread is suspended. Other threads are not affected.

- IUG_%F (`f[n]` parity error only)
- IUR_%R (`r[n]` or `Y` parity error only)

Note – SPARC64 VIIIfx cores are not multi-threaded. The `ASI_UGESR` of the given core records error information for both urgent errors synchronous to thread execution and asynchronous to thread execution.

P.1.4 Restrainable Errors

A restrainable error is an error that does not require immediate handling by system software because it does not seriously affect the currently executing program. A restrainable error causes a disrupting trap with low priority.

There are two types of restrainable errors:

- Uncorrectable errors that do not affect the currently executing instruction sequence.

  An error detected during a cache line writeback or copyback data belongs to this group.

- Degrade Error

  When errors occur frequently, a resource that can be isolated without seriously affecting instruction execution is degraded; that is, the resource is no longer used. Some performance is sacrificed.

Compatibility Note – When SPARC64 VIIIfx detects a correctable error (CE), the error is automatically corrected. Software is not notified.

A restrainable error is reported by the *ECC_error* trap. This trap only occurs when a restrainable error can be signalled and `PSTATE.IE` = 1.

DG_U2$, UE_RAW_L2$INSD

These errors are asynchronous to thread execution. When these errors are detected, the ASI_AFSR registers of all threads in the processor module record the error, and *ECC_error* exceptions are signalled. Suspended threads do not signal the exception.

DG_D1$sTLB, UE_RAW_D1$INSD

These errors are asynchronous to thread execution. When these errors are detected, the ASI_AFSR registers of all threads in the core record the error, and *ECC_error* exceptions are signalled. Suspended threads do not signal the exception.

Threads in other cores are not affected.

UE_DST_BETO

This error is synchronous to thread execution. When this error is detected, the ASI_AFSR register of the thread that caused the error records the error. An *ECC_error* exception is signalled, unless the thread is suspended. Other threads are not affected.

P.1.5 instruction_access_error

This error is synchronous to thread execution. When this error is detected, the ASI_ISFSR, TPC, and ASI_ISFPAR registers of the thread that caused the error record the error. An *instruction_access_error* exception is signalled. Other threads are not affected.

P.1.6 data_access_error

This error is synchronous to thread execution. When this error is detected, the ASI_DSFSR, ASI_DSFAR, and ASI_DSFPAR registers of the thread that caused the error record the error. A *data_access_error* exception is signalled. Other threads are not affected.

P.2 Error Handling and Error Control

P.2.1 Registers Used for Error Handling

TABLE P-1 lists the registers used for error handling. The ASI_ERROR_CONTROL register controls whether an exception is signalled when an error is detected, and ASI_EIDR stores the ID used for error marking. The other registers display information on the error.

TABLE P-1 Registers Used for Error Handling

ASI	VA	Name	Location of Description
$4C_{16}$	00_{16}	ASI_ASYNC_FAULT_STATUS	P.7.1
$4C_{16}$	08_{16}	ASI_URGENT_ERROR_STATUS	P.4.1
$4C_{16}$	10_{16}	ASI_ERROR_CONTROL	P.2.6
$4C_{16}$	18_{16}	ASI_STCHG_ERROR_INFO	P.3.1
50_{16}	18_{16}	ASI_IMMU_SFSR	F.10.9
50_{16}	78_{16}	ASI_IMMU_SFPAR	F.10.12
58_{16}	18_{16}	ASI_DMMU_SFSR	F.10.9
58_{16}	20_{16}	ASI_DMMU_SFAR	F.10.10 of JPS1 **Commonality**
58_{16}	78_{16}	ASI_DMMU_SFPAR	F.10.12
$6E_{16}$	00_{16}	ASI_EIDR	P.2.5

P.2.2 Summary of Behavior During Error Detection

Behavior during error detection is described below.

Conditions that Inhibit Error Detection

Error Type	Conditions Inhibiting Detection
Fatal error	None (always detected).
`error_state` transistion error	When `ASI_ECR.WEAK_ED` = 1, most errors are not detected.
Urgent error	**I_UGE, IAE, DAE:** • When `ASI_ECR.WEAK_ED` = 1 or in a suspended state, most errors are not detected. **A_UGE:** • In a suspended state, most errors are not detected. • Errors that are not associated with register use are restrained when `ASI_ECR.WEAK_ED` = 1, or for individual error conditions. Errors that are associated with register use are restrained for individual error conditions. (There are few individual error conditions.)
Restrainable error	None.

Conditions that Inhibit Exception Signalling when an Error is Detected

Error Type	Conditions Inhibiting Signalling
Fatal error	None (always detected).
`error_state` transistion error	None (always detected).
Urgent error	**I_UGE, IAE, DAE:** • In a suspended state. **A_UGE:** • When `ASI_ECR.UGE_HANLDER` = 1. • When `ASI_ECR.WEAK_ED` = 1. If the exception is masked when detected, the trap is delayed. Once the exception is no longer masked, *async_data_error* is signalled. • In a suspended state.
Restrainable error	• When `ASI_ECR.UGE_HANLDER` = 1. • When `ASI_ECR.WEAK_ED` = 1. • When `PSTATE.IE` = 1. • When the error is masked. The fields `ASI_ECR.RTE_DG` and `ASI_ECR.RTE_UE` mask different types of errors. • In a suspended state.

Behavior During Error Detection

Error Type	Behavior
Fatal error	1. CPU enters the CPU Fatal Error state. 2. CPU notifies the system that a fatal error has occurred. 3. The system halts.
`error_state` transistion error	1. CPU enters `error_state`. 2. A `WDR` is signalled by the CPU.

Error Type	Behavior
Urgent error	**I_UGE:** • When ASI_ECR.UGE_HANLDER = 0, a single-ADE trap occurs. • When ASI_ECR.UGE_HANLDER = 1, a multiple-ADE trap occurs. **A_UGE:** • When exception signalling is not masked, a single-ADE trap occurs. • When exception signalling is masked, notification of the exception is pending. **IAE:** • When ASI_ECR.UGE_HANLDER = 0, an IAE exception is signalled. • When ASI_ECR.UGE_HANLDER = 1, a multiple-ADE trap occurs. **DAE:** • When ASI_ECR.UGE_HANLDER = 0, a DAE exception is signalled. • When ASI_ECR.UGE_HANLDER = 1, a multiple-ADE trap occurs.
Restrainable error	When exception signalling is not masked, an *ECC_error* exception may be signalled even though ASI_AFSR does not display any error information. 1. When error notification is pending and a write to ASI_AFSR occurs, the error information is overwritten. 2. When an UE is detected and an *ECC_error* is signalled, a write to ASI_AFSR erases a pending DG. 3. When a DG is detected and an *ECC_error* is signalled, a write to ASI_AFSR erases a pending UE. When such exceptions are signalled, system software should ignore the exception and continue processing.

Relationship between TPC and the Instruction that Caused the Error

Error Type	Behavior
Fatal error	No relationship.
error_state transistion error	No relationship.
Urgent error	**I_UGE:** • For TLB write errors, TPC points to the instruction that attempted to update the TLB; TPC may also point to the instruction that immediately preceded the instruction that attempted to update the TLB. A TLB write error is detected when a subsequent DONE/RETRY instruction is executed, or an exception is signalled. • For all other errors, TPC points to the instruction that follows the instruction causing the error. **A_UGE:** • No relationship. **IAE, DAE** • TPC points to the instruction that caused the error.
Restrainable error	No relationship.

Other

Priority when Multiple Types of Errors are Detected Simultaneously

Fatal Error	error_state transition error	Urgent Error	Restrainable Error
1. Enter fatal error state (TT = 1)	2. Enter error_state (TT = 2)	3. ADE (TT = 40_{16}) 4. DAE (TT = 32_{16}) 5. IAE (TT = $0A_{16}$)	6. ECC_error_trap (TT = 63_{16})

Completion Method for an Interrupt Instruction

Fatal Error	error_state transition error	Urgent Error	Restrainable Error
Cannot commit.	Cannot commit.	**ADE:** • See P.4.3. **IAE, DAE:** • Conforms to the JPS1 definition for a precise exception.	Conforms to the JPS1 definition for a precise exception.

Error Display Registers

Fatal Error	`error_state` transition error	Urgent Error	Restrainable Error
	ASI_STCHG_ ERROR_INFO	**I_UGE, A_UGE:** • ASI_UGESR **IAE:** • ASI_ISFSR **DAE:** • ASI_DSFSR	ASI_AFSR

Number of Errors Signalled by One Exception

Fatal Error	`error_state` transition error	Urgent Error	Restrainable Error
All fatal errors are detected.	All `error_state` transition errors are detected and displayed in ASI_STCHG_ ERROR_INFO.	**Single ADE:** • All I_UGE and A_UGE are detected. **Multiple ADE:** • If a multiple ADE trap occurs, the first ADE is displayed in ASI_UGESR. **IAE:** • Only one is shown. **DAE:** • Only one is shown.	All restrainable errors are detected and displayed in ASI_AFSR.

P.2.3 Limits to Automatic Correction of Correctable Errors

When a correctable error (CE) is detected, the CPU corrects the input data and proceeds with the operation; however, there are limits to whether the source data can be corrected automatically. The following data cannot be corrected automatically:

- CE in memory
- CE in received interrupt data (`ASI_INTR_DATA_R`)

When other correctable errors are detected, the CPU can automatically correct the source data containing the CE.

For a CE in `ASI_INTR_DATA`, no special action is required by the OS because the error data will be overwritten when the next interrupt is received. For a CE in memory, it is expected that the OS will correct the error.

P.2.4 Error Marking for Cacheable Data

Error Marking for Cacheable Data

When hardware first detects an uncorrectable error (UE) in cacheable data, the data and ECC are replaced with a particular pattern. Using this pattern, the presence of an error can be identified, and the source of the error can be determined. This is called error marking. Error marking specifies the source of the error and prevents a single error from being reported multiple times.

The following data in the system are ECC protected:

- Main memory
- Data bus between memory and ICC
- U2 cache data
- D1 cache data

When the CPU detects an unmarked UE, error marking is performed.

Whether data containing an UE has been marked or not is determined from the ECC syndrome of each doubleword, as shown in TABLE P-2.

TABLE P-2 Syndrome for Marked Data

Syndrome	Error Marking Status	Type of UE
$7F_{16}$	Marked	Marked UE
Multi-bit error pattern other than $7F_{16}$	Not marked yet	Unmarked UE (Raw UE)

The syndrome $7F_{16}$ indicates that a 3-bit error occurred in the doubleword. Error marking introduces the ECC syndrome in the doubleword when the original data and ECC are replaced, as explained in the following section. The probability of syndrome $7F_{16}$ occurring when the data does not contain a marked UE is considered to be zero.

Format for Error-Marking Data

When an unmarked UE is detected in cacheable data, the doubleword containing the error and the corresponding ECC are replaced with error-marking data, which has the format described in TABLE P-3.

TABLE P-3 Format for Error-Marking Data

Data/ECC	Bits	Value
data	63	Error bit. The value is indeterminate.
	62:56	0 (7 bits).
	55:42	ERROR_MARK_ID (14 bits).
	41:36	0 (6 bits).
	35	Error bit. The value is indeterminate.
	34:23	0 (12 bits).
	22	Error bit. The value is indeterminate.
	21:14	0 (8 bits).
	13:0	ERROR_MARK_ID (14 bits).
ECC		This pattern indicates a 3-bit error in bits 63, 35, and 22. That is, this pattern is set so that a syndrome of $7F_{16}$ is detected.

The ERROR_MARK_ID (14 bits) indicates the source of the error. The hardware that detected sets this value.

The format of ERROR_MARK_ID is described in TABLE P-4.

TABLE P-4 ERROR_MARK_ID Bit Description

Bits	Value
13:12	Module_ID. Indicates the hardware where the error occurred. 00_2: Memory system (including DIMM) 01_2: Channel 10_2: CPU 11_2: Reserved
11:0	Source_ID. When Module_ID = 00_2, the 12-bit Source_ID field is always 0. Otherwise, the Source ID is set to the ID of the hardware that detected the error.

ERROR_MARK_ID Set by CPU

TABLE P-5 shows the ERROR_MARK_ID set by the CPU.

TABLE P-5 ERROR_MARK_ID Set by CPU

Type of unmarked UE	Module_ID	Source_ID
Incoming data from memory	00_2 (Memory system)	0
Outgoing data to memory	10_2 (CPU)	$1\ 0000\ 0000_2$ ▯ 000_2
U2 cache data	10_2 (CPU)	$1\ 0000\ 0000_2$ ▯ 000_2
D1 cache data	10_2 (CPU)	$0\ 0000\ 0000_2$ ▯ ASI_EIDR<2:0>

P.2.5 ASI_EIDR

The ASI_EIDR register stores information needed to form the Source_ID of the ERROR_MARK_ID. This information is also used for identifying the interrupt target (see Appendix N.6).

Register name	ASI_EIDR
ASI	$6E_{16}$
VA	00_{16}
Error Detection	Parity
Format	See TABLE P-6

TABLE P-6 ASI_EIDR Bit Description

Bit	Field	Access	Description
63:3	Reserved	R	Always 0.
2:0	ERROR_MARK_ID	RW	When an error occurs in the CPU, this field is copied to the ERROR_MARK_ID of the error data.

Compatibility Note – In SPARC64 VII, software was required to set the value 10_2 into ASI_EIDR<13:12>. In SPARC64 VIIIfx, software no longer needs to set ASI_EIDR<13:12>, as the value of Module_ID_Value is fixed in hardware.

P.2.6 Error Detection Control (ASI_ERROR_CONTROL)

The ASI_ERROR_CONTROL register sets which errors are masked, as well as the behavior during error detection.

Register name	ASI_ERROR_CONTROL (ASI_ECR)
ASI	$4C_{16}$
VA	10_{16}
Error detection	None
Format	See TABLE P-7.
Initial value after reset	After a hard POR, ASI_ERROR_CONTROL.WEAK_ED is set to 1. All other fields are set to 0. For other rests, the values of UGE_HANDLER and WEAK_ED are copied to ASI_STCHG_ERROR_INFO and all fields are set to 0.

The ASI_ERROR_CONTROL register controls how errors are detected, how exceptions are signalled, and how multiple-ADE traps are processed. Registers fields are described below in TABLE P-7.

TABLE P-7 ASI_ERROR_CONTROL Bit Description

Bit	Field	Access	Description
9	RTE_UE	RW	Specifies whether certain restrainable errors (UE, unmarked UE) are signalled. Behavior is described in Appendix P.2.2.
8	RTE_DG	RW	Specifies whether certain restrainable errors (degrade error) are signalled. Behavior is described in Appendix P.2.2.
1	WEAK_ED	RW	Weak Error Detection. Controls whether detection of I_UGE and DAE is inhibited: When WEAK_ED = 0, error detection is not inhibited. When WEAK_ED = 1, error detection is inhibited if the CPU can continue processing. When an I_UGE or DAE is detected during instruction execution while WEAK_ED = 1, the value of the result (in register or memory) is indeterminate. If WEAK_ED = 1 but the CPU cannot ignored an I_UGE or DAE and continue processing, the error is signalled. WEAK_ED masks exception signalling for A_UGE and restrainable errors, as described in Appendix P.2.2. When a multiple-ADE trap occurs, WEAK_ED is set to 1 by hardware.
0	UGE_HANDLER	RW	When a UGE occurs, this bit is used by hardware to determine whether the OS is processing the UGE. 0: Hardware recognizes that the OS is not processing the UGE. 1: Hardware recognizes that the OS is processing the UGE. UGE_HANDLER masks exception signalling for A_UGE and restrainable errors, as described in Appendix P.2.2. The value of UGE_HANDLER is used to determine whether a multiple-ADE trap is caused when I_UGE, IAE, and DAE occur. When an ADE occurs, UGE_HANDLER = 1. A RETRY/DONE resets UGE_HANDLER to 0.
Other	Reserved	R	Always reads as 0.

P.3 Fatal Errors and `error_state` Transition Errors

P.3.1 ASI_STCHG_ERROR_INFO

The `ASI_STCHG_ERROR_INFO` register indicates information for detected `error_state` transition errors. This information is primarily intended for use by OBP (Open Boot PROM) software.

Compatibility Note – In SPARC64 VIIIfx, information on a fatal error is not displayed in `ASI_STCHG_ERROR_INFO`. That is, system software cannot know the details of a fatal error.

Register name	`ASI_STCHG_ERROR_INFO`
ASI	$4C_{16}$
VA	18_{16}
Error Detection	None
Format	See TABLE P-8
Initial value after reset	After a hard POR, all fields are set to 0. For other resets, values are unchanged.
Update policy	When an error is detected, the corresponding bit is set to 1. Writing 1 to bit 0 sets all bits in the register to 0.

TABLE P-8 describes the fields in the ASI_STCHG_ERROR_INFO register. Once a “sticky” bit is set to 1, that value is not modified by hardware.

TABLE P-8 ASI_STCHG_ERROR_INFO bit description *(1 of 2)*

Bit	Field	Access	Description
63:34	Reserved	R	Always 0.
33	ECR_WEAK_ED	R	ASI_ERROR_CONTROL.WEAK_ED is copied into this field on a POR or watchdog reset.
32	ECR_UGE_HANDLER	R	ASI_ERROR_CONTROL.UGE_HANDLER is copied into this field on a POR or watchdog reset.
31:24	Reserved	R	Always 0.
23	EE_MODULE	RW	Indicates a request to degrade the CPU module due to an error state transition error. Sticky.
22	EE_CORE	RW	Indicates a request to degrade the core due to an error state transition error. Sticky.
21	EE_THREAD	RW	Indicates a request to degrade the thread due to an error state transition error. Sticky. Hardware does not set this bit to 1.
20	UGE_MODULE	RW	Indicates a request to degrade the CPU module due to an urgent error. Sticky.
19	UGE_CORE	RW	Indicates a request to degrade the core due to an urgent error. Sticky.
18	UGE_THREAD	RW	Indicates a request to degrade the thread due to an urgent error. Sticky. Hardware does not set this bit to 1.
17	rawUE_MODULE	RW	Indicates that an unmarked UE was detected in L2$. Sticky.
16	rawUE_CORE	RW	Indicates that an unmarked UE was detected in L1$. Sticky.
15	EE_DCUCR_MCNTL_ECR	R	Indicates that an UE was detected in one of the following registers: (A) ASI_DCUCR (A) ASI_MCNTL (A) ASI_ECR
14	EE_OTHER	R	Set to 1 when an error occurs for a case not listed is this table. This bit is always 0 in SPARC64 VIIIfx.
13	EE_TRAP_ADR_UE	R	Indicates that the trap address could not be calculated because a UE occurred in the TBA, TT, or address calculation logic.
12	Reserved	R	Always 0.

TABLE P-8 ASI_STCHG_ERROR_INFO bit description *(2 of 2)*

Bit	Field	Access	Description
11	EE_WDT_IN_MAXTL	R	Indicates that a watchdog timeout occurred while TL = MAXTL.
10	EE_SECOND_WDT	R	Indicats that a second watchdog timeout was detected after an *async_data_error* exception occurred. (*async_data_error* was the first watchdog timeout.)
9	EE_SIR_IN_MAXTL	R	Indicates that an SIR occurred while TL = MAXTL.
8	EE_TRAP_IN_MAXTL	R	Indicates that a trap occurred while TL = MAXTL.
7:1	Reserved	R	Always 0.
0	clear_all	W	Writing 1 to this bit sets all fields in this register to 0.

P.3.2 Error_state Transition Error in Suspended Thread

SPARC64 VIIIfx enters the suspend state using a suspend instruction. Only POR, WDR, XDR, *interrupt_vector* and *interrupt_level_n* exceptions can return it back to the running state. If an error occurred in the resources related to those exceptions, the thread stays suspended forever. To prevent this situation, an urgent error regarding the following registers is reported as error_state transition error in suspended state.

- ASI_EIDR
- STICK, STICK_CMPR
- TICK, TICK_CMPR

In this case, ASI_STCHG_ERROR_INFO.UGE_CORE, along with corresponding bit of ASI_UGESR is set to 1.

P.4 Urgent Error

This section explains the details of urgent errors, such as status monitoring and completion methods for instructions that are forced to complete.

P.4.1 URGENT ERROR STATUS (ASI_UGESR)

Register name	ASI_URGENT_ERROR_STATUS
ASI	$4C_{16}$
VA	08_{16}
Error detection	None
Format	See TABLE P-9
Initial value after reset	After a hard POR, all fields are set to 0. For other resets, the values are unchanged.

The ASI_UGESR displays error information when an *async_data_error* (ADE) occurs, as well as error information for the second error when a multiple ADE occurs.

TABLE P-9 describes the fields of the UGESR. In the table, the prefixes for each field have the following meanings:

- IUG_ Instruction Urgent error
- IAG_ Autonomous Urgent error
- IAUG_ Both I_UGE and A_UGE

TABLE P-9 ASI_UGESR Bit Description *(1 of 2)*

Bit	Field	Access	Description
Setting a bit in ASI_UGESR<22:8> to 1 indicates that the corresponding error caused the single-ADE trap. Each bit in ASI_UGESR<22:16> indicates an error in an internal CPU register. The error detection conditions for these errors are defined in *"Internal Register Error Handling"* (page 286).			
22	IAUG_CRE	R	Uncorrectable error in any of the following registers: (IA) ASI_EIDR (IA) ASI_WATCHPOINT (when enabled) (I) ASI_INTR_R (A) ASI_INTR_DISPATCH_W (UE during write) (IA) STICK (IA) STICK_CMPR
21	IAUG_TSBCTXT	R	Uncorrectable error in any of the following registers: (IA) ASI_DMMU_TSB_BASE (IA) ASI_PRIMARY_CONTEXT (IA) ASI_SECONDARY_CONTEXT (IA) ASI_SHARED_CONTEXT (IA) ASI_IMMU_TSB_BASE
20	IUG_TSBP	R	Uncorrectable error in any of the following registers: (I) ASI_DMMU_TAG_TARGET (I) ASI_DMMU_TAG_ACCESS (I) ASI_IMMU_TAG_TARGET (I) ASI_IMMU_TAG_ACCESS
19	IUG_PSTATE	R	Uncorrectable error in any of the following registers: PSTATE, PC, NPC, CWP, CANSAVE, CANRESTORE, OTHERWIN, CLEANWIN, PIL, WSTATE
18	IUG_TSTATE	R	Uncorrectable error in any of the following registers: TSTATE, TPC, TNPC, TXAR
17	IUG_%F	R	Uncorrectable error in the floating-point registers (including the added registers), FPRS register, FSR, or GSR.
16	IUG_%R	R	Uncorrectable error in the general-purpose integer registers (including the added registers), Y register, CCR, or ASI registers.
14	IUG_WDT	R	First watchdog timeout. A single-ADE trap sets IUG_WDT = 1 and halts execution of the instruction pointed to by TPC; the result of the instruction result is indeterminate.
10	IUG_DTLB	R	When an uncorrectable error occurs in the DTLB during a load, store, or demap, this bit is set to1. Indicates the following: • On a DTLB read via ASI_DTLB_DATA_ACCESSand ASI_DTLB_TAG_ACCESS, an UE occurred in DTLB data or DTLB tag. • A write to the DTLB or a demap failed.TPC indicates either the instruction that caused the error or the following instruction.

TABLE P-9 ASI_UGESR Bit Description *(2 of 2)*

Bit	Field	Access	Description
9	IUG_ITLB	R	When an uncorrectable error occurs in the ITLB during a load, store, or demap, this bit is set to1. Indicates the following: • On a ITLB read viaASI_ITLB_DATA_ACCESS and ASI_ITLB_TAG_ACCESS, an UE occurred in ITLB data or ITLB tag. • A write to the ITLB or a demap failed.TPC indicates either the instruction that caused the error or the following instruction.
8	IUG_COREERR	R	Indicates an error occurred in the CPU core. When an error occurs in an execution resource or a resource that is not software-visible, this bit is set to 1. When an error occurs in a program-visible register and an instruction that reads the register is executed, the error bit corresponding to that register is always set; IUG_COREERR may or may not also be set.
5:4	INSTEND	R	Completion method for trapped instruction. When a watchdog timeout is not detected for a single-ADE trap, INSTEND indicates the completion method for instruction pointed to by TPC. 00_2: Precise 01_2: Retryable but not precise 10_2: *Reserved* 11_2: Not retryable See P.4.3 for details. When a watchdog timeout occurs, the completion method is undefined.
3	PRIV	R	Privileged mode. The value ofPSTATE.PRIV immediately before the single-ADE trap is copied. When this value is unknown because a UE occurred in the PSTATE register, ASI_UGESR.PRIV is set to 1.
2	MUGE_DAE	R	Indicates that a DAE caused multiple UGEs. For a single-ADE trap, MUGE_DAE is set to 0. For a multiple-ADE trap caused by a DAE, MUGE_DAE is set to 1. A multiple-ADE trap not caused by a DAE does not change MUGE_DAE.
1	MUGE_IAE	R	Indicates that a IAE caused multiple UGEs. For a single-ADE trap, MUGE_IAE is set to 0. For a multiple-ADE trap caused by an IAE, MUGE_IAE is set to 1. A multiple-ADE trap not caused by an IAE does not change MUGE_IAE.
0	MUGE_IUGE	R	Indicates that a I_UGE caused multiple UGEs. For a single-ADE trap, MUGE_IUGE is set to 0. For a multiple-ADE trap caused by an I_UGE, MUGE_IUGE is set to 1. A multiple-ADE trap not caused by an I_UGE does not change MUGE_IUGE.
Other	*Reserved*	R	Always 0.

P.4.2 Processing for async_data_error (ADE) Traps

Single-ADE traps and multiple-ADE traps are generated by the conditions defined in P.2.2. This section describes trap processing for these traps in more detail.

1. The following conditions cause ADE traps:
   - When ASI_ERROR_CONTROL.UGE_HANDLER = 0 and I_UGEs and/or A_UGEs are detected, a single-ADE trap is generated.
   - When ASI_ERROR_CONTROL.UGE_HANDLER = 1 and I_UGEs, IAE, and/or DAE are detected, a multiple-ADE trap is generated.
2. State transition, trap target address calculation, and TL processing are performed in the following order:
   a. Perform state transition

      When TL = MAXTL, the CPU enters error_state and abandons the ADE trap.

      When the CPU is in execute state with TL = MAXTL – 1, the CPU enters RED_state.

   b. Calculate trap target address

      When the CPU is in execute state, the address is calculated from TBA, TT, and TL.

      Otherwise, the CPU is in RED_state and the address is set to RSTVaddr + $A0_{16}$.

   c. TL is incremented by 1.
3. Update TSTATE, TPC, TNPC, and TXAR

   The values of PSTATE, PC, NPC, and XAR immediately before the ADE trap occurred are copied to TSTATE, TPC, TNPC, and TXAR respectively. If the original register contained an UE, the UE is also copied.
4. Update values of other registers

   The following 3 groups of registers are updated:

   a. Automatically verified registers

Hardware updates the following registers.

Register	Update Condition	Updated Value
PSTATE	Always	AG = 1, MG = 0, IG = 0, IE = 0, PRIV = 1, AM = 0, PEF = 1, RED = 0 (or 1 depending on the CPU status), MM = 00, TLE = 0, CLE = 0.
PC	Always	ADE trap address.
nPC	Always	ADE trap address + 4.
CCR	When the register contains an UE	0.
FSR, GSR	When the register contains an UE	A 0 is written to all registers that contain an UE. For a single-ADE trap, ASI_UGESR.IUG_%F is set to 1.
CWP, CANSAVE, CANRESTORE, OTHERWIN, CLEANWIN	When the register contains an UE	A 0 is written to all registers that contain an UE. For a single-ADE trap, ASI_UGESR.IUG_PSTATE is set to 1.
TICK	When the register contains an UE	NPT = 1, Counter = 0.
TICK_COMPARE	When the register contains an UE	INT_DIS = 1, TICK_CMPR = 0.
XAR	Always	0
XASR	When the register contains an UE	0

Updating these register removes any errors in these registers.

Errors in registers other than those listed above and errors in TLB entires are not removed.

b. ASI_UGESR

Bits	Field	Update on a Single-ADE Trap	Update on a Multiple-ADE Traps
63:6	Error Description	All bits in this field are updated. Displays all I_UGEs and A_UGEs detected.	Unchanged.
5:4	INSTEND	Indicates the completion method for the instruction pointed to be TPC.	Unchanged.
2	MUGE_DAE	Set to 0.	If a DAE caused the multiple-ADE trap, MUGE_DAE is set to 1. Otherwise, MUGE_DAE is unchanged.
1	MUGE_IAE	Set to 0.	If an IAE caused the multiple-ADE trap, MUGE_IAE is set to 1. Otherwise, MUGE_IAE is unchanged.
0	MUGE_IUGE	Set to 0.	If an I_UGE caused the multiple-ADE trap, MUGE_IUGE is set to 1. Otherwise, MUGE_IUGE is unchanged.

c. ASI_ERROR_CONTROL

On a single-ADE trap, ASI_ERROR_CONTROL.UGE_HANDLER is set to 1. UGE_HANDLER is set to 1 until a RETRY or DONE instruction is executed; this informs hardware that the error is being processed.

On a multiple-ADE trap, ASI_ERROR_CONTROL.WEAK_ED is set to 1, and the CPU runs in weak error detection mode.

5. Set ASI_ERROR_CONTROL.UGE_HANDLER to 0.

When a RETRY or DONE instruction is committed, UGE_HANDLER is set to 0.

P.4.3 Instruction Execution when an ADE Trap Occurs

In SPARC64 VIIIfx, an instruction forced to complete by an *async_data_error* exception completes in one of 3 ways. That is, the instruction pointed to by the TPC is one of 3 types:

- Precise
- Retryable but not precise (not defined in JPS1)
- Not retryable (not defined in JPS1)

For a single-ADE trap, the completion method for the instruction pointed to by the TPC is indicated in ASI_UGESR.INSTEND.

TABLE P-10 describes the difference between each completion method.

TABLE P-10 Instruction Execution when an *async_data_error* Trap Occurs

	Precise	Retryable But Not Precise	Not Retryable
Instructions executed after the last ADE, IAE, or DAE trap but before the instruction pointed to by TPC.	Committed. Instructions that do not cause an UGE complete as specified. The results of instructions that cause an UGE are undefined; that is, an undefined value is written to the destination register or memory.		
Instruction pointed to by TPC	Not executed.	The result of the instruction is incomplete. Only part of the result is written, and there are cases where the result is corrupted. Registers and memory not associated with the instruction are not affected. The following behavior does not occur: • A store to a cacheable address space (both memory and cache). • A store to a noncacheable address space. • An update of the result register when the register is also a source operand register.	The result of the instruction is incomplete. Only part of the result is written, and there are cases where the result is corrupted. Registers and memory not associated with the instruction are not affected. A store to an invalid address is not performed (a store to a valid address may be performed).
Instructions to be executed after the instruction pointed to by TPC	Not executed.	Not executed.	Not executed.
The possibility of resuming the program that signalled the exception when the error was reported by a single-ADE trap and did not cause any damage.	Possible.	Possible.	Impossible.

P.4.4 Expected Software Handling of ADE Traps

Expected software handling of an ADE trap is described by the pseudo C code below. The purpose of this code is to recover from the following errors:

- An error in the CPU internal RAM or registers
- An error in the accumulator
- An error in the CPU internal temporary registers or data bus

```
void
expected_software_handling_of_ADE_trap()
{
   /*
    * From here to Point#1, only %r0-%r7 are used because
    * register window control registers may be invalid.
    * In a single-ADE trap handler, it is recommendeded that
    * only %r0-%r7 be used, if possible.
                */

   ASI_SCRATCH_REGp ← %rX;      /* working register 1 */
   ASI_SCRATCH_REGq ← %rY;      /* working register 2 */
   %rX ← ASI_UGESR;

   if ((%rX && 0x07) ≠ 0) {
        /* multiple-ADE trap */
        invoke panic routine and generate largest possible
        system dump with ASI_ERROR_CONTROL.WEAK_ED == 1;
   }

   if (%rX.IUG_%R == 1) {
        %r1-%r63 ← %r0 (except for %rX and %rY);
        %y ← %r0;
        %tstate.pstate ← %r0;
        /* the asi field in %tstate.pstate may contain the
           error */
   }
   else {
        %rX, %rY, ASI_SCRATCH_REGp and ASI_SCRATCH_REGq are
        used to save needed registers. %r1-%r7 are saved to
        %rX, %rY, ASI_SCRATCH_REGp and ASI_SCRATCH_REGq;

        /*
         * When the processor recovers from an error that
         * occurred in a context with PSTATE.AG == 1,
         * all %r registers must be saved and restored to
         * their original values.
         */
   }

   if (ASI_UGESR.IUG_PSTATE == 1) {
        %tstate.pstate ← %r0;
        %tpc ← %r0;
        %pil ← %r0;
        %wstate ← %r0;
        all registers in the the register window ← %r0;
```

```
     set appropriate values for register window control
       registers (CWP, CANSAVE, CANRESTORE, OTHERWIN,
       CLEANWIN);
}

/*
 * Point#1
 * After this point, the program can use all windowed %r
 * registers except for %r0-%r7 because the register
 * window control registers were verified in the previous
 * step.
 */

if (ASI_UGESR.IAUG_CRE == 1
      || ASI_UGESR.IAUG_TSBCTXT == 1
      || ASI_UGESR.IUG_TSBP == 1
      || ASI_UGESR.IUG_TSTATE == 1
      || ASI_UGESR.IUG_%F==1) {

     verify all registers in which these errors may occur;
}

if (ASI_UGESR.IUG_DTLB == 1) {
     execute demap_all for DTLB;
     /*
      * A locked fDTLB entry is not removed by this
      * operation.
      */
}

if (ASI_UGESR.IUG_ITLB == 1) {
     execute demap_all for ITLB;
     /*
      * A locked fITLB entry is not removed by this
      * operation.
      */
}

if (ASI_UGESR.bits<22:14> == 0 &&
    ASI_UGESR.INSTEND == 0 || ASI_UGESR.INSTEND == 1) {
     ++ADE_trap_retry_per_unit_of_time;
     if (ADE_trap_retry_per_unit_of_time < threshold)
          use RETRY to return to the context prior to the
          trap;
     else
          halt OS because too many ADE trap retries;
} else if (ASI_UGESR.bits<22:18> == 0 &&
```

```
                ASI_UGESR.bits<15:14> == 0 &&
                ASI_UGESR.PRIV == 0) {
            ++ADE_trap_kill_user_per_unit_of_time;
            if (ADE_trap_kill_user_per_unit_of_time
                  < threshold) {
                kill one user process and continue OS processing;
            } else {
                halt OS because too many user processes killed
                by ADE traps;
            }
        } else {
            halt OS because of unrecoverable, urgent error.
        }
    }
```

P.5 Instruction Access Errors

See Appendix F.5, *“Faults and Traps”*, for details.

P.6 Data Access Errors

See Appendix F.5, *“Faults and Traps”*, for details.

P.7 Restrainable Errors

P.7.1 ASI_ASYNC_FAULT_STATUS (ASI_AFSR)

Register name	ASI_ASYNC_FAULT_STATUS (ASI_AFSR)
ASI	$4C_{16}$
VA	00_{16}
Error Detection	None
Format	See TABLE P-11
Initial value after reset	After a hard POR, all fields in ASI_AFSR are set to 0. For other resets, values are unchanged.

The ASI_ASYNC_FAULT_STATUS register indicates restrainable errors that have occurred. Once a bit is set to 1, that value is preserved until system software overwrites the bit. TABLE P-11 describes the fields of the AFSR. In the table, the prefixes for each field indicate the type of restrainable error:

- DG_ Degradation error
- UE_ Uncorrectable Error

TABLE P-11 ASI_ASYNC_FAULT_STATUS Bit Description

Bit	Field	Access	Description
12	Reserved		
11	DG_U2$	RW1C	When a way in the U2 cache of the CPU is removed, this bit is set to 1.
10	DG_D1$sTLB	RW1C	When a way in the I1/D1 cache or the sITLB/sDTLB is removed, this bit is set to 1.
9	Reserved	R	Always reads as 0; writes are ignored.
3	UE_DST_BETO	RW1C	When a write to memory returns a bus error, this bit is set to 1.
2	Reserved	R	Always reads as 0; writes are ignored.
1	UE_RAW_L2$INSD	RW1C	When an unmarked UE is detected in L2 cache data, this bit is set to 1.
0	UE_RAW_D1$INSD	RW1C	When an unmarked UE is detected in D1 cache data, this bit is set to 1.
Other	Reserved	R	Always reads as 0; writes are ignored.

Note – A disrupting bus error or timeout is reported by one of the following fields: AFSR.UE_DST_BETO, DSFSR.BERR, or DSFSR.RTO.

Note – When a write to an address space that sets AFSR.UE_DST_BETO is immediately followed by a read from the same address, the data is returned from the store buffer and a *data_access_error* may not occur. AFSR.UE_DST_BETO is set after the write is executed.

P.7.2 Expected Software Handling for Restrainable Errors

It is recommended that all restrainable errors be recorded. Expected software handling for each restrainable error is described below.

- DG_L1$, DG_U2$ — The following CPU states are reported:
  - Indicates that a way in the I1 cache, D1 cache, U2 cache, sITLB, or sDTLB has been removed; there is the possibility that this will cause a decrease in performance.
  - Indicates that there is the possibility of a decrease in CPU availability. When only one way can be used in the I1 cache, D1 cache, U2 cache, sITLB, or sDTLB and errors are detected in the remaining way, a error_state transition error occurs.

    If necessary, software can stop the use of the CPU that contains the errors.
- UE_DST_BETO — This error occurs in the following cases:
  - There is an incorrect entry in the DTLB.
  - An invalid address space is accessed using a physical address access ASI.

  In both cases, the error is caused by a bug in system software. Using the recorded error information, the system software should be corrected.
- UE_RAW_L2$INSD, and UE_RAW_D1$INSD — These errors handled as follows:
  - If possible, the error in the cache line containing the UE is removed. Note that this causes the data in the cache line to be lost.
- When *ECC_error* exception is generated but the error is not indicated in ASI_AFSR — the *ECC_error* exception is ignored.

  See *"Summary of Behavior During Error Detection"* (page 262) for details.

P.8 Internal Register Error Handling

This section describes error handling for errors that occur in the following registers:

- Nonprivileged and Privileged registers

- ASR registers
- ASI registers

P.8.1 Nonprivileged and Privileged Register Error Handling

The terms used in TABLE P-12 are defined as follows:

Column	Term	Meaning
Condition for Error Detection	InstrAccess	The error is detected when the register is accessed during instruction execution.
Error Correction	W	The error is corrected when a write to the entire register is performed.
	ADE trap	Hardware removes the error by performing a write to the entire register during trap processing of the *async_data_error* exception.

TABLE P-12 describes error handling for errors that occur in nonprivileged and privileged registers. When an urgent error occurs in the PC, nPC, PSTATE, CWP, ASI, or an XAR register, the *async_data_error* trap handler is entered. When registers are copied to the TPC, TNPC, TSTATE, and TXAR, any errors in these registers are also copied.

TABLE P-12 Nonprivileged and Privileged Register Error Handling *(1 of 2)*

Register Name	RW	Error Protection	Condition for Error Detection	Error Type	Error Correction
%rn[1]	RW	Parity	InstrAccess	IUG_%R	W
%fn[1]	RW	Parity	InstrAccess	IUG_%F	W
PC	R	Parity	Always	IUG_PSTATE	ADE trap
nPC	R	Parity	Always	IUG_PSTATE	ADE trap
PSTATE	RW	Parity	Always	IUG_PSTATE	ADE trap, W
TBA	RW	Parity	PSTATE.RED = 0	error_state	W (by OBP)
PIL	RW	Parity	PSTATE.IE = 1 InstrAccess	IUG_PSTATE	W
CWP, CANSAVE, CANRESTORE, OTHERWIN, CLEANWIN	RW	Parity	Always	IUG_PSTATE	ADE trap, W
TT	RW	None	—	—	—
TL	RW	Parity	PSTATE.RED = 0	error_state	W (by OBP)
TPC	RW	Parity	InstrAccess	IUG_TSTATE	W
TNPC	RW	Parity	InstrAccess	IUG_TSTATE	W
TSTATE	RW	Parity	InstrAccess	IUG_TSTATE	W

TABLE P-12 Nonprivileged and Privileged Register Error Handling *(2 of 2)*

Register Name	RW	Error Protection	Condition for Error Detection	Error Type	Error Correction
WSTATE	RW	Parity	Always	IUG_PSTATE	ADE trap, W
VER	R	None	—	—	—
FSR	RW	Parity	Always	IUG_%F	ADE trap, W
Y	RW	Parity	InstrAccess	IUG_%R	W
CCR	RW	Parity	Always	IUG_%R	ADE trap, W
ASI	RW	Parity	Always	IUG_%R	ADE trap, W
TICK	RW	Parity	AUG Always[2]	IUG_COREERR	ADE trap[3] , W
FPRS	RW	Parity	Always	IUG_%F	ADE trap, W

1.Includes the registers added by HPC-ACE.

2.A suspended thread signals an error_state transition error.

3.Set to 0x8000_0000_0000_0000 for correction.

P.8.2 ASR Error Handling

The terms used in TABLE P-13 are defined as follows:

Column	Term	Meaning
Condition for Error Detection	AUG always	The error is detected when ASI_ERROR_CONTROL.UGE_HANDLER = 0 and ASI_ERROR_CONTROL.WEAK_ED = 0.
	InstrAccess	The error is detected when the register is accessed during instruction execution.
Error Type	(I)AUG_*xxx*	Autonomous urgent error. ASI_UGESR.IAUG_*xxx* = 1.
	I(A)UG_*xxx*	Instruction urgent error. ASI_UGESR.IAUG_*xxx* = 1.
Error Correction	W	The error is corrected when a write to the entire register is performed.
	ADE trap	Hardware removes the error by performing a write to the entire register during trap processing of the *async_data_error* exception.

TABLE P-13 describes error handling for ASR errors.

TABLE P-13 ASR Error Handling

ASR Number	Register Name	RW	Error Protection	Condition for Error Detection	Error Type	Error Correction
16	PCR	RW	None	—	—	—
17	PIC	RW	None	—	—	—
18	DCR	R	None	—	—	—
19	GSR	RW	Parity	Always	IUG_%F	ADE trap, W
20	SET_SOFTINT	W	None	—	—	—
21	CLEAR_SOFTINT	W	None	—	—	—
22	SOFTINT	RW	None	—	—	—
23	TICK_COMPARE	RW	Parity	AUG always[1]	IUG_COREERR	ADE trap, W
24	STICK	RW	Parity	AUG always[1]	(I)AUG_CRE	W
				InstrAccess	I(A)UG_CRE	W
25	STICK_COMPARE	RW	Parity	AUG always[1]	(I)AUG_CRE	W
				InstrAccess	I(A)UG_CRE	W
29	XAR	RW	Parity	Always	IUG_COREERR	ADE trap, W
30	XASR	RW	Parity	Always	IUG_COREERR	ADE trap, W
29	TXAR	RW	Parity	InstrAccess	IUG_TSTATE	W

1.A suspended thread signals an `error_state` transition error.

STICK Behavior on Error

When an error occurs in the `STICK` register, countup is stopped regardless of the condition for error detection described in TABLE P-13.

P.8.3 ASI Register Error Handling

The terms used in TABLE P-14 are defined as follows:

Column	Term	Meaning
Error Protection	Parity	Parity protected.
	Triple	Register is triplicated.
	ECC	ECC protected (double-bit error detection, single-bit error correction).
	Gecc	Generated ECC.
	None	Not protected.

Column	Term	Meaning
Condition for Error Detection	Always	Error is always detected.
	AUG always	Error is detected when ASI_ERROR_CONTROL.UGE_HANDLER = 0 and ASI_ERROR_CONTROL.WEAK_ED = 0.
	LDXA	Error is detected when the register is read by an instruction.
	ITLB write	Error is detected on a write to the ITLB or when a demap operation updates the ITLB.
	DTLB write	Error is detected on a write to the DTLB or when a demap operation updates the DTLB.
	Used by TLB	Error is detected when the register is referenced during a search of the TLB.
	Enabled	Error is detected when the function is enabled.
	intr_receive	Error is detected when an interrupt packet is received. When there is an UE in the interrupt packet, a vector_interrupt exception is generated and ASI_INTR_RECEIVE.BUSY is set to 0. Setting ASI_INTR_RECEIVE.BUSY allows a new interrupt packet to be received.
Error Type	error_state	error_state transition error.
	(I)AUG_*xxxx*	Autonomous urgent error. ASI_UGESR.IAUG_*xxxx* = 1.
	I(A)UG_*xxxx*	Instruction urgent error. ASI_UGESR.IAUG_*xxxx* = 1.
	Other	Bit in ASI_UGESR that corresponds to the error is set to 1.
Error Correction	RED trap	When a RED_state trap occurs, the value of the register is updated and the error is corrected.
	W	A write to the ASI register corrects the error.
	W_other_I	Error is corrected by updating all of the following registers: • ASI_IMMU_TAG_ACCESS • When ASI_UGESR.IAUG_TSBCTXT = 1 for a single-ADE trap, ASI_IMMU_TSB_BASE, ASI_PRIMARY_CONTEXT, ASI_SECONDARY_CONTEXT, ASI_SHARED_CONTEXT
	W_other_D	Error is corrected by updating all of the following registers: • ASI_DMMU_TAG_ACCESS • When ASI_UGESR.IAUG_TSBCTXT = 1 for a single-ADE trap, ASI_DMMU_TSB_BASE, ASI_PRIMARY_CONTEXT, ASI_SECONDARY_CONTEXT, ASI_SHARED_CONTEXT
	Interrupt receive	Error is corrected when the interrupt packet is received.

TABLE P-14 describes error handling for ASI register errors.

TABLE P-14 Handling of ASI Register Errors *(1 of 2)*

ASI	VA	Register Name	RW	Error Protect	Error Detect Condition	Error Type	Correction
45_{16}	00_{16}	DCU_CONTROL	RW	Parity	Always	error_state	RED trap
	08_{16}	MEMORY_CONTROL	RW	Parity	Always	error_state	RED trap
48_{16}	00_{16}	INTR_DISPATCH_STATUS	R	Parity	LDXA or register update	I(A)UG_CRE (UE)	None
49_{16}	00_{16}	INTR_RECEIVE	RW	Parity	LDXA	I(A)UG_CRE (UE)	None
$4A_{16}$	—	SYS_CONFIG	R	None	—	—	—
$4B_{16}$	00_{16}	STICK_CNTL	RW	Triple	Always	—	Always
$4C_{16}$	00_{16}	ASYNC_FAULT_STATUS	RW1C	None	—	—	—
$4C_{16}$	08_{16}	URGENT_ERROR_STATUS	R	None	—	—	—
$4C_{16}$	10_{16}	ERROR_CONTROL	RW	Parity	Always	error_state	RED trap
$4C_{16}$	18_{16}	STCHG_ERROR_INFO	R, W1AC	None	—	—	—
$4F_{16}$	00_{16}–38_{16}	SCRATCH_REGs	RW	Parity	LDXA	IUG_COREERR	W
50_{16}	00_{16}	IMMU_TAG_TARGET	R	Parity	LDXA	IUG_TSBP	W_other_I
50_{16}	18_{16}	IMMU_SFSR	RW	None	—	—	—
50_{16}	28_{16}	IMMU_TSB_BASE	RW	Parity	LDXA	I(A)UG_TSBCTXT	W
50_{16}	30_{16}	IMMU_TAG_ACCESS	RW	Parity	LDXA	IUG_TSBP	W (W_other_I)
50_{16}	60_{16}	IMMU_TAG_ACCESS_EXT	RW	Parity	LDXA	IUG_TSBP	W
50_{16}	78_{16}	IMMU_SFPAR	RW	Parity	LDXA	I(A)UG_CRE	W
53_{16}	—	SERIAL_ID	R	None	—	—	—
54_{16}	—	ITLB_DATA_IN	W	Parity	ITLB write	IUG_ITLB	DemapAll
55_{16}	—	ITLB_DATA_ACCESS	RW	Parity	LDXA	IUG_ITLB	DemapAll
					ITLB write	IUG_ITLB	DemapAll
56_{16}	—	ITLB_TAG_READ	R	Parity	LDXA	IUG_ITLB	DemapAll
57_{16}	—	IMMU_DEMAP	W	Parity	ITLB write	IUG_ITLB	DemapAll
58_{16}	00_{16}	DMMU_TAG_TARGET	R	Parity	LDXA	IUG_TSBP	W_other_D
58_{16}	08_{16}	PRIMARY_CONTEXT	RW	Parity	LDXA	I(A)UG_TSBCTXT	W
					Used by TLB AUG always	I(A)UG_TSBCTXT	W
						(I)AUG_TSBCTXT	W
58_{16}	10_{16}	SECONDARY_CONTEXT	RW	Parity	= P_CONTEXT	IAUG_TSBCTXT	W
58_{16}	18_{16}	DMMU_SFSR	RW	None	—	—	—
58_{16}	20_{16}	DMMU_SFAR	RW	Parity	LDXA	IAUG_CRE	W
58_{16}	28_{16}	DMMU_TSB_BASE	RW	Parity	LDXA	I(A)UG_TSBCTXT	W
58_{16}	30_{16}	DMMU_TAG_ACCESS	RW	Parity	LDXA	IUG_TSBP	W (W_other_D)

TABLE P-14 Handling of ASI Register Errors *(2 of 2)*

ASI	VA	Register Name	RW	Error Protect	Error Detect Condition	Error Type	Correction
58_{16}	38_{16}	DMMU_WATCHPOINT	RW	Parity	Enabled	(I)AUG_CRE	W
					LDXA	I(A)UG_CRE	W
58_{16}	60_{16}	DMMU_TAG_ACCCESS_EXT	RW	Parity	LDXA	IUG_TSBP	W
58_{16}	68_{16}	SHARED_CONTEXT	RW	Parity	= P_CONTEXT	(I)AUG_TSBCTXT	W
58_{16}	78_{16}	DMMU_SFPAR	RW	Parity	LDXA	I(A)UG_CRE	W
$5C_{16}$	—	DTLB_DATA_IN	W	Parity	DTLB write	IUG_DTLB	DemapAll
$5D_{16}$	—	DTLB_DATA_ACCESS	RW	Parity	LDXA	IUG_DTLB	DemapAll
					DTLB write	IUG_DTLB	DemapAll
$5E_{16}$	—	DTLB_TAG_READ	R	Parity	LDXA	IUG_DTLB	DemapAll
$5F_{16}$	—	DMMU_DEMAP	W	Parity	DTLB write	IUG_DTLB	DemapAll
60_{16}	—	IIU_INST_TRAP	RW	Parity	LDXA	No match at error	W
67_{16}	—	FLUSH_L1I	W	None	—	—	—
$6D_{16}$	00_{16}- 58_{16}	BARRIER_INIT	RW	Parity	Always if assigned or LDXA	Fatal Error	—
$6E_{16}$	00_{16}	EIDR	RW	Parity	Always[1]	IAUG_CRE	W
$6F_{16}$	00_{16}- 58_{16}	BARRIER_ASSIGN	RW	Parity	Always if assigned	Fatal Error	—
74_{16}	addr	CACHE_INV	W	None	—	—	—
77_{16}	40_{16}–50_{16}	INTR_DATA0:2_W	W	Gecc	None	—	W
77_{16}	70_{16}	INTR_DISPATCH_W	W	Gecc	store	(I)AUG_CRE	W
$7F_{16}$	40_{16}–50_{16}	INTR_DATA0:2_R	R	ECC	LDXA	IAUG_CRE	Interrupt Receive
					intr_receive	BUSY = 0	
$E7_{16}$	00_{16}	SCCR	RW	Parity	Always	IUG_COREERR	W
FE_{16}	00_{16}- 58_{16}	LBSY, BST	RW	Parity	Always if assigned	Fatal Error	—

1.Notified as error_state transition error in suspended state.

P.9 Cache Error Handling

This section describes error handling for cache tag errors and cache data errors.

P.9.1 Error Handling for Cache Tag Errors

D1 Cache Tag Errors and I1 Cache Tag Errors

The D1 (Data level-1) and the I1 (Instruction level-1) cache tags are duplicated in the U2 (Unified level-2) cache. The D1 cache tags, the I1 cache tags, and the duplicated cache tags in the U2 cache are all parity protected.

When a parity error is detected in a D1 cache tag or a duplicate D1 cache tag, hardware copies the other cache tag to the tag containing the error. If this action corrects the error, program execution is not affected.

Similarly, when a parity error is detected in an I1 cache tag or a duplicate I1 cache tag, hardware copies the other cache tag to the tag containing the error. If this action corrects the error, program execution is not affected.

If copying the cache tag does not correct the error, the action is repeated. When the error is permanent, a watchdog timeout or a FATAL error is eventually detected.

U2 Cache Tag Errors

The U2 cache tags are ECC protected. Single-bit errors are corrected, and double-bit errors are detected.

When a correctable error is detected in a U2 cache tag, hardware corrects the error by writing the corrected data to the U2 cache tag. The error is not reported to system software.

When an uncorrectable error is detected in a U2 cache tag, a fatal error is signalled and the CPU enters CPU Fatal Error state.

P.9.2 Error Handling for I1 Cache Data Errors

Each doubleword in I1 cache data is parity protected.

When a parity error is detected in I1 cache data during instruction fetch, hardware performs the following sequence of actions:

1. Reread the I1 cache line containing the parity error from the U2 cache.

   Any UE in the data read from the U2 cache is marked, since error marking is performed for all outgoing data, that is, data leaving the U2 cache.

2. For each doubleword read from the U2 cache,

   a. When the doubleword does not contain an UE, the data is saved to the I1 cache. This data is supplied to the instruction fetch unit when needed.

An I1 cache error that is corrected by refilling the I1 cache is not reported to system software.

b. When the doubleword contains a marked UE, the parity bit for the corresponding doubleword in I1 cache data is set. This data is supplied to the instruction fetch unit when needed.

3. The instruction fetch unit handles an instruction containing an error in the following way.

   The instruction is discarded when the instruction containing the parity error is fetched but is not executed and does not update the software-visible state.

   When the fetched instruction executes and commits, an *instruction_access_error* exception is generated. ASI_ISFSR indicates that a marked UE was detected and displays the corresponding ERROR_MARK_ID.

P.9.3 Error Handling for D1 Cache Data Errors

Each doubleword in D1 cache data is ECC protected. Single-bit errors are corrected, and double-bit errors are detected.

Correctable Errors in D1 Cache Data

When a correctable error is detected in D1 cache data, the data is corrected automatically by hardware. A correctable error is not reported to system software.

Marked Uncorrectable Errors in D1 Cache Data

When a marked uncorrectable error (UE) is detected in D1 cache data during a cache line writeback to the U2 cache, the D1 cache data and ECC are written to the U2 cache without any changes. That is, a marked UE in D1 cache data is written back to the U2 cache; this is not reported to system software.

When a marked UE is detected in D1 cache data during an access by a load/store instruction (except for doubleword stores), a *data_access_error* exception is generated. This exception is precise, and ASI_DSFSR displays the ERROR_MARK_ID of the marked UE.

Unmarked UE in D1 Cache Data During Cache Line Writeback

When an unmarked UE is detected in D1 cache data during a cache line writeback to the U2 cache, error marking of the doubleword containing the error is performed. The value in ASI_EIDR is used for the ERROR_MARK_ID. Only corrected data or data containing marked a UE is written back to the U2 cache.

Marking the UE sets ASI_AFSR.UE_RAW_D1$INSD to 1.

Unmarked UE in D1 Cache Data on a Read by a Memory Access Instruction

When an unmarked UE is detected in D1 cache data during a read by a memory access instruction, hardware performs the following sequence of actions:

1. Hardware writes back the D1 cache line and refills the data from the U2 cache.

   The D1 cache line is written back to the U2 cache, regardless of whether the U2 data is the same or has been updated. Error marking is performed during writeback. The value in `ASI_EIDR` is used for the `ERROR_MARK_ID`. The D1 cache line is refilled from the U2 cache, and `ASI_AFSR.UE_RAW_D1$INSD` is set to 1.

2. Normally, step 1 performs error marking for unmarked errors; during this processing, however, a new UE may be introduced in the same doubleword. In this case, step 1 is repeated until the doubleword contains no unmarked errors, or until D1 cache way reduction occurs.

3. At this point, all unmarked UEs in D1 cache data have been marked. The load or store instruction accesses the doubleword with the marked UE. The memory access instruction then accesses the data containing the marked UE. Subsequent behavior is described in the subsection *"Marked Uncorrectable Errors in D1 Cache Data"* (page 294).

P.9.4 Error Handling for U2 Cache Data Errors

Each doubleword in U2 cache data is ECC protected. Single-bit errors are corrected, and double-bit errors are detected.

Correctable Errors in U2 Cache Data

When a correctable error is detected in incoming U2 cache fill data from memory, the error is automatically corrected by hardware. No exception is signalled.

When a correctable error is detected in U2 cache data requested by the I1/D1 cache or that is being written to memory or another cache, the error is automatically corrected by hardware. The error is not reported to system software.

Marked Uncorrectable Errors in U2 Cache Data

For U2 cache data, a doubleword containing a marked UE is handled in the same manner as a corrected doubleword. No error is reported when a marked UE is detected in U2 cache data.

When a marked UE is detected in U2 cache fill data from memory, the doubleword containing the marked UE is stored without any changes in the U2 cache.

When a marked UE is detected in D1 cache data being written back to the U2 cache, the doubleword containing the marked UE is stored without any changes in the U2 cache. Data containing an unmarked UE is not written back. See Appendix P.9.3, *"Error Handling for D1 Cache Data Errors"* (page 294).

When a marked UE is detected in U2 cache data requested by the I1/D1 cache or that is being written to memory or another cache, the doubleword containing the marked UE is sent without any changes.

Unmarked UE in U2 Cache Data

When an unmarked UE is detected in U2 cache fill data from memory, error marking is performed for the doubleword containing the unmarked UE. The value used for ERROR_MARK_ID is 0. The doubleword and associated ECC are replaced with the marked data, and the updated data is stored in the U2 cache. No exception is signalled.

When an unmarked UE is detected in data read from the U2 cache (I1 cache fill, D1 cache fill, write to memory or another cache), error marking is performed for the doubleword containing the unmarked UE. The value in ASI_EIDR is used for ERROR_MARK_ID, and ASI_AFSR.UE_RAW_L2$INSD is set to 1.

P.9.5 Automatic I1, D1, and U2 Cache Way Reduction

When errors occur frequently in the I1, D1, or U2 cache, hardware degrades the appropriate cache way, while maintaining cache coherency. This is called way reduction.

Conditions for Cache Way Reduction

Hardware counts the number of errors that occur in each cache way for each cache. The following errors are counted:

- For each I1 cache way,
  - Parity errors in I1 cache tags and duplicate I1 cache tags
  - Parity errors in I1 cache data
- For each D1 cache way,
  - Parity errors in D1 cache tags and duplicate D1 cache tags
  - Correctable errors in D1 cache data
  - Unmarked UEs in D1 cache data
- For each U2 cache way,
  - Correctable errors and UEs in U2 cache tags
  - Correctable errors in U2 cache data
  - Unmarked UEs in U2 cache data

If the counter for a cache way exceeds the specified threshold value within a set amount of time, that cache way is degraded. The procedure for way reduction is described below.

I1 Cache Way Reduction

Procedure for degrading way *w* of the I1 cache:

1. When one cache way has already been degraded, the entry containing the error is invalidated.
2. Otherwise,
   - All entries in way *w* are invalidated, and way *w* is never refilled.
   - `ASI_AFSR.DG_L1STLB` is set to 1, and a restrainable error is signalled.

D1 Cache Way Reduction

Procedure for degrading way *w* of the I1 cache:

1. When one cache way has already been degraded, the entry containing the error is written back to the U2 cache and invalidated.
2. Otherwise,
   - All entries in way *w* are invalidated, and way *w* is never refilled. Data that has been updated in the D1 cache but not the U2 cache is written back to the U2 cache before the entry is invalidated.
   - `ASI_AFSR.DG_L1$STLB` is set to 1, and a restrainable error is signalled.

U2 Cache Way Reduction

U2 cache way reduction is performed when `DCUCR.WEAK_SPCA` = 0. When `DCUCR.WEAK_SPCA` = 1, way reduction is pending; U2 cache way reduction is started once `DCUCR.WEAK_SPCA` = 0.

Procedure for removing way *w* of the U2 cache:

1. When all cache ways have already been degraded, and only one cache way remains,
   - All entries in way *w* are invalidated (that is, all active entries are invalidated), but cache way *w* can still be used. U2 cache data is invalidated to preserve data coherency for the entire system.
   - `ASI_AFSR.DG_U2` is set to 1, and a restrainable error is signalled even though the U2 cache configuration has not been changed.
2. Otherwise,
   - All entries in all cache ways, including way *w*, are invalidated to preserve data coherency for the entire system.
   - Way *w* can no longer be used.
   - `ASI_AFSR.DG_U2` is set to 1, and a restrainable error is signalled.

P.10 TLB Error Handling

This section describes error processing for TLB entries, as well as sTLB way reduction.

P.10.1 Error Processing for TLB Entries

TABLE P-15 describes the error protection implemented for each SPARC64 VIIIfx TLB.

TABLE P-15 Error Protection and Error Detection for TLB Entries

TLB type	Field	Error protection	Errors that can be detected
sITLB, sDTLB	tag	Parity	Parity error (Uncorrectable)
	data	Parity	Parity error (Uncorrectable)
fITLB, fDTLB	lock bit	Triplication	None; the value is determined by majority
	tag, except lock bit	Parity	Parity error (Uncorrectable)
	data	Parity	Parity error (Uncorrectable)

TLB errors are detected during address translation for memory accesses and when TLB entries are accessed directly via the ASI registers.

TLB Error Detected on Access Via ASI Register

When an error is detected in a DTLB entry on an access via the ASI_DTLB_DATA_ACCESS or ASI_DTLB_TAG_ACCESS register, ASI_UGESR.IUG_DTLB is set to 1 and an instruction urgent error is signalled.

When an error is detected in a ITLB entry on an access via the ASI_ITLB_DATA_ACCESS or ASI_ITLB_TAG_ACCESS register, ASI_UGESR.IUG_ITLB is set to 1 and an instruction urgent error is signalled.

sTLB Error Detected During Address Translation

When an error is discovered in a sTLB entry during address translation, that entry is invalidated. The error is not reported to system software.

fTLB Error Detected During Address Translation

Both fTLB tags and data are duplicated. When an fTLB parity error is detected during address translation, the error can be corrected automatically by replacing the copy containing the parity error with the duplicated tag or data. The error is not reported to system software. If parity errors are detected in both copies, a fatal error is signalled.

F.APPENDIX **Q**

Performance Instrumentation

This appendix describes the SPARC64 VIIIfx performance counters (PA). Please see the following sections:

Q.1 PA Overview

For information on the performance counter registers, please refer to *"Performance Control Register (PCR) (ASR 16)"* (page 27) and *"Performance Instrumentation Counter (PIC) Register (ASR 17)"* (page 28).

Q.1.1 Sample Pseudo-codes

Counter Clear/Set

The PICs are read/write registers. Writing zero will clear the counter; writing any other value will set the counter. The following pseudocode procedure clears all PICs (assuming privileged access):

```
/* Clear PICs without updating SL/SU values */
pic_init = 0x0;
pcr = rd_pcr();
pcr.ulro = 0x1;       /* don't update SU/SL on write */
pcr.ovf = 0x0;        /* clear overflow bits */
```

```
pcr.ut = 0x0;
pcr.st = 0x0;          /* disable counts   */
for (i=0; i<=pcr.nc; i++) {
   /* select the PIC to be written */
   pcr.sc = i;
   wr_pcr(pcr);
   wr_pic(pic_init);/* clear PIC[i] */
}
```

Counter Event Selection and Start

Counter events are selected through the `PCR.SC` and `PCR.SU`/`PCR.SL` fields. The following pseudocode selects events and enables counters (assuming privileged access):

```
pcr.ut = 0x0;          /* Disable user counts */
pcr.st = 0x0;          /* Disable system counts also */
pcr.ulro = 0x0;        /* Make SU/SL writeable */
pcr.ovro = 0x1;        /* Overflow is read-only */
/* Select events without enabling counters */
for(i=0; i<=pcr.nc; i++) {
   pcr.sc = i;
   pcr.sl = select an event;
   pcr.su = select an event;
   wr_pcr(pcr);
}
/* Start counting */
pcr.ut = 0x1;
pcr.st = 0x1;
pcr.ulro = 0x1;        /* SU/SL is read-only */
/* Clear overflow bits here if needed */
wr_pcr(pcr);
```

Counter Stop and Read

The following pseudocode disables and reads counters (assuming privileged access):

```
pcr.ut = 0x0;          /* Disable user counts */
pcr.st = 0x0;          /* Disable system counts, too */
pcr.ulro = 0x1;        /* Make SU/SL read-only */
pcr.ovro = 0x1;        /* Overflow is read-only */
for(i=0; i<=pcr.nc; i++) {
   pcr.sc = i;
   wr_pcr(pcr);
   pic = rd_pic();
   picl[i] = pic.picl;
   picu[i] = pic.picu;
}
```

Q.2 Description of PA Events

The performance counter (PA) events can be classified into the following groups:

1. Instruction and trap statistics
2. MMU and L1 cache events
3. L2 cache events
4. Bus transaction events

There are 2 types of PA events that can be measured in SPARC64 VIIIfx, standard and supplemental events.

Standard events in SPARC64 VIIIfx have been verified for correct behavior; they are guaranteed to be compatibile[1] with future processors.

Supplemental events are primarily intended to be used for debugging the hardware.

a. The behavior of supplemental events may not be fully verified. There is a possibility that some of these events may not behave as specified in this document.

b. The definition of these events may be changed without notice. Compatibility with future processors is not guaranteed.

All PA events defined in SPARC64 VIIIfx are shown in TABLE Q-1. Shaded events are supplemental events. For details on each event, refer to the descriptions in the following sections. Unless otherwise indicated, speculative instructions are also counted by the PA events.

1. Provided that a feature is not removed due to design changes.

TABLE Q-1 PA Events and Encodings

Encoding	Counter							
	picu0	**picl0**	**picu1**	**picl1**	**picu2**	**picl2**	**picu3**	**picl3**
0000000	cycle_counts							
0000001	instruction_counts							
0000010	instruction_flow_counts	*Reserved*			instruction_flow_counts	*Reserved*		xma_inst
0000011	iwr_empty	*Reserved*			iwr_empty	*Reserved*		
0000100	*Reserved*							
0000101	op_stv_wait							
0000110	effective_instruction_counts							
0000111	SIMD_load_store_instructions	SIMD_floating_instructions	SIMD_fma_instructions	sxar1_instructions	sxar2_instructions	unpack_sxar1	unpack_sxar2	*Reserved*
0001000	load_store_instructions							
0001001	branch_instructions							
0001010	floating_instructions							
0001011	fma_instructions							
0001100	prefetch_instructions							
0001101	*Reserved*	ex_load_instructions	ex_store_instructions	fl_load_instructions	fl_store_instructions	SIMD_fl_load_instructions	SIMD_fl_store_instructions	*Reserved*
0001110	*Reserved*							
0001111	*Reserved*							
0010000	*Reserved*							
0010001	*Reserved*							
0010010	rs1	flush_rs	*Reserved*					
0010011	1iid_use	2iid_use	3iid_use	4iid_use	*Reserved*	*sync_intlk*	*regwin_intlk*	*Reserved*
0010100	*Reserved*							
0010101	*Reserved*	toq_rsbr_phantom	*Reserved*	flush_rs	*Reserved*		rs1	*Reserved*
0010110	trap_all	trap_int_vector	trap_int_level	trap_spill	trap_fill	trap_trap_inst	trap_IMMU_miss	trap_DMMU_miss
0010111	*Reserved*	trap_SIMD_load_across_pages	*Reserved*					
0011000	*Reserved*							
0011001	*Reserved*							

TABLE Q-1 PA Events and Encodings *(Continued)*

Encoding	Counter							
	picu0	**picl0**	**picu1**	**picl1**	**picu2**	**picl2**	**picu3**	**picl3**
0011010	*Reserved*		single_sxar_commit	*Reserved*				suspend_cycle
0011011	rsf_pmmi	*Reserved*		0iid_use	flush_rs	*Reserved*		decall_intlk
0011100	*Reserved*							
0011101	op_stv_wait_pfp_busy_ex	op_stv_wait_sxmiss	op_stv_wait_sxmiss_ex	op_stv_wait_nc_pend	cse_window_empty_sp_full	op_stv_wait_pfp_busy	*Reserved*	
0011110	cse_window_empty	eu_comp_wait	branch_comp_wait	0endop	op_stv_wait_ex	fl_comp_wait	1endop	2endop
0011111	inh_cmit_gpr_2write	*Reserved*			3endop	*Reserved*	sleep_cycle	op_stv_wait_swpf
0100000	uITLB_miss2	uDTLB_miss2	uITLB_miss	uDTLB_miss	L1I_miss	L1D_miss	L1I_wait_all	L1D_wait_all
0100001	*Reserved*							
0100010	*Reserved*							
0100011	L1I_thrashing	L1D_thrashing	*Reserved*					
0100100	swpf_success_all	swpf_fail_all	*Reserved*		swpf_lbs_hit	*Reserved*		
0100101	*Reserved*							
0100110	*Reserved*							
0100111	*Reserved*							
0110000	*Reserved*		L2_miss_dm	L2_miss_pf	L2_read_dm	L2_read_pf	L2_wb_dm	L2_wb_pf
0110001	bi_count	*Reserved*		cpd_count	cpu_mem_read_count	cpu_mem_write_count	IO_mem_read_count	IO_mem_write_count
0110010	L2_miss_wait_dm_bank0	L2_miss_wait_pf_bank0	L2_miss_count_dm_bank0	L2_miss_count_pf_bank0	L2_miss_wait_dm_bank1	L2_miss_wait_pf_bank1	L2_miss_count_dm_bank1	L2_miss_count_pf_bank1
0110011	L2_miss_count_dm_bank2	L2_miss_count_pf_bank2	L2_miss_wait_dm_bank2	L2_miss_wait_pf_bank2	L2_miss_count_dm_bank3	L2_miss_count_pf_bank3	L2_miss_wait_dm_bank3	L2_miss_wait_pf_bank3
0110100	lost_pf_pfp_full	lost_pf_by_abort	IO_pst_count	*Reserved*				
0110101	*Reserved*							
0110110	*Reserved*							
0111111	*Disabled (No PIC is counted up)*							
1111111	*Disabled (No PIC is counted up)*							

Q.2.1 Instruction and Trap Statistics

Standard PA Events

1 *cycle_counts*

Counts the number of cycles when the performance counter is enabled. This counter is similar to the `TICK` register but can count user cycles and system cycles separately, based on the settings of `PCR.UT` and `PCR.ST`.

2 *instruction_counts* (Non-Speculative)

Counts the number of committed instructions, including `SXAR1` and `SXAR2`.

SPARC64 VIIIfx commits up to 4 instructions per cycle; however, this number normally does not include `SXAR1` and `SXAR2`. Thus, there are cases where *instruction_counts* / *cycle_counts* is a value larger than 4.

3 *effective_instruction_counts* (Non-Speculative)

Counts the number of committed instructions. `SXAR1` and `SXAR2` are not included.

Instructions per cycle (IPC) can be derived by combining this event with *cycle_counts*.

IPC = *effective_instruction_counts* / *cycle_counts*

If *effective_Instruction_counts* and *cycle_counts* are collected for both user and system modes, the IPC in either user or system mode can be derived.

4 *load_store_instructions* (Non-Speculative)

Counts the number of committed load/store instructions. Also counts atomic load-store instructions. SIMD load/store instructions are counted separately by a different event.

5 *branch_instructions* (Non-Speculative)

Counts the number of committed branch instructions. Also counts the `CALL`, `JMPL`, and `RETURN` instructions.

6 *floating_instructions* (Non-Speculative)

Counts the number of committed 2-operand floating-point instructions. The counted instructions are `FPop1` (TABLE E-5), `FPop2` (TABLE E-6), and `IMPDEP1` with opf<8:4> = 16_{16} or 17_{16}. SIMD versions of these instructions are not counted.

Compatibility Note – In CPUs up to and including SPARC64 VII, this event only counted `FPop1` and `FPop2` instructions.

---

7 *fma_instructions* (Non-Speculative)

Counts the number of committed 3-operand floating-point instructions. The counted instructions are `FM{ADD,SUB}{s,d}`, `FNM{ADD,SUB}{s,d}`, and `FTRIMADDd`. SIMD versions of these instructions are not counted.

Compatibility Note – In CPUs up to and including SPARC64 VII, this event was called *impdep2_instructions* and only counted floating-point multiply-add/subtract instructions.

Two operations are executed per instruction; the number of operations is obtained by multiplying by 2.

8 *prefetch_instructions* (Non-Speculative)

Counts the number of committed prefetch instructions.

9 *SIMD_load_store_instructions* (Non-Speculative)

Counts the number of committed SIMD load/store instructions.

10 *SIMD_floating_instructions* (Non-Speculative)

Counts the number of committed 2-operand SIMD floating-point instructions. The counted instructions are the same as *floating_instructions*.

Two operations are executed per instruction; the number of operations is obtained by multiplying by 2.

11 *SIMD_fma_instructions* (Non-Speculative)

Counts the number of committed 3-operand SIMD floating-point instructions. The counted instructions are the same as *fma_instructions*.

Four operations are executed per instruction; the number of operations is obtained by multiplying by 4.

12 *sxar1_instructions* (Non-Speculative)

Counts the number of committed `SXAR1` instructions.

13 *sxar2_instructions* (Non-Speculative)

Counts the number of committed `SXAR2` instructions.

14 *trap_all* (Non-Speculative)

Counts the occurrences of all trap events. The number of occurrences counted equals the sum of the occurrences counted by all trap PA events.

15 *trap_int_vector* (Non-Speculative)

Counts the occurrences of *interrupt_vector_trap*.

16 *trap_int_level* (Non-Speculative)

Counts the occurrences of *interrupt_level_n*.

17 *trap_spill* (Non-Speculative)

Counts the occurrences of *spill_n_normal* and *spill_n_other*.

18 *trap_fill* (Non-Speculative)

Count the occurrences of *fill_n_normal* and *fill_n_other*.

19 *trap_trap_inst* (Non-Speculative)

Counts the occurrences of *trap_instruction*.

20 *trap_IMMU_miss* (Non-Speculative)

Counts the occurrences of *fast_instruction_access_MMU_miss*.

21 *trap_DMMU_miss* (Non-Speculative)

Counts the occurrences of *fast_data_instruction_access_MMU_miss*.

22 *trap_SIMD_load_across_pages* (Non-Speculative)

Counts the occurrences of *SIMD_load_across_pages*.

Supplemental PA Events

23 *xma_inst* (Non-Speculative)

Counts the number of committed `FPMADDX` and `FPMADDXHI` instructions.

24 *unpack_sxar1* (Non-Speculative)

Counts the number of unpacked `SXAR1` instructions that are committed.

25 *unpack_sxar2* (Non-Speculative)

Counts the number of unpacked `SXAR2` instructions that are committed.

26 *instruction_flow_counts* (Non-Speculative)

Counts the number of committed instruction flows. In SPARC64 VIIIfx, there are instructions that are processed internally as several separate instructions, called instruction flows. This event does not count packed `SXAR1` and `SXAR2` instructions.

27 *ex_load_instructions* (Non-Speculative)

Counts the number of committed integer-load instructions. Counts the *LD(S,U)B{A}*, *LD(S,U)H{A}*, *LD(S,U)W{A}*, *LDD{A}*, and *LDX{A}* instructions.

28 *ex_store_instructions* (Non-Speculative)

Counts the number of committed integer-store and atomic instructions. Counts the *STB{A}*, *STH{A}*, *STW{A}*, *STD{A}*, *STX{A}*, *LDSTUB{A}*, *SWAP{A}*, and *CAS{X}A* instructions.

29 *fl_load_instructions* (Non-Speculative)

Counts the number of committed floating-point load instructions. Counts the *LDF{A}*, *LDDF{A}*, and *LD{X}FSR* instructions.

This event does not count SIMD load instructions or *LDQF{A}*

30 *fl_store_instructions* (Non-Speculative)

Counts the number of committed floating-point store instructions. Counts the *STF{A}*, *STDF{A}*, *STFR*, *STDFR*, and *ST{X}FSR* instructions.

This event does not count SIMD store instructions or *STQF{A}*.

31 *SIMD_fl_load_instructions* (Non-Speculative)

Counts the number of committed floating-point SIMD load instructions. Counted instructions are the SIMD versions of *LDF{A}* and *LDDF{A}*.

32 *SIMD_fl_store_instructions* (Non-Speculative)

Counts the number of committed floating-point SIMD store instructions. Counted instructions are the SIMD versions of *STF{A}*, *STDF{A}*, *STFR*, and *STDFR*.

33 *iwr_empty*

Counts the number of cycles that the IWR (Issue Word Register) is empty. IWR is a four-entry register that holds instructions during instruction decode; the IWR may be empty if an instruction cache miss prevents instruction fetch.

34 *rs1* (Non-Speculative)

Counts the number of cycles in which normal execution is halted due to the following:

- a trap or interrupt
- to update privileged registers
- to guarantee memory order
- RAS-initiated hardware retry

35 *flush_rs* (Non-Speculative)

Counts the number of pipeline flushes due to misprediction. Since SPARC64 VIIIfx supports speculative execution, instructions that should not have been executed may be executed due to misprediction. When it is determined that the predicted path is incorrect, these instructions are cancelled. A pipeline flush occurrs at this time.

misprediction rate = *flush_rs* / *branch_instructions*

36 *0iid_use*

Counts the number of cycles where no instruction is issued. SPARC64 VIIIfx issues up to four instructions per cycle; when no instruction is issued, *0iid_use* is incremented.

In SPARC64 VIIIfx, there are instructions that are processed internally as several separate instructions, called instruction flows. Each of these instruction flows is counted. `SXAR` instructions are also counted.

37 *1iid_use*

Counts the number of cycles where one instruction is issued.

38 *2iid_use*

Counts the number of cycles where two instructions are issued.

39 *3iid_use*

Counts the number of cycles where three instructions are issued.

40 *4iid_use*

Counts the number of cycles where four instructions are issued.

41 *sync_intlk*

Counts the number of cycles where instruction issue is inhibited by a pipeline sync.

42 *regwin_intlk*

Counts the number of cycles where instruction issue is inhibited by a register window switch.

43 *decall_intlk*

Counts the number of cycles where instruction issue is inhibited by a static interlock condition at the decode stage. *decall_intlk* includes *sync_intlk* and *regwin_intlk*; stall cycles due to dynamic conditions (such as reservation station full) are not counted.

44 *rsf_pmmi* (Non-Speculative)

Counts the number of cycles where mixing single-precision and double-precision floating-point operations prevents instructions from issuing.

45 *toq_rsbr_phantom*

Counts the number of instructions that are predicted taken but are not actually branch instructions. Branch prediction in SPARC64 VIIIfx is done prior to instruction decode; branch prediction occurs whether the instruction is a branch instruction or not. Instructions that are not branch instructions may be incorrectly predicted as taken branches.

46 *op_stv_wait* (Non-Speculative)

Counts the number of cycles where no instructions are committed because the oldest, uncommitted instruction is a memory access waiting for data. *op_stv_wait* does not count cycles where a store instruction is waiting for data (atomic instructions are counted).

Note that *op_stv_wait* does not measure the cache miss latency, since any cycles prior to becoming the oldest, uncommitted instruction are not counted.

47 *op_stv_wait_nc_pend* (Non-Speculative)

Counts *op_stv_wait* for noncacheable accesses.

48 *op_stv_wait_ex* (Non-Speculative)

Counts *op_stv_wait* for integer memory access instructions. Does not distinguish between the L1 cache and L2 cache.

49 *op_stv_wait_sxmiss* (Non-Speculative)

Counts *op_stv_wait* caused by an L2$ miss. Does not distinguish between integer and floating-point loads.

50 *op_stv_wait_sxmiss_ex* (Non-Speculative)

Counts *op_stv_wait* caused by an integer-load L2$ miss.

51 *op_stv_wait_pfp_busy* (Non-Speculative)

Counts *op_stv_wait* caused by a memory access instruction that cannot be executed due to the lack of an available prefetch port.

52 *op_stv_wait_pfp_busy_ex* (Non-Speculative)

Counts *op_stv_wait* caused by an integer memory access instruction that cannot be executed due to the lack of an available prefetch port.

53 *op_stv_wait_swpf* (Non-Speculative)

Counts *op_stv_wait* caused by a prefetch instruction that cannot be executed due to the lack of an available prefetch port.

54 *cse_window_empty_sp_full* (Non-Speculative)

Counts the number of cycles where no instructions are committed because the CSE is empty and the store ports are full.

55 *cse_window_empty* (Non-Speculative)

Counts the number of cycles where no instructions are committed because the CSE is empty.

56 *branch_comp_wait* (Non-Speculative)

Counts the number of cycles where no instructions are committed and the oldest, uncommitted instruction is a branch instruction. Measuring *branch_comp_wait* has a lower priority than measuring *eu_comp_wait*.

57 *eu_comp_wait* (Non-Speculative)

Counts the number of cycles where no instructions are committed and the oldest, uncommitted instruction is an integer or floating-point instruction. Measuring *eu_comp_wait* has a higher priority than measuring *branch_comp_wait*.

58 *fl_comp_wait* (Non-Speculative)

Counts the number of cycles where no instructions are committed and the oldest, uncommitted instruction is a floating-point instruction.

59 *0endop* (Non-Speculative)

Counts the number of cycles where no instructions are committed. *0endop* also counts cycles where the only instruction that commits is an `SXAR` instruction.

60 *1endop* (Non-Speculative)

Counts the number of cycles where one instruction is committed.

61 *2endop* (Non-Speculative)

Counts the number of cycles where two instructions are committed.

62 *3endop* (Non-Speculative)

Counts the number of cycles where three instructions are committed.

63 *inh_cmit_gpr_2write* (Non-Speculative)

Counts the number of cycles where fewer than four instructions are committed due to a lack of GPR write ports (only 2 integer registers can be updated each cycle).

64 *suspend_cycle* (Non-Speculative)

Counts the number of cycles where the instruction unit is halted by a `SUSPEND` or `SLEEP` instruction.

65 *sleep_cycle* (Non-Speculative)

Counts the number of cycles where the instruction unit is halted by a `SLEEP` instruction

66 *single_sxar_commit* (Non-Speculative)

Counts the number of cycles where the only instruction committed is an unpacked `SXAR` instruction. These cycles are also counted by *0endop*.

Q.2.2 MMU and L1 cache Events

Standard PA Events

1 *uITLB_miss*

Counts the occurrences of instruction uTLB misses.

2 *uDTLB_miss*

Counts the occurrences of data uTLB misses.

Note – Main TLB misses are counted by *trap_IMMU_miss* and *trap_DMMU_miss*.

---

3 *L1I_miss*

Counts the occurrences of I1 cache misses.

4 *L1D_miss*

Counts the occurrences of D1 cache misses.

5 *L1I_wait_all*

Counts the total time spent processing L1 instruction cache misses, i.e. the total miss latency. In SPARC64 VIIIfx, the L1 cache is a non-blocking cache that can process multiple cache misses in parallel; *L1I_wait_all* only counts the miss latency for one of these misses. That is, the overlapped miss latencies are not counted.

6 *L1D_wait_all*

Counts the total time spent processing L1 data cache misses, i.e. the total miss latency. In SPARC64 VIIIfx, the L1 cache is a non-blocking cache that can process multiple cache misses in parallel; *L1D_wait_all* only counts the miss latency for one of these misses. That is, the overlapped miss latencies are not counted.

Supplemental PA Events

7 *uITLB_miss2*

Counts the number of reads from the fITLB caused by an instruction fetch uTLB miss.

8 *uDTLB_miss2*

Counts the number of reads from the fDTLB caused by a data access uTLB miss.

9 *swpf_success_all*

Counts the number of `PREFETCH` instructions not lost in the SU and sent to the SX .

10 *swpf_fail_all*

Counts the number of prefetch instructions lost in the SU.

11 *swpf_lbs_hit*

Counts the number of prefetch instructions that hit in the L1 cache.

The number of prefetch instructions sent to the SU
= *swpf_success_all* + *swpf_fail_all* + *swpf_lbs_hit*

12 *L1I_thrashing*

Counts the occurrences of an L2 read request being issued twice in the period between acquiring and releasing a store port. When instruction fetch causes an L1 instruction cache miss, the requested data is updated in the L1I$. This counter is incremented if the updated data is evicted before it can be read.

13 *L1D_thrashing*

Counts the occurrences of an L2 read request being issued twice in the period between acquiring and releasing a store port. When a memory access instruction causes an L1 data cache miss, the requested data is updated in the L1D$. This counter is incremented if the updated data is evicted before it can be read.

Q.2.3 L2 cache Events

L2 cache events may be due to the actions of a CPU core or external requests. Events caused by a CPU core are counted separately for each core; those caused by external requests are counted for all cores.

Most L2 cache events are categorized as either demand (dm) or prefetch (pf) events, but these events do not necessarily correspond to load/store/atomic instructions and prefetch instructions. This is because:

- When a load/store instruction cannot be executed due to a lack of resources needed to move data into the L1 cache, data is first moved into the L2 cache. Once L1 cache resources become available, the load/store instruction is executed. That is, only the request to move data into the L2 cache is processed as a prefetch request.
- The hardware prefetch mechanisms generates prefetch requests.
- L1 cache prefetch instructions are processed as demand requests.

It follows that the demand and prefetch L2 cache events correspond to the following:

- A demand (dm) request to the L2 cache is an instruction fetch, load/store instruction, or L1 prefetch instruction that was able to acquire the resources needed to access memory.
- A prefetch (pf) request to the L2 cache is an instruction fetch, load/store instruction, or L1 prefetch instruction that could not acquire the resources needed to access memory; a hardware prefetch is also a prefetch access.

Standard PA Events

1 *L2_read_dm*

Counts the number of L2 cache references by demand requests. A single block load/store instruction is counted as 8 cache references.

External cache-reference requests are not counted.

2 *L2_read_pf*

Counts L2 cache references by prefetch requests. A single block load/store instruction is counted as 8 cache references.

3 *L2_miss_dm*

Counts the number of L2 cache misses caused by demand requests.

This counter is the sum of the *L2_miss_count_dm_bank*{*0,1,2,3*}.

4 *L2_miss_pf*

Counts the number of L2 cache misses caused by prefetch requests.

This counter is the sum of the *L2_miss_count_pf_bank*{*0,1,2,3*}.

5 *L2_miss_count_dm_bank{0,1,2,3}*

Counts the number of L2 cache misses for each bank caused by demand requests.

Note – Consider the case where a prefetch to an address misses in the L2 cache, which issues a memory access request. If the corresponding demand request arrives before the data is returned, the resulting L2 cache demand miss is not counted.

---

6 *L2_miss_count_pf_bank{0,1,2,3}*

Counts the number of L2 cache misses for each bank caused by prefetch requests.

7 *L2_miss_wait_dm_bank{0,1,2,3}*

Counts the total time spent processing L2 cache misses for each bank caused by demand requests, i.e. the total miss latency for each bank. The latency of each memory access request is counted.

Note – Consider the case where a prefetch to an address misses in the L2 cache, which issues a memory access request. If the corresponding demand request arrives before the data is returned, *L2_miss_wait_dm_bank{0,1,2,3}* counts the cycles after the demand request arrives and before the data is returned.

8 *L2_miss_wait_pf_bank{0,1,2,3}*

Counts the total time spent processing L2 cache misses for each bank caused by prefetch requests, i.e. the total miss latency for each bank. The latency of each memory access request is counted.

The L2 cache miss latency can be derived by dividing *L2_miss_wait_** by *L2_miss_count_**.

Note – The L2 cache miss latency can be obtained from *L2_miss_count_** and *L2_miss_wait_**. Consider the case where a demand request arrives while a prefetch request is being processed; because of the way these events are defined, measuring the prefetch and demand latencies separately may overestimate the demand latency and underestimate the prefetch latency.

9 *L2_wb_dm*

Counts the occurrences of writeback by demand L2-cache misses.

10 *L2_wb_pf*

Counts the occurrences of writeback by prefetch L2-cache misses.

Supplemental PA Events

11 *lost_pf_pfp_full*

Counts the number of prefetch requests lost due to PF port full.

12 *lost_pf_by_abort*

Counts the number of prefetch requests lost due to SX pipe abort.

Bus Transaction EventsStandard PA Events

1 *cpu_mem_read_count*

Counts the number of memory read requests issued by the CPU.

2 *cpu_mem_write_count*

Counts the number of memory write requests issued by the CPU.

3 *IO_mem_read_count*

Counts the number of memory read requests issued by I/O.

4 *IO_mem_write_count*

Counts the number of memory write requests issued by I/O.

Only ICC-FST is counted by this event. ICC-PST can be counted using *IO_pst_count*.

5 *bi_count*

Counts the number of external cache-invalidate requests received by the CPU chip. These requests that do not check the cache data before invalidating.

For this event, the same value is counted by all cores.

6 *cpi_count*

Counts the number of external cache-copy-and-invalidate requests received by the CPU chip. These requests copy updated cache data to memory before invalidating; cache data that is consistent with memory does not need to be copied and is invalidated.

For this event, the same value is counted by all cores.

Implementation Note – This PA event does not exist in SPARC64 VIIIfx; compatibility, however, is preserved.

---

7 *cpb_count*

Counts the number of external cache-copyback requests received by the CPU chip. These request copy updated cache data to memory.

For this event, the same value is counted by all cores.

Implementation Note – This PA event does not exist in SPARC64 VIIIfx; compatibility, however, is preserved.

---

8 *cpd_count*

Counts the number of external cache-read requests received by the CPU chip. These requests, such as a DMA read request, read the updated data in the cache without writing the data to memory.

For this event, the same value is counted by all cores.

9 *IO_pst_count*

Counts the number of memory write requests (ICC-PST) issued by I/O.

Q.3 Cycle Accounting

Cycle accounting can be generally defined as a method for analyzing the factors contributing to performance bottlenecks. The total time (number of CPU cycles) required to execute an instruction sequence can be classified as time spent in various CPU execution states (executing instructions, waiting for a memory access, waiting for execution to complete, etc). This can provide a good grasp of the performance bottlenecks involved and allow performance to be analyzed and improved. In fact, SPARC64 VIIIfx defines a large number of PA events that record detailed information about CPU execution states; this enables efficient analysis of bottlenecks and is useful for performance tuning.

In this document, however, cycle accounting is specifically defined as the analysis of instructions as they are committed in order. SPARC64 VIIIfx is an out-of-order execution CPU with multiple execution units; the CPU is generally in a state where executing instructions and waiting instructions are thoroughly mixed together. One instruction may be waiting for data from memory, another executing a floating-point multiply, and yet another waiting for confirmation of the branch direction. Simply analyzing the reasons why individual instructions are waiting is not useful. Cycle accounting classifies cycles by the number of instructions committed; when a cycle commits no instructions, the conditions that prevented instructions from committing are analyzed.

SPARC64 VIIIfx commits up to 4 instructions per cycle. The more cycles that commit the maximum number of instructions, the better the execution efficiency. Cycles that do not commit any instructions have an extremely negative effect on performance, and it is important to perform a detailed analysis. The main causes are:

- Waiting for a memory access to return data.
- Waiting for instruction execution to complete.
- Instruction fetch is unable to supply the pipeline with instructions.

The chart in TABLE Q-2 lists useful PA events for cycle accounting, as well as how those PA events can be used to analyze execution efficiency.

The diagram in FIGURE Q-1 shows the relationship between the various *op_stv_wait_** events. The PA events marked with a † in the chart and diagram are synthetic events; that is, they are calculated from other PA events.

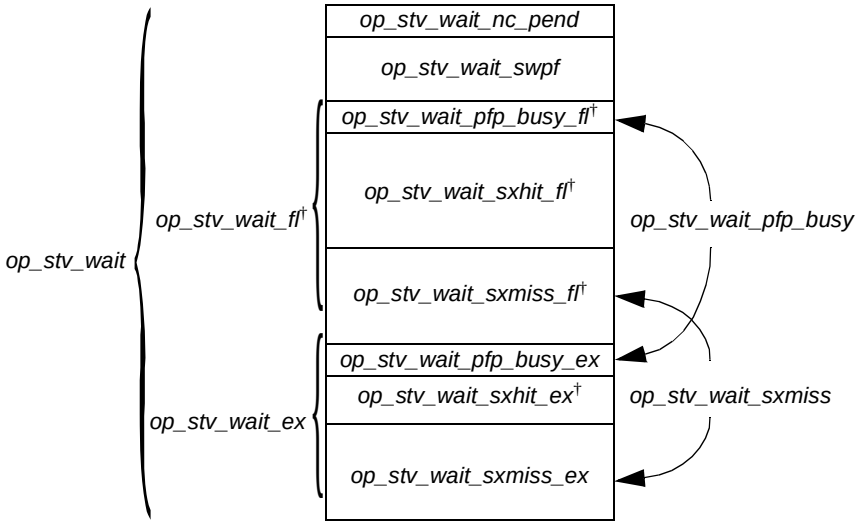


FIGURE Q-1 Breakdown of *op_stv_wait*

TABLE Q-2 Useful Performance Events for Cycle Accounting

Instructions Committed per Cycle	Cycles	Remarks
4	*cycle_counts* *- 3endop - 2endop* *- 1endop - 0endop*	N/A (Four instructions are committed in a cycle)
3	*3endop*	*inh_cmit_gpr_2write* measures one of the conditions that can prevent subsequent instruction(s) from committing.
2	*2endop*	
1	*1endop*	
0	Execution: *eu_comp_wait* *+ branch_comp_wait*	*eu_comp_wait* *= ex_comp_wait*† *+ fl_comp_wait*
	Instruction Fetch: *cse_window_empy*	*cse_window_empty* *= cse_window_empty_sp_full* *+ sleep_state* + misc.†
	L1D cache miss: *op_stv_wait* - L2 cache miss (see below)	
	L2 cache miss: *op_stv_wait_sxmiss* *+ op_stv_wait_nc_pend*	
	Others: *0endop* *- op_stv_wait* *- cse_window_empy* *- eu_comp_wait* *- branch_comp_wait* *-(instruction_flow_counts* *- instruction_counts*)	

F.APPENDIX **R**

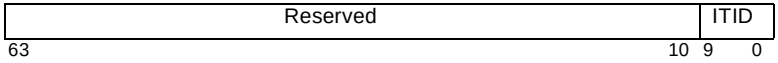
System Programmer's Model

This appendix describes CPU components that have not been discussed elsewhere.

Information about how to control the CPU via the service processor is out of the scope of this document and is not discussed.

R.1 System Config Register

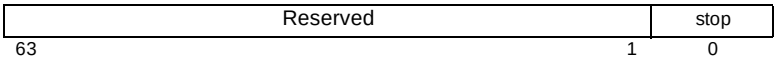
Register Name	`ASI_SYS_CONFIG`
ASI	$4A_{16}$
VA	—
Access Type	Supervisor read/write (write is ignored)



Bit	Field	Access	Description
63:10	TBD	TBD	TBD
9:0	`ITID`	R	Thread ITID (Interrupt Target ID)。

R.2 STICK Control Register

Register Name	ASI_STICK_CNTL
ASI	$4B_{16}$
VA	00_{16}
Access Type	Supervisor read/write



Bit	Field	Access	Description
63:1	—		
0	stop	RW	When stop is 1, STICK count-up is halted. When stop is 0, STICK count-up is restarted.

The STICK_CNTL register is used to enable/disable STICK count-up and is shared by all cores. If any core sets STICK_CNTL, the STICK counters of all cores are enabled/disabled at the same time.

STICK count-up is halted while STICK.stop = 1. This has the following effects:

- Setting the STICK_CMPR does not post an interrupt, as the value is never reached.

  Of course, if STICK.stop = 1 and

  - STICK_CMPR.INT_DIS = 0
  - STICK_CMPR.STICK_CMPR = STICK.counter

  the value is already reached, and SOFTINT.SM is set. A level-14 interrupt is posted when PSTATE.IE = 1 and PIL < 14.

- Cores executing the SLEEP instruction do not wake up.

When multiple cores attempt to write STICK_CNTL at the same time, the requests are processed one at a time. The order in which they are processed is dependent on the hardware implementation.

Programming Note – The STICK_CNTL register is managed via a core.

After a write to STICK_CNTL, a read/write of the STICK register does not execute until the the write commits and a FLUSH instruction is executed. The time required for the write to commit is undefined. The core that wrote STICK_CNTL reads STICK_CNTL to check that the write has committed. When a read/write of the STICK register is performed before the write commits, the value written to/read from STICK is not preserved.

F.APPENDIX **S**

Summary of Specification Differences

This appendix summarizes the differences between the SPARC64 VIIIfx specification and the SPARC V9, SPARC JPS1, and SPARC64 VII specifications. This appendix is provided for the convenience of the reader and is not a formal specification. Please refer to the other chapters in this document for formal definitions of specific items.

TABLE S-1 lists the differences between the SPARC64 VIIIfx specification and the SPARC V9, SPARC JPS1, and SPARC64 VII specifications. The “Binary Compatibility” column indicates whether software that conforms to the specification for SPARC V9, SPARC JPS1, or SPARC64 VII will run on the SPARC64 VIIIfx CPU.[1]

1. Software that uses aspects of the architecture that are *reserved* by the SPARC V9, SPARC JPS1, or SPARC64 VII specification is not compatible. TABLE S-1 does not list *reserved* items.

TABLE S-1 Summary of Specification Differences *(1 of 4)*

Item	Specification				Binary Compatibility			Page
	V9	JPS1	SPARC64 VII	SPARC64 VIIIfx	V9	JPS1	SPARC64 VII	
Architecture								
Core, thread	undef		4 cores, 2 threads per core	8 cores, 1 thread per core			no	10
Integer registers	160 registers			192 registers				20
Floating-point register	32 single-precision registers 32 double-precision registers			32 single-precision registers 256 double-precision registers Double-precision registers can be used for single-precision operations.				20
ASR	undef	`%pcr`, `%pic`, `%dcr`, `%gsr`, `%softint`, `%tick_cmpr`, `%sys_tick`, `%sys_tick_cmpr`		`%pcr`, `%pic`, `%dcr`, `%gsr`, `%softint`, `%tick_cmpr`, `%sys_tick`, `%sys_tick_cmpr`, `%xar`, `%xasr`, `%txar`				26
Physical address	undef	at least 43 bits	47 bits	41 bits		no		178, 183
RSTVaddr	undef	impl-dep	PA = 7fff f000 0000_{16}	PA = 1ff f000 0000_{16}			no	45
Cache	undef		• L1: 64KB/2way(I), 64KB/2way(D), 64byte line • L2: 6MB/12way, 256byte line/4sublines	• L1: 32KB/2way(I), 32KB/2way(D), 128byte line Sector cache. • L2: 6MB/12way, 128byte line Index hashing, sector cache.			no (index hash)	12, 12, 230, 231
`SXflush`	undef		yes	no			no	—
TLB	undef		32(fTLB)+2048/4way(sTLB), I,D TLBs. fTLB is the victim cache for the sTLB.	16(fTLB)+256/4way(sITLB), 512/4way(sDTLB) No victim cache functionality. Error injection function deleted.			no	175, 193
Page size	undef	8KB, 64KB, 512KB, 4MB	8KB, 64KB, 512KB, 4MB, 32MB, 256MB	8KB, 64KB, 512KB, 4MB, 32MB, 256MB, 2GB				177
`TSB`	undef	On a TLB miss, hardware computes pointers into the TSB.		No hardware support. Deleted ASIs: • I/D TSB Primary Extension • D TSB Secondary Extension • I/D TSB Nuclues Extension • I/D TSB 8KB ptr • I/D TSB 64KB ptr • D TSB Direct ptr The `split` field in TSB Base is deleted.		no		179, 185, 194

TABLE S-1 Summary of Specification Differences *(2 of 4)*

Item	Specification				Binary Compatibility			Page
	V9	JPS1	SPARC64 VII	SPARC64 VIIIfx	V9	JPS1	SPARC64 VII	
Hardware Barrier	undef		BPU 2, BB 12/BPU, BST 24bit/ BPU	No BPU, BB 12, BST 8bit/BB			no	222
Hardware Prefetch	undef		Yes. Cannot be managed by software, so it is not described in the specification.	Yes. Can be managed by software.				237
Interrupt Registers	undef	8 registers		3 registers		no		242
Instructions								
impdep1	undef	VIS	VIS, `SLEEP`, `SUSPEND`	`SLEEP`, `SUSPEND`, `FCMP(EQ,LE,LT,NE,GT,GE)E(s,d)`, `FCMP(EQ,NE)(s,d)`, `FMAX(s,d)`, `FMIN(s,d)`, `FRCPA(s,d)`, `FRSQRTA(s,d)`, `FTRISSELd`, `FTRISMULd`				78, 79, 116, 118, 120, 125
impdep2	undef	undef	`F{N}M(ADD,SUB)(s,d)`, `FPMADDX{HI}`	`F{N}M(ADD,SUB)(s,d)`, `FPMADDX{HI}`, `FTRIMADDd`, `FSELMOV(s,d)`				72, 80, 124
load/store			`QUAD_LDD_PHYS`	`QUAD_LDD_PHYS`, `ST{D}FR`, `XFILL`				89, 124, 135
Other			`POPC`	`POPC`, `SXAR`	V8[1]			95, 133
SIMD	no			yes				
block load, block store (bld/bst) behavior	undef	• Data in the cache is invalidated, and bst commit is written to memory. • Register dependency is ignored.	• Data in the cache is invalidated, and bst commit is written to memory. • Register dependency is detected. • Internally, memory model for bld/bst is RMO. Ordering between preceding and succeeding instructions does not conform to V9. • If the TTE is invalidated during a bld/bst, a fast_data_access_MMU_miss occurs.	• bst commit is stored in the cache. • Conforms to TSO.		no		68
`rd` update on a load exception	impdep. #44		Not updated.	Not updated for non-SIMD. There are cases where `rd` is updated for SIMD.				82, 86

TABLE S-1 Summary of Specification Differences *(3 of 4)*

Item	Specification				Binary Compatibility			Page
	V9	JPS1	SPARC64 VII	SPARC64 VIIIfx	V9	JPS1	SPARC64 VII	
LDDF/ STDF_mem_address_not_aligned	impdep. #109, #110		Exception signalled.	Exception signalled for non-SIMD. Exception not signalled for SIMD.				82, 86, 101, 105
Instruction attributes	no			Can specify SIMD, cache sector, and disable hardware prefetch.				29
Traps								
Types			*async_data_error*	*async_data_error*, *illegal_action*, *SIMD_load_across_pages*				53
Priority			*async_data_error* is priority 2.	*async_data_error* is priority 2, *illegal_action* is priority 8.5, *SIMD_load_across_pages* is priority 12, and *fp_exception_other* (ftt = *unimplemented_FPop*) is priority 8.2. When *fp_exception_ieee754* and *fp_exception_other* (ftt = unfinished_FPop) occur simultaneously for a SIMD operation, *fp_exception_other* takes priority.		The behavior of *fp_exception_other* differs, but compatbility is unaffected.		50
Registers saved				For these added registers,、 • on a trap TXAR[TL] ← XAR XAR ← 0 • on a DONE/RETRY XAR ← TXAR[TL] TXAR[TL] is unchanged				51
Register Functions								
%ver.impl			7	8			no	26
%fsr.cexc update	At most 1 bit is set.			There are cases where a SIMD operation sets 2 bits.				24
PA Event types			6 bits	7 bits				27, 304
watchpoint		VA, PA can be specified separately.		VA, PA share a register.		no		36
AFAR		optional	Fixed value of 0. Readable.	Deleted.			no	—
EIDR			bits <13:0> are valid. Software sets the value 10_{02} in bits <13:12>. Used as the error mark ID.	bits <2:0> are valid. bits <13:12> have a fixed value of 10_{02} in hardware.				270

TABLE S-1 Summary of Specification Differences *(4 of 4)*

Item	Specification				Binary Compatibility			Page
	V9	JPS1	SPARC64 VII	SPARC64 VIIIfx	V9	JPS1	SPARC64 VII	
SYS_CONFIG			JB_CONFIG_REGISTER UC_S, UC_SW, CLK_MODE, ITID fields are defined.	SYS_CONFIG Only the ITID field is defined.			no	323
Other								
Display cause of fatal error			Cause can be identified from STCHG_ERROR_INFO.	Cause of fatal error is not displayed.			no	272
STICK start/stop			No (controlled by SC).	Yes.			no	324

1.SXAR is not V8-compatible.

Index

SYMBOLS

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z