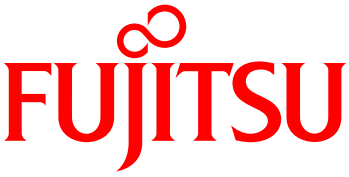


SPARC64™ VI Extensions

Fujitsu Limited

Release 1.3, 27 Mar. 2007

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

Copyright© 2006 Fujitsu Limited, 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan. All rights reserved.

This product and related documentation are protected by copyright and distributed under licenses restricting their use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Fujitsu Limited and its licensors, if any.

The product(s) described in this book may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

SPARC64™ is a registered trademark of SPARC International, Inc., licensed exclusively to Fujitsu Limited.

UNIX is a registered trademark of The Open Group in the United States and other countries.
Sun, Sun Microsystems, the Sun logo, Solaris, and all Solaris-related trademarks and logos are registered trademarks of Sun Microsystems, Inc.
Fujitsu and the Fujitsu logo are trademarks of Fujitsu Limited.

This publication is provided “as is” without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or noninfringement. This publication could include technical inaccuracies or typographical errors. changes are periodically added to the information herein; these changes will be incorporated in new editions of the publication. hal computer systems, inc. may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

Contents

F.CHAPTER **1**

Overview

1.1 Navigating *the SPARC64™ VI Extensions*

The SPARC64 VI processor fully implements the instruction set architecture that conforms to **Commonality**.

- *SPARC Joint Programming Specification 1 (JPS1): Commonality*

This *SPARC64 VI Extensions* describes implementation specific portions of SPARC64 VI. We suggest that you approach this specification as follows.

1. **Familiarize yourself with the SPARC64 VI processor and its components by reading the following sections in this specification:**
   - *The SPARC64 VI processor* on page 2
   - *Component Overview* on page 4
   - *Processor Pipeline* on page 30
2. **Study the terminology in Chapter 2, *Definitions*.**
3. **For details of architectural changes, see the remaining chapters in this Specification as your interests dictate.**

1.2 Fonts and Notational Conventions

Please refer to Section 1.2 of **Commonality** for font and notational conventions.

1.3 The SPARC64 VI processor

The SPARC64 VI processor is a high-performance, high-reliability, and high-integrity processor that fully implements the instruction set architecture that conforms to SPARC V9, as described in **Commonality**. In addition, the SPARC64 VI processor implements the following features:

- 64-bit virtual address space and 43-bit physical address space
- Advanced RAS features that enable high-integrity error handling
- Multi threaded Processing (MTP)

Microarchitecture for High Performance

The SPARC64 VI is an out-of-order execution superscalar processor that issues up to four instructions per cycle. Instructions in the predicted path are issued in program order and are stored temporarily in *reservation stations* until they are dispatched out of program order to the appropriate execution units. Instructions commit in program order when no exceptions occur during execution and all prior instructions commit (that is, the result of the instruction execution becomes visible). Out-of-order execution in SPARC64 VI contributes to high performance.

SPARC64 VI implements a large branch history buffer to predict its instruction path. The history buffer is large enough to sustain a good prediction rate for large-scale programs such as DBMS and to support the advanced instruction fetch mechanism of SPARC64 VI. This instruction fetch scheme predicts the execution path beyond the multiple conditional branches in accordance with the branch history. It then tries to prefetch instructions on the predicted path as much as possible to reduce the effect of the performance penalty caused by instruction cache misses.

High Integration

SPARC64 VI integrates an on-board, associative, level-2 cache. The level-2 cache is unified for instruction and data. It is the lowest layer in the cache hierarchy.

This integration contributes to both performance and reliability of SPARC64 VI. It enables shorter access time and more associativity and thus contributes to higher performance. It contributes to higher reliability by eliminating the external connections for level-2 cache.

High Reliability and High Integrity

SPARC64 VI implements the following advanced RAS features for reliability and integrity beyond that of ordinary microprocessors.

1. Advanced RAS features for caches

- Strong cache error protection:
  - ECC protection for D1 (Data level 1) cache data, U2 (unified level 2) cache data, and the U2 cache tag.
  - Parity protection for I1 (Instruction level 1) cache data.
  - Parity protection and duplication for the I1 cache tag and the D1 cache tag.
- Automatic correction of all types of single-bit error:
  - Automatic single-bit error correction for the ECC protected data.
  - Invalidation and refilling of I1 cache data for the I1 cache data parity error.
  - Copying from duplicated tag for I1 cache tag and D1 cache tag parity errors.
- Dynamic way reduction while cache consistency is maintained.
- Error marking for cacheable data with uncorrectable errors:
  - Special error-marking pattern for cacheable data with uncorrectable errors. The identification of the module that first detects the error is embedded in the special pattern.
  - Error-source isolation with faulty module identification in the special error-marking. The identification information enables the processor to avoid repetitive error logging for the same error cause.

2. Advanced RAS features for the core

- Strong error protection:
  - Parity protection for all data paths.
  - Parity protection for most of software-visible registers and internal temporary registers.
  - Parity prediction or residue checking for the accumulator output.
- Hardware instruction retry
- Support for software instruction retry (after failure of hardware instruction retry)
- Error isolation for software recovery:
  - Error indication for each programmable register group.
  - Indication of retryability of the trapped instruction.
  - Use of different error traps to differentiate degrees of adverse effects on the CPU and the system.

3. Extended RAS interface to software

- Error classification according to the severity of the effect on program execution:
  - Urgent error (nonmaskable): Unable to continue execution without OS intervention; reported through a trap.
  - Restrainable error (maskable): OS controls whether the error is reported through a trap, so error does not directly affect program execution.
- Isolated error indication to determine the effect on software
- Asynchronous data error (ADE) trap for additional errors:
  - Relaxed instruction end method (precise, retryable, not retryable) for the *async_data_error* exception to indicate how the instruction should end; depends on the executing instruction and the detected error.

- Some ADE traps that are deferred but retryable.
- Simultaneous reporting of all detected ADE errors at the error barrier for correct handling of retryability.

Multi threaded Processing.

SPARC64 VI is a quadruple threaded processor, which has two dual threaded physical cores. The two threads belong to the same physical core sharing most of the physical resources, while the two cores do not share physical resources except L2 Cache and system interface.

1.3.1 Component Overview

The SPARC64 VI processor contains these components.

- Instruction control Unit (IU)
- Execution Unit (EU)
- Storage Unit (SU)
- Secondary cache and eXternal access Unit (SXU)

FIGURE 1-1 illustrates the major units; the following subsections describe them.

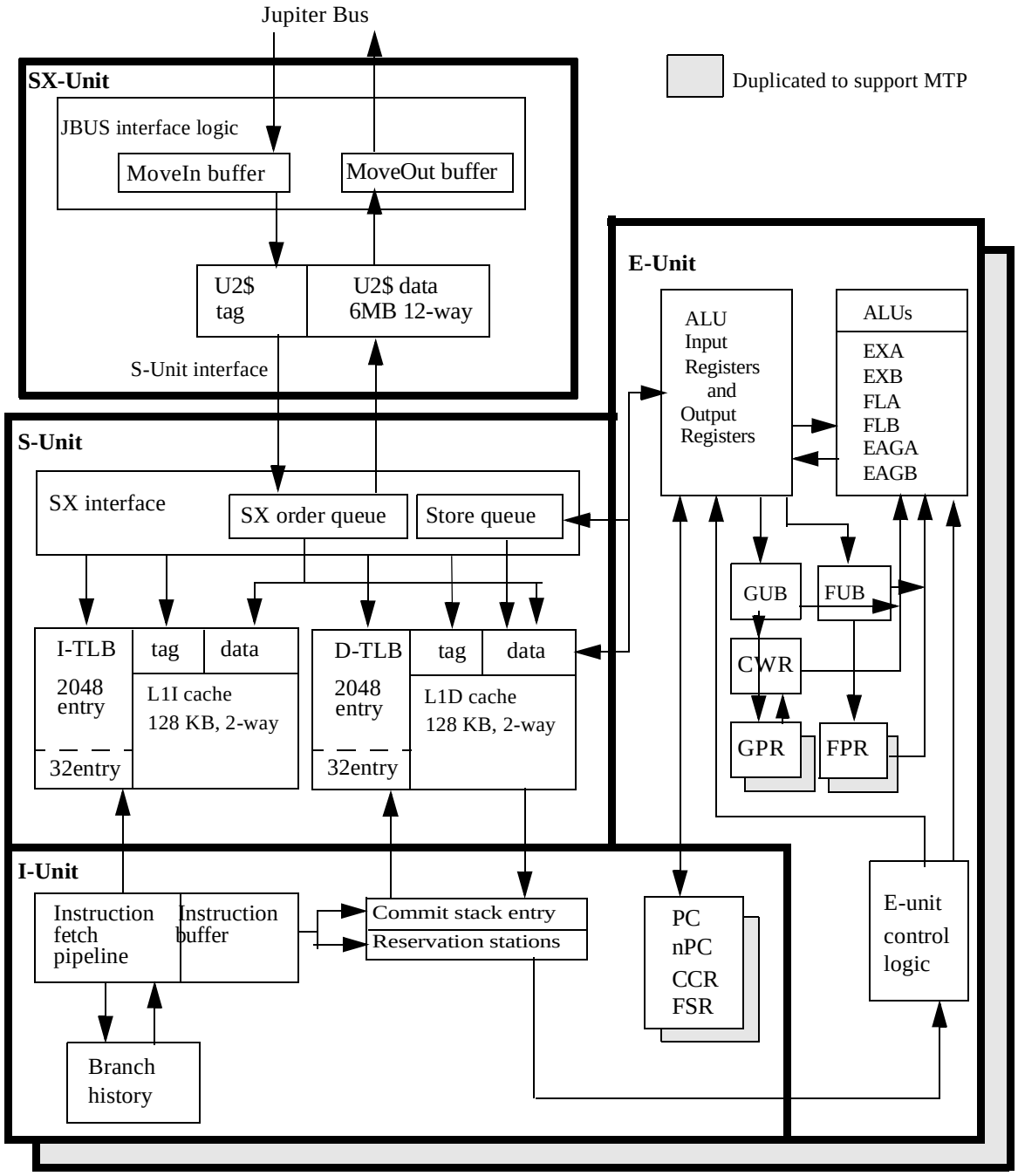


FIGURE 1-1 SPARC64 VI Block Diagram

1.3.2 Instruction Control Unit (IU)

The IU predicts the instruction execution path, fetches instructions on the predicted path, distributes the fetched instructions to appropriate reservation stations, and dispatches the instructions to the execution pipeline. The instructions are executed out of order, and the IU commits the instructions in order. Major blocks are defined in TABLE 1-1.

TABLE 1-1 Instruction Control Unit Major Blocks

Name	Description
Instruction fetch pipeline	Four stages: fetch address generation, iTLB and L1I cache access, iTLB and L1I cache match, and a write to I-buffer.
Branch history	A table to predict branch target and direction.
Instruction buffer	A buffer to hold instructions fetched.
Reservation station	Six reservation stations to hold instructions until they can execute: `RSBR` for branch and the other control-transfer instructions; `RSA` for load/store instructions; `RSEA` and `RSEB` for integer arithmetic instructions; `RSFA` and `RSFB` for floating-point arithmetic and VIS instructions.
Commit stack entries	A buffer to hold information about instructions issued but not yet committed.
`PC`, `nPC`, `CCR`, `FSR`	Program-visible registers for instruction execution control.

1.3.3 Execution Unit (EU)

The EU carries out execution of all integer arithmetic, logical, shift instructions, all floating-point instructions, and all VIS graphic instructions. TABLE 1-2 describes the EU major blocks.

TABLE 1-2 Execution Unit Major Blocks

Name	Description
`GUB`	General register (gr) renaming register file.
`GPR`	Gr architecture register file.
`FUB`	Floating-point (fr) renaming register file.
`FPR`	Fr architecture register file.
EU control logic	Controls the instruction execution stages: instruction selection, register read, and execution.
Interface registers	Input/output registers to other units.
Two integer execution pipelines (EXA, EXB)	64-bit ALU and shifters.

TABLE 1-2 Execution Unit Major Blocks *(Continued)*

Name	Description
Two floating-point and graphics execution pipelines (FLA, FLB)	Each floating-point execution pipeline can execute floating point multiply, floating point add/sub, floating-point multiply and add, floating point div/sqrt, and floating-point graphics instruction.
Two virtual address adders for memory access pipeline (`EAGA`, `EAGB`)	Two 64-bit virtual addresses for load/store.

1.3.4 Storage Unit (SU)

The SU handles all sourcing and sinking of data for load and store instructions. TABLE 1-3 describes the SU major blocks.

TABLE 1-3 Storage Unit Major Blocks

Name	Description
Instruction level-1 cache	128-Kbyte, 2-way associative, 64-byte line; provides low latency instruction source.
Data level-1 cache	128-Kbyte, 2-way associative, 64-byte line, writeback; provides the low latency data source for loads and stores.
Instruction Translation Buffer	2048 entries, 2-way associative TLB (sTLB). 32 entries, fully associative TLB (fTLB).
Data Translation Buffer	2048 entries, 2-way associative TLB (sTLB). 32 entries, fully associative TLB (fTLB).
Store Buffer and Write Buffer	Decouples the pipeline from the latency of store operations. Allows the pipeline to continue flowing while the store waits for data, and eventually writes into the data level 1 cache.

1.3.5 Secondary Cache and External Access Unit (SXU)

The SXU controls the operation of unified level-2 caches and the external data access interface (Jupiter Bus). TABLE 1-4 describes the major blocks of the SXU.

TABLE 1-4 Secondary Cache and External Access Unit Major Blocks

Name	Description
Unified level-2 cache	6-Mbyte, 12-way associative, 256-byte line (four 64-byte sublines), writeback; provides low latency data source for both instruction level-1 cache and data level-1 cache.
Movein buffer	Catches returning data from memory system in response to the cache line read request.

TABLE 1-4 Secondary Cache and External Access Unit Major Blocks

Name	Description
Moveout buffer	Holds writeback data to memory.
Jupiter Bus interface control logic	Send/receive transaction packets to/from Jupiter Bus interface connected to the system.

F.CHAPTER **2**

Definitions

This chapter defines concepts unique to SPARC64 VI., the Fujitsu implementation of SPARC JPS1. For definition of terms that are common to all implementations, please refer to Chapter 2 of **Commonality**.

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

completed Term applied to an instruction after it has *finished*, has sent a non-error status to the issue unit, and all of its source operands are non-speculative. **Note:** Although the state of the machine has been temporarily altered by completion of an instruction, the state has not yet been permanently changed and the old state can be recovered until the instruction has been *committed.*

executed Term applied to an instruction that has been processed by an execution unit such as a load unit. An instruction is in execution as long as it is still being processed by an execution unit.

fetched Term applied to an instruction that is obtained from the I2 instruction cache or from the on-chip internal cache and sent to the issue unit.

finished Term applied to an instruction when it has completed execution in a functional unit and has forwarded its result onto a result bus. Results on the result bus are transferred to the register file, as are the waiting instructions in the instruction queues.

instruction initiated Term applied to an instruction when it has all of the resources that it needs (for example, source operands) and has been selected for execution.

instruction dispatched Synonym: **instruction initiated**.

instruction issued Term applied to an instruction when it has been dispatched to a reservation station.

instruction retired Term applied to an instruction when all machine resources (serial numbers, renamed registers) have been reclaimed and are available for use by other instructions. An instruction can only be retired after it has been *committed.*

instruction stall Term applied to an instruction that is not allowed to be issued. Not every instruction can be issued in a given cycle. The SPARC64 VI implementation imposes certain issue constraints based on resource availability and program requirements.

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

machine sync The state of a machine when all previously executing instructions have committed; that is, when no issued but uncommitted instructions are in the machine.

Memory Management Unit (MMU) Refers to the address translation hardware in SPARC64 VI that translates 64-bit virtual address into physical address. The MMU is composed of the mITLB, mDTLB, uITLB, uDTLB, and the ASI registers used to manage address translation.

mTLB Main TLB. Split into I and D, called mITLB and mDTLB, respectively. Contains address translations for the uITLB and uDTLB. When the uITLB or uDTLB do not contain a translation, they ask the mTLB for the translation. If the mTLB contains the translation, it sends the translation to the respective uTLB. If the mTLB does not contain the translation, it generates a fast access exception to a software translation trap handler, which will load the translation information (TTE) into the mTLB and retry the access. *See also* **TLB**.

uDTLB Micro Data TLB. A small, fully associative buffer that contains address translations for data accesses. Misses in the uDTLB are handled by the mTLB.

uITLB Micro Instruction TLB. A small, fully associative buffer that contains address translations for instruction accesses. Misses in the uTLB are handled by the mTLB.

MTP Multi Threaded Processor. A processor module containing more than one thread. (May also be used as an acronym for Multi threaded Processing.)

non-speculative A distribution system whereby a result is guaranteed known correct or an operand state is known to be valid. SPARC64 VI employs speculative distribution, meaning that results can be distributed from functional units before the point at which guaranteed validity of the result is known.

physical core A physical core includes an execution pipeline and associated structures, such as caches, that are required for performing the execution of instructions from one or more software threads. A physical core contains one or more threads. The physical core provides the necessary resources for each thread to make forward progress at a reasonable rate.

processor module A *processor module* is the unit on which a shared interface is provided to control the configuration and execution of a collection of threads. A *processor module* contains one or more physical cores, each of which contains one or more threads. On a more

physical side, a *processor module* is a physical module that plugs into a system. And a *processor module* is expected to appear logically as a single agent on the system interconnect fabric.

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

rename registers A large set of hardware registers implemented by SPARC64 VI that are invisible to the programmer. Before instructions are *issued*, source and destination registers are mapped onto this set of rename registers. This allows instructions that normally would be blocked, waiting for an architecture register, to proceed in parallel. When instructions are *committed*, results in renamed registers are posted to the architects registers in the proper sequence to produce the correct program results.

reservation station A holding location that buffers dispatched instructions until all input operands are available. SPARC64 VI implements dataflow execution based on operand availability. When operands are available, the instructions in the reservation station are scheduled for execution. Reservation stations also contain special tag-matching logic that captures the appropriate operand data. Reservation stations are sometimes referred to as queues (for example, the integer queue).

scan A method used to initialize all of the machine state within a chip. In a chip that has been designed to be scannable, all of the machine state is connected in one or several loops called “scan rings.” Initialization data can be scanned into the chip through the scan rings. The state of the machine also can be scanned out through the scan rings.

sleeping Describes a thread that is suspended from operation. While sleeping, a thread is not issuing instructions for execution but still maintains cache coherency. Unlike *suspended*, a *sleeping* thread awakes automatically within limited cycles.

speculative A distribution system whereby a result is not guaranteed as known to be correct or an operand state is not known to be valid. SPARC64 VI employs speculative distribution, meaning results can be distributed from functional units before the point at which guaranteed validity of the result is known.

superscalar An implementation that allows several instructions to be issued, executed, and committed in one clock cycle. SPARC64 VI *issues* up to 4 instructions per clock cycle.

suspended Describes a thread that is suspended from operation. When suspended, a thread is not issuing instructions for execution but still maintains cache coherency. Unlike *sleeping*, a *suspended* thread does not awake automatically without certain events.

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

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

thread A term that identifies the hardware state used to hold a software thread in order to execute it. thread is specifically the software visible architecture state (PC, next PC, general purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.) of a thread and any micro architecture state required by hardware for its execution.

F.CHAPTER **3**

Architectural Overview

Please refer to Chapter 3 in **Commonality**.

F.CHAPTER **4**

Data Formats

Please refer to Chapter 4 in **Commonality**.

F.CHAPTER 5

Registers

The SPARC64 VI processor includes two types of registers: general-purpose—that is, working, data, control/status—and ASI registers.

The SPARC V9 architecture also defines two implementation-dependent registers: the IU Deferred-Trap Queue and the Floating-Point Deferred-Trap Queue (`FQ`); SPARC64 VI does not need or contain either queue. All processor traps caused by instruction execution are precise, and there are several disrupting traps caused by asynchronous events, such as interrupts, asynchronous error conditions, and `RED_state` entry traps.

For general information, please see parallel subsections of Chapter 5 in **Commonality**. For easier referencing, this chapter follows the organization of Chapter 5 in **Commonality**.

For information on MMU registers, please refer to Section F.10, *Internal Registers and ASI Operations*, on page 99.

The chapter contains these sections:

5.1 Nonprivileged Registers

Most of the definitions for the registers are as described in the corresponding sections of **Commonality**. Only SPARC64 VI-specific features are described in this section.

5.1.7 Floating-Point State Register (FSR)

Please refer to Section 5.1.7 of **Commonality** for the description of `FSR`.

The sections below describe SPARC64 VI-specific features of the `FSR` register.

FSR_nonstandard_fp (NS)

SPARC V9 defines the FSR.NS bit which, when set to 1, causes the FPU to produce implementation-dependent results that may not conform to IEEE Std 754-1985. SPARC64 VI implements this bit.

When FSR.NS = 1, denormal input operands and denormal results that would otherwise trap are flushed to 0 of the same sign and an inexact exception is signalled (that may be masked by FSR.TEM.NXM). See Section B.6, *Floating-Point Nonstandard Mode*, on page 69 for details.

When FSR.NS = 0, the normal IEEE Std 754-1985 behavior is implemented.

FSR_version (*ver*)

For each SPARC V9 IU implementation (as identified by its VER.impl field), there may be one or more FPU implementations or none. This field identifies the particular FPU implementation present. For the first SPARC64 VI, FSR.ver = 0 (impl. dep. #19); however, future versions of the architecture may set FSR.ver to other values. Consult the SPARC64 VI Data Sheet for the setting of FSR.ver for your chipset.

FSR_floating-point_trap_type (*ftt*)

The complete conditions under which SPARC64 VI triggers *fp_exception_other* with trap type *unfinished_FPop* is described in Section B.6, *Floating-Point Nonstandard Mode*, on page 69 (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.

In SPARC64 VI, the cexc bits are set according to the following pseudocode:

```
if (<LDFSR or LDXFSR commits>)
   <update using data from LDFSR or LDXFSR>;
else if (<FPop commits with ftt = 0>)
   <update using value from FPU>
else if (<FPop commits with IEEE_754_exception>)
   <set one bit in the CEXC field as supplied by FPU>;
else if (<FPop commits with unfinished_FPop error>)
   <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 VI follows case (1); that is, it implements all three fields in conformance with IEEE Std 754-1985. See FSR Conformance in Section 5.1.7 of **Commonality** for more information about other implementation methods.

5.1.9 Tick (TICK) Register

SPARC64 VI implements TICK.counter register as a 63-bit register (impl. dep. #105).

Implementation Note – On SPARC64 VI, the counter part of the value returned when the TICK register is read is the value of TICK.counter when the RDTICK instruction is *executed*. The difference between the counter values read from the TICK register on two reads reflects the number of processor cycles executed between the *executions* of the RDTICK instructions, not their *commits*. In longer code sequences, the difference between this value and the value that would have been obtained when the instructions are committed would have been small.

5.2 Privileged Registers

Please refer to Section 5.2 of **Commonality** for the description of privileged registers.

5.2.6 Trap State (TSTATE) Register

SPARC64 VI implements only bits 2:0 of the TSTATE.CWP field. Writes to bits 4 and 3 are ignored, and reads of these bits always return zeroes.

Note – Spurious setting of the PSTATE.RED bit by privileged software should not be performed, since it will take the SPARC64 VI into RED_state without the required sequencing.

5.2.9 Version (VER) Register

TABLE 5-1 shows the values for the VER register for SPARC64 VI.

TABLE 5-1 VER Register Encoding

Bits	Field	Value
63:48	manuf	0004_{16} (impl. dep. #104)
47:32	impl	6
31:24	mask	*n* (The value of n depends on the processor chip version)
15:8	maxtl	5
4:0	maxwin	7

The manuf field contains Fujitsu's 8-bit JEDEC code in the lower 8 bits and zeroes in the upper 8 bits. The manuf, impl, and mask fields are implemented so that they may change in future SPARC64 processor versions. 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 **Commonality** for details of the ASRs.

Performance Control Register (PCR) (ASR 16)

SPARC64 VI implements the PCR register as described in **Commonality**, with additional features as described in this section.

In SPARC64 VI, the accessibility of PCR when PSTATE.PRIV = 0 is determined by PCR.PRIV. If PSTATE.PRIV = 0 and PCR.PRIV = 1, an attempt to execute either RDPCR or WRPCR will cause a *privileged_action* exception. If PSTATE.PRIV = 0 and PCR.PRIV = 0, RDPCR operates without privilege violation and WRPCR causes a *privileged_action* exception only when an attempt is made to change (that is, write 1 to) PCR.PRIV (impl. dep. #250).

See Appendix Q, *Performance Instrumentation* for a detailed discussion of the PCR and PIC register usage and event count definitions.

The Performance Control Register in SPARC64 VI is illustrated in FIGURE 5-1 and described in TABLE 5-2.

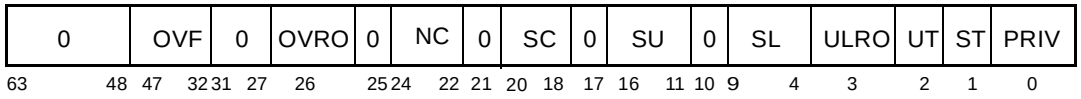


FIGURE 5-1 SPARC64 VI Performance Control Register (PCR) (ASR 16)

TABLE 5-2 PCR Bit Description

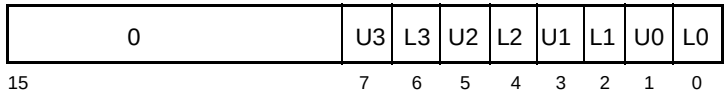
Bit	Field	Description
47:32	**OVF**	Overflow Clear/Set/Status. Used to read counter overflow status (via RDPCR) and clear or set counter overflow status bits (via WRPCR). PCR.OVF is a SPARC64 VI-specific field (impl. dep. #207). The following figure depicts the bit layout of SPARC64 VI OVF field for four counter pairs. Counter status bits are cleared on write of 0 to the appropriate OVF bit. 
26	**OVRO**	Overflow read-only. Write-only/read-as-zero field specifying PCR.OVF update behavior for WRPCR.PCR. The OVRO field is implementation -dependent (impl. dep. #207). WRPCR.PCR with PCR.OVRO = 1 inhibits updating of PCR.OVF for the current write only. The intention of PCR.OVRO is to write PCR while preserving current PCR.OVF value. PCR.OVF is maintained internally by hardware, so a subsequent RDPCR.PCR returns accurate overflow status at the time.
24:22	**NC**	Number of counter pairs. Three-bit, read-only field specifying the number of counter pairs, encoded as 0–7 for 1–8 counter pairs (impl. dep. #207). For SPARC64 VI, the hardcoded value of NC is 3 (indicating presence of 4 counter pairs).
20:18	**SC**	Select PIC. In SPARC64 VI, three-bit field specifying which counter pair is currently selected as PIC (ASR 17) and which SU/SL values are visible to software. On write, PCR.SC selects which counter pair is updated (unless PCR.ULRO is set; see below). On read, PCR.SC selects which counter pair is to be read through PIC (ASR 17).
16:11	**SU**	Defined (as S1) in **Commonality**.
9:4	**SL**	Defined (as S0) in **Commonality**.
3	**ULRO**	Implementation-dependent field (impl. dep. #207) that specifies whether SU/SL are read-only. In SPARC64 VI, this field is write-only/read-as-zero, specifying update behavior of SU/SL on write. When PCR.ULRO = 1, SU/SL are considered as read-only; the values set on PCR.SU/PCR.SL are not written into SU/SL. When PCR.ULRO = 0, SU/SL are updated. PCR.ULRO is intended to switch the visible PIC by writing PCR.SC, without affecting current selection of SU/SL of that PIC. On PCR read, PCR.SU/PCR.SL always shows the current setting of the PIC regardless of PCR.ULRO.
2	**UT**	Defined in **Commonality**.
1	**ST**	Defined in **Commonality**.

TABLE 5-2 PCR Bit Description *(Continued)*

Bit	Field	Description
0	**PRIV**	Defined in **Commonality**, with the additional function of controlling PCR accessibility as described above (impl. dep. #250).

Performance Instrumentation Counter (PIC) Register (ASR 17)

The PIC register is implemented as described in **Commonality**.

Four PICs are implemented in SPARC64 VI. Each is accessed through ASR 17, using PCR.SC as a select field. Read/write access to the PIC will access the PICU/PICL counter pair selected by PCR. For PICU/PICL encoding of specific event counters, see Appendix Q, *Performance Instrumentation*.

Counter Overflow. On overflow, counters wrap to 0, SOFTINT register bit 15 is set, 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 are generated simultaneously, then multiple overflow status bits will be set. If overflow status bits are already set, then they remain set on counter overflow.

Overflow status bits are cleared by software writing 0 to the appropriate bit of PCR.OVF and may be set by writing 1 to the appropriate bit. Setting these bits by software does not generate a level 15 interrupt.

Dispatch Control Register (DCR) (ASR 18)

The DCR is not implemented in SPARC64 VI. Zero is returned on read, and writes to the register are ignored. The DCR is a privileged register; attempted access by nonprivileged (user) code generates a *privileged_opcode* exception.

5.2.12 Registers Referenced Through ASIs

Data Cache Unit Control Register (DCUCR)

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

The Data Cache Unit Control Register contains fields that control several memory-related hardware functions. The functions include Instruction, Prefetch, write and data caches, MMUs, and watchpoint setting. SPARC64 VI implements most of DCUCUR's functions described in Section 5.2.12 of **Commonality**.

After a power-on reset (POR), all fields of DCUCR, including implementation-dependent fields, are set to 0. After a WDR, XIR, or SIR reset, all fields of DCUCR, including implementation-dependent fields, are set to 0.

The Data Cache Unit Control Register is illustrated in FIGURE 5-2 and described in TABLE 5-3. In the table, bits are grouped by function rather than by strict bit sequence.

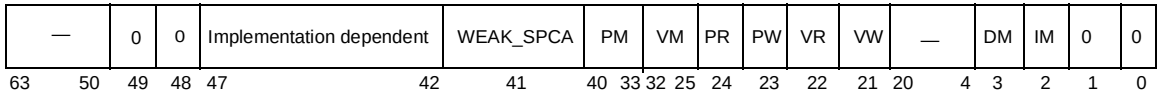


FIGURE 5-2 DCU Control Register Access Data Format (ASI 45_{16})

TABLE 5-3 DCUCR Description

Bits	Field	Type	Use — Description
49:48	**CP, CV**	RW	Not implemented in SPARC64 VI (impl. dep. #232). It reads as 0 and writes to it are ignored.
47:42	impl. dep.		Not used. It reads as 0 and writes to it are ignored.
41	**WEAK_SPCA**	RW	Disable speculative memory access (impl. dep. #240). Branch prediction mechanism is disabled and no load, store, and instruction fetch in the speculative path is issued. Due to absence of branch prediction, all CTI instruction is considered as not taken, and subsequent instructions beyond CTI are being fetched. Instruction fetch is eventually stopped by internal resource limitation, so the memory area being accessed beyond CTI is predictable. L2 cache flush by supervisor software is always executed regardless of DCUCR.WEAK_SPCA setting. Autonomous L2 cache flush by RAS is pending until all DCUCR.WEAK_SPCA in a CPU module is set to 0. Although DCUCR is per thread resource, some resources such as branch history table are shared two threads in a core, DCUCR.WEAK_SPCA setting in one thread may affect to the other thread's behavior.
40:33	**PM<7:0>**		Defined in **Commonality**.
32:25	**VM<7:0>**		Defined in **Commonality**.
24, 23	**PR, PW**		Defined in **Commonality**.
22, 21	**VR, VW**		Defined in **Commonality**.
20:4	**—**		*Reserved.*
3	**DM**		Defined in **Commonality**.
2	**IM**		Defined in **Commonality**.
1	**DC**	RW	Not implemented in SPARC64 VI (impl. dep. #252). It reads as 0 and writes to it are ignored.
0	**IC**	RW	Not implemented in SPARC64 VI (impl. dep. #253). It reads as 0 and writes to it are ignored.

Implementation Note – When DCUCR.WEAK_SPCA = 1, the memory area being accessed beyond CTI does not exceed 1KB of that CTI.

Implementation Note – In SPARC64 VI, a thread with DCUCR.WEAK_SPCA = 1 causes branch history table to be unusable on all threads in a core. This is because branch history table is shared by two threads and it is cleared by setting DCUCR.WEAK_SPCA = 1, which takes certain amount of cycles, usually longer cycle than a thread to run without thread switching.

Programming Note – Supervisor software should issue membar #Sync immediately after setting DCUCR.WEAK_SPCA = 1, to make sure no speculative memory access is issued thereafter.

Programming Note – Changing IM(IMMU enable) and DM(DMMU Enable) in DCUCR requires the following instruction sequence for SPARC64 VI to work correctly.

```
# DCUCR.IM update
stxa DCUCR
flush

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

Data Watchpoint Registers

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

SPARC64 VI employs a conservative check of the PA/VA watchpoint for partial store instructions. See Section A.42, *Partial Store (VIS I)*, on page 65 for details.

Instruction Trap Register

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

In SPARC64 VI, the least significant 11 bits (bits 10:0) of a `CALL` or branch (`BPcc`, `FBPfcc`, `Bicc`, `BPr`) instruction in the instruction cache are identical to their architectural encoding (as it appears in main memory) (impl. dep. #245).

5.2.13 Floating-Point Deferred-Trap Queue (FQ)

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

5.2.14 IU Deferred-Trap Queue

SPARC64 VI neither has nor needs an IU deferred-trap queue (impl. dep. #16)

F.CHAPTER **6**

Instructions

This chapter presents SPARC64 VI implementation-specific instruction details and the processor pipeline information in these subsections:

For additional, general information, please see parallel subsections of Chapter 6 in **Commonality**. For easy referencing, we follow the organization of Chapter 6 in **Commonality**.

6.1 Instruction Execution

SPARC64 VI is an advanced superscalar implementation of SPARC V9. Several instructions may be issued and executed in parallel. Although SPARC64 VI provides serial program execution semantics, some of the implementation characteristics described below are part of the architecture visible to software for correctness and efficiency.

6.1.1 Data Prefetch

SPARC64 VI employs speculative (out of program order) execution of instructions; in most cases, the effect of these instructions can be undone if the speculation proves to be incorrect.[1] However, exceptions can occur because of speculative data prefetching. Formally, SPARC64 VI employs the following rules regarding speculative prefetching:

1. An *async_data_error* may be signalled during speculative data prefetching.

1. If a memory operation *y* resolves to a volatile memory address (*location[y]*), SPARC64 VI will not speculatively prefetch *location[y]* for any reason; *location[y]* will be fetched or stored to only when operation *y* is *committable*.

2. If a memory operation *y* resolves to a nonvolatile memory address (*location[y]*), SPARC64 VI *may* speculatively prefetch *location[y]* subject, adhering to the following sub-rules:

   a. If an operation *y* can be speculatively prefetched according to the prior rule, operations with store semantics are speculatively prefetched for ownership only if they are prefetched to cacheable locations. Operations without store semantics are speculatively prefetched even if they are noncacheable as long as they are not volatile.

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

SPARC64 VI provides two mechanisms to avoid speculative execution of a load:

1. Avoid speculation by disallowing speculative accesses to certain memory pages or I/O spaces. This can be done by setting the `E` (side-effect) bit in the `PTE` for all memory pages that should not allow speculation. All accesses made to memory pages that have the `E` bit set in their `PTE` will be delayed until they are no longer speculative or until they are cancelled**.** See Appendix F, *Memory Management Unit*, for details.

2. Alternate space load instructions that force program order, such as `ASI_PHYS_BYPASS_WITH_EBIT[_L]` (AS I = 15_{16}, $1D_{16}$), will not be speculatively executed.

6.1.2 Instruction Prefetch

The processor prefetches instructions to minimize cases where the processor must wait for instruction fetch. In combination with branch prediction, prefetching may cause the processor to access instructions that are not subsequently executed. In some cases, the speculative instruction accesses will reference data pages. SPARC64 VI does not generate a trap for any exception that is caused by an instruction fetch until all of the instructions before it (in program order) have been committed.[1]

6.1.3 Syncing Instructions

SPARC64 VI has instructions called *syncing instructions*, that stop execution for the number of cycles it takes to clear the pipeline and to synchronize the processor. There are two types of synchronization, *pre* and *post*. A presyncing instruction waits for all previous instructions

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

to commit, commits by itself, and then issues successive instructions. A postsyncing instruction issues by itself and prevents the successive instructions from issuing until it is committed. Some instructions have both pre- and postsync attributes.

In SPARC64 VI almost all instructions commit in order, but store instruction commit before becoming globally visible. A few syncing instructions cause the processor to discard prefetched instructions and to refetch the successive instructions.

6.2 Instruction Formats and Fields

Instructions are encoded in five major 32-bit formats and several minor formats. Please refer to Section 6.2 of **Commonality** for illustrations of four major formats. FIGURE 6-1 illustrates Format 5, unique to SPARC64 VI.

Format 5 (op = 2, op3 = 37_{16}): `FMADD`, `FMSUB`, `FNMADD`, and `FNMSUB` *(in place of* `IMPDEP2B`)

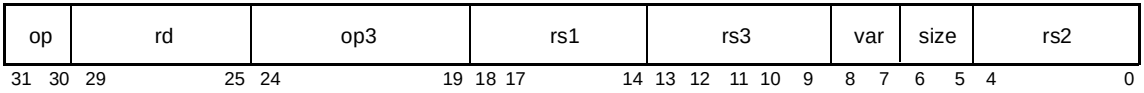


FIGURE 6-1 Summary of Instruction Formats: Format 5

Instruction fields are those shown in Section 6.2 of **Commonality**. Three additional fields are implemented in SPARC64 VI. They are described in TABLE 6-1.

TABLE 6-1 Instruction Fields Specific to SPARC64 VI

Bits	Field	Description
13:9	**`rs3`**	This 5-bit field is the address of the third `f` register source operand for the floating-point multiply-add and multiply-subtract instruction.
8.7	**`var`**	This 2-bit field specifies which specific operation (variation) to perform for the floating-point multiply-add and multiply-subtract instructions
6.5	**`size`**	This 2-bit field specifies the size of the operands for the floating-point multiply-add and multiply-subtract instructions.

Since `size` = 00 is not `IMPDEP2B` and since `size` = 11 assumes quad operations but is not implemented in SPARC64 VI, the instruction with `size` = 00 or 11 generates an *illegal_instruction* exception in SPARC64 VI.

6.3 Instruction Categories

SPARC V9 instructions comprise the categories listed below. All categories are described in Section 6.3 of **Commonality**. Subsections in bold face are SPARC64 VI implementation dependencies.

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

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`)

Instructions other than `CALL` and `JMPL` are described in their entirety in Section 6.3.2 of **Commonality**. SPARC64 VI implements `CALL` and `JMPL` as described below.

CALL and JMPL Instructions

SPARC64 VI writes all 64 bits of the `PC` into the destination register when `PSTATE.AM` = 0. The upper 32 bits of `r[15]` (`CALL`) or of `r[rd]` (`JMPL`) are written as zeroes when `PSTATE.AM` = 1 (impl. dep. #125).

SPARC64 VI implements `JMPL` and `CALL` return prediction hardware in a form of special stack, called the Return Address Stack (RAS). Whenever a `CALL` or `JMPL` that writes to `%o7` (`r[15]`) occurs, SPARC64 VI "pushes" the return address (PC+8) onto the RAS. When either of the synthetic instructions *retl* (`JMPL [%o7+8]`) and *ret* (`JMPL [%i7+8]`) are subsequently executed, the return address is predicted to be the address stored on the top of

the RAS and the RAS is “popped.” If the prediction in the RAS is incorrect, SPARC64 VI backs up and starts issuing instructions from the correct target address. This backup takes a few extra cycles.

Programming Note – For maximum performance, software and compilers must take into account how the RAS works. For example, tricks that do nonstandard returns in hopes of boosting performance may require more cycles if they cause the wrong RAS value to be used for predicting the address of the return. Heavily nested calls can also cause earlier entries in the RAS to be overwritten by newer entries, since the RAS only has a limited number of entries. Eventually, some return addresses will be mis-predicted because of the overflow of the RAS.

6.3.7 Floating-Point Operate (FPop) Instructions

The complete conditions of generating an *fp_exception_other* exception with `FSR.ftt` = *unfinished_FPop* are described in Section B.6, *Floating-Point Nonstandard Mode*, on page 69.

The SPARC64 VI-specific `FMADD` and `FMSUB` instructions (described below) are also floating-point operations. They require the floating-point unit to be enabled; otherwise, an *fp_disabled* trap is generated. They also affect the `FSR`, like FPop instructions. However, these instructions are not included in the FPop category and, hence, reserved encodings in these opcodes generate an *illegal_instruction* exception, as defined in Section 6.3.9 of **Commonality**.

6.3.8 Implementation-Dependent Instructions

SPARC64 VI uses the `IMPDEP2` instruction to implement the Floating-Point Multiply-Add/Subtract and Negative Multiply-Add/Subtract instructions; these have an `op3` field = 37_{16} (`IMPDEP2`). See *Floating-Point Multiply-Add/Subtract* on page 55 for full definitions of these instructions. Opcode space is reserved in `IMPDEP2` for the quad-precision forms of these instructions. However, SPARC64 VI does not currently implement the quad-precision forms, and the processor generates an *illegal_instruction* exception if a quad-precision form is specified. Since these instructions are not part of the required SPARC V9 architecture, the operating system does not supply software emulation routines for the quad versions of these instructions.

SPARC64 VI uses the `IMPDEP1` instruction to implement the graphics acceleration instructions.

6.4 Processor Pipeline

The pipeline of SPARC64 VI consists of fifteen stages, shown in FIGURE 6-2. Each stage is referenced by one or two letters as follows:

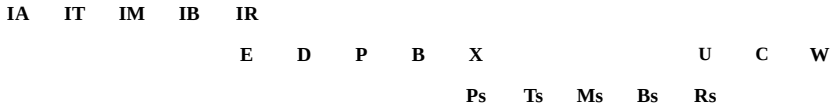


FIGURE 6-2 SPARC64 VI pipeline stages

6.4.1 Instruction Fetch Stages

- IA: Instruction Address generation
- IT: Instruction TLB Tag access
- IM: Instruction TLB tag Match
- IB: Instruction cache read to Buffer
- IR: Instruction read Result

IA through IR stages are dedicated to instruction fetch. These stages work in concert with the cache access unit to supply instructions to subsequent stages. The instructions fetched from memory or cache are stored in the Instruction Buffer (I-buffer).

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

Instruction fetch stages are designed so that they work independently of subsequent stages as much as possible. And they can fetch instructions even when execution stages stall. These stages fetch until the Instruction Buffer I-Buffer is full; further fetches are possible by requesting prefetches to the L1 cache.

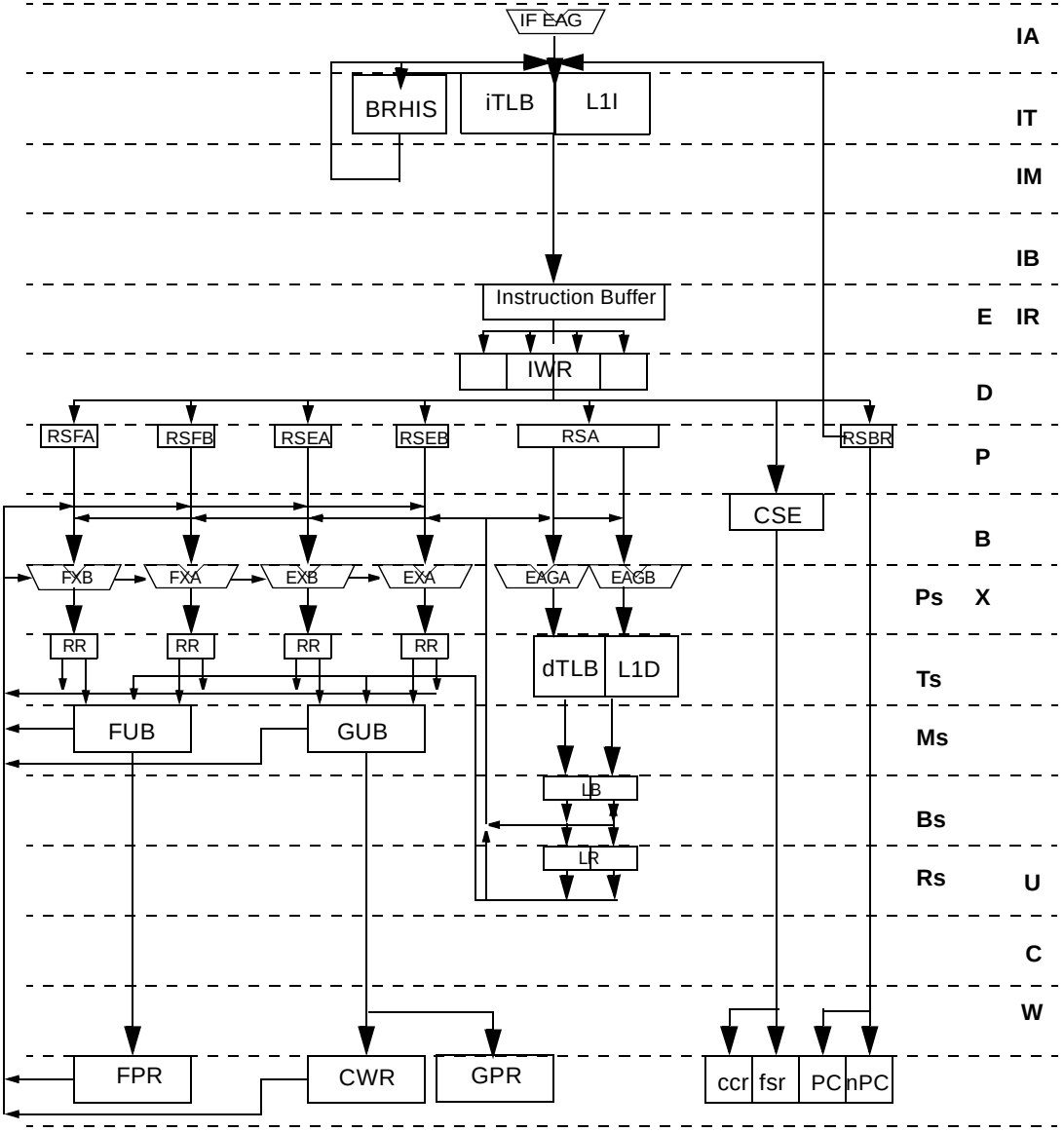


FIGURE 6-3 SPARC64 VI Pipeline Diagram

6.4.2 Issue Stages

- E: Entry
- D: Decode

SPARC64 VI is an out-of-order execution CPU. It has six execution units (two arithmetic and logic units, two floating-point units, two load/store units). Each unit except the load/store unit has its own reservation station. E and D stages are issue stages that decode instructions and dispatch them to the target RS. SPARC64 VI can issue up to four instructions per cycle.

The resources needed to execute an instruction are assigned in the issue stages. The resources to be allocated include the following:

- Commit stack entry (CSE)
- Renaming registers of integer (GUB) and floating-point (FUB)
- Entries of reservations stations
- Memory access ports

Resources needed for an instruction are specific to the instruction, but all resources must be assigned at these stages. In normal execution, assigned resources are released at the very last stage of the pipeline, 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 used by them are released immediately. This behavior enables the decoder to restart issuing instructions as quickly as possible.

6.4.3 Execution Stages

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

Instructions in reservation stations will be executed when certain conditions are met, for example, the values of source registers are known, the execution unit is available. Execution latency varies from one to many, depending on the instruction.

1. An entry in a reservation station is released at the X-stage.

Execution Stages for Cache Access

Memory access requests are passed to the cache access pipeline 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 6.4.1, *Instruction Fetch Stages*, for details. Stages in instruction fetch and cache access correspond as follows:

Instruction Fetch Stages	Cache Access
IA	**Ps**
IT	**Ts**
IM	**Ms**
IB	**Bs**
IR	**Rs**

When an exception is signalled, fetch ports and store ports used by memory access instructions are released. The cache access pipeline itself remains working in order to complete outgoing memory accesses. When data is returned, it is then stored to the cache.

6.4.4 Completion Stages

- W: Write
- After an out-of-order execution, execution reverts to program order to complete. Exception handling is done in the completion stages. Exceptions occurring in execution stages are not handled immediately but are signalled when the instruction is completed.[1]

1. RAS-related exception may be signalled before completion.

F.CHAPTER **7**

Traps

Please refer to Chapter 7 of **Commonality**. Section numbers in this chapter correspond to those in Chapter 7 of **Commonality**.

This chapter adds SPARC64 VI-specific information in the following sections:

7.1 Processor States, Normal and Special Traps

Please refer to Section 7.1 of **Commonality**.

7.1.1 RED_state

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 VI this virtual address is FFFF FFFF F000 0000_{16}, which translates to physical address 0000 07FF F000 0000_{16} in RED_state (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 VI has the following implementation-dependent behavior in RED_state (impl. dep. #115):

- While in RED_state, all internal ITLB-based translation functions are disabled. DTLB-based translations are disabled upon entry but may be reenabled by software while in RED_state. However, ASI-based access functions to the TLBs are still available.
- While mTLBs and uTLBs are disabled, all accesses are assumed to be noncacheable and strongly ordered for data access.
- XIR errors are not masked and can cause a trap.

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).

Although the standard behavior of the CPU upon an entry into error_state is to internally generate a *watchdog_reset* (WDR), the CPU optionally stays halted upon an entry to error_state depending on a setting in the OPSR register (impl. dep #40, #254).

7.2 Trap Categories

Please refer to Section 7.2 of **Commonality**.

An exception or interrupt request can cause any of the following trap types:

- Precise trap
- Deferred trap
- Disrupting trap
- Reset trap

7.2.2 Deferred Traps

Please refer to Section 7.2.2 of **Commonality**.

SPARC64 VI implements a deferred trap to signal certain error conditions (impl. dep. #32). Please refer to the description of `I_UGE` error on "Relation between `%tpc` and the instruction that caused the error" row in TABLE P-2 on page 161 for details. See also *Instruction End-Method at ADE Trap* on page 176.

7.2.4 Reset Traps

Please refer to Section 7.2.4 of **Commonality**.

In SPARC64 VI, a watchdog reset (WDR) occurs when the processor has not committed an instruction for 2^{33} processor clocks.

7.2.5 Uses of the Trap Categories

Please refer to Section 7.2.5 of **Commonality**.

All exceptions that occur as the result of program execution are precise in SPARC64 VI (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 exception is precise in SPARC64 VI.

7.3 Trap Control

Please refer to Section 7.3 of **Commonality**.

7.3.1 PIL Control

SPARC64 VI receives external interrupts from Jupiter Bus. They cause an *interrupt_vector_trap* (TT = 60_{16}). The interrupt vector trap handler reads the interrupt information and then schedules SPARC V9-compatible interrupts by writing bits in the SOFTINT register. Please refer to Section 5.2.11 of **Commonality** for details.

During handling of SPARC V9-compatible interrupts by SPARC64 VI, the PIL register is checked. If an interrupt has sufficient priority, SPARC64 VI will stop issuing new instructions, will flush all uncommitted instructions, and then will vector to the trap handler. The only exception to this process occurs when SPARC64 VI is processing a higher-priority trap.

SPARC64 VI takes a normal disrupting trap upon receipt of an interrupt request.

7.4 Trap-Table Entry Addresses

Please refer to Section 7.4 of **Commonality**.

7.4.2 Trap Type (TT)

Please refer to Section 7.4.2 of **Commonality**.

SPARC64 VI implements all mandatory SPARC V9 and SPARC JPS1 exceptions, as described in Chapter 7 of **Commonality**, plus the exception listed in TABLE 7-1, which is specific to SPARC64 VI (impl. dep. #35; impl. dep. #36).

TABLE 7-1 Exceptions Specific to SPARC64 VI

Exception or Interrupt Request	TT	Priority
async_data_error	040_{16}	2

7.4.4 Details of Supported Traps

Please refer to Section 7.4.4 in **Commonality**.

SPARC64 VI Implementation-Specific Traps

SPARC64 VI supports the following implementation-specific trap type:

- *async_data_error*

7.5 Trap Processing

Please refer to Section 7.5 of **Commonality**.

7.6 Exception and Interrupt Descriptions

Please refer to Section 7.6 of **Commonality**.

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

Please refer to Section 7.6.4 of **Commonality**.

SPARC64 VI implements all six traps that are implementation dependent in SPARC V9 but mandatory in JPSI (impl. dep. #35). See Section 7.6.4 of **Commonality** for details.

7.6.5 SPARC JPS1 Implementation-Dependent Traps

Please refer to Section 7.6.5 of **Commonality**.

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

- ***async_data_error*** [`tt` = 040_{16}] (Preemptive or disrupting) (impl. dep. #218) — SPARC64 VI implements the *async_data_error* exception to signal the following errors.
  - Uncorrectable errors in the internal architecture registers (general registers–gr, floating-point registers–fr, ASR, ASI registers)

- Uncorrectable errors in the core pipeline
- Watch dog time-out first time
- TLB access error upon access by an `ldxa` or `stxa` instruction

Multiple errors may be reported in a single generation of the *async_data_error* exception. Depending on the situation, the *async_data_error* trap becomes a precise trap, a disrupting trap, or a preemptive trap upon error detection. The `TPC` and `TNPC` stacked by the exception may indicate the exact instruction, the preceding instruction, or the subsequent instruction inducing the error. See Appendix for details of the *async_data_error* exception in SPARC64 VI.

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 **Commonality** and defined in Appendix D, *Formal Specification of the Memory Models*, also in **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 VI behaves in a manner that guarantees adherence to whichever memory model is currently in effect**.**

This chapter describes the following major SPARC64 VI-specific details of memory models.

For general information, please see parallel subsections of Chapter 8 in **Commonality**. For easier referencing, this chapter follows the organization of Chapter 8 in **Commonality**, listing subsections whether or not there are implementation-specific details.

8.1 Overview

Note – The words *"hardware memory model"* denote the underlying hardware memory models as differentiated from the "SPARC *V9 memory model*," which is the memory model the programmer selects in `PSTATE.MM`.

SPARC64 VI supports only one mode of memory handling to guarantee correct operation under any of the three SPARC V9 memory ordering models (impl. dep. #113):

- **Total Store Order** — All loads are ordered with respect to loads, and all stores are ordered with respect to loads and stores. This behavior is a superset of the requirements for the SPARC V9 memory models TSO, PSO, and RMO. When `PSTATE.MM` selects PSO or RMO, SPARC64 VI operates in this mode. Since programs written for PSO (or RMO) will always work if run under Total Store Order, this behavior is safe but does not take advantage of the reduced restrictions of PSO (or RMO).

8.4 SPARC V9 Memory Model

Please refer to Section 8.4 of **Commonality**.

In addition, this section describes SPARC64 VI-specific details about the processor/memory interface model.

8.4.5 Mode Control

SPARC64 VI implements Total Store Ordering for all `PSTATE.MM`. Writing 11_2 into `PSTATE.MM` also causes the machine to use TSO (impl. dep. #119). However, the encoding 11_2 should not be used, since future version of SPARC64 VI may use this encoding for a new memory model.

8.4.6 Synchronizing Instruction and Data Memory

All caches in a SPARC64 VI-based system (uniprocessor or multiprocessor) have a unified cache consistency protocol and implement strong coherence between instruction and data caches. Writes to any data cache cause invalidations to the corresponding locations in all

instruction caches; references to any instruction cache cause corresponding modified data to be flushed and corresponding unmodified data to be invalidated from all data caches. The flush operation is still operative in SPARC64 VI, however.

F.CHAPTER **9**

Multi-Threaded Processing

SPARC64 VI can process two threads in each of the two cores in the same processor module to provide a dense, high throughput system. This chapter specifies a required interface between hardware and software to handle multiple threads on the same processor module.

9.1 MTP structure

9.1.1 General MTP structure

Three structures are known for Multi threaded Processor.

1. Chip Multi Processing

   One processor module includes multiple physical cores, where each physical core is able to run a single thread independently from other cores at any given time. This structure is called Chip Multi-Processing (CMP).

2. Multi-thread (MT)

   One processor module includes a single physical core. The core is able to run multiple threads in parallel from the software's point of view. Although there is only a single physical core, the physical core behaves as if it were multiple virtual processors. This is because the core includes multiple software visible resources (PC, next PC, general purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.). This virtual processor is called a thread.

   There are two types of Multi-thread implementations.

   a. Vertical Multi-thread (VMT)

The physical core is able to run only a single thread at any given time. But multiple threads can run in parallel from the software's point of view by using time-sharing technique. That is, the core includes multiple software visible resources (PC, next PC, general purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.), and hardware switches threads to run in relatively-short time.

b. Simultaneous Multi-thread (SMT)

The physical core is able to run multiple threads at any given time. That is, the core includes multiple software visible resources (PC, next PC, general purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.) as well as multiple execution units, and multiple threads run at the same time.

9.1.2 MTP structure of SPARC64 VI

SPARC64 VI implements a combination of CMP and VMT. That is, it has two physical cores where each core has two threads with VMT structure. In other words, four threads are able to run in parallel. The two threads which belong to the same physical core share most of the physical resources, while the two physical cores do not share physical resources except L2 cache and system interface.

Major hardware triggers to switch threads for SPARC64 VI are:

- *Interrupt*
- *Software-invisible hardware triggers to increase the MT efficiency (L2 Cache miss, timer, ...)*

Also it is possible for software to switch threads explicitly as described in *How to control threads* on page 47.

How to execute multiple threads in parallel on SPARC64 VI is illustrated in FIGURE 9-1.

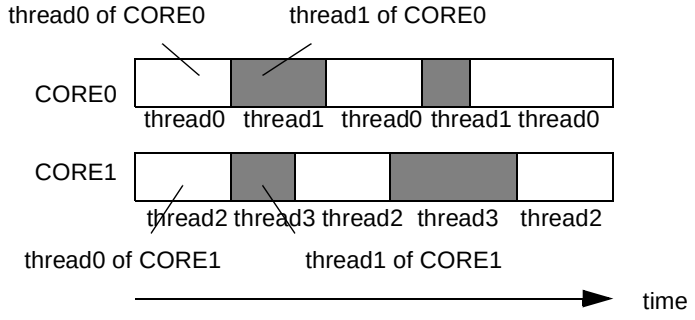


FIGURE 9-1 Multiple threads in SPARC64 VI

9.2 MTP Programming Model

9.2.1 Thread independency

In principle, because the software visible resources are not shared between threads, each thread of SPARC64 VI is independent of each other like a conventional Symmetric Multi Processor. Even for supervisor software, this true except the followings:

Shared TLBs

Thread0 and thread1 belongs to the same physical core share fTLB and sTLB logically. See *Translation Lookaside Buffer Hardware* on page 117 for details.

Error handling

An error asynchronous to thread execution is always signalled to all related threads. See *Error Classes and Signalling* on page 153 for details.

Performance

Since each thread has its own software visible resources, they are independent of each other from the programming model point of view. But this is not true for performance. Since threads belonging to the same physical core share most of physical resources, it is highly recommended for the OS to schedule threads in the following manner.

- Run threads belonging to the same process space on thread0 and thread1
- Suspend thread1 to run a single threaded program

Note – Since threads belonging to different physical cores share none of physical resources except the L2 cache and the system interface, it is not required to pay special attention to them.

9.2.2 How to control threads

There are special instructions for switching threads. Threads may be relegated to a suspended or sleep state to halt their execution. See *Suspend* on page 58 and *Sleep* on page 59 for details.

9.2.3 Shared registers between threads

The following ASR and ASI registers are shared among all the threads within a processor module.

- ASI_L2_DIAG_TAG_READ_REG
- ASI_SERIAL_ID

F.APPENDIX **A**

Instruction Definitions

This appendix describes the SPARC64 VI-specific implementation of the instructions in Appendix A of **Commonality**. If an instruction is not described in this appendix, then no SPARC64 VI implementation-dependency applies.

- See TABLE A-1 of **Commonality** for the location at which general information about the instruction can be found.
- Section numbers refer to the parallel section numbers in Appendix A of **Commonality**.

TABLE A-1 lists four instructions that are unique to SPARC64 VI.

TABLE A-1 Implementation-Specific Instructions

Operation	Name	Page
FMADD(s,d)	Floating-point multiply add	55
FMSUB(s,d)	Floating-point multiply subtract	55
FNMADD(s,d)	Floating-point multiply negate add	55
FNMSUB(s,d)	Floating-point multiply negate subtract	55
POPC	Population Count	66
SUSPEND	Suspend a thread	58
SLEEP	Put a thread to sleep	59

Each instruction definition consists of these parts:

1. A table of the opcodes defined in the subsection with the values of the field(s) that uniquely identify the instruction(s).
2. An illustration of the applicable instruction format(s). In these illustrations a dash (—) indicates that the field is *reserved* for future versions of the architecture and shall be 0 in any instance of the instruction. If a conforming SPARC V9 implementation encounters nonzero values in these fields, its behavior is undefined.
3. A list of the suggested assembly language syntax, as described in Appendix , *Assembly Language Syntax*.

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). 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.

   Also, any instruction that is not implemented in hardware shall generate an *illegal_instruction* exception (or *fp_exception_other* exception with `ftt` = *unimplemented_FPop* for floating-point instructions) when it is executed.

   The *illegal_instruction* trap can occur during chip debug on any instruction that has been programmed into the processor's `IIU_INST_TRAP` (ASI = 60_{16}, VA = 0). These traps are also not listed under each instruction.

   The following traps *never* occur in SPARC64 VI:

   - *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 include any timing information (in either cycles or clock time).

The following SPARC64 VI-specific extensions are described.

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

The following notes summarize behavior of block load/store instructions in SPARC64 VI.

1. Block load and store operations are not atomic, in that they are internally decomposed into eight independent, 8-byte load/store operations in SPARC64 VI. Each load/store is always issued and performed in the RMO memory model and obeys all prior MEMBAR and atomic instruction-imposed ordering constraints.

2. Block load/store instructions are out of the scope of V9 memory models, meaning that self-consistency of memory reference instruction is not always maintained if block load/store instructions are involved in the execution flow. The following table describes the implemented ordering constraints for block load/store instructions with respect to the other memory reference instructions with an operand address conflict in SPARC64 VI:

Program Order for conflicting bld/bst/ld/st		Ordered/
first	**next**	**Out-of-Order**
store	blockstore	Ordered
store	blockload	Ordered
load	blockstore	Ordered
load	blockload	Ordered
blockstore	store	Out-of-Order
blockstore	load	Out-of-Order
blockstore	blockstore	Out-of-Order
blockstore	blockload	Out-of-Order
blockload	store	Ordered
blockload	load	Ordered
blockload	blockstore	Ordered
blockload	blockload	Ordered

   To maintain the memory ordering even for the memory address conflicts, MEMBAR instructions shall be inserted into appropriate locations in the program.

   Although self-consistency with respect to the block load/store and the other memory reference instructions is not maintained in some cases, register conflicts between the other instructions and block load/store instructions are maintained in SPARC64 VI. The read-after-write, write-after-read, and write-after-write obstructions between a block load/store instruction and the other arithmetic instructions are detected and handled appropriately.

3. Block load instructions operate on the cache if the operand is present.

4. The block store with commit instruction always stores the operand in main storage and invalidates the line in the L1D and L2 cache if it is present.

5. The block store instruction stores the operand into main storage if it is not present in the L1D and the status of the line is invalid, shared, or owned. In case the line is not present in the L1D cache and is exclusive or modified on the L2 cache, the block store instruction modifies only the line in L2 cache. If the line is present in the L1D and the status is either clean/shared or clean/owned, the line is stored in main storage. If the line is present in the L1D and the status is clean/exclusive, the line in the L1D is invalidated and the operand is stored in the L2 cache. If the line is in the L1D and the status is modified/modified or clean/modified, the operand is stored in the L1D or L2 with L1D invalidation, respectively. The following table summarizes each cache status before block store and the results of the block store. Blank cells mean that no action occurred in the corresponding cache or memory, and the data, if it exists, is unchanged.

	Storage	**Status**				
Cache status before bst	L1	Invalid		Valid		
	L2	E, M	I, S, O	E	M	S, O
Action	L1	—	—	invalidate	update/ invalidate	—
	L2	update	—	update	—/update	—
	Memory	—	update	—	—	update

Exceptions

fp_disabled
PA_watchpoint
VA_watchpoint
illegal_instruction (misaligned rd)
mem_address_not_aligned (see *Block Load and Store ASIs* on page 128)
data_access_exception (see *Block Load and Store ASIs* on page 128)
LDDF_mem_address_not_aligned (see *Block Load and Store ASIs* on page 128)
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.12 Call and Link

SPARC64 VI clears the upper 32 bits of the PC value in r[15] when PSTATE.AM is set (impl. dep. #125). The value written into r[15] is visible to the instruction in the delay slot.

SPARC64 VI has a special hardware table, called Return Address Stack, to predict the return address from a subroutine. Though the return prediction stack achieves better performance in normal cases, there is a special use of the CALL instruction (call .+8) that may have an undesirable effect on the return address stack. In this case, the CALL instruction is used to read the PC contents, not to call a subroutine. In SPARC64 VI, the return address of the CALL (PC + 8) is not stored in its return address stack, to avoid a detrimental performance effect. When a ret or retl is executed, the value in the return address stack is used to predict the return address.

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 encoding, and which (if any) exceptions they may cause.

SPARC64 VI uses IMPDEP1 to encode VIS, SUSPEND, and SLEEP instructions (impl. dep. #106).

SPARC64 VI uses IMPDEP2B to encode the Floating-Point Multiply Add/Subtract instructions (impl. dep. #106). See Section A.24.1, *Floating-Point Multiply-Add/Subtract*, on page 55 for details.

See I.1.2, *Implementation-Dependent and Reserved Opcodes*, in **Commonality** for information about extending the SPARC V9 instruction set by means of the implementation-dependent instructions.

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

Exceptions implementation-dependent (IMPDEP2)

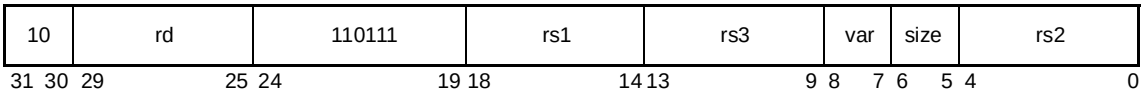
A.24.1 Floating-Point Multiply-Add/Subtract

SPARC64 VI uses IMPDEP2B opcode space to encode the Floating-Point Multiply Add/ Subtract instructions.

Opcode	Variation	Size†	Operation
FMADDs	00	01	Multiply-Add Single
FMADDd	00	10	Multiply-Add Double
FMSUBs	01	01	Multiply-Subtract Single
FMSUBd	01	10	Multiply-Subtract Double
FNMADDs	11	01	Negative Multiply-Add Single
FNMADDd	11	10	Negative Multiply-Add Double
FNMSUBs	10	01	Negative Multiply-Subtract Single
FNMSUBd	10	10	Negative Multiply-Subtract Double

† 11 is reserved for quad.

Format (5)



Operation	Implementation
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 Floating-point Multiply-Add instructions multiply the registers specified by the `rs1` field times the registers specified by the `rs2` field, add that product to the registers specified by the `rs3` field, then write the result into the registers specified by the `rd` field.

The Floating-point Multiply-Subtract instructions multiply the registers specified by the `rs1` field times the registers specified by the `rs2` field, subtract from that product the registers specified by the `rs3` field, and then write the result into the registers specified by the `rd` field.

The Floating-point Negative Multiply-Add instructions multiply the registers specified by the `rs1` field times the registers specified by the `rs2` field, *negate* the product, *subtract* from that negated value the registers specified by the *rs3* field, and then write the result into the registers specified by the `rd` field.

The Floating-point Negative Multiply-Subtract instructions multiply the registers specified by the `rs1` field times the registers specified by the `rs2` field, *negate* the product, *add* that negated product to the registers specified by the `rs3` field, and then write the result into the registers specified by the `rd` field.

The instruction is treated as fused multiply and add/subtract operations on SPARC64 VI. That is, a multiply operation is first performed with infinite precision without rounding step, and then an add/subtract operation is performed with a complete rounding step. Consequently, at most one rounding error can be incurred.

Programming Note – SPARC64 V treats the instruction as separate multiply and add/subtract operations. That is, a multiply operation is first performed with a complete rounding step (as if it were a single multiply operation), and then an add/subtract operation is performed with a complete rounding step (as if it were a single add/subtract operation). Consequently, at most two rounding errors can be incurred.
Also `fnmadd` and `fnmsub` behavior with `rs1=NaN` or `rs2=NaN` is different between SPARC64 V and SPARC64 VI. SPARC64 VI outputs one of the `NaN` inputs as it is, while SPARC64 V outputs the one with the sign bit inverted.

The behavior of SPARC64 VI in handling traps in Floating-point Multiply-Add/Subtract instructions is described in TABLE A-2. If a trapping *invalid* exception or a denormal source operand with `FSR.NS=1` is detected in the multiply part in the process of a Floating-point Multiply-Add/Subtract instruction, the execution of the instruction is aborted, the exception condition is recorded in `FSR.cexc`, the `aexc` is not modified, and the CPU traps with the exception condition. The add/subtract part of the instruction is only performed when the multiply-part of the instruction does not have a trapping *invalid* exception.

If there are trapping IEEE754 exception conditions in the add/subtract part, only the trapping exception condition is recorded in the `cexc`, and the `aexc` is not modified. If there are no trapping IEEE754 exception conditions, nontrapping exception condition of the add/subtract part is written into the `cexc` and the `cexc` is accumulated into the `aexc`. The boundary

conditions of an *unfinished_FPop* trap for Floating-point Multiply-Add/Subtract instructions are the same as the FMUL boundary conditions for the source operand 1 and 2, and the same as the FADD ones for the source operand 3 and the destination.

TABLE A-2 IEEE754 Exceptions in Floating-Point Multiply-Add/Subtract Instructions

FMUL	IEEE754 trap (*inv* or *nx* only)	No trap	No trap
FADD	—	IEEE754 trap	No trap
cexc	Exception condition of FMUL	Exception condition of FADD	Nontrapping exception conditions of FADD
aexc	No change	No change	Logical OR of the cexc (above) and the aexc

Detailed contents of cexc depending on the various conditions are described in TABLE A-3 and TABLE A-4. The following terminology is used: uf, of, inv, and nx are nontrapping IEEE exception conditions—underflow, overflow, invalid operation, and inexact, respectively.

TABLE A-3 Non-Trapping cexc When FSR.NS = 0

		FADD		
	none	nx	of nx	inv
FMUL none	none	nx	of nx	inv
inv	inv	—	—	inv

TABLE A-4 Non-Trapping cexc When FSR.NS = 1

		FADD			
	none	nx	of nx	uf nx	inv
FMUL none	none	nx	of nx	uf nx	inv
inv	inv	—	—	—	inv
nx	nx	nx	of nx	uf nx	inv nx

In the tables, the conditions with "—" do not exist.

Programming Note – The Floating-point Multiply-Add instructions are encoded in the SPARC V9 IMPDEP2 opcode space, and they are specific to the SPARC64 VI implementation. They *cannot* be used in any programs that will be executed on any other SPARC V9 processor, unless that implementation exactly matches the SPARC64 VI use of the IMPDEP2 opcode.

Exceptions

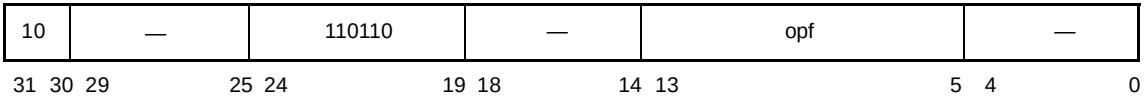
fp_disabled
fp_exception_ieee_754 (NV, NX, OF, UF)
illegal_instruction (size = 00_2 or 11_2) (*fp_disabled* is not checked for these encoding)
fp_exception_other (*unfinished_FPop*)

A.24.2 Suspend

The suspend instructions in this section are specific to SPARC64 VI.

opcode	opf	operation
SUSPEND^{P}	0 1000 0010	suspend a thread

Format (3)



Assembly Language Syntax
`suspend`

Description The instruction puts the thread executed it into the SUSPENDED state. The instruction sets `PSTATE.IE` to “1”. Exit conditions from the SUSPENDED state are:

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

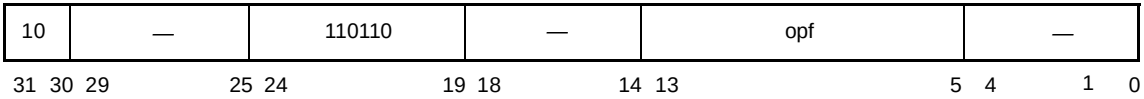
Exceptions: *privileged_opcode*

A.24.3 Sleep

The sleep instructions in this section are specific to SPARC64 VI.

opcode	opf	operation
SLEEP	0 1000 0011	put a thread to sleep

Format (3)



Assembly Language Syntax
`sleep`

Description The instruction puts the thread executed it to sleep. Conditions to wake up are:

- POR,WDR,XIR
- *interrupt_vector* trap
- *interrupt_level_n* trap
- After a certain period, where the period is implementation-dependent.
  The value of SPARC64 VI is about 1.6 micro-seconds. The period is measured by an external clock to SPARC64 VI, and the same clock is used to increment `STICK`.

Note – When the instruction is executed with `PSTATE.IE=0`, the thread will not wake up even if there is an *interrupt_vector.*

If a given thread (A) executes the SLEEP instruction while the other thread (B) in the same core is already in the sleep state, then the thread (A) is relegated to the sleep state and the thread (B) wakes up instead.

Exceptions: None

A.29 Jump and Link

SPARC64 VI clears the upper 32 bits of the PC value in r[rd] when PSTATE.AM is set (impl. dep. #125). The value written into r[rd] is visible to the instruction in the delay slot.

If either of the low-order two bits of the jump address is nonzero, a *mem_address_not_aligned* exception occurs. However, when the JMPL instruction causes a *mem_address_not_aligned* trap, DSFSR and DSFAR are not updated (impl. dep. #237).

If the JMPL instruction has r[rd] = 15, SPARC64 VI stores PC + 8 in a hardware table called return address stack (RAS). When a RET (jmpl %i7+8, %g0) or RETL (jmpl %o7+8, %g0) is executed, the value in the RAS is used to predict the return address.

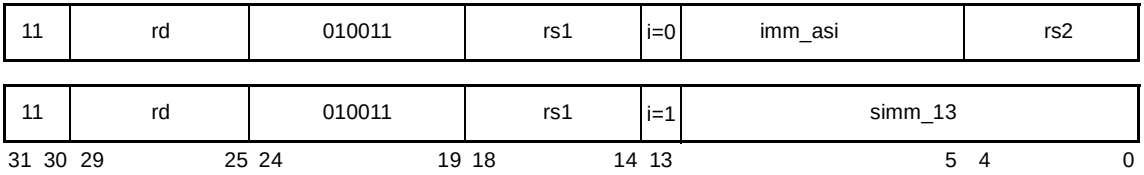
JMPL with rd = 0 can be used to return from a subroutine. The typical return address is "*r*[31] + 8" if a non leaf routine (one that uses the SAVE instruction) is entered by a CALL instruction, or "*r*[15] + 8" if a leaf routine (one that does not use the SAVE instruction) is entered by a CALL instruction or by a JMPL instruction with rd = 15.

A.30 Load Quadword, Atomic [Physical]

The Load Quadword ASIs in this section are specific to SPARC64 VI, as an extension to SPARC JPS1.

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 item, using physical addressing. The data are placed in an even/odd pair of 64-bit registers. The lowest-address 64 bits are placed in the even-numbered register; the highest-address 64 bits are placed in the odd-numbered register. The reference is made from the nucleus context.

In addition to the usual traps for LDDA using a privileged ASI, a *data_access_exception* exception occurs for a noncacheable access or for the use of the quadword-load ASIs with any instruction other than LDDA. A *mem_address_not_aligned* exception is generated if the access is not aligned on a 16-byte boundary.

ASIs 34_{16} and $3C_{16}$ are supported in SPARC64 VI in addition to those for Load Quadword Atomic for virtually addressed data (ASIs 24_{16} and $2C_{16}$).

The memory access for a load quad instruction with `ASI_QUAD_LDD_PHYS{_L}` behaves as if the following `TTE` is set:

- `TTE.NFO`= 0
- `TTE.CP`= 1
- `TTE.CV`= 0

- TTE.E = 0
- TTE.P = 1
- TTE.W = 0

Note – TTE.IE depends on the endianness of the ASI. When the ASI is 034_{16}, TTE.IE = 0; TTE.IE = 1 when the ASI is $03C_{16}$.

Therefore, the atomic quad load physical instruction can only be applied to a cacheable memory area. Semantically, ASI_QUAD_LDD_PHYS{_L} (034_{16} and $03C_{16}$) is a combination of ASI_NUCLEUS_QUAD_LDD and ASI_PHYS_USE_EC.

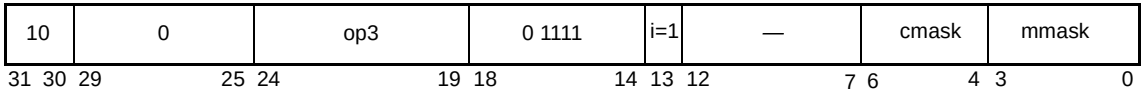
With respect to little endian memory, a Load Quadword Atomic instruction behaves as if it comprises two 64-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

Exceptions:

privileged_action
PA_watchpoint (recognized on only the first 8 bytes of a transfer)
illegal_instruction (misaligned rd)
mem_address_not_aligned
data_access_exception
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.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-5 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-5 Order Constraints Imposed by `mmask` Bits

Mask Bit	Name	Description
mmask<3>	#StoreStore	The effects of all stores appearing before the MEMBAR instruction must be visible to all processors before the effect of any stores following the MEMBAR. Equivalent to the deprecated STBAR instruction. Has no effect on SPARC64 VI since all stores are performed in program order.
mmask<2>	#LoadStore	All loads appearing before the MEMBAR instruction must have been performed before the effects of any stores following the MEMBAR are visible to any other processor. This has no effect on SPARC64 VI since all stores are performed in program order and must occur after performance of any load.
mmask<1>	#StoreLoad	The effects of all stores appearing before the MEMBAR instruction must be visible to all processors before loads following the MEMBAR may be performed.
mmask<0>	#LoadLoad	All loads appearing before the MEMBAR instruction must have been performed before any loads following the MEMBAR may be performed. This has no effect on SPARC64 VI since all loads are performed after any prior loads.

The cmask field is encoded in bits 6 through 4 of the instruction. Bits in the cmask field, described in TABLE A-6, 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-6 Bits in the cmask Field

Mask Bit	Function	Name	Description
cmask<2>	Synchronization barrier	#Sync	All operations (including nonmemory reference operations) appearing before 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 before the MEMBAR must have been performed before any memory operation after the MEMBAR may be initiated. Equivalent to #Sync in SPARC64 VI.
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 VI.

A.42 Partial Store (VIS I)

Please refer A.42 in **Commonality** for general details.

Watchpoint exceptions on partial store instructions occur conservatively on SPARC64 VI. The DCUCR Data Watchpoint masks are only checked for nonzero value (watchpoint enabled). The byte store mask (`r[rs2]`) in the partial store instruction is ignored, and a watchpoint exception can occur even if the mask is zero (that is, no store will take place) (impl. dep. #249).

Implementation Note – For a partial store instruction to noncacheable area with *mask* = 0, SPARC64 VI still issues a Jupiter Bus transaction with zero-byte mask.

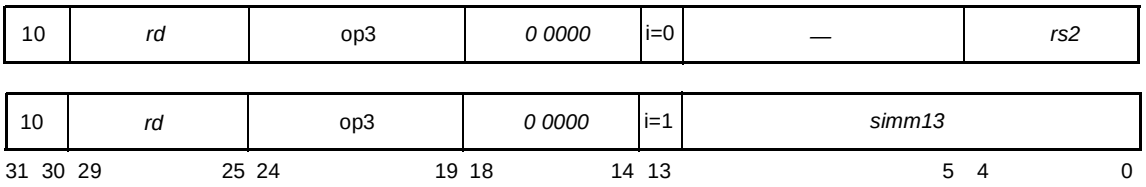
Exceptions:

fp_disabled
PA_watchpoint
VA_watchpoint
illegal_instruction (misaligned rd)
mem_address_not_aligned (see *Partial Store ASIs* on page 128)
data_access_exception (see *Partial Store ASIs* on page 128)
LDDF_mem_address_not_aligned (see *Partial Store ASIs* on page 128)
data_access_error
fast_data_access_MMU_miss
fast_data_access_protection

A.48 Population Count

opcode	op3	operation
POPC	10 1110	Population Count

Format (3)



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

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

Note – Unlike SPARC64 V, SPARC64 VI implements the instruction in hardware.

Exceptions: *illegal_instruction* (instruction<18:14> ≠ 0)

A.49 Prefetch Data

Please refer to Section A.49, *Prefetch Data*, of **Commonality** for principal information.

The `prefetcha` instruction of SPARC64 VI works 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 an ASI other than the above is specified, `prefetcha` is executed as a nop.

TABLE A-7 describes prefetch variants implemented in SPARC64 VI.

TABLE A-7 Prefetch Variants

fcn	Fetch to:	Status	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	
21	L2	S,E	
22	L1D	M,E	
23	L2	M,E	
24-31	*implementation dependent*		NOP

SPARC64 VI does not causes a *fast_data_access_MMU_miss* miss on fcn = 20, 21, 22 and 23 (impl. dep. #103(2)).

A.51 Read State Register

In SPARC64 VI, an RDPCR instruction will generate a *privileged_action* exception if `PSTATE.PRIV` = 0 and `PCR.PRIV` = 1. If `PSTATE.PRIV` = 0 and `PCR.PRIV` = 0, RDPCR will not cause any access privilege violation exception (impl. dep. #250).

A.59 SHUTDOWN (VIS I)

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

A.70 Write State Register

In SPARC64 VI, a WRPCR instruction will cause a *privileged_action* exception if `PSTATE.PRIV` = 0 and `PCR.PRIV` = 1. If `PSTATE.PRIV` = 0 and `PCR.PRIV` = 0, WRPCR causes a *privileged_action* exception only when an attempt is made to change (that is, write 1 to) `PCR.PRIV` (impl. dep. #250).

A.71 Deprecated Instructions

The deprecated instructions in A.71 of **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 VI, STBAR behaves as NOP since the hardware memory models always enforce the semantics of these MEMBARs for all memory accesses.

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.

Please see Appendix B of **Commonality** for choices for these implementation dependencies, to ensure that SPARC V9 implementations are as consistent as possible.

Following is information specific to the SPARC64 VI implementation of SPARC V9 in these sections:

B.1 Traps Inhibiting Results

Please refer to Section B.1 of **Commonality**.

The SPARC64 VI hardware, in conjunction with kernel or emulation code, produces the results described in this section.

B.6 Floating-Point Nonstandard Mode

In this section, the hardware boundary conditions for the *unfinished_FPop* exception and the nonstandard mode of SPARC64 VI floating-point hardware are discussed.

SPARC64 VI floating-point hardware has its specific range of computation. If either the values of input operands or the value of the intermediate result shows that the computation may not fall in the range that hardware provides, SPARC64 VI generates an *fp_exception_other* exception (tt = 022_{16}) with FSR.ftt = 02_{16} (*unfinished_FPop*) and the operation is taken over by software.

The kernel emulation routine completes the remaining floating-point operation in accordance with the IEEE 754-1985 floating-point standard (impl. dep. #3).

SPARC64 VI implements a nonstandard mode, enabled when FSR.NS is set (see *FSR_nonstandard_fp (NS)* on page 16). Depending on the setting in FSR.NS, the behavior of SPARC64 VI with respect to the floating-point computation varies.

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

SPARC64 VI may invoke an *fp_exception_other* (tt = 022_{16}) exception with FSR.ftt = *unfinished_FPop* (ftt = 02_{16}) in FsTOd, FdTOs, FADD(s,d), FSUB(s,d), FsMULd(s,d), FMUL(s,d), FDIV(s,d), FSQRT(s,d) floating-point instructions. In addition, Floating-point Multiply-Add/Subtract instructions generate the exception, since the instruction is the combination of a multiply and an add/subtract operation: FMADD(s,d), FMSUB(s,d), FNMADD(s,d), and FNMADD(s,d).

The following basic policies govern the detection of boundary conditions:

1. When one of the operands is a denormalized number and the other operand is a normal non-zero floating-point number (except for a NaN or an infinity), an *fp_exception_other* with *unfinished_FPop* condition is signalled. The cases in which the result is a zero or an overflow are excluded.

2. When both operands are denormalized numbers, except for the cases in which the result is a zero or an overflow, an *fp_exception_other* with *unfinished_FPop* condition is signalled.

3. When both operands are normal, the result before rounding is a denormalized number and TEM.UFM = 0, and *fp_exception_other* with *unfinished_FPop* condition is signalled, except for the cases in which the result is a zero.

When the result is expected to be a constant, such as an exact zero or an infinity, and an insignificant computation will furnish the result, SPARC64 VI tries to calculate the result without signalling an *unfinished_FPop* exception.

Implementation Note – Detecting the exact boundary conditions requires a large amount of hardware. SPARC64 VI detects approximate boundary conditions by calculating the exponent intermediate result (the exponent before rounding) from input operands, to avoid the hardware cost. Since the computation of the boundary conditions is approximate, the detection of a zero result or an overflow result shall be pessimistic. SPARC64 VI generates an *unfinished_FPop* exception pessimistically.

The equations to calculate the result exponent to detect the boundary conditions from the input exponents are presented in TABLE B-1, where Er is the approximation of the biased result exponent before rounding and is calculated only from the input exponents (esrc1, esrc2). Er is to be used for detecting the boundary condition for an *unfinished_FPop*.

TABLE B-1 Result Exponent Approximation for Detecting *unfinished_FPop* Boundary Conditions

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 input operands. When the corresponding input operand is a denormalized number, the value is 0.

From Er, eres is calculated. eres is a biased result exponent, after mantissa alignment and before rounding, where the appropriate adjustment of the exponent is applied to the result mantissa: left-shifting or right-shifting the mantissa to the implicit 1 at the left of the binary point, subtracting or adding the shift-amount to the exponent. The result mantissa is assumed to be 1.xxxx in calculating eres. If the result is a denormalized number, eres is less than zero.

TABLE B-2 describes the boundary condition of each floating-point instruction that generates an *unfinished_FPop* exception.

TABLE B-2 *unfinished_FPop* Boundary Conditions

Operation	Boundary Conditions
FdTOs	–25 < eres < 1 and TEM.UFM = 0.
FsTOd	Second operand (rs2) is a denormalized number.
FADDs, FSUBs, FADDd, FSUBd	1. One of the operands is a denormalized number, and the other operand is a normal, nonzero floating-point number (except for a NaN and an infinity)[1]. 2. Both operands are denormalized numbers. 3. Both operands are normal nonzero floating-point numbers (except for a NaN and an infinity), eres < 1, and TEM.UFM = 0.

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

Operation	Boundary Conditions
FMULs, FMULd	1. One of the operands is a denormalized number, the other operand is a normal, nonzero floating-point number (except for a NaN and an infinity), and single precision: -25 < Er double precision: -54 < Er 2. Both operands are normal, nonzero floating-point numbers (except for a NaN and an infinity), TEM.UFM = 0, and single precision: –25 < eres < 1 double precision: –54 < eres < 1
FsMULd	1. One of the operands is a denormalized number, and the other operand is a normal, nonzero floating-point number (except for a NaN and an infinity). 2. Both operands are denormalized numbers.
FDIVs, FDIVd	1. The dividend (operand1; rs1) is a normal, nonzero floating-point number (except for a NaN and an infinity), the divisor (operand2; rs2) is a denormalized number, and single precision: Er < 255 double precision: Er < 2047 2. The dividend (operand1; rs1) is a denormalized number, the divisor (operand2; rs2) is a normal, nonzero floating-point number (except for a NaN and an infinity), and single precision: –25 < Er double precision: –54 < Er 3. Both operands are denormalized numbers. 4. Both operands are normal, nonzero floating-point numbers (except for a NaN and an infinity), TEM.UFM = 0 and single precision: –25 < eres < 1 double precision: –54 < eres < 1
FSQRTs, FSQRTd	The input operand (operand2; rs2) is a positive nonzero and is a denormalized number.

1.Operation of 0 and denormalized number generates a result in accordance with the IEEE754-1985 standard.

Pessimistic Zero

If a condition in TABLE B-3 is `true`, SPARC64 VI generates the result as a pessimistic zero, meaning that the result is a denormalized minimum or a zero, depending on the rounding mode (`FSR.RD`).

TABLE B-3 Conditions for a Pessimistic Zero

	Conditions		
Operations	**One operand is denormalized[1]**	**Both are denormalized**	**Both are normal fp-number[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.Both operands are non-zero, non-NaN, and non-infinity numbers.

2.Both may be zero, but both are non-NaN and non-infinity numbers.

Pessimistic Overflow

If a condition in TABLE B-4 is `true`, SPARC64 VI regards the operation as having an overflow condition.

TABLE B-4 Pessimistic Overflow Conditions

Operations	Conditions
FDIVs	The divisor (operand2; rs2) is a denormalized number and, $Er \geq 255$.
FDIVd	The divisor (operand2; rs2) is a denormalized number and, $E \geq 2047$.

B.6.2 Operation Under FSR.NS = 1

When `FSR.NS` = 1 (nonstandard mode), SPARC64 VI zeroes all the input denormalized operands before the operation and signals an inexact exception if enabled. If the operation generates a denormalized result, SPARC64 VI zeroes the result and also signals an inexact exception if enabled. The following list defines the operation in detail.

- If either operand is a denormalized number and both operands are non-zero, non-NaN, and non-infinity numbers, the input denormalized operand is replaced with a zero with same sign, and the operation is performed. If enabled, inexact exception is signalled; an *fp_exception_ieee_754* (`tt` = 021_{16}) is generated, with `nxc`=1 in `FSR.cexc` (`FSR.ftt`=01_{16}; *IEEE754_exception*). However, if the operation is `FDIV(s,d)` and either a *division_by_zero* or an *invalid_operation* condition is detected, or if the operation is `FSQRT(s,d)` and an *invalid_operation* condition is detected, the inexact condition is not reported.
- If the result before rounding is a denormalized number, the result is flushed to a zero with a same sign and signals either an underflow exception or an inexact exception, depending on `FSR.TEM`.

As observed from the preceding, when `FSR.NS` = 1, SPARC64 VI generates neither an *unfinished_FPop* exception nor a denormalized number as a result. TABLE B-5 summarizes the behavior of SPARC64 VI floating-point hardware depending on `FSR.NS`.

Note – The result and behavior of SPARC64 VI of the shaded column in the tables Table B-5 and Table B-6 conform to IEEE754-1985 standard.

Note – Throughout Table B-5 and Table B-6, lowercase exception conditions such as nx, uf, of, dv and nv are nontrapping IEEE 754 exceptions. Uppercase exception conditions such as NX, UF, OF, DZ and NV are trapping IEEE 754 exceptions.

TABLE B-5 Floating-Point Exceptional Conditions and Results

FSR.NS	Denorm : Norm[1]	Result Denorm[2]	Pessimistic Zero	Pessimistic Overflow	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	n/a	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	No	Yes	—	—	1	—	—	UF
					0	—	1	NX
							0	uf + nx, a signed zero
		No			—	—	—	Conforms to IEEE754-1985
	Yes	—						TABLE B-6

1. One of the operands is a denormalized number, and the other operand is a normal or a denormalized number (non- zero, non-NaN, and non-infinity).
2. The result before rounding turns out to be a denormalized number.
3. Dmin = denormalized minimum.
4. If the FPop is either `FADD{s,d}`, or `FSUB{s,d}` and the operation is 0 ± denormalized number, SPARC64 VI does not generate an *unfinished_FPop* and generates a result according to IEEE754-1985 standard.
5. Nmax = normalized maximum.

TABLE B-6 describes how SPARC64 VI behaves when `FSR.NS` = 1 (nonstandard mode).

TABLE B-6 Non arithmetic Operations Under `FSR.NS` = 1

Operations	op1= denorm	op2= denorm	UFM	NXM	DVM	NVM	Result
FsTOd	—	Yes	—	1	—	—	NX
				0	—	—	nx, a signed zero
FdTOs	—	Yes	1	—	—	—	UF
			0	1	—	—	NX
				0	—	—	uf + nx, a signed zero

TABLE B-6 Non arithmetic Operations Under FSR.NS = 1 *(Continued)*

Operations	op1= denorm	op2= denorm	UFM	NXM	DVM	NVM	Result
FADDs, FSUBs, FADDd, FSUBd	Yes	No	—	1	—	—	NX
				0	—	—	nx, op2
	No	Yes		1	—	—	NX
				0	—	—	nx, op1
	Yes	Yes		1	—	—	NX
				0	—	—	nx, a signed zero
FMULs, FMULd, FsMULd	Yes	—	—	1	—	—	NX
				0	—	—	nx, a signed zero
	—	Yes		1	—	—	NX
				0			nx, a signed zero
FDIVs, FDIVd	Yes	No	—	1	—	—	NX
				0	—	—	nx, a signed zero
	No	Yes		—	1	—	DZ
				—	0	—	dz, a signed infinity
	Yes	Yes		—	—	1	NV
				—	—	0	nv, dNaN[1]
FSQRTs, FSQRTd	—	Yes and op2 > 0	—	1	—	—	NX
				0	—	—	nx, zero
		Yes and op2 < 0		—	—	1	NV
				—	—	0	nv, dNaN

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

F.APPENDIX **C**

Implementation Dependencies

This appendix summarizes implementation dependencies. 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 on page 78. These numbers have been removed from the body of this document for SPARC64 VI to make the document more readable. TABLE C-1 has been modified to include descriptions of the manner in which SPARC64 VI has resolved each implementation dependency.

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.1 Definition of an Implementation Dependency

Please refer to Section C.1 of **Commonality**.

C.2 Hardware Characteristics

Please refer to Section C.2 of **Commonality**.

C.3 Implementation Dependency Categories

Please refer to Section C.3 of **Commonality**.

C.4 List of Implementation Dependencies

TABLE C-1 provides a complete list of how each implementation dependency is treated in the SPARC64 VI implementation.

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

Nbr	SPARC64 VI Implementation Notes	Page
1	**Software emulation of instructions** The operating system emulates all instructions that generate *illegal_instruction* or *unimplemented_FPop* exceptions.	—
2	**Number of IU registers** SPARC64 VI supports eight register windows (NWINDOWS = 8). SPARC64 VI supports an additional two global register sets (Interrupt globals and MMU globals) for a total of 160 integer registers.	—
3	**Incorrect IEEE Std. 754-1985 results** See **Section B.6, *Floating-Point Nonstandard Mode*** for details.	69
4–5	*Reserved.*	
6	**I/O registers privileged status** This dependency is beyond the scope of this publication. It should be defined in each system that uses SPARC64 VI.	—
7	**I/O register definitions** This dependency is beyond the scope of this publication. It should be defined in each system that uses SPARC64 VI.	—
8	**RDASR/WRASR target registers** SPARC64 VI does not define implementation dependent ASR register.	—

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

Nbr	SPARC64 VI Implementation Notes	Page
9	**RDASR/WRASR privileged status** SPARC64 VI does not define implementation dependent ASR register.	—
10–12	*Reserved.*	
13	**VER.impl** VER.impl = 6 for the SPARC64 VI processor.	18
14–15	*Reserved.*	—
16	**IU deferred-trap queue** SPARC64 VI neither has nor needs an IU deferred-trap queue.	23
17	*Reserved.*	—
18	**Nonstandard IEEE 754-1985 results** SPARC64 VI flushes denormal operands and results to zero when FSR.NS = 1. For the treatment of denormalized numbers, please refer to Section B.6, *Floating-Point Nonstandard Mode* for details.	16
19	**FPU version, FSR.ver** FSR.ver = 0 for SPARC64 VI.	16
20–21	*Reserved.*	
22	**FPU TEM, cexc, and aexc** SPARC64 VI implements all bits in the TEM, cexc, and aexc fields in hardware.	15
23	**Floating-point traps** In SPARC64 VI floating-point traps are always precise; no FQ is needed.	23
24	**FPU deferred-trap queue (FQ)** SPARC64 VI neither has nor needs a floating-point deferred-trap queue.	23
25	**RDPR of FQ with nonexistent FQ** Attempting to execute an RDPR of the FQ causes an *illegal_instruction* exception.	23
26–28	*Reserved.*	—
29	**Address space identifier (ASI) definitions** The ASIs that are supported by SPARC64 VI are defined in Appendix , *Address Space Identifiers.*	—
30	**ASI address decoding** SPARC64 VI supports all of the listed ASIs.	125
31	**Catastrophic error exceptions** SPARC64 VI contains a watchdog timer that times out after no instruction has been committed for a specified number of cycles. If the timer times out, the CPU tries to invoke an *async_data_error* trap. If the counter continues to count to reach 2^{33}, the processor enters `error_state`. Upon an entry to `error_state`, the processor optionally generates a WDR reset to recover from `error_state`.	144
32	**Deferred traps** SPARC64 VI signals a deferred trap in a few of its severe error conditions. SPARC64 VI does not contain a deferred trap queue.	37, 153

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

Nbr	SPARC64 VI Implementation Notes	Page
33	**Trap precision** There are no deferred traps in SPARC64 VI other than the trap caused by a few severe error conditions. All traps that occur as the result of program execution are precise.	37
34	**Interrupt clearing** For details of interrupt handling see Appendix , *Interrupt Handling*.	137
35	**Implementation-dependent traps** SPARC64 VI supports the following traps that are implementation dependent: • *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} through 067_{16}) • *fast_data_access_MMU_miss* (tt = 068_{16} through $06B_{16}$) • *fast_data_access_protection* (tt = $06C_{16}$ through $06F_{16}$) • *async_data_error* (tt = 040_{16})	39
36	**Trap priorities** SPARC64 VI's 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)	39
37	**Reset trap** SPARC64 VI implements power-on reset (POR) and watchdog reset.	37
38	**Effect of reset trap on implementation-dependent registers** See Section O.2, *RED_state and error_state*.	145
39	**Entering error_state on implementation-dependent errors** CPU watchdog timeout at 2^{33} ticks, a normal trap, or an SIR at TL = MAXTL causes the CPU to enter error_state.	36
40	**Error_state processor state** SPARC64 VI optionally takes a watchdog reset trap after entry to error_state. Most error-logging register state will be preserved. (See also impl. dep. #254.)	36
41	*Reserved.*	
42	**FLUSH instruction** SPARC64 VI implements the FLUSH instruction in hardware.	—
43	*Reserved.*	
44	**Data access FPU trap** The destination register(s) are unchanged if an access error occurs.	—

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

Nbr	SPARC64 VI Implementation Notes	Page
45–46	*Reserved.*	
47	**RDASR** SPARC64 VI does not define implementation dependent ASR register.	—
48	**WRASR** SPARC64 VI does not define implementation dependent ASR register.	—
49–54	*Reserved.*	
55	**Floating-point underflow detection** See *FSR_underflow* in Section 5.1.7 of **Commonality** for details.	—
56–100	*Reserved.*	
101	**Maximum trap level** MAXTL = 5.	18
102	**Clean windows trap** SPARC64 VI generates a *clean_window* exception; register windows are cleaned in software.	—
103	**Prefetch instructions** SPARC64 VI implements PREFETCH variations 0–3 and 20–23 with the following implementation-dependent characteristics: • The prefetches have observable effects in privileged code. • All variants never cause a *fast_data_access_MMU_miss* trap. • All prefetches are for 64-byte cache lines, which are aligned on a 64-byte boundary. • See Section A.49, *Prefetch Data*, for implemented variations and their characteristics. • Prefetches will work normally if the ASI is ASI_PRIMARY, ASI_SECONDARY, or ASI_NUCLEUS, `ASI_PRIMARY_AS_IF_USER`, `ASI_SECONDARY_AS_IF_USER`, and their little-endian pairs.	67
104	**VER.manuf** VER.manuf = 0004_{16}. The least significant 8 bits are Fujitsu's JEDEC manufacturing code.	18
105	**TICK register** SPARC64 VI implements 63 bits of the TICK register; it increments on every clock cycle.	17
106	**IMPDEP*n* instructions** SPARC64 VI uses the IMPDEP21 opcode for SUSPEND and SLEEP instructions, and the IMPDEP2 opcode for the Multiply Add/Subtract instructions. SPARC64 VI also conforms to Sun's specification for VIS-1 and VIS-2.	54
107	**Unimplemented LDD trap** SPARC64 VI implements LDD in hardware.	—
108	**Unimplemented STD trap** SPARC64 VI implements STD in hardware.	—

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

Nbr	SPARC64 VI Implementation Notes	Page
109	***LDDF_mem_address_not_aligned*** If the address is word aligned but not doubleword aligned, SPARC64 VI generates the *LDDF_mem_address_not_aligned* exception. The trap handler software emulates the instruction.	—
110	***STDF_mem_address_not_aligned*** If the address is word aligned but not doubleword aligned, SPARC64 VI generates the *STDF_mem_address_not_aligned* exception. The trap handler software emulates the instruction.	—
111	***LDQF_mem_address_not_aligned*** SPARC64 VI generates an *illegal_instruction* exception for all LDQFs. The processor does not perform the check for *fp_disabled*. The trap handler software emulates the instruction.	—
112	***STQF_mem_address_not_aligned*** SPARC64 VI generates an *illegal_instruction* exception for all STQFs. The processor does not perform the check for *fp_disabled*. The trap handler software emulates the instruction.	—
113	**Implemented memory models** SPARC64 VI implements Total Store Order (TSO) for all the memory models specified in `PSTATE.MM`. See Chapter 8, *Memory Models*, for details.	41
114	**RED_state trap vector address (RSTVaddr)** RSTVaddr is a constant in SPARC64 VI, where: VA = FFFF FFFF F000 0000_{16} and PA = 07FF F000 0000_{16}	36
115	**RED_state processor state** See *RED_state* on page 36 for details of implementation-specific actions in RED_state.	36
116	**SIR_enable control flag** See Section A.60 `SIR` in **Commonality** for details.	—
117	**MMU disabled prefetch behavior** Prefetch and nonfaulting Load always succeed when the MMU is disabled.	99
118	**Identifying I/O locations** This dependency is beyond the scope of this publication. It should be defined in a system that uses SPARC64 VI.	—
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, since future versions of SPARC64 VI may use this encoding for a new memory model.	42
120	**Coherence and atomicity of memory operations** Although SPARC64 VI implements the Jupiter Bus based cache coherency mechanism, this dependency is beyond the scope of this publication. It should be defined in a system that uses SPARC64 VI.	—

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

Nbr	SPARC64 VI Implementation Notes	Page
121	**Implementation-dependent memory model** SPARC64 VI implements TSO, PSO, and RMO memory models. See Chapter 8, *Memory Models*, for details. Accesses to pages with the E (Volatile) bit of their MMU page table entry set are also made in program order.	—
122	**FLUSH latency** Since the `FLUSH` instruction synchronizes the processor, its total latency varies depending on many portions of the SPARC64 VI processor's state. Assuming that all prior instructions are completed, the latency of `FLUSH` is 18 processor cycles.	—
123	**Input/output (I/O) semantics** This dependency is beyond the scope of this publication. It should be defined in a system that uses SPARC64 VI.	—
124	**Implicit ASI when TL > 0** See Section 5.1.7 of **Commonality** for details.	—
125	**Address masking** When PSTATE.AM = 1, SPARC64 VI *does* mask out the high-order 32 bits of the PC when transmitting it to the destination register.	28, 53, 60
126	**Register Windows State Registers width** NWINDOWS for SPARC64 VI is 8; therefore, only 3 bits are implemented for the following registers: CWP, CANSAVE, CANRESTORE, OTHERWIN. If an attempt is made to write a value greater than NWINDOWS – 1 to any of these registers, the extraneous upper bits are discarded. The CLEANWIN register contains 3 bits.	—
127–201	*Reserved.*	
202	***fast_ECC_error* trap** *fast_ECC_error* trap is not implemented in SPARC64 VI.	—
203	**Dispatch Control Register bits 13:6 and 1** SPARC64 VI does not implement DCR.	20
204	**DCR bits 5:3 and 0** SPARC64 VI does not implement DCR.	20
205	**Instruction Trap Register** SPARC64 VI implements the Instruction Trap Register.	22
206	**SHUTDOWN instruction** In privileged mode the SHUTDOWN instruction executes as a NOP in SPARC64 VI.	68
207	**PCR register bits 47:32, 26:17, and bit 3** SPARC64 VI uses these bits for the following purposes: • Bits 47:32 for set/clear/show status of overflow (OVF). • Bit 26 for validity of OVF field (OVRO). • Bits 24:22 for number of counter pair (NC). • Bits 20:18 for counter selector (SC). • Bit 3 for validity of SU/SL field (ULRO). Other implementation-dependent bits are read as 0 and writes to them are ignored.	18, 197

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

Nbr	SPARC64 VI Implementation Notes	Page
208	**Ordering of errors captured in instruction execution** The order in which errors are captured during instruction execution is implementation dependent. Ordering can be in program order or in order of detection.	—
209	**Software intervention after instruction-induced error** Precision of the trap to signal an instruction-induced error for which recovery requires software intervention is implementation dependent**.**	—
210	**ERROR output signal** The causes and the semantics of ERROR output signal are implementation dependent.	—
211	**Error logging registers' information** The information that the error logging registers preserves beyond the reset induced by an ERROR signal is implementation dependent**.**	—
212	**Trap with fatal error** Generation of a trap along with ERROR signal assertion upon detection of a fatal error is implementation dependent**.**	—
213	**AFSR.PRIV** SPARC64 VI does not implement the AFSR.PRIV bit.	—
214	**Enable/disable control for deferred traps** SPARC64 VI does not implement a control feature for deferred traps.	—
215	**Error barrier** DONE and RETRY instructions may implicitly provide an error barrier function as MEMBAR #Sync. Whether DONE and RETRY instructions provide an error barrier is implementation dependent.	—
216	***data_access_error* trap precision** *data_access_error* trap is always precise in SPARC64 VI.	—
217	***instruction_access_error* trap precision** *instruction_access_error* trap is always precise in SPARC64 VI.	—
218	***async_data_error*** *async_data_error* trap is implemented in SPARC64 VI, using tt = 40_{16}. See Appendix , *Error Handling* for details.	39
219	**Asynchronous Fault Address Register (`AFAR`) allocation** SPARC64 VI does not implement an AFAR.	180
220	**Addition of logging and control registers for error handling** SPARC64 VI implements various features for sustaining reliability. See Appendix P for details.	—
221	**Special/signalling ECCs** The method to generate "special" or "signalling" ECCs and whether processor-ID is embedded into the data associated with special/signalling ECCs is implementation dependent.	—

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

Nbr	SPARC64 VI Implementation Notes	Page
222	**TLB organization** SPARC64 VI has the following TLB organization: • Level-1 micro ITLB (uITLB), 32-way fully associative • Level-1 micro DTLB (uDTLB), 32-way fully associative • Level-2 IMMU-TLB—consisting of sITLB (set-associative Instruction TLB) and fITLB (fully associative Instruction TLB). • Level-2 DMMU-TLB—consisting of sDTLB (set-associative Data TLB) and fDTLB (fully associative Data TLB).	93
223	**TLB multiple-hit detection** On SPARC64 VI, TLB multiple hit detection is supported. However, the multiple hit is not detected at every TLB reference. When the micro-TLB (uTLB), which is the cache of sTLB and fTLB, matches the virtual address, the multiple hit in sTLB and fTLB is not detected. The multiple hit is detected only when the micro-TLB mismatches and the main TLB is referenced.	94
224	**MMU physical address width** The SPARC64 VI MMU implements 43-bit physical addresses. The `PA` field of the `TTE` holds a 43-bit physical address. Bits 46:43 of each TTE always read as 0 and writes to them are ignored. The MMU translates virtual addresses into 43-bit physical addresses. Each cache tag holds bits 42:6 of physical addresses.	95
225	**TLB locking of entries** In SPARC64 VI, when a TTE with its lock bit set is written into TLB through the Data In register, the TTE is automatically written into the corresponding fully associative TLB and locked in the TLB. Otherwise, the TTE is written into the corresponding sTLB of fTLB, depending on its page size.	95
226	**`TTE` support for CV bit** SPARC64 VI does not support the CV bit in TTE. Since I1 and D1 are virtually indexed caches, unaliasing is supported by SPARC64 VI. See also impl. dep. #232.	95
227	**TSB number of entries** SPARC64 VI supports a maximum of 16 million entries in the common TSB and a maximum of 32 million lines the Split TSB.	96
228	**`TSB_Hash` supplied from TSB or context-ID register** `TSB_Hash` is generated from the context-ID register in SPARC64 VI.	96
229	**`TSB_Base` address generation** SPARC64 VI generates the `TSB_Base` address directly from the TLB Extension Registers. By maintaining compatibility with UltraSPARC I/II, SPARC64 VI provides mode flag `MCNTL.JPS1_TSBP`. When `MCNTL.JPS1_TSBP` = 0, the `TSB_Base` register is used.	96
230	***data_access_exception* trap** SPARC64 VI generates *data_access_exception* only for the causes listed in Section 7.6.1 of **Commonality**.	97
231	**MMU physical address variability** The width of physical address is 47 bit in SPARC64 VI.	99

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

Nbr	SPARC64 VI Implementation Notes	Page
232	**DCU Control Register CP and CV bits** SPARC64 VI does not implement CP and CV bits in the DCU Control Register. See also impl. dep. #226.	20, 99
233	**TSB_Hash field** SPARC64 VI does not implement `TSB_Hash`.	99
234	**TLB replacement algorithm** For fTLB, SPARC64 VI implements a pseudo-LRU. For sTLB, LRU is used.	105
235	**TLB data access address assignment** The MMU TLB data-access address assignment and the purpose of the address are implementation dependent.	105
236	**`TSB_Size` field width** In SPARC64 VI, TSB_Size is 4 bits wide, occupying bits 3:0 of the TSB register. The maximum number of TSB entries is, therefore, 512×2^{15} (16M entries).	108
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 register.	60, 97, 108
238	**TLB page offset for large page sizes** On SPARC64 VI, even for a large page, written data for TLB Data Register is preserved for bits representing an offset in a page, so the data previously written is returned regardless of the page size.	95
239	**Register access by ASIs 55_{16} and $5D_{16}$** In SPARC64 VI, VA<63:19> of IMMU ASI 55_{16} and DMMU ASI $5D_{16}$ are ignored. An access to virtual addresses 40000_{16} to $60FF8_{16}$ is treated as an access 00000_{16} to $20FF8_{16}$	100
240	**DCU Control Register bits 47:41** SPARC64 VI uses bit 41 for `WEAK_SPCA`, which enables/disables memory access in speculative paths.	20
241	**Address Masking and `DSFAR`** When PSTATE.AM = 1, SPARC64 VI writes zeroes to the more significant 32 bits of DSFAR.	?
242	**TLB lock bit** In SPARC64 VI, only the fITLB and the fDTLB support the lock bit. The lock bit in sITLB and sDTLB is read as 0 and writes to it are ignored.	95
243	**Interrupt Vector Dispatch Status Register BUSY/NACK pairs** In SPARC64 VI, 32 BUSY/NACK pairs are implemented in the Interrupt Vector Dispatch Status Register.	140
244	**Data Watchpoint Reliability** No implementation-dependent features of SPARC64 VI reduce the reliability of data watchpoints.	22

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

Nbr	SPARC64 VI Implementation Notes	Page
245	**Call/Branch displacement encoding in I-Cache** In SPARC64 VI, 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 (as they appear in main memory).	?
246	**VA<38:29> for Interrupt Vector Dispatch Register Access** SPARC64 VI ignores all 10 bits of VA<38:29> when the Interrupt Vector Dispatch Register is written.	140
247	**Interrupt Vector Receive Register SID fields** SID_H and SID_L values are undefined.	140
248	**Conditions for *fp_exception_other* with *unfinished_FPop*** SPARC64 VI triggers *fp_exception_other* with trap type *unfinished_FPop* under the standard conditions described in **Commonality** Section 5.1.7.	16
249	**Data watchpoint for Partial Store instruction** Watchpoint exceptions on Partial Store instructions occur conservatively on SPARC64 VI. The DCUCR Data Watchpoint masks are only checked for nonzero value (watchpoint enabled). The byte store mask (r[rs2]) in the Partial Store instruction is ignored, and a watchpoint exception can occur even if the mask is zero (that is, no store will take place).	65
250	**PCR accessibility when PSTATE.PRIV = 0** In SPARC64 VI, the accessibility of PCR when PSTATE.PRIV = 0 is determined by PCR.PRIV. If PSTATE.PRIV = 0 and PCR.PRIV = 1, an attempt to execute either RDPCR or WRPCR will cause a *privileged_action* exception. If PSTATE.PRIV = 0 and PCR.PRIV = 0, RDPCR operates without privilege violation and WRPCR generates a *privileged_action* exception only when an attempt is made to change (that is, write 1 to) PCR.PRIV.	18, 20, 68
251	*Reserved.*	—
252	**DCUCR.DC (Data Cache Enable)** SPARC64 VI does not implement DCUCR.DC.	20
253	**DCUCR.IC (Instruction Cache Enable)** SPARC64 VI does not implement DCUCR.IC.	20
254	**Means of exiting error_state** The standard behavior of a SPARC64 VI CPU upon entry into error_state is to reset itself by internally generating a *watchdog_reset* (WDR). However, OPSR can be set so that when error_state is entered, the processor remains halted in error_state instead of generating a *watchdog_reset*.	36, 151
255	**LDDFA with ASI $E0_{16}$ or $E1_{16}$ and misaligned destination register number** No exception is generated based on the destination register *rd*.	128

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

Nbr	SPARC64 VI Implementation Notes	Page
256	**LDDFA with ASI $E0_{16}$ or $E1_{16}$ and misaligned memory address** For LDDFA with ASI $E0_{16}$ or $E1_1$ and a memory address aligned on a 2^n-byte boundary, a SPARC64 V processor behaves as follows: $n \geq 3$ ($\geq$ 8-byte alignment): no exception related to memory address alignment is generated. $n = 2$ (4-byte alignment): *LDDF_mem_address_not_aligned* exception is generated. $n \leq 1$ ($\leq$ 2-byte alignment): *mem_address_not_aligned* exception is generated.	128
257	**LDDFA with ASI $C0_{16}$–$C5_{16}$ or $C8_{16}$–CD_{16} and misaligned memory address** For LDDFA with $C0_{16}$–$C5_{16}$ or $C8_{16}$–CD_{16} and a memory address aligned on a 2^n-byte boundary, a SPARC64 V processor behaves as follows: $n \geq 3$ ($\geq$ 8-byte alignment): no exception related to memory address alignment is generated. $n = 2$ (4-byte alignment): *LDDF_mem_address_not_aligned* exception is generated. $n \leq 1$ ($\leq$ 2-byte alignment): *mem_address_not_aligned* exception is generated.	128
258	**ASI_SERIAL_ID** SPARC64 VI provides an identification code for each processor.	127

F.APPENDIX **D**

Formal Specification of the Memory Models

Please refer to Appendix D of **Commonality**.

F.APPENDIX **E**

Opcode Maps

Please refer to Appendix E in *SPARC Joint Programming Specification 1 (JPS1): Commonality*. TABLE E-1 lists the opcode maps for the SPARC64 VI IMPDEP2 instructions, and lists the one for the IMPDEP1 instructions.

TABLE E-1 `IMPDEP2` (op = 2, op3 = 37_{16})

		var (instruction <8:7>)			
		00	**01**	**10**	**11**
size (instruction<6:5>)	**00**	*(not used — reserved)*			
	01	FMADDs	FMSUBs	FNMADDs	`FNMADDs`
	10	FMADDd	FMSUBd	SNMSUBd	`FNMSUBd`
	11	*(reserved for quad operations)*			

TABLE E-2 IMPDEP1: opf<8:0> for VIS opcodes (op = 2, op3 = 36_{16})

		opf <8:4>									
		00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09-1F**
opf <3:0>	**0**	EDGE8	ARRAY8	FCMPLE16	—	—	FPADD16	FZERO	FAND	SHUTDOWN	—
	1	EDGE8N	—	—	FMUL 8x16	—	FPADD16S	FZEROS	FANDS	SIAM	—
	2	EDGE8L	ARRAY16	FCMPNE16	—	—	FPADD32	FNOR	FXNOR	SUSPEND	—
	3	EDGE8LN	—	—	FMUL 8x16AU	—	FPADD32S	FNORS	FXNORS	SLEEP	—
	4	EDGE16	ARRAY32	FCMPLE32	—	—	FPSUB16	FANDNOT2	FSRC1	—	—
	5	EDGE16N	—	—	FMUL 8x16AL	—	FPSUB16S	FANDNOT2S	FSRC1S	—	—
	6	EDGE16L	—	FCMPNE32	FMUL 8SUx16	—	FPSUB32	FNOT2	FORNOT2	—	—
	7	EDGE16LN	—	—	FMUL 8ULx16	—	FPSUB32S	FNOT2S	FORNOT2S	—	—
	8	EDGE32	ALIGN ADDRESS	FCMPGT16	FMULD 8SUx16	FALIGNDATA	—	FANDNOT1	FSRC2	—	—
	9	EDGE32N	BMASK	—	FMULD 8ULx16	—	—	FANDNOT1S	FSRC2S	—	—
	A	EDGE32L	ALIGN ADDRESS _LITTLE	FCMPEQ16	FPACK32	—	—	FNOT1	FORNOT1	—	—
	B	EDGE32LN	—	—	FPACK16	FPMERGE	—	FNOT1S	FORNOR1S	—	—
	C	—	—	FCMPGT32	—	BSHUFFLE	—	FXOR	FOR	—	—
	D	—	—	—	FPACKFIX	FEXPAND	—	FXORS	FORS	—	—
	E	—	—	FCMPEQ32	PDIST	—	—	FNAND	FONE	—	—
	F	—	—	—	—	—	—	FNANDS	FONES	—	—

F.APPENDIX **F**

Memory Management Unit

The Memory Management Unit (MMU) architecture of SPARC64 VI conforms to the MMU architecture defined in Appendix F of **Commonality** but with some model dependency. See Appendix F in **Commonality** for the basic definitions of the SPARC64 VI MMU.

Section numbers in this appendix correspond to those in Appendix F of **Commonality**. Figures and tables, however, are numbered consecutively.

This appendix describes the implementation dependencies and other additional information about the SPARC64 VI MMU. For SPARC64 VI implementations, we first list the implementation dependency as given in TABLE C-1 of **Commonality**, then describe the SPARC64 VI implementation.

F.1 Virtual Address Translation

IMPL. DEP. #222: TLB organization is JPS1 implementation dependent.

SPARC64 VI has the following TLB organization:

- Level-1 micro ITLB (uITLB), 32-way fully associative
- Level-1 micro DTLB (uDTLB), 32-way fully associative
- Level-2 IMMU-TLB consists of sITLB (set-associative Instruction TLB) and fITLB (fully associative Instruction TLB).
- Level-2 DMMU-TLB consists of sDTLB (set-associative Data TLB) and fDTLB (fully associative Data TLB).

TABLE F-1 shows the organization of SPARC64 VI TLBs.

Hardware contains micro-ITLB and micro-DTLB as the temporary memory of the main TLBs, as shown in TABLE F-1. In contrast to the micro-TLBs, sTLB and fTLB are called main TLBs.

The micro-TLBs are coherent to main TLBs and are not visible to software, with the exception of TLB multiple hit detection. Hardware maintains the consistency between micro-TLBs and main TLBs.

No other details on micro-TLB are provided because software cannot execute direct operations to micro-TLB and its configuration is invisible to software.

TABLE F-1 Organization of SPARC64 VI TLBs

Feature	sITLB and sDTLB	fITLB and fDTLB
Entries	2048	32
Associativity	2-way set associative	Fully associative
Locked translation entry	Not supported	Supported
Unlocked translation entry	Supported	Supported

IMPL. DEP. #223: Whether TLB multiple-hit detections are supported in JPS1 is implementation dependent.

On SPARC64 VI, TLB multiple hit detection is supported. However, the multiple hit is not detected at every TLB reference. When the micro-TLB (uTLB), which is the cache of sTLB and fTLB, matches the virtual address, the multiple hit in sTLB and fTLB is not detected. The multiple hit is detected only when the micro-TLB mismatches and main TLB is referenced.

F.2 Translation Table Entry (TTE)

The size field of TTE is extended from 2bits to 3bits on SPARC64 VI to support over 4M pages. The MSB of the size is located at bit48 of TTE.

TABLE F-2 TSB and `TTE` Bit Description

Bits	Field Name	Description
Data <48, 62:61>	**size**	The page size of this entry, encoded as shown below. Size<2:0> Page Size 000 = 8 KB 001 = 64 KB 010 = 512 KB 011 = 4 MB 100 = 32 MB 101 = 256 MB
Data <46:13>	**PA**	The physical page number.

IMPL DEP. in Commonality TABLE F-1: TTE_Data bits 46:43 are implementation dependent.

On SPARC64 VI, TTE_Data bits 46:43 is used for PA<46:43>.

IMPL. DEP. #224: Physical address width support by the MMU is implementation dependent in JPS1; minimum PA width is 43 bits.

The SPARC64 VI MMU implements 47-bit physical addresses. The PA field of the TTE holds a 47-bit physical address. The MMU translates virtual addresses into 47-bit physical addresses. Each cache tag holds bits 46:6 of physical addresses.

IMPL. DEP. #238: When page offset bits for larger page size (PA<15:13>, PA<18:13>, and PA<21:13> for 64-Kbyte, 512-Kbyte, and 4-Mbyte, respectively) are stored in the TLB, it is implementation dependent whether the data returned from those fields by a Data Access read are zero or the data previously written to them.

On SPARC64 VI, the data returned from PA<15:13>, PA<18:13>, PA<21:13>, PA<24:13>, and PA<27:13> for 64-Kbyte, 512-Kbyte, 4-Mbyte, 32-Mbyte, and 256-Mbyte pages, respectively, by a Data Access read are the data previously written to them.

IMPL. DEP. #225: The mechanism by which entries in TLB are locked is implementation dependent in JPS1.

In SPARC64 VI, when a TTE with its lock bit set is written into TLB through the Data In register, the TTE is automatically written into the corresponding fully associative TLB and locked in the TLB. Otherwise, the TTE is written into the corresponding sTLB or fTLB, depending on its page size.

IMPL. DEP. #242: An implementation containing multiple TLBs may implement the L (lock) bit in all TLBs but is only required to implement a lock bit in one TLB for each page size. If the lock bit is not implemented in a particular TLB, it is read as 0 and writes to it are ignored.

In SPARC64 VI, only the fITLB and the fDTLB support the lock bit as described in TABLE F-1. The lock bit in sITLB and sDTLB is read as 0 and writes to it are ignored.

IMPL. DEP. #226: Whether the CV bit is supported in TTE is implementation dependent in JPS1. When the CV bit in TTE is not provided and the implementation has virtually indexed caches, the implementation should support hardware unaliasing for the caches.

In SPARC64 VI, no TLB supports the CV bit in TTE. SPARC64 VI supports hardware unaliasing for the caches. The CV bit in any TLB entry is read as 0 and writes to it are ignored.

F.3.3 TSB Organization

IMPL. DEP. #227: The maximum number of entries in a TSB is implementation dependent in JPS1. See impl. dep. #228 for the limitation of `TSB_size` in TSB registers.

SPARC64 VI supports a maximum of 16 million lines in the common TSB and a maximum 32 million lines in the split TSB. The maximum number *N* in FIGURE F-4 of **Commonality** is 16 million ($16 * 2^{20}$).

F.4.2 TSB Pointer Formation

IMPL. DEP. #228: Whether `TSB_Hash` is supplied from a TSB Extension Register or from a context-ID register is implementation dependent in JPS1. Only for cases of direct hash with context-ID can the width of the `TSB_size` field be wider than 3 bits.

On SPARC64 VI, `TSB_Hash` is supplied from a context-ID register. The width of the `TSB_size` field is 4 bits.

IMPL. DEP. #229: Whether the implementation generates the TSB Base address by exclusive-ORing the TSB Base Register and a TSB Extension Register or by taking the `TSB_Base` field directly from the TSB Extension Register is implementation dependent in JPS1. This implementation dependency is only to maintain compatibility with the TLB miss handling software of UltraSPARC I/II.

On SPARC64 VI, when `ASI_MCNTL.JPS1_TSBP` = 1, the TSB Base address is generated by taking `TSB_Base` field directly from the TSB Extension Register.

TSB Pointer Formation

On SPARC64 VI, the number N in the following equations ranges from 0 to 15; N is defined to be the `TSB_Size` field of the TSB Base or TSB Extension Register.

SPARC64 VI supports the TSB Base from TSB Extension Registers as follows when `ASI_MCNTL.JPS1_TSBP` = 1.

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

```
8K_POINTER = TSB_Extension[63:13+N] || (VA[21+N:13] ⊕ TSB_Hash) ||
0000

64K_POINTER = TSB_Extension[63:13+N] || (VA[24+N:16] ⊕ TSB_Hash) ||
0000
```

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

```
8K_POINTER = TSB_Extension[63:14+N] || 0 || (VA[21+N:13] ⊕ TSB_Hash)
|| 0000
```

```
64K_POINTER = TSB_Extension[63:14+N] || 1 || (VA[24+N:16] ⊕
TSB_Hash) || 0000
```

Value of TSB_Hash for both a shared TSB and a split TSB

When 0 <= N <= 4,

TSB_Hash = context_register[N+8:0]

Otherwise, when 5 <= N <= 15,

TSB_Hash[12:0] = context_register[12:0]

TSB_Hash[N+8:13] = 0 (N-4 bits zero)

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 *data_access_exception* trap.

SPARC64 VI signals a *data_access_exception* for the causes, as defined in F.5 in **Commonality**. However, caution is needed to deal with an invalid ASI. See Section F.10.9 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 VI, 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.

Additional information: On SPARC64 VI, the two precise traps—*instruction_access_error* and *data_access_error*—are recorded by the MMU in addition to those in TABLE F-2 of **Commonality**. A modification (the two traps are added) of that table is shown below.

TABLE F-3 MMU Trap Types, Causes, and Stored State Register Update Policy

			Registers Updated (Stored State in MMU)				
Ref #	Trap Name	Trap Cause	I-SFSR	I-MMU Tag Access	D-SFSR, SFAR	D-MMU Tag Access	Trap Type
1.	*fast_instruction_access_MMU_miss*	I-TLB miss	X2	X			64_{16}–67_{16}
2.	*instruction_access_exception*	Several (see below)	X2	X			08_{16}
3.	*fast_data_access_MMU_miss*	D-TLB miss			X3	X	68_{16}–$6B_{16}$
4.	*data_access_exception*	Several (see below)			X3	X1	30_{16}
5.	*fast_data_access_protection*	Protection violation			X3	X	$6C_{16}$-$6F_{16}$

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	Trap Type
6.	*privileged_action*	Use of privileged ASI			X3		37_{16}
7.	watchpoint	Watchpoint hit			X3		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)	X2				$0A_{16}$
10	*data_access_error*	Several (see below)			X3		32_{16}

- X1: The contents of the context field of the D-MMU Tag Access Register are undefined after a *data_access_exception*.
- X2: `I-SFSR` is updated according to its update policy described in Section F.10.9
- X3: `D-SFSR` and `D-SFAR` are updated according to the update policy described in Section F.10.9

The traps with Ref #1~8 in TABLE F-3 conform to the specification defined in Section F.5 of **Commonality**.

The additional traps (Ref #9 and #10) are described below.

Ref 9: *instruction_access_error* — Signalled upon detection of at least one of the following errors.

- An uncorrectable error is detected upon an instruction fetch reference.
- A bus error response from the Jupiter Bus is detected upon an instruction fetch reference.
- fITLB multiple hits are detected in a fITLB lookup for an instruction reference.
- An fITLB entry parity error is detected in an fTLB lookup for an instruction reference.

Ref 10: *data_access_error* — Signalled upon the detection of at least one of the following errors.

- An uncorrectable error is detected upon an instruction operand access.
- A bus error response from the Jupiter Bus is detected upon an operand access.
- fDTLB multiple hits are detected in an fDTLB lookup for an operand access.
- An fDTLB entry parity error is detected in a fDTLB lookup for an instruction operand access.

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 impl. dep. #224 on page 95 for the variability of the width of physical address. The physical address width to pass to the Jupiter Bus interface is 47 bits.

IMPL. DEP. #232: Whether `CP` and `CV` bits exist in the DCU Control Register is implementation dependent in JPS1.

On SPARC64 VI, `CP` and `CV` bits do not exist in the DCU Control Register.

When DMMU is disabled, the processor behaves as if the TTE bits were set as:

- `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.

On SPARC64 VI, the PREFETCH instruction completes without memory access when the DMMU is disabled.

A *data_access_exception* is generated at the execution of the nonfaulting load instruction when the DMMU is disabled, as defined in Section F.5 of **Commonality**.

F.10 Internal Registers and ASI Operations

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.

On SPARC64 VI, the `TSB_Hash` field is not implemented in the I/D Primary/Secondary/Nucleus TSB Extension Register. See *TSB Pointer Formation* on page 96 for details.

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* on page 105.

Additional information: The `ASI_DCUCR` register also affects the MMUs. `ASI_DCUCR` is described in Section 5.2.12 of **Commonality**. The SPARC64 VI implementation dependency in `ASI_DCUCR` is described in *Data Cache Unit Control Register (DCUCR)* on page 20.

SPARC64 VI also has an additional MMU internal register `ASI_MCNTL` (Memory Control Register) that is shared between the IMMU and the DMMU. The register is illustrated in FIGURE F-1 and described in TABLE F-4.

ASI_MCNTL (Memory Control Register)

ASI:45_{16}
VA:08_{16}
Access Modes:Supervisor read/write

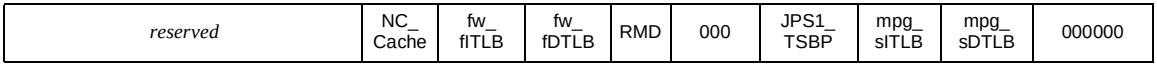


FIGURE F-1 Format of `ASI_MCNTL`

TABLE F-4 `MCNTL` Field Description

Bits	Field Name	RW	Description
Data <16>	**NC_Cache**	R/W	Force instruction caching. When set, the instruction lines fetched from a noncacheable area are cached in the instruction cache. The NC_Cache has no effect on operand references. If MCNTL.NC_Cache = 1, the CPU fetches a noncacheable line in four consecutive 16-byte fetches and stores the entire 64 bytes in the I-Cache. NC_Cache is provided for use by OBP, and OBP should clear the bit before exiting. A write to ASI_FLUSH_L1I must be performed before MCNTL.NC_CACHE = 0 is set. Otherwise, noncacheable instructions may remain in the L1 cache.
Data <15>	**fw_fITLB**	R/W	Force write to fITLB. This is the mITLB version of fTLB force write. When fw_fITLB = 1, a TTE write to mITLB through ITLB Data In Register is directed to fITLB. fw_fITLB is provided for use by OBP to register the TTEs that map the address translations themselves into fDTLB.
Data <14>	**fw_fDTLB**	R/W	Force write to fDTLB. When fw_fDTLB = 1, a TTE write to mDTLB through DTLB Data In Register is directed to fDTLB. fw_fDTLB is provided for use by OBP to register the TTEs that map the address translations themselves into fDTLB.

TABLE F-4 MCNTL Field Description

Data <13:12>	**RMD**	R	TLB RAM MODE. The value is always 2. This field is read-only and writes to this field are ignored.
Data <8>	**JPS1_TSBP**	R/W	TSB-pointer context-hashing enable. When JPS1_TSBP = 0, SPARC64 VI does not apply the context-ID hashing for 8-Kbyte or 64-Kbyte TSB pointer generation. The pointer generation strategy is compatible with UltraSPARC. When JPS1_TSBP = 1, SPARC64 VI is in JPS1_TSBP mode, meaning that the CPU applies the context-ID hashing to generate an 8-Kbyte or 64-Kbyte page TSB pointer.
Data<7>	**mpg_sITLB**	RW	This bit enables translating multiple page sizes on sITLBs. When this bit is set, page size fields in the context register are activated, and sITLB simultaneously have multiple page sizes dedicated for each context. When this bit is cleared, the page size field in the context register and the tag_access_ext register are ignored and default page sizes (8K for the first sTLB and 4M for the second) are used.
Data<6>	**mpg_sDTLB**	RW	This bit enables translating multiple page sizes on the sDTLB. When this bit is set, page size fields in the context register are activated, and sDTLB simultaneously have multiple page sizes dedicated for each context. When this bit is cleared, page size field in the context register and the tag_access_ext register are ignored and default page sizes (8K for the first sTLB and 4M for the second) are used.

Setting "10" into mpg_sITLB and mpg_sDTLB is not allowed. SPARC64 VI behavior is undefined with this setting.

F.10.2 Context Registers

sTLBs consists of two parts, where the first sTLB is 1024-entry two-way associative and the second sTLB is 1024 entry two-way associative. Normally the first sTLB holds 8KB pages and the second sTLB holds 4M pages for translations. But software can program sTLBs to be used for 8 KB, 64 KB, 512 KB, 4 MB, 32MB and 256MB page translations, by setting MCNTL#mpg_sTLB. Each sTLB can hold any of the 6 page sizes, but are programmed to only one page size at any given time. Each sTLB can be programmed to either the same or different page sizes.

Each sTLB page size (`PgSz`) is programmable independently, one PgSz per context (Primary/ Secondary/ Nucleus). PgSz specified Kernel can set the PgSz fields in ASI_PRIMARY_CONTEXT_REG and ASI_SECONDARY_CONTEXT_REG. PgSz specified in ASI_PRIMARY_CONTEXT_REG are used for both sITLBs and sDTLBs. When both sDTLBs are programmed to have identical page size, the behavior is a "single" 4-way 2048-entry sDTLB.

The following is the page size bit encoding:

- 000 = 8 KB
- 001 = 64 KB
- 010 = 512 KB
- 011 = 4 MB
- 100 = 32 MB
- 101 = 256 MB

Note – SPARC64 VI behavior with undefined page size (110,111) is undefined.

ASI_PRIMARY_CONTEXT

ASI: 58_{16}
VA: 08_{16}
Access Modes: Supervisor read/write

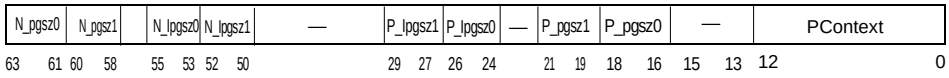


FIGURE F-2 IMMU and DMMU Primary Context Register

TABLE F-5 IMMU and DMMU Primary Context Register

Bit	Field	Type	Description
63:61	N_pgsz0	RW	Nucleus context's page size at the first sDTLB
60:58	N_pgsz1	RW	Nucleus context's page size at the second sDTLB
55:53	N_Ipgsz0	RW	Nucleus context's page size at the first sITLB
52:50	N_Ipgsz1	RW	Nucleus context's page size at the second sITLB
29:27	P_Ipgsz1	RW	Primary context's page size at the second sITLB
26:24	P_Ipgsz0	RW	Primary context's page size at the first sITLB
21:19	P_pgsz1	RW	Primary context's page size at the second sDTLB
18:16	P_pgsz0	RW	Primary context's page size at the first sDTLB
12:0	`PContext`	RW	Primary context

The value written to any of PgSz fields can be read regardless of MCNTL.mpg_sITLB/ mpg_sDTLB setting.

ASI_SECONDARY_CONTEXT

ASI: 58_{16}
VA: 10_{16}
Access Modes:Supervisor read/write

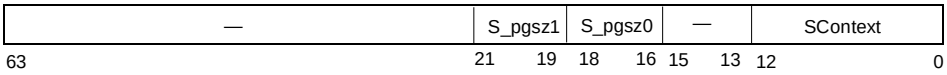


FIGURE F-3 DMMU Secondary Context Register

TABLE F-6 DMMU Secondary Context Register

Bit	Field	Type	Description
21:19	S_pgsz1	RW	Secondary context's page size at the second sDTLB.
18:16	S_pgsz0	RW	Secondary context's page size at the first sDTLB.
12:0	`SContext`	RW	Secondary context

The value written to any of PgSz fields can be read regardless of MCNTL.mpg_sITLB/ mpg_sDTLB setting.

F.10.3 Instruction/Data MMU TLB Tag Access Registers

When the MMU signals a trap due to a miss, exception, or protection, hardware automatically saves the missing VA and context to the Tag Access Register (`ASI_I/DMMU_TAG_ACCESS`). To ease indexing of the sTLBs when the TTE data is presented (via STXA `ASI_I/DTLB_DATA_IN_REG`), the missing page size information of sTLBs is captured into a new Extension Register, called `ASI_I/DMMU_TAG_ACCESS_EXT`.

Note – If SIZE of TTE to be written is different from PgSz of the ASI register, the TTE is written into fTLB rather than sTLB.

The `ASI_I/DMMU_TAG_ACCESS_EXT` register value on an *instruction/data_access_exception* is not valid (undefined).

The register values are not valid (undefined) when corresponding `ASI_MCNCTL#mpg_sI/DTLB` value is zero.

ASI_I/DMMU_TAG_ACCESS_EXT

ASI:50_{16}(IMMU) / 58_{16}(DMMU)
VA:60_{16}
Access Modes:Supervisor read/write

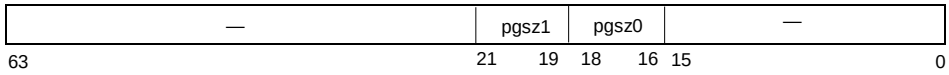


FIGURE F-4 I/D MMU Tag access extension Register

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.

For fTLB, SPARC64 VI implements a pseudo-LRU. For sTLB, LRU is used.

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 on SPARC64 VI are shown in TABLE F-7.

TABLE F-7 MMU TLB Data Access Address Assignment

VA Bit	Field	Description
17:16	**TLB#**	TLB to be accessed: fTLB or sTLB is designated as follows. 00: fTLB (32 entries) 01: reserved 10: sTLB(2048 entries of 8-Kbyte page and 4-Mbyte page) 11: reserved
15	**ER**	Error insertion into mTLB: When set on a write, an entry with parity error is inserted into a selected TLB location. This field is ignored for a TLB entry read operation.
13:3	TLB index	Index number of the TLB. Specifies an index number for the TLB reference. When fTLB is specified in TLB# field, the upper 6-bits of the specified index are ignored. When sTLB is specified in TLB# field, Index 0-511 addresses way0 of 8K-byte page sTLB Index 512-1023 addresses way1 of 8K-byte page sTLB Index 1024-1535 addresses way0 of 4M-byte page sTLB Index 1536-2047 addresses way1 of 4M-byte page sTLB When the entry to be written has a lock bit set and the specified TLB is the sTLB, the entry is written into the sTLB with its lock bit cleared. When the entry to be written into the fTLB, the entry is written without lock bit modification.
Other	*Reserved*	Ignored.

sTLB index hash

When sTLB is referenced by instruction fetch or operand access, indexing sTLB uses hashed VA and CONTEXT as follows. Notice that the bits taken from CONTEXT is inversed.

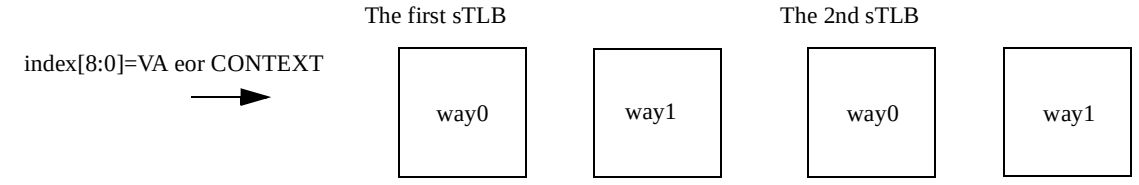
- 8 KB page : index[8:0] = VA[21:13] eor CONTEXT[0:8]
- 64 KB page : index[8:0] = VA[24:16] eor CONTEXT[0:8]
- 512 KB page: index[8:0] = VA[27:19] eor CONTEXT[0:8]
- 4 MB page : index[8:0] = VA[30:22] eor CONTEXT[0:8]
- 32MB page : index[8:0] = VA[33:25] eor CONTEXT[0:8]
- 256MB page : index[8:0] = VA[36:28] eor CONTEXT[0:8]

The hash function is not applied to an access with TLB Data Access Register.

Note – To avoid use of hash function, pages with TTE#G = 1 is always written into fTLB on TLB Data In.

---

sTLB reference instruction fetch and operand access



sTLB Data Access

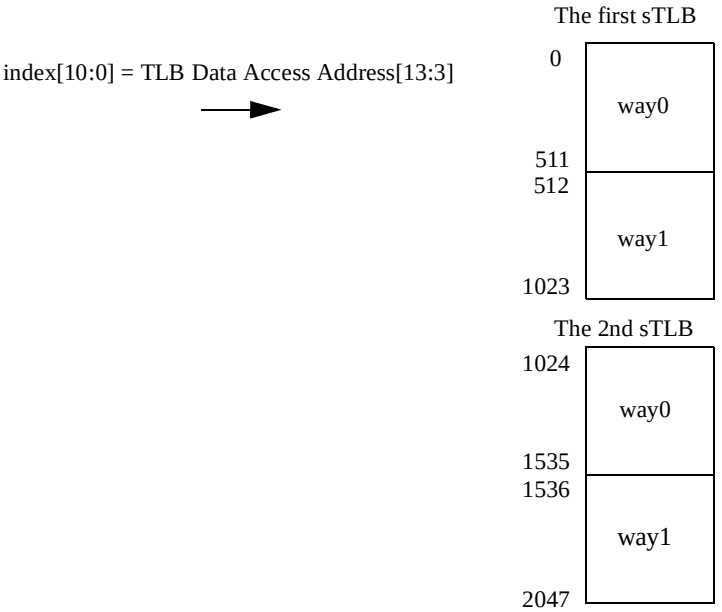


FIGURE F-5 Indexing sTLB

I/D MMU TLB Tag Access Register

On an ASI store to the TLB Data Access or Data In Register, SPARC64 VI verifies the consistency between the Tag Access Register and the data to be written. If their indexes are inconsistent, the TLB entry is not updated. However, SPARC64 VI does not verify the

consistency if TTE.V = 0 for the TTE to be written. This enables demapping of specified TLB entries through the TLB Data Access Register. Software can use this feature to validate faulty TLB entries.

On verifying the consistency, the bits position and length that is interpreted as index against the data in Tag Access Register varies on the page size and MCNTL.RMD. For example, bits[21:13] of VA in 8KB page, or bits[30:22] of VA n 4MB page, is considered as index and compared with the index field of TLB Data Access or Data In Register.

I/D TSB Base Registers

IMPL. DEP. #236: The width of the TSB_Size field in the TSB Base Register is implementation dependent; the permitted range is from 2 to 6 bits. The least significant bit of TSB_Size is always at bit 0 of the TSB Base Register. Any bits unimplemented at the most significant end of TSB_Size read as 0, and writes to them are ignored.

On SPARC64 VI, the width of the TSB_Size field in the TSB Base Register is 4 bits. The number of entries in the TSB ranges from 512 entries at TSB_Size = 0 (8 Kbyte for common TSB, 16 Kbyte for split TSB), to 16 million entries at TSB_Size = 15 (256 Mbyte for common TSB, 512 Mbyte for split TSB).

F.10.7 I/D TSB Extension Registers

IMPL DEP. in Commonality FIGURE F-13: Bits 11:3 in I/D TSB Extension Register are an implementation-dependent field.

On SPARC64 VI, bits 11:0 in I/D TSB Extension Registers are assigned as follows.

- Bits 11:4 — Reserved. Always read as 0, and writes to it are ignored.
- Bits 3:0 — TSB_Size field is expanded to be a 4-bit field in SPARC64 VI.

F.10.9 I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR)

IMPL DEP. in Commonality FIGURE F-15 and TABLE F-12: Bits 63:25 in I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR) are an implementation-dependent field.

The format of I/D-MMU SFSR in SPARC64 VI is shown in FIGURE F-6.

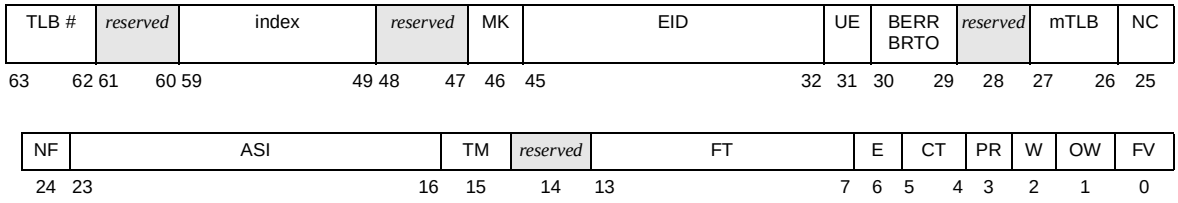


FIGURE F-6 MMU I/D Synchronous Fault Status Registers (I-SFSR, D-SFSR)

The specification of bits 24:0 in the SPARC64 VI SFSR conforms to the specification defined in Section F.10.9 in **Commonality**. Bits 63:25 in SPARC64 VI SFSR are implementation dependent. TABLE F-8 describes the I-SFSR bits, and TABLE F-8 describes the D-SFSR bits.

TABLE F-8 I-SFSR Bit Description

Bits	Field Name	RW	Description
Data<63:62 >	**TLB#**	R/W	Faulty TLB# log. Recorded upon an mITLB error to identify the faulty TLB (fITLB: 00_2 or sITLB: 10_2). The priority of error logging for multiple error conditions (parity error and multiple-hit error) is as follows: fTLB parity high sTLB parity sTLB multihit fTLB multihit low
Data <59:49>	**index**	R/W	Faulty TLB index log. Recorded upon an mITLB error and is the index number for the faulty TLB. The priority of error logging for multiple error conditions (parity error and multiple-hit error) is as follows: fTLB parity high sTLB parity sTLB multihit fTLB multihit low On multiple hit error, any one of the index numbers is shown.
Data <46>	**MK**	R/W	Marked UE. On SPARC64 VI, all uncorrectable errors are reported as marked, so this bit is always set whenever ISFSR.UE = 1. See Section P.2.4, *Error Marking for Cacheable Data Error*, on page 164 for details.
Data <45:32>	**EID**	R/W	Error mark ID. Valid for a marked UE. See Section P.2.4, *Error Marking for Cacheable Data Error*, on page 164 for ERROR_MARK_ID.
Data <31>	**UE**	R/W	Instruction error status; uncorrectable error. When UE = 1, an uncorrectable error in a fetched instruction word has been detected. Valid only for an *instruction_access_error* exception.
Data <30>	**BERR**	RW	Bus error response has been received from an instruction fetch transaction. Valid only for a *instruction_access_error* exception.

TABLE F-8 I-SFSR Bit Description

Bits	Field Name	RW	Description
Data<29>	**BRTO**	RW	Bus time-out response has been received from an instruction fetch transaction. Valid only for a *instruction_access_error* exception.
Data <27:26>	**mITLB**<1:0>	R/W	mITLB error status. Either a multiple-hit status (mITLB<1>) or a parity error status (mITLB<0>) has been encountered upon a mITLB lookup. Valid only for an *instruction_access_error* exception.
Data <25>	**NC**	R/W	Noncacheable reference. The reference that has invoked an exception is a noncacheable reference. Valid for an *instruction_access_error* exception caused by ISFSR.UE, ISFSR.BERR, or ISFSR.BRTO only. For other causes of the trap, the value is unknown.
Data <23:16>	**ASI**<7:0>	R/W	ASI. The 8-bit address space identifier applied to the reference that has invoked an exception. This field is valid for the exception in which the ISFSR.FV bit is set. A recorded ASI is 80_{16}(ASI_PRIMARY) or 04_{16} (ASI_NUCLEUS) depending on the trap level (when TL > 0, the ASI is ASI_NUCLEUS.).
Data <15>	**TM**	R/W	Translation miss. When TM = 1, it signifies an occurrence of a mITLB miss upon an instruction reference.
Data <13:7>	**FT**<6:0>	R/W	Fault type. Saves and indicates an exact condition that caused the recorded exception. See TABLE F-9 for the field encoding. In the IMMU, FT is valid only for an *instruction_access_exception*. The ISFSR.FT always reads as 0 for a *fast_instruction_access_MMU_miss* and reads 01_{16} for an *instruction_access_exception*, since no other fault types apply.
Data <5:4>	**CT**<1:0>	R/W	Context type; Saves the context attribute for the reference that invokes an exception. For nontranslating ASI or invalid ASI, ISFSR.CT = 11_{02}. 00_{02}: Primary 01_{02}: Reserved 10_{02}: Nucleus 11_{02}: Reserved
Data <3>	**PR**	R/W	Privileged. Indicates the CPU privilege status during the instruction reference that generates the exception. This field is valid when ISFSR.FV = 1.
Data <1>	**OW**	R/W	Overwritten. Set when ISFSR.FV = 1 upon the detection of a exception. This means that the fault valid bit is not yet cleared when another fault is detected.
Data <0>	**FV**	R/W	Fault valid. Set when the IMMU detects an exception. The bit is not set on an IMMU miss. When the Fault Valid bit is not set, the values of the remaining fields in the ISFSR are undefined, except for an IMMU miss.

TABLE F-9 describes the field encoding for ISFSR.FT.

TABLE F-9 Instruction Synchronous Fault Status Register FT (Fault Type) Field

FT<6:0>	Error Description
01_{16}	Privilege violation. Set when TTE.P = 1 and PSTATE.PRIV = 0 for the instruction reference.
02_{16}	*Reserved*
04_{16}	*Reserved*
08_{16}	*Reserved*

TABLE F-9 Instruction Synchronous Fault Status Register FT (Fault Type) Field

FT<6:0>	Error Description
10_{16}	*Reserved*
20_{16}	*Reserved,* since there is no virtual hole.
40_{16}	*Reserved,* since there is no virtual hole.

ISFSR is updated either upon a occurrence of a *fast_instruction_access_MMU_miss*, an *instruction_access_exception*, or an *instruction_access_error* trap. TABLE F-10 shows the detailed update policy of each field, and TABLE F-11 describes the fields.

TABLE F-10 ISFSR Update Policy

	Field	TLB#, index	FV	OW	PR, CT[1]	FT	TM	ASI	UE, BERR, BRTO, mITLB, NC[2]
	Fresh fault or miss[3]								
Miss	MMU miss	—	0	0	V	—	1	—	—
Exception	Access exception	—	1	0	V	V	0	V	—
Error	Access error	V[4]	1	0	V	—	0	V	V
Overwrite policy[5]									
Error on exception		U[4]	1	1	U	K	K	U	U
Exception on error		K	1	1	U	U	K	U	K
Error on miss		U	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 when the ASI is not a translating ASI. The value 11 is recorded in ISFSR.CT for an illegal value in the ASI (00_{16}–03_{16}, 12_{16}–13_{16}, 16_{16}–17_{16}, $1A_{16}$–$1B_{16}$, $1E_{16}$–23_{16}, $2D_{16}$–$2F_{16}$, and 35_{16}–$3B_{16}$).

2.Valid only for the *instruction_access_error* caused by ISFSR.UE, ISFSR.BERR, or ISFSR.BRTO.

3.Types: 0 – logical 0; 1 –logical 1; V– Valid field to be updated; "—" – not a valid field

4.Updated when mITLB is signified.

5.Types: 0 – logical 0; 1 – logical 1; K – keep; U – Update as per fault/miss

TABLE F-11 D-SFSR Bit Description *(1 of 3)*

Bits	Field Name	RW	Description
Data <63:62>	**TLB#**	R/W	Faulty TLB# log. Recorded upon an mDTLB error to identify the faulty TLB (fDTLB: 00_2 or sDTLB: 10_2). The priority of error logging for multiple error conditions (parity error and multiple-hit error) is as follows: fTLB parity high sTLB parity sTLB multihit fTLB multihit low

TABLE F-11 D-SFSR Bit Description *(2 of 3)*

Bits	Field Name	RW	Description
Data <59:49>	**index**	R/W	Faulty TLB index log. Recorded upon an mDTLB error. Index number for the faulty TLB. The priority of error logging for multiple error conditions (parity error and multiple-hit error) is as follows: fTLB parity high sTLB parity sTLB multihit fTLB-multihit low On multiple hit error, any one of the index numbers is shown.
Data <46>	**MK**	R/W	Marked UE. On SPARC64 VI, all uncorrectable errors are reported as marked, so this bit is always set whenever DSFSR.UE = 1. See Section P.2.4, *Error Marking for Cacheable Data Error*, on page 164 for details.
Data <45:32>	**EID**	R/W	Error-mark ID. Valid for a marked UE. See Section P.2.4, *Error Marking for Cacheable Data Error*, on page 164 for details about ERROR_MARK_ID.
Data <31>	**UE**	R/W	Operand access error status. Uncorrectable error. When UE = 1, it signifies an occurrence of an uncorrectable error in an operand fetch reference. Valid only for a *data_access_error* exception.
Data <30>	**BERR**	RW	Bus error response has been received from an operand fetch transaction. Valid only for a *data_access_error* exception.
Data<29>	**BRTO**	RW	Bus time-out response has been received from an operand fetch transaction. Valid only for a *data_access_error* exception.
Data <27:26>	**mDTLB**<1:0>	R/W	mDTLB error status. Either a multiple-hit status (mDTLB<1>) or a parity error status (mDTLB<0>) has been encountered upon a mDTLB lookup. Valid only for a *data_access_error* exception.
Data <25>	**NC**	R/W	Noncacheable reference. The reference that invoked an exception is a non-cacheable reference. This field indicates that the faulty reference is a non-cacheable operand access. Valid only for an *data_access_error* exception caused by DSFSR.UE, DSFSR.BERR, or DSFSR.BRTO. For other causes of the trap, the value is unknown.
Data <24>	**NF**	R/W	Nonfaulting load. The instruction which generated the exception was a nonfaulting load instruction.
Data <23:16>	**ASI**<7:0>	R/W	ASI. The 8-bit address space identifier applied to the reference that has invoked an exception. This field is valid for the exception in which the DSFSR.FV bit is set. When the reference does not specify an ASI, the reference is regarded as with an implicit ASI and a recorded ASI is as follows: 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)
Data <15>	**TM**	R/W	Translation miss. When TM = 1, it signifies an occurrence of a mDTLB miss upon an operand reference.
Data <13:7>	**FT**<6:0>	R/W	Fault type. Saves and indicates an exact condition that caused the recorded exception. The encoding of this field is described in TABLE F-12.

TABLE F-11 D-SFSR Bit Description *(3 of 3)*

Bits	Field Name	RW	Description
Data <6>	**E**	R/W	Side-effect page. Associated with faulting data access. The reference is mapped to the translation with an E bit set, or the ASI for the reference was either 015_{16} or $01D_{16}$. Valid only for an *data_access_error* exception caused by DSFSR.UE, DSFSR.BERR, or DSFSR.BRTO. For other causes of the trap, the value is unknown.
Data	**CT**<1:0>	R/W	Context type. Saves the context attribute for the reference that invokes an exception. For nontranslating ASI or invalid ASI, DSFSR.CT = 11_{02}. 00_{02}: Primary 01_{02}: Secondary 10_{02}: Nucleus 11_{02}: Reserved When a *data_access_exception* trap is caused by an invalid combination of an ASI and an opcode (e.g., atomic load quad, block load/store, block commit store, partial store, or short floating-point load/store instructions), the recording of the DSFSR.CT field is based on the encoding of the ASI specified by the instruction.
Data <3>	**PR**	R/W	Privileged. Indicates the CPU privilege status during the operand reference that generates the exception. This field is valid when DSFSR.FV = 1.
Data <2>	**W**	R/W	Write. W = 1 if the reference is for an operand write operation (a store or atomic load/store instruction).
Data <1>	**OW**	R/W	Overwritten. Set when DSFSR.FV = 1 upon detection of a exception. This means that the fault valid bit is not yet cleared when another fault is detected.
Data <0>	**FV**	R/W	Fault valid. Set when the DMMU detects an exception. The bit is not set on a DMMU miss. When the FV bit is not set, the values of the remaining fields in the DSFSR and DSFAR are undefined, except for a DMMU miss.

TABLE F-12 defines the encoding of the FT<6:0> field.

TABLE F-12 MMU Synchronous Fault Status Register FT (Fault Type) Field

FT<6:0>	Error Description
01_{16}	Privilege violation. An attempt was made to access a privileged page (TTE.P = 1) under nonprivileged mode (PSTATE.PRIV = 0) or through a *_AS_IF_USER ASI. This exception has priority over a *fast_data_access_protection* exception.
02_{16}	Nonfaulting load instruction to page marked with the E bit. This bit is zero for internal ASI accesses.
04_{16}	An attempt was made to access a noncacheable page or an internal ASI by an atomic instruction (CASA, CASXA, SWAP, SWAPA, LDSTUB, LDSTUBA) or an atomic quad load instruction (LDDA with ASI = 024_{16}, $02C_{16}$, 034_{16}, or $03C_{16}$).

TABLE F-12 MMU Synchronous Fault Status Register FT (Fault Type) Field *(Continued)*

FT<6:0>	Error Description
08_{16}	An attempt was made to access an alternate address space with an illegal ASI value, an illegal VA, an invalid read/write attribute, or an illegally sized operand. If the quad load ASI is used with the other opcode than LDDA, this bit is set. **Note:** Since an illegal ASI check is done prior to a TTE unmatch check, DSFSR.FT<3> = 1 causes the value of other bits of DSFSR.FT to be undetermined and generates a *data_access_exception* exception (which otherwise has lower priority than *fast_data_access_MMU_miss*). Note, too, that a reference to an internal ASI may generate a *mem_address_not_aligned* exception.
10_{16}	Access other than nonfaulting load was made to a page marked NFO. This bit is zero for internal ASI accesses.
20_{16}	*Reserved*, since there is no virtual hole.
40_{16}	*Reserved*, since there is no virtual hole.

Multiple bits of DSFSR.FT may be set by a trap as long as the cause of the trap matches multiply in TABLE F-12.

DSFSR is updated upon various traps, including *fast_data_access_MMU_miss*, *data_access_exception*, *fast_data_access_protection*, *PA_watchpoint*, *VA_watchpoint*, *privileged_action*, *mem_address_not_aligned*, and *data_access_error* traps. TABLE F-13 shows the detailed update policy of each field.

TABLE F-13 DSFSR Update Policy

	Field	TLB#, index	FV	OW	W, PR, NF, CT[1]	FT	TM	ASI	UE, BERR, BRTO, mDTLB, NC[2], E[2]	DSFAR
	Fresh fault or miss[3]									
Miss	MMU miss	—	0	0	V	—	1	—	—	V
Exception	Access exception	—	1	0	V	V	0	V	—	V
Faults	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[4]	—	1	0	V	—	0	V	—	V
	Access misaligned	—	1	0	V	—	0	V	—-	V
	Access error	V[5]	1	0	V	—	0	V	V	V
Overwrite Policy[6]										
Exception on fault		K	1	1	U	U	K	U	K	U
Fault on exception		U[4]	1	1	U	K	K	U	U	U
Exception on miss[7]		K	1	K	U	U	1	U	K	U
Fault on miss		U[4]	1	K	U	K	1	U	U	U

TABLE F-13 DSFSR 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 when the ASI is not a translating ASI. The value 11 is recorded in DSFSR.CT for an illegal value in ASI (00_{16}–03_{16}, 12_{16}–13_{16}, 16_{16}–17_{16}, $1A_{16}$–$1B_{16}$, $1E_{16}$–23_{16}, $2D_{16}$–$2F_{16}$, or 35_{16}–$3B_{16}$).

2.Valid only for the *data_access_error* caused by DSFSR.UE,or DSFSR.BERR, or DSFSR.BRTO.

3.Types: 0 – logic 0; 1 – logic 1; V – Valid field to be updated; "—" – not a valid field

4.Memory reference instruction only.

5.Updated when mDTLB is signified.

6.Types: 0 – logic 0; 1 – logic 1; V– Valid field to be updated; "—" – not a valid field

7.Fault/exception on miss means the miss happened first, then a fault/exception was encountered before software had a chance to clear the DSFSR register.

F.10.12 Synchronous Fault Physical Addresses

This section describes how the IMMU and DMMU obtain a fault physical address.

IMMU Synchronous Fault Physical Address

The Instruction Synchronous Fault Physical Address Register is newly added to capture physical memory address of the fault recorded in the IMMU Synchronous Fault Status Register (I-SFSR). The registers are updated on *instruction_access_error* exception, while the value is valid only when corresponding I-SFSR#MK=1, I-SFSR#UE=1, I-SFSR#BERR=1 or I-SFSR#BRTO=1.

The value of bit2:0 is undefined.

ASI:50_{16}
VA:78_{16}
Access Modes:Supervisor read/write

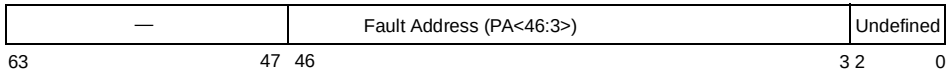


FIGURE F-7 MMU Instruction Synchronous Fault Physical Address Register (I-SFPAR)

DMMU Synchronous Fault Physical Address

The Data Synchronous Fault Physical Address Register is newly added to capture physical memory address of the fault recorded in the DMMU Synchronous Fault Status Register (D-SFSR). The registers are updated on *data_access_error* exception, while the value is valid only when corresponding D-SFSR#MK=1, D-SFSR#UE=1, D-SFSR#BERR=1 or D-SFSR#BRTO=1.

The value of bit2:0 is undefined.

ASI:58_{16}
VA:78_{16}
Access Modes:Supervisor read/write

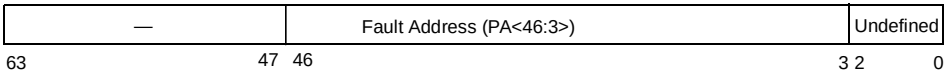


FIGURE F-8 MMU Data Synchronous Fault Physical Address Register (D-SFPAR)

F.11 MMU Bypass

On SPARC64 VI, two additional ASIs are supported as DMMU bypass accesses:
ASI_ATOMIC_QUAD_LDD_PHYS (ASI 34_{16}) and
ASI_ATOMIC_QUAD_LDD_PHYS_LITTLE (ASI $3C_{16}$)

TABLE F-14 shows the bypass attribute bits on SPARC64 VI. The first four rows conform to the bypass attribute bits defined in TABLE F-15 of **Commonality**.

TABLE F-14 Bypass Attribute Bits on SPARC64 VI

ASI	ASI	Attribute Bits							
NAME	VALUE	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

Unlike other software visible resources, thread0 and thread1 within the same core logically shares fTLBs and sTLBs. That is, a TLB entry written by one thread can be referenced by the other thread.

Note – Threads belonging to different physical core do not share TLBs.

If two identical TTEs are written, no multiple-hit error is detected during a virtual address translation. One of the two TTEs is used for the translation, instead. In other words, it is allowed for both the threads to write identical contents into a TLB independently. Hardware guarantees no multi-hit error will occur in this case.

However, it is not allowed to write two different TTEs with the same VA and CONTEXT into a TLB. This might result in a multi-hit error.

F.12.2 TLB Replacement Policy

Automatic TLB Replacement Rule

On an automatic replacement write to the TLB, the MMU picks the entry to write according to the following rules:

1. If the following conditions are satisfied—
   - the new entry unlocked, TTE.G = 0 page
   - and page size is either 8KB or 4MB when `ASI_MCNTL.mpg_sITLB/mpg_sDTLB = 0`,
     or page size matches either pgsz0/1 field of the relevant CONTEXT register when `ASI_MCNTL.mpg_sITLB/mpg_sDTLB = 1`
   - and `ASI_MCNTRL.fw_fITLB` = 0 for IMMU automatic replacement
   - and `ASI_MCNTRL.fw_fDTLB` = 0 for DMMU automatic replacement

   —then the replacement is directed to the sTLB (2-way TLB). Otherwise, the replacement occurs in the fully associative TLB (fTLB).
2. If replacement is directed to the 2-way TLB, then the replacement set index is generated from the TLB Tag Access Register with bits determined by the page size, then exclusive-or'ed with inversed CONTEXT register value to hash on both I-MMU and D-MMU.
3. If replacement is directed to the fully associative TLB (fTLB), then the following alternatives are evaluated:

a. The first invalid entry is replaced (measuring from entry 0). If there is no invalid entry, then

b. the first unused, unlocked (LRU, but clear) entry will be replaced (measuring from entry 0). If there is no unused unlocked entry, then

c. all used bits are reset, and the process is repeated from Step 3b.

If fTLB is the target of the automatic replacement and all entries in the fTLB have their lock bit set, the automatic replacement operation is ignored and the entries in the target fTLB remain unchanged.

Restriction of sTLB Entry Direct Replacement

In SPARC64 VI, no restriction check is applied to the stxa address and the contents of I/D TLB Data Access Register.

F.12.4 Instruction/Data MMU TLB Tag Access Registers

Demap Operations

For Demap Page in sTLBs, the page size used to index sTLBs is derived based on the Context bits (Primary/Secondary/Nucleus). Hardware will automatically select proper `PgSz` bits based on the "context" field (Primary/Secondary/Nucleus) defined in ASI_I/DMMU_DEMAP. These two `PgSz` fields are used to properly index the first sTLB and the second sTLB.

In addition, the selected PgSz based on the context bits is used to check if the demap operation is valid or not for Demap Page and Demap Context operations with sTLBs. That is, if the PgSz is different from SIZE of the corresponding TLB entry, the TLB entry will not be demapped.

Note – Valid context ID should be specified on Demap Page and Context operations. Demap operation with non-existing Context ID (01_2 for IMMU and 11_2 for IMMU/DMMU) might demap unexpected sTLB entries.

Demap All operations with sTLBs are straight forward.

F.APPENDIX **G**

Assembly Language Syntax

Please refer to Appendix G of ***Commonality***.

F.APPENDIX **H**

Software Considerations

Please refer to Appendix H of ***Commonality***.

F.APPENDIX **I**

Extending the SPARC V9 Architecture

Please refer to Appendix I of ***Commonality***.

F.APPENDIX **J**

Changes from SPARC V8 to SPARC V9

Please refer to Appendix J of ***Commonality***.

F.APPENDIX **K**

Programming with the Memory Models

Please refer to Appendix K of ***Commonality***.

F.APPENDIX **L**

Address Space Identifiers

Every load or store address in a SPARC V9 processor has an 8-bit Address Space Identifier (ASI) appended to the VA. The VA plus the ASI fully specifies the address. For instruction loads and for data loads or stores that do not use the load or store alternate instructions, the ASI is an implicit ASI generated by the hardware. If a load alternate or store alternate instruction is used, the value of the ASI can be specified in the `%asi` register or as an immediate value in the instruction. In practice, ASIs are not only used to differentiate address spaces but are also used for other functions, such as referencing registers in the MMU unit.

This chapter summarizes SPARC64 VI enhanced ASIs. Please refer to **Commonality** for Sections L.1 and L.2.

L.3 SPARC64 VI ASI Assignments

For SPARC64 VI, all accesses made with ASI values in the range 00_{16}–$7F_{16}$ when `PSTATE.PRIV` = 0 cause a *privileged_action* exception.

Warning – The software should follow the ASI assignments and VA assignments in TABLE L-1. Some illegal ASI or VA accesses will cause the machine to enter unknown states.

TABLE L-1 SPARC64 VI ASI Assignments *(1 of 3)*

Value	ASI Name (Suggested Macro Syntax)	Type	VA_{16}	Description	Page
00_{16}–33_{16}	(JPS1)				
34_{16}	`ASI_ATOMIC_QUAD_LDD_PHYS`	R	—		61
35_{16}–$3B_{16}$	(JPS1)				
$3C_{16}$	`ASI_ATOMIC_QUAD_LDD_PHYS_LITTLE`	R	—		61
$3D_{16}$–44_{16}	(JPS1)				

TABLE L-1 SPARC64 VI ASI Assignments *(2 of 3)*

Value	ASI Name (Suggested Macro Syntax)	Type	VA_{16}	Description	Page
45_{16}	ASI_DCU_CONTROL_REG (ASI_DCUCR)	RW	00		20
45_{16}	ASI_MEMORY_CONTROL_REG (ASI_MCNTL)	RW	08		100
46_{16}–49_{16}	(JPS1)				
$4A_{16}$	ASI_JB_CONFIG_REGISTER	RW	00		217
$4B_{16}$	(JPS1)				
$4C_{16}$	ASI_ASYNC_FAULT_STATUS	RW	00		108
$4C_{16}$	ASI_URGENT_ERROR_STATUS (ASI_UGESR)	R	08		171
$4C_{16}$	ASI_ERROR_CONTROL	RW	10		167
$4C_{16}$	ASI_STCHG_ERROR_INFO	RW	18		169
$4D_{16}$	ASI_ASYNC_FAULT_ADDR_D1	R	00	Always read as zero	180
$4D_{16}$	ASI_ASYNC_FAULT_ADDR_U2	R	08	Always read as zero	181
$4E_{16}$	(JPS1)				
$4F_{16}$	ASI_SCRATCH_REG0	RW	00		127
$4F_{16}$	ASI_SCRATCH_REG1	RW	08		127
$4F_{16}$	ASI_SCRATCH_REG2	RW	10		127
$4F_{16}$	ASI_SCRATCH_REG3	RW	18		127
$4F_{16}$	ASI_SCRATCH_REG4	RW	20		127
$4F_{16}$	ASI_SCRATCH_REG5	RW	28		127
$4F_{16}$	ASI_SCRATCH_REG6	RW	30		127
$4F_{16}$	ASI_SCRATCH_REG7	RW	38		127
50_{16}	(JPS1)		00-58		
50_{16}	ASI_IMMU_TAG_ACCESS_EXT	RW	60		104
50_{16}	ASI_IMMU_SFPAR	RW	78		115
51_{16}–57_{16}	(JPS1)				
58_{16}	ASI_DMMU_TAG_ACCESS_EXT	RW	60		104
58_{16}	ASI_DMMU_SFPAR	RW	78		115
59_{16}–66_{16}	(JPS1)				
67_{16}	ASI_FLUSH_L1I	W	—		133
68_{16}–69_{16}	(JPS1)				
$6A_{16}$	ASI_L2_CTRL	RW	—		134
$6B_{16}$	ASI_L2_DIAG_TAG_READ	R	00_{16}-$7FFC0_{16}$		134
$6C_{16}$	ASI_L2_DIAG_TAG_READ_REG	R	TBD		135

TABLE L-1 SPARC64 VI ASI Assignments *(3 of 3)*

Value	ASI Name (Suggested Macro Syntax)	Type	VA_{16}	Description	Page
$6D_{16}$	(JPS1)				
$6E_{16}$	ASI_ERROR_IDENT (ASI_EIDR)	RW	00		167
$6F_{16}$–73_{16}	(JPS1)				
74_{16}	ASI_CACHE_INV	W	—		135
75_{16}–FF_{16}	(JPS1)				

L.3.2 Special Memory Access ASIs

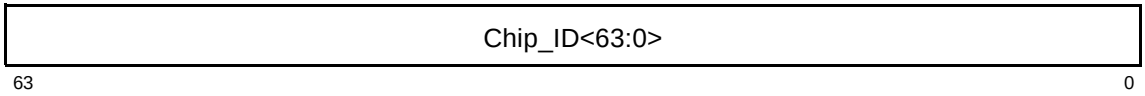
Please refer to Section L.3.3 in **Commonality**.

In addition to the ASIs described in **Commonality**, SPARC64 VI supports the ASIs described below.

ASI 53_{16} (ASI_SERIAL_ID)

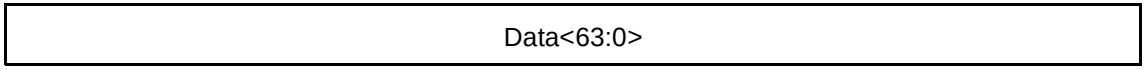
SPARC64 VI provides an identification code for each processor. In other words, this ID is unique for each processor chip. In conjunction with the Version Register (please refer to *Version (VER) Register* on page 18), software can attain completely unique chip identification code.

This register is defined as read-only. A write to this register causes *data_access_error.*



ASI $4F_{16}$ (ASI_SCRATCH_REGx)

SPARC64 VI provides eight of 64-bit registers that can be used temporary storage for supervisor software.



[1] Register Name: ASI_SCRATCH_REG*x* (*x* = 0–7)
[2] ASI: $4F_{16}$
[3] VA: VA<5:3> = register number
The other VA bits must be zero.
[4] RW: Supervisor read/write

Block Load and Store ASIs

ASIs $E0_{16}$ and $E1_{16}$ exist only for use with STDFA instructions as Block Store with Commit operations (see *Block Load and Store Instructions (VIS I)* on page 51). Neither ASI $E0_{16}$ nor ASI $E1_{16}$ should be used with LDDFA; however, if either is used, the LDDFA behaves as follows:

1. No exception is generated based on the destination register *rd* (impl. dep. #255).
2. For LDDFA with ASI $E0_{16}$ or $E1_1$ and a memory address aligned on a 2^n-byte boundary, a SPARC64 VI processor behaves as follows (impl. dep. #256):

   $n \geq 3$ (≥ 8-byte alignment): no exception related to memory address alignment is generated, but a *data_access_exception* is generated (see case 3, below).
   $n = 2$ (4-byte alignment): *LDDF_mem_address_not_aligned* exception is generated.

   $n \leq 1$ (≤ 2-byte alignment): *mem_address_not_aligned* exception is generated.
3. If the memory address is correctly aligned, a *data_access_exception* with an AFSR.FTYPE = "invalid ASI" is generated.

Partial Store ASIs

ASIs $C0_{16}$–$C5_{16}$ and $C8_{16}$–CD_{16} exist for use with the STDFA instruction for Partial Store operations (see *Partial Store (VIS I)* on page 65). None of these ASIs should be used with LDDFA; however, if one of them is used, the LDDFA behaves as follows on a SPARC64 VI processor (impl. dep. #257):

1. For LDDFA with $C0_{16}$–$C5_{16}$ or $C8_{16}$–CD_{16} and a memory address aligned on a 2^n-byte boundary, a SPARC64 VI processor behaves as follows:

   $n \geq 3$ (≥ 8-byte alignment): no exception related to memory address alignment is generated.

   $n = 2$ (4-byte alignment): *LDDF_mem_address_not_aligned* exception is generated.

   $n \leq 1$ (≤ 2-byte alignment): *mem_address_not_aligned* exception is generated.
2. If the memory address is correctly aligned, SPARC64 VI generates a *data_access_exception* with AFSR.FTYPE = "invalid ASI."

F.APPENDIX **M**

Cache Organization

This appendix describes SPARC64 VI cache organization in the following sections:

M.1 Cache Types

SPARC64 VI has two levels of on-chip caches, with these characteristics:

- Level-1 cache is split for instruction and data; level-2 cache is unified.
- Level-1 caches are virtually indexed, physically tagged (VIPT), and level-2 caches are physically indexed, physically tagged (PIPT).
- Level-1 caches are 64 bytes in line size, and level-2 cache are 256 bytes in line size (4 64byte sub-line).
- All lines in the level-1 caches are included in the level-2 cache.
- Between level-1 caches, or level-1 and level-2 caches, coherency is maintained by hardware. In other words,
  - eviction of a cache line from a level-2 cache causes flush-and-invalidation of all level-1 caches, and
  - self-modification of an instruction stream modifies a level-1 data cache with invalidation of a level-1 instruction cache.
- Level-1 caches are shared by the two threads in the core, and Level-2 is shared by all the threads in the processor module.

M.1.1 Level-1 Instruction Cache (L1I Cache)

TABLE M-1 shows the characteristics of a level-1 instruction cache.

TABLE M-1 L1I Cache Characteristics

Feature	Value
Size	128 Kbytes
Associativity	2-way
Line Size	64-byte
Indexing	Virtually indexed, physically tagged (VIPT)
Tag Protection	Parity and duplicate
Data Protection	Parity

Although an L1I cache is VIPT, `TTE.CV` is ineffective since SPARC64 VI has unaliasing features in hardware.

Instruction fetches bypass the L1I cache when they are noncacheable accesses. Noncacheable accesses occur under one of three conditions:

- `PSTATE.RED` = 1
- `DCUCR.IM` = 0
- `TTE.CP` = 0

When `MCNTL.NC_CACHE` = 1, SPARC64 VI treats all instructions as cacheable, regardless of the conditions listed above. See *ASI_MCNTL (Memory Control Register)* on page 100 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 by `ASI_FLUSH_L1I` before it exits.

M.1.2 Level-1 Data Cache (L1D Cache)

The level-1 data cache is a writeback cache. Its characteristics are shown in TABLE M-2.

TABLE M-2 L1D Cache Characteristics

Feature	Value
Size	128 Kbytes
Associativity	2-way
Line Size	64-byte
Indexing	Virtually indexed, physically tagged (VIPT)
Tag Protection	Parity and duplicate
Data Protection	ECC

Although L1D cache is VIPT, `TTE.CV` is ineffective since SPARC64 VI has unaliasing features in hardware.

Data accesses bypass the L1D cache when they are noncacheable accesses. Noncacheable accesses occur under one of three conditions:

- The ASI used for the access is either `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

Unlike the L1I cache, the L1D cache does not use `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 in TABLE M-3.

TABLE M-3 L2 Cache Characteristics

Feature	Value
Size	6 Mbyte (max)
Associativity	12-way (max)
Line Size	256-byte consists of 4 64-byte sublines
Indexing	Physically indexed, physically tagged (PIPT)
Tag Protection	ECC
Data Protection	ECC

The L2 cache is bypassed when the access is noncacheable. `MCNTL.NC_CACHE` is not used on the L2 cache.

M.2 Cache Coherency Protocols

The CPU uses the enhanced MOESI cache-coherence protocol; these letters are acronyms for cache line states as follows:

M	Exclusive modified
O	Shared modified (owned)
E	Exclusive clean
S	Shared clean
I	Invalid

A subset of the MOESI protocol is used in the on-chip caches as well as the D-Tags in the system controller. TABLE M-4 shows the relationships between the protocols.

TABLE M-4 Relationships Between Cache Coherency Protocols

L2-Cache	L1D-Cache	SAT (store ownership)	L1I-Cache
Invalid (I)	Invalid (I)	Invalid (I)	Invalid (I)
Shared Clean (S)	Invalid (I) or Clean (C)	Invalid (I)	Invalid (I) or Valid (V)
Shared Modified (O)			
Exclusive Clean (E)			
Exclusive Modified (M)			
	Exclusive Modified (M)	Valid (V)	Invalid (I)

TABLE M-5shows the encoding of the MOESI states in the L2 Cache.

TABLE M-5 L2 Cache MOESI States

Bit 2 (Valid)	Bit 1 (Exclusive)	Bit 0 (Modified)	State
0	—	—	Invalid (I)
1	0	0	Shared clean (S)
1	1	0	Exclusive clean (E)
1	0	1	Shared modified (O)
1	1	1	Exclusive modified (M)

M.3 Cache Control/Status Instructions

Several ASI instructions are defined to manipulate the caches. The following conventions are common to all of the load and store alternate instructions defined in this section:

1. The opcode of the instructions should be ldda, ldxa, lddfa, stda, stxa, or stdfa. Otherwise, a *data_access_exception* exception with `D-SFSR.FT` = 08_{16} (Invalid ASI) is generated.
2. No operand address translation is performed for these instructions.
3. VA<2:0> of all of the operand address should be 0. Otherwise, a *mem_address_not_aligned* exception is generated.
4. The don't-care bits (designated "—" in the format) in the VA of the load or store alternate can be of any value. It is recommended that software use zero for these bits in the operand address of the instruction.
5. The don't-care bits (designated "—" in the format) in DATA are read as zero and ignored on write.
6. The instruction operations are not affected by `PSTATE.CLE`. They are always treated as big-endian.
7. The instructions are all strongly ordered regardless of load or store and the memory model. Therefore, no speculative executions are performed.

Multiple Asynchronous Fault Address Registers are maintained in hardware, one for each major source of asynchronous errors. These ASIs are described in *ASI_ASYNC_FAULT_STATUS (ASI_AFSR)* on page 180. The following subsections describe all other cache-related ASIs in detail.

M.3.1 Flush Level-1 Instruction Cache (`ASI_FLUSH_L1I`)

[1]	Register Name:	ASI_FLUSH_L1I
[2]	ASI:	67_{16}
[3]	VA:	Any
[4]	RW	Supervisor write

`ASI_FLUSH_L1I flushes and` invalidates the entire level-1 instruction cache. VA can be any value. A write to this ASI with any VA and any data causes flushing and invalidation.

M.3.2 Level-2 Cache Control Register (ASI_L2_CTRL)

[1]	Register Name:	ASI_L2_CTRL
[2]	ASI:	$6A_{16}$
[3]	VA:	any
[4]	RW	Supervisor read/write
[5]	Data	

ASI_L2_CTRL is a control register for L2 training, interface, and size configuration. It is illustrated below and described in TABLE M-6.

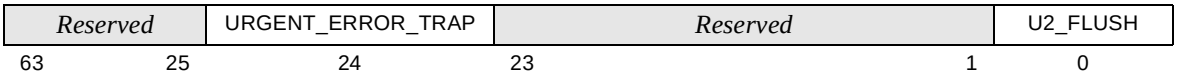


TABLE M-6 ASI_L2_CTRL Register Bits

Bit	Field	RW	Description
24	**URGENT_ERROR_TRAP**	RW1C	This bit is set to 1 when one of the error exceptions (*instruction_access_error*, *data_access_error*, or *asynchronous_data_error*) is generated. The bit remains set to 1 until supervisor software explicitly clears it by writing 1 to the bit.
0	**U2_FLUSH**	W	Set this bit to 1 causes entire level-2 cache to flush. Until the flushing of the level-2 cache completes, the processor ceases operation and does not perform further instruction execution. Writing 0 to this bit is ignored.

M.3.3 L2 Diagnostics Tag Read (ASI_L2_DIAG_TAG_READ)

This ASI instruction is a diagnostic read of L2 cache tag, as well as tag 2 of L1I and L1D.

ASI_L2_DIAG_TAG_READ works in concert with ASI_L2_DIAG_TAG_READ_REG. A read to ASI_L2_DIAG_TAG_READ returns 0, with the side effect of setting the tag to ASI_L2_DIAG_TAG_READ_REG0-6.

[1]	Register Name:	ASI_L2_DIAG_TAG
[2]	ASI:	$6B_{16}$
[3]	VA:	VA<18:6>: Index number of the tag. 0000_{16}–$7FFC0_{16}$
[4]	RW	Supervisor read
[5]	Data	Meaningless.

M.3.4 L2 Diagnostics Tag Read Registers (ASI_L2_DIAG_TAG_READ_REG)

ASI_L2_DIAG_TAG_READ_REG0-6 holds the tag that is specified by the read of ASI_L2_DIAG_TAG_READ.

[1]	Register Name:	ASI_L2_DIAG_TAG_READ_REG
[2]	ASI:	$6C_{16}$
[3]	VA:	VA<6:3> internal register number
[4]	RW	Supervisor Read
[5]	Data	TBD.

M.3.5 Cache invalidation (ASI_CACHE_INV)

The following ASI is newly added on SPARC64 VI.

[1]	Register Name:	ASI_CACHE_INV
[2]	ASI:	74_{16}
[3]	VA:	Physical Address
[4]	RW	Supervisor write
[5]	Data	

ASI_CACHE_INV flushes and invalidates cache lines of all processor modules in the same partition. The cache lines to be invalidated is specified by the VA field which keeps the physical address (That is, ASI_CACHE_INV is bypass ASI). Thus PSTATE_address_mask is ignored. Also the Physical Address Data Watchpoint Register (ASI 58_{16}, VA=40_{16}) is ignored unlike other bypass ASIs.

The ASI is write-only and read to it causes *data_access_exception* with AFSR.FTYPE = "invalid ASI".

Note – DCUCR.WEAK_SPCA has to be set to "1" before executing the instruction.

F.APPENDIX **N**

Interrupt Handling

Interrupt handling in SPARC64 VI is described in these sections:

N.1 Interrupt Dispatch

When a processor wants to dispatch an interrupt to another processor, it first sets up the interrupt data registers (`ASI_INTR_W` data 0-7) with the outgoing interrupt packet data by using ASI instructions. It then performs an `ASI_INTR_W` (interrupt dispatch) write to trigger delivery of the interrupt. The interrupt packet and the associated data are forwarded to the target processor by the system controller. The processor polls the `BUSY` bit in the `INTR_DISPATCH_STATUS` register to determine whether the interrupt has been dispatched successfully.

FIGURE N-1 illustrates the steps required to dispatch an interrupt.

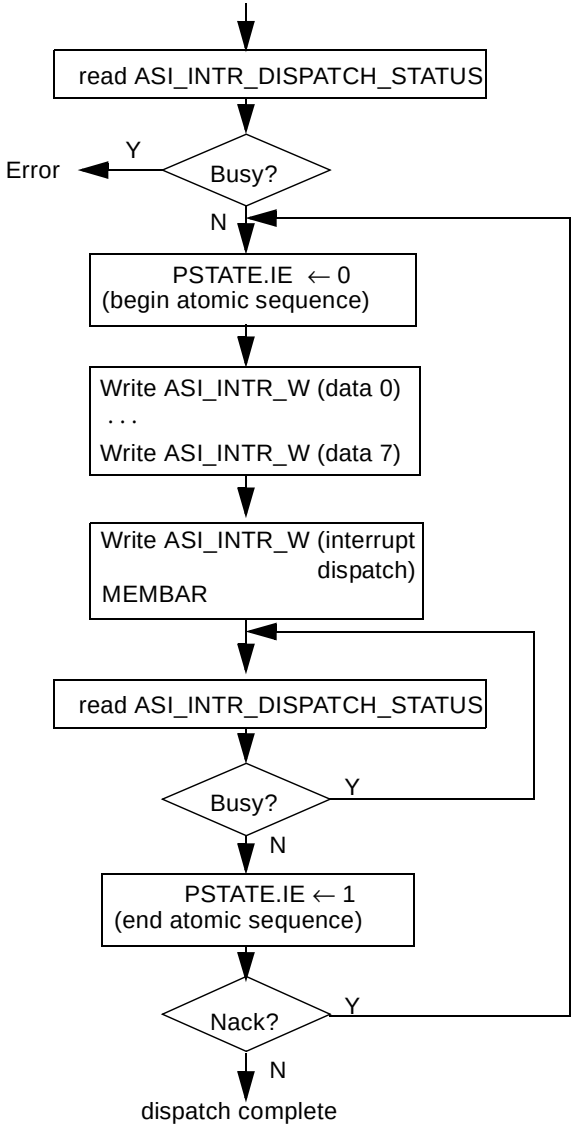


FIGURE N-1 Dispatching an Interrupt

N.2 Interrupt Receive

When an interrupt packet is received, eight interrupt data registers are updated with the associated incoming data and the BUSY bit in the ASI_INTR_RECEIVE register is set. If interrupts are enabled (PSTATE.IE = 1), then the processor takes a trap and the interrupt data registers are read by the software to determine the appropriate trap handler. The handler may reprioritize this interrupt packet to a lower priority.

If an incoming packet is marked as error, BUSY bit in the ASI_INTR_RECEIVE register is not set. In this case, other interrupt related ASI registers may also be corrupted and should not be accessed. See Section P.8.3, *ASI Register Error Handling*, on page 184 for detail.

FIGURE N-2 is an example of the interrupt receive flow.

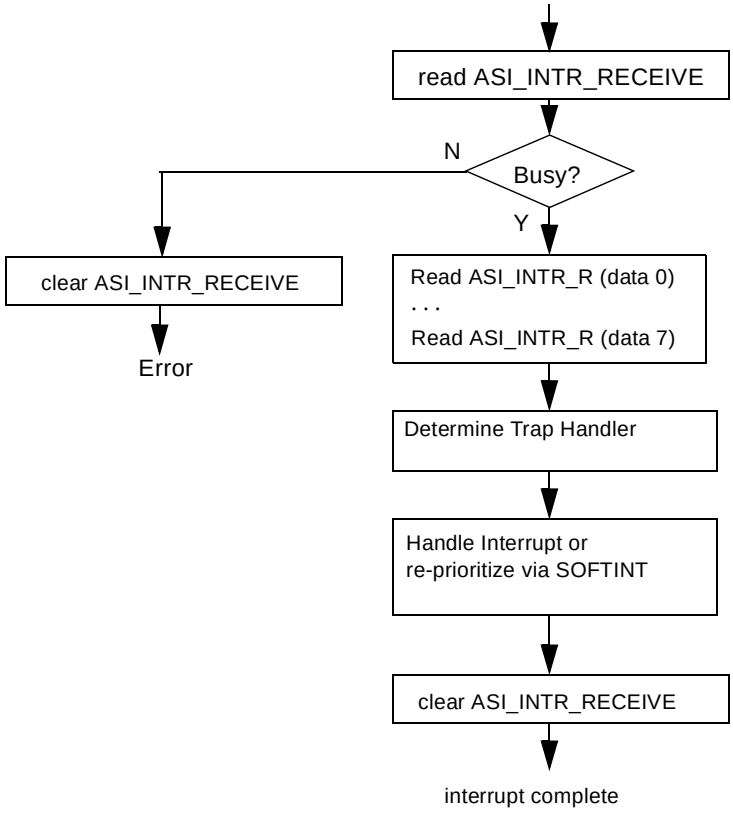


FIGURE N-2 Interrupt Receive Flow

N.3 Interrupt Global Registers

Please refer to Section N.3. of **Commonality**.

N.4 Interrupt-Related ASR Registers

Please refer to Section N.4 of **Commonality** for details of these registers.

N.4.2 Interrupt Vector Dispatch Register

SPARC64 VI ignores all 10 bits of VA<38:29> when the Interrupt Vector Dispatch Register is written (impl. dep. #246).

N.4.3 Interrupt Vector Dispatch Status Register

In SPARC64 VI, 32 BUSY/NACK pairs are implemented in the Interrupt Vector Dispatch Status Register (impl. dep. #243).

N.4.5 Interrupt Vector Receive Register

SPARC64 VI sets a 10-bit value in the SID_H and SID_L fields of the Interrupt Vector Receive Register, but the value to be set is undefined. (impl. dep. #247).

N.5 How to identify interrupt target

SPARC64 VI has multiple threads in a processor module. As a result, SPARC64 VI needs a mechanism to identify which thread should receive a given interrupt (*interrupt_vector*).

ASI_EIDR is used to identify the thread to receive a given interrupt (*interrupt_vector*).

The firmware is supposed to initialize ASI_EIDR with the Interrupt Target Identifier (ITID) on boot. The behavior of SPARC64 VI when it receives an interrupt packet is as follows.

a. If at least one of the ASI_EIDRs remain uninitialized, and none of the initialized ASI_EIDR values are equal to the ITID value in the interrupt packet

The interrupt packet is sent to the thread specified by ITID#1:0 in the packet.

b. If all of the ASI_EIDRs have been initialized, but none or more than one of the ASI_EIDR values are equal to the ITID value in the interrupt packet

Which thread receives the packet or none receives it is undefined. The sender sees `ASI_INTR_DISPATCH_STATUS#NACK=0` in both the cases, though.

c. If one but only one of the initialized ASI_EIDR values is equal to the ITID value in the interrupt packet,

The interrupt packet is sent to the thread of which ASI_EIDR value matches with the ITID value in the packet.

F.APPENDIX **O**

Reset, RED_state, and error_state

The appendix contains these sections:

O.1 Reset Types

This section describes the four reset types: power-on reset, watchdog reset, externally initiated reset, and software-initiated reset.

POR and XIR are applied to all the threads within a processor module. In other words, all the threads go through the same trap process. WDR, SIR, are RED_state are applied only to the particular thread which invoked the reset. Other threads are unaffected and continues to run.

O.1.1 Power-on Reset (POR)

For execution of the power-on reset on SPARC64 VI, an external facility must issue the required sequence of JTAG commands to the processor.

While the reset pin is asserted or the Power ready signal is de-asserted, the processor stops and executes only the specified JTAG command. The processor does not change any software-visible resources in the processor except the change by JTAG command execution and does not change any memory system state.

On POR, the processor enters `RED_state` with `TT` = 1 trap to RSTVaddr + 20_{16} and starts the instruction execution.

O.1.2 Watchdog Reset (WDR)

The watchdog reset trap is generated internally in the following cases:

- Second watchdog timeout detection while TL < MAXTL.
- First watchdog timeout detection while TL = MAXTL
- When a trap occurs while TL = MAXTL

When triggered by a watchdog timeout, a WDR trap has TT = 2 and control transfers to RSTVaddr + 40_{16}. Otherwise, the TT of the trap is preserved, causing an entry into error_state.

O.1.3 Externally Initiated Reset (XIR)

When SPARC64 VI receives a packet requesting XIR through Jupiter Bus, it generates a trap of TT = 3 and causes the processor to transfer execution to RSTVaddr + 60_{16} and enter RED_state.

O.1.4 Software-Initiated Reset (SIR)

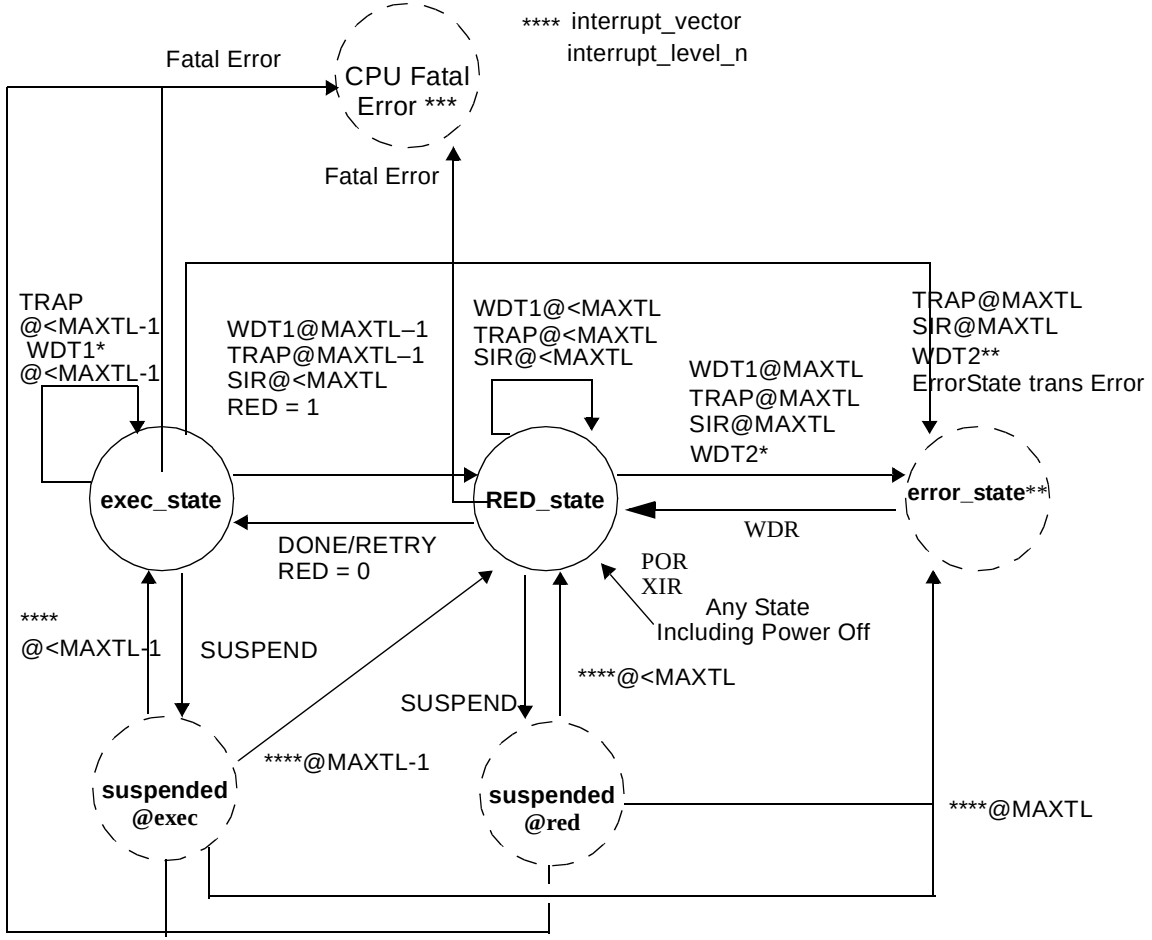
Any processor can initiate a software-initiated reset with an SIR instruction.

If TL (Trap Level) < MAXTL (5), an SIR instruction causes a trap of TT = 4 and causes the processor to execute instructions from RSTVaddr + 80_{16} and enter RED_state.

If a processor executes an SIR instruction while TL = 5, it enters error_state and ultimately generates a watchdog reset trap.

O.2 RED_state and error_state

The suspended_state is added to support MTP effectively. There is no way for a given thread to tell if the other thread is in the suspended_state or not .



* WDT1 is the first watchdog timeout.

** WDT2 is the second watchdog timeout. WDT2 takes the CPU into error_state. In a normal setting, error_state immediately generates a watchdog reset trap and brings the CPU into RED_state. Thus, the state is transient. When the OPSR (Operation Status Register) specifies the stop on error_ state, an entry into error_state does not cause a watchdog reset and the CPU remains in the error_state.

***CPU_fatal_error_state signals the detection of a fatal error to the system through P_FERR signal to the system, and the system causes a FATAL reset. Soft POR will be applied to the all threads in the system at the FATAL reset.

FIGURE O-1 Processor State Diagram

O.2.1 RED_state

Once the processor enters RED_state for any reason except when a power-on reset (POR) is performed, the software should not attempt to return to execute_state; if software attempts a return, then the state of the processor is unpredictable.

When the processor processes a reset or a trap that enters RED_state, it takes a trap at an offset relative to the RED_state trap table (RSTVaddr); in the processor this is at virtual address VA = FFFFFFFFF0000000$_{16}$ and physical address PA = 000007FFF0000000$_{16}$.

The following list further describes the processor behavior upon entry into RED_state, and during RED_state:

- Whenever the processor enters RED_state, all fetch buffers are invalidated.
- When the processor enters RED_state because of a trap or reset, the DCUCR register is updated by hardware to disable several hardware features. Software must set these bits when required (for example, when the processor exits from RED_state).
- When the processor enters RED_state *not* because of a trap or reset (that is, when the PSTATE.RED bit has been set by WRPR), these register bits are unchanged—unlike the case above. The only side effect is the disabling of the instruction MMU.
- When the processor is in RED_state, it behaves as if the IMMU is disabled (DCUCR.IM is clear), regardless of the actual values in the respective control register.
- Caches continue to snoop and maintain coherence while the processor is in RED_state.

O.2.2 error_state

The processor enters error_state when a trap occurs and TL = MAXTL (5) or when the second watchdog time-out has occurred.

On the normal setting, the processor immediately generates a watchdog reset trap (WDR) and transitions to RED_state. Otherwise, the OPSR (Operating Status Register) specifies the stop on error_state, that is, the processor does not generate a watchdog reset after error_state transition and remains in the error_state.

O.2.3 CPU Fatal Error state

The processor enters CPU fatal error state when a fatal error is detected on the processor. A fatal error is one that breaks the cache coherency or the system data integrity. See Appendix , *Error Handling*, for details of the SDC error.

The processor reports the fatal error detection to the system, and the system causes the fatal reset. Soft POR will be applied to the all CPUs in the system at the fatal reset.

O.3 Processor State after Reset and in RED_state

TABLE O-1 shows the various processor states after resets and when entering `RED_state`.

In this table, it is assumed that `RED_state` entry happens as a result of resets or traps. If `RED_state` entry occurs because the WRPR instruction sets the `PSTATE.RED` bit, no processor state will be changed except the `PSTATE.RED` bit itself; the effects of this are described in *RED_state* on page 146.

TABLE O-1 Nonprivileged and Privileged Register State after Reset and in `RED_state`

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 = 07FF F000 0000_{16}				
PC nPC		RSTV | 20_{16} RSTV | 24_{16}	RSTV | 40_{16} RSTV | 44_{16}	RSTV | 60_{16} RSTV | 64_{16}	RSTV | 80_{16} RSTV | 84_{16}	RSTV | $A0_{16}$ RSTV | $A4_{16}$
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 (TSO)				
	TLE	0	Unchanged			
	CLE	0	Copied from TLE			
TBA<63:15>		Unknown/Unchanged	Unchanged			
Y		Unknown/Unchanged	Unchanged			
PIL		Unknown/Unchanged	Unchanged			
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] TNPC[TL]		Unknown/Unchanged Unknown/Unchanged	PC nPC			

TABLE O-1 Nonprivileged and Privileged Register State after Reset and in `RED_state` *(Continued)*

Name		POR[1]	WDR[2]	XIR	SIR	RED_state
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} 6_{16} Mask dependent 5_{16} 7_{16}				
FSR		0	Unchanged			
FPRS		Unknown/Unchanged	Unchanged			

1.Hard POR occurs when power is cycled. Values are unknown following hard POR. Soft POR occurs when the reset signal is asserted. Values are unchanged following soft POR.

2.The first watchdog time-out trap is taken in execute_state (i.e. `PSTATE.RED` = 0), subsequent watchdog time-out traps as well as watchdog traps due to a trap @ `TL` = `MAX_TL` are taken in `RED_state`. See Section O.1.2, *Watchdog Reset (WDR)*, on page 144 for more details.

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			

TABLE O-2 ASR State after Reset and in RED_state *(Continued)*

ASR	Name	POR[1]	WDR[2]	XIR	SIR	RED_state
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			

1.Hard POR occurs when power is cycled. Values are unknown following hard POR. Soft POR occurs when the reset signal is asserted. Values are unchanged following soft POR.

2.The first watchdog time-out trap is taken in execute_state (i.e. `PSTATE.RED` = 0), subsequent watchdog time-out traps, as well as watchdog traps due to a trap @ `TL` = `MAX_TL`, are taken in `RED_state`. See Section O.1.2, *Watchdog Reset (WDR)*, on page 144or more details

TABLE O-3 ASI Register State After Reset and in `RED_state` *(1 of 3)*

ASI	VA	Name	POR[1]	WDR[2]	XIR	SIR	RED_state
45	00	DCUCR	0	0			
45	08	MCNTL	0	0			
48	00	INST_DISPATCH_STATUS	0	Unchanged			
49	00	INTR_RECEIVE	Unknown/Unchanged	Unchanged			
4A	00	JBUS_CONFIG UC_S UC_SW CLK_MODE ITID	 Pre-defined/Unchanged Pre-defined/Unchanged Pre-defined/Unchanged Pre-defined/Unchanged	 Unchanged Unchanged Unchanged Unchanged			
4C	00	AFSR	Unknown/Unchanged	Unchanged			
4C	08	UGESR	Unknown/Unchanged	Unchanged			
4C	10	ERROR_CONTROL WEAK_ED Others	 1 Unknown/Unchanged	 1 Unchanged			
4C	18	STCHG_ERR_INFO	Unknown/Unchanged	Unchanged			
4D	00	AFAR_D1	Constant Value	Constant Value			
4D	08	AFAR_U2	Constant Value	Constant Value			
4F	--	SCRATCH_REGs	Unknown/Unchanged	Unchanged			
50	00	IMMU_TAG_TARGET	Unknown/Unchanged	Unchanged			
50	18	IMMU_SFSR	Unknown/Unchanged	Unchanged			
50	28	IMMU_TSB_BASE	Unknown/Unchanged	Unchanged			

TABLE O-3 ASI Register State After Reset and in `RED_state` *(2 of 3)*

ASI	VA	Name	POR[1]	WDR[2]	XIR	SIR	RED_state
50	30	IMMU_TAG_ACCESS	Unknown/Unchanged	Unchanged			
50	48	IMMU_TAG_TSB_PEXT	Unknown/Unchanged	Unchanged			
50	58	IMMU_TAG_TSB_NEXT	Unknown/Unchanged	Unchanged			
50	60	IMMU_TAG_ACCESS_EXT	Unknown/Unchanged	Unchanged			
50	78	IMMU_SFPAR	Unknown/Unchanged	Unchanged			
51	—	IMMU_TSB_8KB_PTR	Unknown/Unchanged	Unchanged			
52	—	IMMU_TSB_64KB_PTR	Unknown/Unchanged	Unchanged			
53	—	SERIAL_ID	Constant value	Constant value			
54	—	ITLB_DATA_IN	Unknown/Unchanged	Unchanged			
55	—	ITLB_DATA_ACCESS	Unknown/Unchanged	Unchanged			
56	—	ITLB_TAG_READ	Unknown/Unchanged	Unchanged			
57	—	ITLB_DEMAP	Unknown/Unchanged	Unchanged			
58	00	DMMU_TAG_TARGET	Unknown/Unchanged	Unchanged			
58	08	PRIMARY_CONTEXT	Unknown/Unchanged	Unchanged			
58	10	SECONDARY_CONTEXT	Unknown/Unchanged	Unchanged			
58	18	DMMU_SFSR	Unknown/Unchanged	Unchanged			
58	20	DMMU_SFAR	Unknown/Unchanged	Unchanged			
58	28	DMMU_TSB_BASE	Unknown/Unchanged	Unchanged			
58	30	DMMU_TAG_ACCESS	Unknown/Unchanged	Unchanged			
58	38	DMMU_VA_WATCHPOINT	Unknown/Unchanged	Unchanged			
58	40	DMMU_PA_WATCHPOINT	Unknown/Unchanged	Unchanged			
58	48	DMMU_TSB_PEXT	Unknown/Unchanged	Unchanged			
58	50	DMMU_TSB_SEXT	Unknown/Unchanged	Unchanged			
58	58	DMMU_TSB_NEXT	Unknown/Unchanged	Unchanged			
58	60	DMMU_TAG_ACCESS_EXT	Unknown/Unchanged	Unchanged			
58	78	DMMU_SFPAR	Unknown/Unchanged	Unchanged			
59	—	DMMU_TSB_8KB_PTR	Unknown/Unchanged	Unchanged			
5A	—	DMMU_TSB_64KB_PTR	Unknown/Unchanged	Unchanged			
5B	—	DMMU_TSB_DIRECT_PTR	Unknown/Unchanged	Unchanged			
5C	—	DTLB_DATA_IN	Unknown/Unchanged	Unchanged			
5D	—	DTLB_DATA_ACCESS	Unknown/Unchanged	Unchanged			
5E	—	DTLB_TAG_READ	Unknown/Unchanged	Unchanged			
5F	—	DMMU_DEMAP	Unknown/Unchanged	Unchanged			
60	—	IIU_INST_TRAP	0	Unchanged			
6E	—	EIDR	0/Unchanged	Unchanged			

TABLE O-3 ASI Register State After Reset and in RED_state *(3 of 3)*

ASI	VA	Name	POR[1]	WDR[2]	XIR	SIR	RED_state
6F	—	BARRIER_SYNC_P	Unknown/Unchanged	Unchanged			
77	40:68	INTR_DATA0:5_W	Unknown/Unchanged	Unchanged			
77	70	INTR_DISPATCH_W	Unknown/Unchanged	Unchanged			
77	80:88	INTR_DATA6:7_W	Unknown/Unchanged	Unchanged			
7F	40:88	INTR_DATA0:7_R	Unknown/Unchanged	Unchanged			

1. Hard POR occurs when power is cycled. Values are unknown following hard POR. Soft POR occurs when the reset signal is asserted. Values are unchanged following soft POR

2. The first watchdog time-out trap is taken in execute_state (i.e. PSTATE.RED = 0), subsequent watchdog time-out traps as well as watchdog traps due to a trap @ TL = MAX_TL, are taken in RED_state. See Section O.1.2, *Watchdog Reset (WDR),* on page 144 for more details.

O.3.1 Operating Status Register (OPSR)

OPSR is the control register in the CPU that is scanned in during the hardware power-on reset sequence before the CPU starts running.

The value of the OPSR is specified outside of the CPU and is never changed by software. OPSR is set by scan-in during hardware power-on reset and by a JTAG command after hardware POR.

Most of the OPSR settings are not visible to the software.

F.APPENDIX **P**

Error Handling

This appendix describes the processor behavior to a programmer writing an operating system, firmware, or recovery code for SPARC64 VI. Section headings differ from those of Appendix P of **Commonality**.

P.1 Error Classes and Signalling

On SPARC64 VI, an error is classified into one of the following four categories, depending on the degree to which it obstructs program execution:

- 1.Fatal error
- 2.Error state transition error
- 3.Urgent error
- 4.Restrainable error

SPARC64 VI includes two COREs in the same processor module, where each core contains two threads. When an error is detected, how to identify the threads where an error is logged and gets reported depends on the error type.

An error detected in the course of an instruction or occurring in a resource specific to a thread (ex. IUG_%R) are called synchronous to thread execution. In this case, the error is logged and reported to the thread executing the instruction or the thread includes the resource with the error. By their nature, *instruction_access_error* and *data_access_error* belong to this category.

An error independent from instruction execution or occurring in the shared resources between multiple threads is called asynchronous to tread execution. In this case, the error is logged and reported to all the threads related to the resource causing the error.

Error marking is essentially asynchronous to thread execution. When an L1$ or an L2$ raw uncorrectable error is detected, ASI_EIDR of the valid (that is, not degraded) threads with the smallest thread ID (core0-thread0 < core0-thread1 < core1-thread0 < core1-thread1) related to that cache is used for error marking.

Another issue is how to log and report an error when a corresponding thread is in the suspended state. Except for fatal errors, the error logging and report are postponed until the corresponding thread exits from the suspended state.

P.1.1 Fatal Error

A fatal error is one of the following errors that damages the entire system.

a. Error breaking data integrity on the system

All errors that break cache coherency are in this category.

b. Invalid system control flow is detected and therefore validity of the subsequent system behavior cannot be guaranteed.

When the CPU detects a fatal error, the CPU enters FATAL `error_state` and reports the fatal error occurrence to the system controller. The system controller transfers the entire system state to the FATAL state and stops the system. After the system stops, a FATAL reset, which is a type of power-on reset, will be issued to the whole system.

All fatal errors are asynchronous to thread execution. If a fatal error is detected in a given thread, all the threads within the processor module log the cause into ASI_STCHG_ERROR_INFO and go through the POR sequence even if they are in the suspended state.

P.1.2 `error_state` transition Error

An `error_state` transition error is a serious error that prevents the CPU from reporting the error by generating a trap. However, any damage caused by the error is limited to within the CPU.

When the CPU detects an `error_state` transition error, it enters `error_state`. The CPU exits `error_state` by causing a watchdog reset, entering `RED_state`, and starting instruction execution at the watchdog reset trap handler.

EE asynchronous to thread execution

The following error_state transition errors are asynchronous to thread execution. If such an EE is detected in a given thread, both the threads within the core which caused the error log it into ASI_STCHG_ERROR_INFO and go through WDR, unless they are in the suspended state. The threads in the other core are unaffected.

- EE_TRAP_ADR_UE
- EE_OTHER

EE synchronous to thread execution

The following error_state transition errors are synchronous to thread execution. If such an EE is detected in a given thread, only that thread logs the cause of the error into ASI_STCHG_ERROR_INFO and goes through WDR. All the other threads are unaffected.

- EE_SIR_IN_MAXTL
- EE_TRAP_IN_MAXTL
- EE_WDT_IN_MAXTL
- EE_SECOND_WDT

P.1.3 Urgent Error

An urgent error (UGE) is an error that requires immediate processing by privileged software, which is reported by an error trap. The types of urgent errors are listed below and then described in further detail.

- Instruction-obstructing error
  - I_UGE: Instruction urgent error
  - IAE: Instruction access error
  - DAE: Data access error
- Urgent error that is independent of the instruction execution
  - A_UGE: Autonomous urgent error

Instruction-Obstructing Error

An instruction-obstructing error is one that is detected by instruction execution and results in the instruction being unable to complete.

When the instruction-obstructing error is detected while ASI_ERROR_CONTROL.WEAK_ED = 0 (as set by privileged software for a normal program execution environment), then an exception is generated to report the error. This trap is nonmaskable.

Otherwise, when ASI_ERROR_CONTROL.WEAK_ED = 1, as with multiple errors or a POST/OBP reset routine, one of the following actions occurs:

- Whenever possible, the CPU writes an unpredictable value to the target of the damaged instruction and the instruction ends.
- Otherwise, an error exception is generated and the damaged instruction is executed as when ASI_ERROR_CONTROL.WEAK_ED = 0 is set.

The three types of instruction-obstructing errors are described below.

- **I_UGE (instruction urgent error)** — All of the instruction-obstructing errors except IAE (instruction access error) and DAE (data access error). There are two categories of I_UGEs.
  - **An uncorrectable error in an internal program-visible register that obstructs instruction execution.**

    An uncorrectable error in the PSTATE, PC, NPC, CCR, ASI, FSR, or GSR register is treated as an I_UGE that obstructs the execution of any instruction. See Sections P.8.1 and P.8.2 for details.

    The first-time watchdog time-out is also treated as this type of I_UGE.
  - **An error in the hardware unit executing the instruction, other than an error in a program-visible register.**

    Among these errors are ALU output errors, errors in temporary registers during instruction execution**,** CPU internal data bus errors, and so forth.

  I_UGE is a preemptive error with the characteristics shown in TABLE P-2.
- **IAE (instruction access error)** — The *instruction_access_error* exception, as specified in JPS1 **Commonality**. On SPARC64 VI, only an uncorrectable error in the cache or main memory during instruction fetch is reported to software as an IAE.

  IAE is a precise error.
- **DAE (data access error)** — The *data_access_error* exception, as specified in JPS1 **Commonality**. On SPARC64 VI, only an uncorrectable error in the cache or main memory during access by a load, store, or load-store instruction is reported to software as a DAE.

  DAE is a precise error.

Urgent Error Independent of Instruction Execution

- **A_UGE (Autonomous Urgent Error)** — An error that requires immediate processing and that occurs independently of instruction execution.

  In normal program execution, ASI_ERROR_CONTROL.WEAK_ED = 0 is specified by privileged software. In this case, the A_UGE trap is suppressed only in the trap handler used to process UGE (that is, the *async_data_error* trap handler).

  Otherwise, in special program execution such as the handling of the occurrence of multiple errors or the POST/OBP reset routine, ASI_ERROR_CONTROL.WEAK_ED = 1 is specified by the program. In this case, no A_UGE generates an exception.

  There are two categories of A_UGEs:

  - **An error in an important resource that will cause a fatal error or error_state transition error when the resource is used.**

    When the resource with the error is used, the program cannot continue execution, and an error_state transition error or a fatal error is detected.

  - **The error in an important resource that is expected to invoke the operating system "panic" process**

    The operating system panic process is expected when this error is detected because the normal processing cannot be expected to continue after this error occurs.

  The A_UGE is a disrupting error with the following deviations.

  - The trap for A_UGE is not masked by PSTATE.IE.
  - The instruction designated by TPC may not end precisely. The instruction end-method is reported in the trap status register for A_UGE.

Traps for Urgent Errors

When an urgent error is detected and not masked, the error is reported to privileged software by 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 urgent errors are asynchronous to thread execution. If such an urgent error is detected in a given thread, both of the threads within the core which caused the error log it into ASI_UGESR and activate an ADE trap, unless they are in the suspended state. The threads in the other core are unaffected.

- IAUG_CRE
- IAUG_TSBCTXT

- IUG_TSBP
- IUG_PSTATE
- IUG_TSTATE
- IUG_%F (except %f*n* parity error)
- IUR_%R (except %rn and Y parity error)
- IUG_WDT
- IUG_DTLB
- IUG_ITLB
- IUG_COREERR

Urgent error synchronous to thread execution

The following urgent errors are synchronous to thread execution. If such an urgent error is detected in a given thread, only that thread logs the cause of the error into ASI_UGESR and activate an ADE trap, unless they are in the suspended state. All the other threads are unaffected.

- IUG_%F (%f*n* parity error only)
- IUR_%R (%r*n* and Y parity error only)

P.1.4 Restrainable Error

A restrainable error is one that does not adversely affect the currently executing program and that does not require immediate handling by privileged software. A restrainable error causes a disrupting trap with low priority.

There are three types of restrainable errors.

- **Correctable Error (**CE**), corrected by hardware**

  Upon detecting the CE, the hardware uses the data corrected by hardware. So a CE has no deleterious effect on the CPU.

  When a CE is detected, data seen by the CPU is always corrected by hardware. But it depends on the CE type whether the source data containing the CE is corrected or not.

- Uncorrectable error without direct damage to the currently executing instruction sequence.

  An error detected in cache line writeback or copyback data is of this type.

- Degradation

  SPARC64 VI can isolate an internal hardware resource that generates frequent errors and continue processing without deleterious effect on software during program execution. However, performance is degraded by the resource isolation. This degradation is reported as a restrainable error.

The restrainable error can be reported to privileged software by the *ECC_error* trap.

When `PSTATE.IE` = 1 and the trap enable mask for any restrainable error is 1, the *ECC_error* exception is generated for the restrainable error.

DG_U2$, DG_U2$x, UE_RAW_L2$INSD

DG_U2$, DG_U2$x, and UE_RAW_L2$INSID are asynchronous to thread execution. If such an error is detected, all the threads within the processor module logs the cause of the error into ASI_AFSR and activate an ECC_error trap, unless they are in the suspended state.

DG_D1$sTLB, UE_RAW_D1$INSD

These restrainable errors are asynchronous to thread execution. If such an error is detected, both the threads within the core which caused the error log it into ASI_AFSR and activate an ECC_error trap, unless they are in the suspended state. The threads in the other core are unaffected.

UE_DST_BETO

An UE_DST_BETO error is synchronous to thread execution. If such an error is detected in a given thread, only that thread logs the cause of the error into ASI_AFSR and activates an ECC_error trap, unless they are in the suspended state. All the other threads in the other core are unaffected.

P.1.5 instruction_access_error

instruction_access_error is synchronous to thread execution. If such an error is detected in a given thread, only that thread logs the cause of the error into ASI_ISFSR/TPC/ASI_ISFPAR, and activates an *instruction_access_error* trap. All the other threads are unaffected.

P.1.6 data_access_error

data_access_error is synchronous to thread execution. If such an error is detected in a given thread, only that thread logs the cause of the error into ASI_DSFSR/ASI_DSFAR/ ASI_DSFPAR, and activates an *data_access_error* trap. All the other threads are unaffected.

P.2 Action and Error Control

P.2.1 Registers Related to Error Handling

The following registers are related to the error handling.

- **ASI registers: Indicate an error.** All ASI registers in TABLE P-1 except `ASI_EIDR` and `ASI_ERROR_CONTROL` are used to specify the nature of an error to privileged software.
- **`ASI_ERROR_CONTROL`: Controls error action.** This register designates error detection masks and error trap enable masks.
- **`ASI_EIDR`: Marks errors.** This register identifies the error source ID for error marking.

TABLE P-1 lists the registers related to the error handling.

TABLE P-1 Registers Related to Error Handling

ASI	VA	R/W	Checking Code	Name	Defined in
$4C_{16}$	00_{16}	RW1C	None	ASI_ASYNC_FAULT_STATUS	P.7.1
$4C_{16}$	08_{16}	R	None	ASI_URGENT_ERROR_STATUS	P.4.1
$4C_{16}$	10_{16}	RW	Parity	ASI_ERROR_CONTROL	P.2.1
$4C_{16}$	18_{16}	R,W1AC	None	ASI_STCHG_ERROR_INFO	P.3.1
50_{16}	18_{16}	RW	None	ASI_IMMU_SFSR	F.10.9
58_{16}	18_{16}	RW	None	ASI_DMMU_SFSR	F.10.9
58_{16}	20_{16}	RW	Parity	ASI_DMMU_SFAR	F.10.10 of **Commonality**
$6E_{16}$	00_{16}	RW	Parity	ASI_EIDR	P.2.5

P.2.2 Summary of Actions Upon Error Detection

TABLE P-2 summarizes what happens when an error is detected.

TABLE P-2 Action Upon Detection of an Error *(1 of 3)*

	Fatal Error (FE)	Error State Transition Error (EE)	Urgent Error (UGE)	Restrainable Error (RE)
Error detection mask (the condition to suppress error detection)	None	When ASI_ECR.WEAK_ED = 1, the error detection is suppressed incompletely.	**I_UGE, IAE, DAE** When ASI_ECR.WEAK_ED = 1 or in the SUSPENDED state, error detection is suppressed incompletely. **A_UGE** In the SUSPENDED state, error detection is suppressed incompletely. Error detection except the register usage is suppressed when ASI_ECR.WEAK_ED = 1 or upon a condition unique to each error. Error detection at the register usage is suppressed by conditions unique to each error. Only some A_UGEs have the above unique conditions to suppress error detection; most do not.	None
Trap mask (the condition to suppress the error trap occurrence)	None	None	**I_UGE, IAE, IAE** the SUSPENDED state. **A_UGE** ASI_ECR.UGE_HANDLER = 1 or ASI_ECR.WEAK_ED = 1 The A_UGE detected during the trap is suppressed, is kept pending in the hardware, and causes the ADE trap when the trap is enabled or the SUSPENDED state.	ASI_ECR.UGE_HANDLER = 1 or ASI_ECR.WEAK_ED = 1 or PSTATE.IE = 0 or ASI_ECR.RTE_*xx* = 0, where RTE_*xx* is the trap enable mask for each error group. RTE_*xx* is RTE_CEDG or RTE_UE or the SUSPENDED state.

TABLE P-2 Action Upon Detection of an Error *(2 of 3)*

	Fatal Error (FE)	Error State Transition Error (EE)	Urgent Error (UGE)	Restrainable Error (RE)
Action upon the error detection	1. CPU enters CPU fatal state. 2. CPU informs the system of fatal error occurrence. 3. The FATAL reset (which is a form of POR reset) is issued to the whole system. 4. POR is sent to all CPUs in the system.	1. CPU enters error_state. 2. Watchdog reset (WDR) is set on the CPU.	**Detection of I_UGE** When ASI_ECR.UGE_HANDLER = 0, a single-ADE trap is set. Otherwise, when ASI_ECR.UGE_HANDLER = 1, a multiple-ADE trap is set. **Detection of A_UGE** When the trap is enabled, a single-ADE trap is set. When the trap is disabled, the trap condition is kept pending in hardware. **Detection of IAE** When ASI_ECR.UGE_HANDLER = 0, an IAE trap is set. Otherwise, a multiple-ADE trap is set. **Detection of DAE** When ASI_ECR.UGE_HANDLER = 0, a DAE trap is set. Otherwise, a multiple-ADE trap is set.	An *ECC_error* trap can occur even though ASI_AFSR does not indicate any detected error(s) corresponding to any trap-enable bit (RTE_UE or RTE_CEDG) set to 1 in ASI_ECR, in the following cases: 1. A pending detected error is erased from ASI_ASFR (by writing 1 to ASI_AFSR) after the error is detected but before the *ECC_error* trap is generated. 2. A pending CE or DG is erased by writing 1 to ASI_AFSR after the *ECC_error* trap is set by the UE error detection. 3. A pending UE is erased by writing 1 to ASI_AFSR after the *ECC_error* trap is set by CE or DG detection. Privileged software should ignore an *ECC_error* trap when the AFSR contains no errors corresponding to those enabled in ASI_ECR to cause a trap.
Priority of action when multiple types of errors are simultaneously detected	1 — CPU fatal state	2 — error_state	3 — ADE trap 4 — DAE trap 5 — IAE trap	6 — *ECC_error* trap
tt (trap type)	1 (RED_state)	2 (RED_state)	ADE: 40_{16} DAE: 32_{16} IAE: $0A_{16}$	63_{16}
Trap priority	1	1	ADE — 2 DAE — 12 IAE — 3	32
End-method of trapped instruction	Abandoned	Abandoned.	**ADE trap** Precise, retryable or nonretryable. See P.4.3. **IAE trap, DAE trap** Precise.	Precise

TABLE P-2 Action Upon Detection of an Error *(3 of 3)*

	Fatal Error (FE)	Error State Transition Error (EE)	Urgent Error (UGE)	Restrainable Error (RE)
Relation between TPC and instruction that caused the error	None	None	**I_UGE** For errors other than TLB write errors, the error was caused by the instruction pointed to by TPC or by the instruction subsequent in the control flow to the one indicated by TPC. For a TLB write error, the instruction pointed to by TPC or the already executed instruction previous in the control flow to the one indicated by TPC wrote a TLB entry and the TLB write failed. The TLB write error is detected after the instruction execution and before any trap, RETRY, or DONE instruction. **A_UGE** None. **IAE, DAE** The instruction pointed to by TPC caused the error.	None
Register that indicates the error	ASI_STCHG_ ERROR_INFO	ASI_STCHG_ ERROR_INFO	**I_UGE, A_UGE** ASI_UGESR **IAE** ASI_ISFSR **DAE** ASI_DSFSR	ASI_AFSR
Number of errors indicated at trap	All FEs are detected and accumulated in ASI_STCHG_ ERROR_INFO	All EEs are detected and accumulated in ASI_STCHG_ ERROR_INFO	**Single-ADE trap** All I_UGEs and A_UGEs detected at trap. **Multiple-ADE trap** The multiple-ADE indication + UGEs at first ADE trap. **IAE** One error **DAE** One error	All restrainable errors detected and accumulated in ASI_AFSR.
Error address indication register	None	None	I_UGE, A_UGE: None IAE: TPC DAE: ASI_DFAR	ASI_AFAR_D1 ASI_AFAR_U2

P.2.3 Extent of Automatic Source Data Correction for Correctable Error

Upon detection of the following correctable errors (CE), the CPU corrects the input data and uses the corrected data; however, the source data with the CE is not corrected automatically.

- CE in memory (DIMM)
- CE in ASI_INTR_DATA_R

Upon detection of other correctable errors, the CPU automatically corrects the source data containing the CE.

For correctable errors in ASI_INTR_DATA, no special action is required by privileged software because the erroneous data will be overwritten when the next interrupt is received. For CE in memory (DIMM), it is expected that privileged software will correct the error in memory.

P.2.4 Error Marking for Cacheable Data Error

Error Marking for Cacheable Data

Error marking for cacheable data involves the following action:

- When a hardware unit first detects an uncorrected error in the cacheable data, the hardware unit replaces the data and ECC of the cacheable data with a special pattern that identifies the original error source and signifies that the data is already marked.

The error marking helps identify the error source and prevent multiple error reports by a single error even after several cache lines transfer with uncorrected data.

The following data are protected by the single-bit error correction and double-bit error detection ECC code attached to every doubleword:

- Main memory (DIMM)
- Jupiter Bus packet data containing cache line data and interrupt packet data
- U2 (unified level 2) cache data
- D1 cache data
- The cacheable area block held by the channel

The ECC applied to these data is the ECC specified for Jupiter Bus.

When the CPU and channel detect an uncorrected error in the above cacheable data that is not yet marked, the CPU and channel execute error marking for the data block with an UE.

Whether the data with UE is marked or not is determined by the syndrome of the doubleword data, as shown in TABLE P-3.

TABLE P-3 Syndrome for Data Marked for Error

Syndrome	Error Marking Status	Type of Uncorrected Error
$7F_{16}$	Marked	Marked UE
Multibit error pattern except for $7F_{16}$	Not marked yet	Raw UE

The syndrome $7F_{16}$ indicates a 3-bit error in the specified location in the doubleword. The error marking replaces the original data and ECC to the data and ECC, as described in the following section. The probability of syndrome $7F_{16}$ occurrence other than the error marking is considered to be zero.

The Format of Error-Marking Data

When the raw UE is detected in the cacheable data doubleword, the erroneous doubleword and its ECC are replaced in the data by error marking, as listed in TABLE P-4.

TABLE P-4 Format of Error-Marked Data

Data/ECC	Bit	Value
data	63	Error bit. The value is unpredictable.
	62:56	0 (7 bits).
	55:42	ERROR_MARK_ID (14 bits).
	41:36	0 (6 bits).
	35	Error bit. The value is unpredictable.
	34:23	0 (12 bits).
	22	Error bit. The value is unpredictable.
	21:14	0 (8 bits).
	13:0	ERROR_MARK_ID (14 bits).
ECC		The pattern indicates 3-bit error in bits 63, 35, and 22, that is, the pattern causing the $7F_{16}$ syndrome.

The `ERROR_MARK_ID` (14 bits wide) identifies the error source. The hardware unit that detects the error provides the error source_ID and sets the `ERROR_MARK_ID` value.

The format of ERROR_MARK_ID<13:0> is defined in TABLE P-5.

TABLE P-5 ERROR_MARK_ID Bit Description

Bit	Value
13:12	Module_ID: Indicates the type of error source hardware as follows: 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 set to 0. Otherwise, the identification number of each Module type is set to Source ID.

ERROR_MARK_ID Set by CPU

TABLE P-6 shows the ERROR_MARK_ID set by the CPU.

TABLE P-6 ERROR_MARK_ID Set by CPU

Type of data with RAW UE	Module_ID value (binary)	Source_ID value
Incoming data from Jupiter Bus	00_2 (Memory system)	0
Outgoing data to Jupiter Bus	ASI_EIDR<13:12>. 10_2 (CPU) is expected.	ASI_EIDR (Identifier of self CPU)
U2 cache data, D1 cache data	ASI_EIDR<13:12>. 10_2 (CPU) is expected.	ASI_EIDR (Identifier of self CPU)

P.2.5 ASI_EIDR

The ASI_EIDR register designates the source ID in the ERROR_MARK_ID of the CPU.

[1] Register name: ASI_EIDR
[2] ASI: $6E_{16}$
[3] VA: 00_{16}
[4] Error checking: Parity.
[5] Format & function: See TABLE P-7.

TABLE P-7 ASI_EIDR Bit Description

Bit	Name	RW	Description
63:14	*Reserved*	R	Always 0.
13:0	ERROR_MARK_ID	RW	ERROR_MARK_ID for the error caused by the CPU.

P.2.6 Control of Error Action (ASI_ERROR_CONTROL)

Error detection masking and the action after error detection are controlled by the value in ASI_ERROR_CONTROL, as defined in TABLE P-8.

[1] Register name: ASI_ERROR_CONTROL (ASI_ECR)
[2] ASI: $4C_{16}$
[3] VA: 10_{16}
[4] Error checking: None
[5] Format & function: See TABLE P-8.
[6] Initial value at reset: Hard POR: ASI_ERROR_CONTROL.WEAK_ED is set to 1. Other fields are set to 0.
Other resets: After UGE_HANDLER and WEAK_ED are copied into ASI_STCHG_ERROR_INFO, all fields in ASI_ERROR_CONTROL are set to 0.

The ASI_ERROR_CONTROL register controls error detection masking, error trap occurrence masking, and the multiple-ADE trap occurrence. The register fields are described in TABLE P-8.

TABLE P-8 ASI_ERROR_CONTROL Bit Description

Bit	Name	RW	Description
9	**RTE_UE**	RW	Restrainable Error Trap Enable submask for UE and Raw UE. The bit works as defined in TABLE P-2.
8	**RTE_CEDG**	RW	Restrainable Error Trap Enable submask for Corrected Error (CE) and Degradation (DG). The bit works as defined in TABLE P-2.

TABLE P-8 ASI_ERROR_CONTROL Bit Description *(Continued)*

Bit	Name	RW	Description
1	**WEAK_ED**	RW	Weak Error Detection. Controls whether the detection of I_UGE and DAE is suppressed: When WEAK_ED = 0, error detection is not suppressed. When WEAK_ED = 1, error detection is suppressed if the CPU can continue processing. When I_UGE or DAE is detected during instruction execution while WEAK_ED = 1, the value of the output register or the store target memory location becomes unpredictable. Even if WEAK_ED = 1, I_UGE or DAE is detected and the corresponding trap is set when the CPU cannot continue processing by ignoring the error. WEAK_ED is the trap disabling mask for A_UGE and restrainable errors, as defined in TABLE P-2. When a multiple-ADE trap is set (I_UGE, IAE, or DAE detection while ASI_ERROR_CONTROL.UGE_HANDLER = 1), WEAK_ED is set to 1 by hardware.
0	**UGE_HANDLER**	RW	Designates whether hardware can expect a UGE handler to run in privileged software (operating system) when a UGE error occurs: 0: Hardware recognizes that the privileged software UGE handler does not run. 1: Hardware expects that the privileged software UGE handler runs. UGE_HANDLER is the trap disabling mask for A_UGE and restrainable errors, as defined in TABLE P-2. The value of UGE_HANDLER determines whether a multiple-ADE trap is caused or not upon detection of I_UGE, IAE, and DAE. When an *async_data_error* trap occurs, UGE_HANDLER is set to 1. When a RETRY or DONE instruction is completed, UGE_HANDLER is set to 0.
Other	*Reserved*	R	Always reads as 0.

P.3 Fatal Error and `error_state` Transition Error

P.3.1 ASI_STCHG_ERROR_INFO

The `ASI_STCHG_ERROR_INFO` register stores detected FATAL error and `error_state` transition error information, for access by OBP (Open Boot PROM) software.

[1]	Register name:	`ASI_STCHG_ERROR_INFO`
[2]	ASI:	$4C_{16}$
[3]	VA:	18_{16}
[4]	Error checking:	None
[5]	Format & function:	See TABLE P-9
[6]	Initial value at reset:	Hard POR: All fields are set to 0. Other resets: Values are unchanged.
[7]	Update policy:	Upon detection of each related error, the corresponding bit in ASI_STCHG_ERROR_INFO is set to 1. Writing 1 to bit 0 erases all error indications in ASI_STCHG_ERROR_INFO (sets all bits in the register, including bit 0, to 0).

TABLE P-9 describes the fields in the `ASI_STCHG_ERROR_INFO` register.

TABLE P-9 ASI_STCHG_ERROR_INFO bit description

Bit	Name	RW	Description
63:34	*Reserved*	R	Always 0.
33	ECR_WEAK_ED	R	ASI_ERROR_CONTROL.WEAK_ED is copied into this field at the beginning of a POR or watchdog reset.
32	ECR_UGE_HANDLER	R	ASI_ERROR_CONTROL.UGE_HANDLER is copied into this field at the beginning of the POR or watchdog reset.
31:24	*Reserved*	R	Always 0.
23	EE_MODULE	RW	Error state transient error requires module degradation, Sticky
22	EE_CORE	RW	Error state transient error requires core degradation, Sticky
21	EE_THREAD	RW	Error state transient error requires thread degradation, Sticky
20	UGE_MODULE	RW	Urgent error requires module degradation, Sticky
19	UGE_CORE	RW	Urgent error requires core degradation, Sticky
18	UGE_THREAD	RW	Urgent error requires thread degradation, Sticky
17	rawUE_MODULE	RW	RawUE detected in L2$, sticky
16	rawUE_CORE	RW	RawUE detected in L1$, sticky

TABLE P-9 ASI_STCHG_ERROR_INFO bit description

Bit	Name	RW	Description
15	EE_DCUCR_MCNTL_ECR	R	Uncorrectable error in any of the following: (A) ASI_DCUCR (A) ASI_MCNTL (A) ASI_ECR
14	EE_OTHER	R	Set to 1 upon detection of `error_state` transition errors not listed elsewhere. The field is always 0 for SPARC64 VI.
13	EE_TRAP_ADR_UE	R	When hardware calculated the trap address to cause a trap, the valid address could not be obtained because of a UE in `%tba`, a UE in `%tt`, or a UE in the address calculator.
12	FE_OPSR		An uncorrectable error occurred in `OPSR` (Operation Status Register); valid CPU operation after such an error cannot be guaranteed. `OPSR` is the hardware mode-setting register. `OSPR` is not visible to software and is set by a JTAG command.
11	EE_WDT_IN_MAXTL	R	A watchdog time-out occurred while `TL` = `MAXTL`.
10	EE_SECOND_WDT	R	A second watchdog time-out was detected after an *async_data_error* exception with watchdog time-out indication (first watchdog time-out) was generated.
9	EE_SIR_IN_MAXTL	R	An SIR occurred while `TL` = `MAXTL`.
8	EE_TRAP_IN_MAXTL	R	A trap occurred while `TL` = `MAXTL`.
7:3	*Reserved*	R	Always 0.
2	FE_OTHER	R	Set to 1 upon detection of urgent errors not listed elsewhere.
1	FE_U2TAG_UE	R	Upon detection of the corresponding error, set to 1.
0	FE_JBUS_UE	RW	An uncorrected error in the Jupiter bus. Writing 1 to this bit sets all fields in this register to 0.

Compatibility Note – EE_OPSR in SPARC64 V is changed to FE_OPSR in SPARC64 VI. There are no changes in the other error_state transition errors.

P.3.2 `error_state` Transition Error in Suspended Thread

SPARC64 VI allows itself to enter suspend state by suspend instruction. Only POR, WDR, XDR, *interrupt_vector* and *interrupt_level_n* exceptions can resume it back to running state. If an error occurred the resources related to those exceptions, this thread suspends forever. To prevent to fall into this situation, an urgent error regarding following registers are reported as `error_state` transition error in suspend 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 presents details about urgent errors: status monitoring, actions, and end-methods.

P.4.1 URGENT ERROR STATUS (ASI_UGESR)

[1]	Register name:	ASI_URGENT_ERROR_STATUS
[2]	ASI:	$4C_{16}$
[3]	VA:	08_{16}
[4]	Error checking:	None
[5]	Format & function:	See TABLE P-10.
[6]	Initial value at reset:	Hard POR: All fields are set to 0. Other resets: The values of all ASI_UGESR fields are unchanged.

The ASI_UGESR register contains the following information when an *async_data_error* (ADE) exception is generated.

- Detected I_UGEs and A_UGEs, and related information
- The type of second error to cause multiple *async_data_error* traps

TABLE P-10 describes the fields of the ASI_UGESR register. In the table, the prefixes in the name field have the following meaning:

- IUG_ Instruction Urgent error
- IAG_ Autonomous Urgent error
- IAUG_ The error detected as both I_UGE and A_UGE

TABLE P-10 ASI_UGESR Bit Description *(1 of 4)*

Bit	Name	RW	Description
Each bit in ASI_UGESR<22:8> indicates the occurrence of its corresponding error in a single-ADE trap as follows: 0: The error is not detected. 1: The error is detected. Each bit in ASI_UGESR<22:16> indicates an error in a CPU register. The error detection conditions for these errors are defined in *Internal Register Error Handling* on page 182.			

TABLE P-10 ASI_UGESR Bit Description *(2 of 4)*

Bit	Name	RW	Description
22	**IAUG_CRE**	R	Uncorrectable error in any of the following: (IA) ASI_EIDR (IA) ASI_PA_WATCH_POINT when enabled (IA) ASI_VA_WATCH_POINT when enabled (I) ASI_AFAR_D1 (I) ASI_AFAR_U2 (I) ASI_INTR_R (A) ASI_INTR_DISPATCH_W (UE at store) (IA) SOFTINT (IA) STICK (IA) STICK_COMP
21	**IAUG_TSBCTXT**	R	Uncorrectable error in any of the following: (IA) ASI_DMMU_TSB_BASE (IA) ASI_DMMU_TSB_PEXT (IA) ASI_DMMU_TSB_SEXT (IA) ASI_DMMU_TSB_NEXT (IA) ASI_PRIMARY_CONTEXT (IA) ASI_SECONDARY_CONTEXT (IA) ASI_IMMU_TSB_BASE (IA) ASI_IMMU_TSB_PEXT (IA) ASI_IMMU_TSB_NEXT
20	**IUG_TSBP**	R	Uncorrectable error in any of the following: (I) ASI_DMMU_TAG_TARGET (I) ASI_DMMU_TAG_ACCESS (I) ASI_DMMU_TSB_8KB_PTR (I) ASI_DMMU_TSB_64KB_PTR (I) ASI_DMMU_TSB_DIRECT_PTR (I) ASI_IMMU_TAG_TARGET (I) ASI_IMMU_TAG_ACCESS (I) ASI_IMMU_TSB_8KB_PTR (I) ASI_IMMU_TSB_64KB_PTR
19	**IUG_PSTATE**	R	Uncorrectable error in any of the following: `%pstate`, `%pc`, `%npc`, `%cwp`, `%cansave`, `%canrestore`, `%otherwin`, `%cleanwin`, `%pil`, `%wstate`
18	**IUG_TSTATE**	R	Uncorrectable error in any of `%tstate`, `%tpc`, `%tnpc`.
17	**IUG_%F**	R	Uncorrectable error in any floating-point register or in the FPRS, FSR, or GSR register.
16	**IUG_%R**	R	Uncorrectable error in any general-purpose (integer) register, or in the Y, CCR, or ASI register.
14	**IUG_WDT**	R	Watchdog timeout first time. Indicates the first watchdog timeout. If IUG_WDT = 1 when a single-ADE trap occurs, the instruction pointed to by TPC is abandoned and its result is unpredictable.

TABLE P-10 ASI_UGESR Bit Description *(3 of 4)*

Bit	Name	RW	Description
10	**IUG_DTLB**	R	Uncorrectable error in DTLB during load, store, or demap. Indicates that one of the following errors was detected during a data TLB access: • An uncorrectable error in TLB data or TLB tag was detected when an LDXA instruction attempted to read ASI_DTLB_DATA_ACCESS or ASI_DTLB_TAG_ACCESS. TPC indicates either the instruction causing the error or the previous instruction. • A store to the data TLB or a demap of the data TLB failed. TPC indicates either the instruction causing the error or the instruction following the one that caused the error.
9	**IUG_ITLB**	R	Uncorrectable error in ITLB during load, store, or demap. Indicates that one of the following errors was detected during an instruction TLB access: • An uncorrectable error in TLB data or TLB tag was detected when an LDXA instruction attempted to read ASI_ITLB_DATA_ACCESS or ASI_ITLB_TAG_ACCESS. TPC indicates either the instruction causing the error or the previous instruction. • A store to the instruction TLB or a demap of the instruction TLB failed. TPC indicates either the instruction causing the error or the following instruction.
8	**IUG_COREERR**	R	CPU core error. Indicates an uncorrectable error in a CPU internal resource used to execute instructions, which cannot be directly accessed by software. When there is an uncorrectable error in a program-visible register and the instruction reading the register with UE is executed, the error in the register is always indicated. In this case, IUG_COREERR may or may not be indicated simultaneously with the register error.
5:4	**INSTEND**	R	Trapped instruction end-method. Upon a single *async_data_error* trap without watchdog time-out detection, INSTEND indicates the instruction end-method of the trapped instruction pointed to by TPC as follows: 00_2: Precise 01_2: Retryable but not precise 10_2: *Reserved* 11_2: Not retryable See Section P.4.3 for the instruction end-method for the *async_data_error* trap. When a watchdog time-out is detected, the instruction end-method is undefined.
3	**PRIV**	R	Privileged mode. Upon a single *async_data_error* trap, the PRIV field is set as follows: When the value of PSTATE.PRIV immediately before the single-ADE trap is unknown because of an uncorrectable error in PSTATE, ASI_UGESR.PRIV is set to 1. Otherwise, the value of PSTATE.PRIV immediately before the single-ADE trap is copied to ASI_UGESR.PRIV.
2	**MUGE_DAE**	R	Multiple UGEs caused by DAE. Upon a single-ADE, MUGE_DAE is set to 0. Upon a multiple-ADE trap caused by a DAE, MUGE_DAE is set to 1. Upon a multiple-ADE trap not caused by a DAE, MUGE_DAE is unchanged.
1	**MUGE_IAE**	R	Multiple UGEs caused by IAE. Upon a single-ADE trap, MUGE_IAE is set to 0. Upon a multiple-ADE trap caused by an IAE, MUGE_IAE is set to 1. Upon a multiple-ADE trap not caused by an IAE, MUGE_IAE is unchanged.

TABLE P-10 ASI_UGESR Bit Description *(4 of 4)*

Bit	Name	RW	Description
0	**MUGE_IUGE**	R	Multiple UGEs caused by I_UGE. Upon a single-ADE trap, MUGE_IUGE is set to 0. Upon a multiple-ADE trap caused by an I_UGE, MUGE_IUGE is set to 1. Upon a multiple-ADE trap not caused by an I_UGE, MUGE_IUGE is unchanged.
Other	*Reserved*	R	Always 0.

P.4.2 Action of *async_data_error* (ADE) Trap

The single-ADE trap and the multiple-ADE trap are generated upon the conditions defined in TABLE P-2 on page 161. The actions upon their occurrence are defined in more detail in this section. For convenience, the shorthand ADE is used to refer to *async_data_error*.

1. Conditions that cause an ADE trap:

An ADE trap occurs when one of the following conditions is satisfied:

- 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 change, trap target address calculation, and TL manipulation.

The following actions are executed in this order:

a. State transition

if (TL = MAXTL), the CPU enters error_state and abandons the ADE trap;

else if (CPU is in execution state && (TL = MAXTL – 1)), then the CPU enters RED_state.

b. Trap target address calculation

When the CPU is in execution state, trap target address is calculated by %tba, %tt, and %tl.

Otherwise, the CPU is in RED_state and the trap target address is set to RSTVaddr + $A0_{16}$.

c. TL increases: TL ← TL + 1.

3. Save the old value into TSTATE, TPC, and TNPC.

PSTATE, PC, and NPC immediately before the ADE trap are copied into TSTATE, TPC, and TNPC, respectively. If the copy source register contains an uncorrectable error, the copy target register also contains the UE.

4. Set the specific register setting:

The following three sets of registers are updated:

a. Update and validation of specific registers.

Hardware writes the registers listed in TABLE P-11**.**

TABLE P-11 Registers Written for Update and Validation

Register	Condition For Writing	Value Written
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 UE	0.
FSR, GSR	When the register contains UE	If either FSR or GSR contains a UE, 0 is written to that register. When 0 is written to FSR and/or GSR upon a single-ADE trap, ASI_UGESR.IUG_%F is set to 1.
CWP, CANSAVE, CANRESTORE, OTHERWIN, CLEANWIN	When the register contains UE	Any register among CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN that contains a UE is written to 0. When 0 is written to one of these registers upon a single-ADE trap, ASI_UGESR.IUG_PSTATE = 1 is set to 1.
TICK	When the register contains UE	NPT = 1, Counter = 0.
TICK_COMPARE	When the register contains UE	INT_DIS = 1, TICK_CMPR = 0.

The error(s) in a written register are removed by setting the correct value to the error checking (parity) code during the full write of the register.

Errors in registers other than those listed above and any errors in the TLB entry remain.

b. Update of `ASI_UGESR`, as shown in TABLE P-12**.**

TABLE P-12 `ASI_UGESR` Update for Single and Multiple-ADE Exceptions

Bit	Field	Update upon a Single-ADE Trap	Update upon a Multiple-ADE Traps
63:6	Error indication	All bits in this field are updated. All I_UGEs and A_UGEs detected at the trap are indicated simultaneously.	Unchanged.
5:4	**INSTEND**	The instruction end-method of the instruction referenced by TPC is set.	Unchanged.
2	**MUGE_DAE**	Set to 0.	If the multiple-ADE trap was caused by a DAE, MUGE_DAE is set to 1. Otherwise, MUGE_DAE is unchanged.
1	**MUGE_IAE**	Set to 0.	If the multiple-ADE trap was caused by an IAE, MUGE_IAE is set to 1. Otherwise, MUGE_IAE is unchanged.
0	**MUGE_IUGE**	Set to 0.	If the multiple-ADE trap was caused by an I_UGE, MUGE_IUGE is set to 1. Otherwise, MUGE_IUGE is unchanged.

c. Update of `ASI_ERROR_CONTROL`

Upon a single-ADE trap, `ASI_ERROR_CONTROL.UGE_HANDLER` is set to 1. During the period after the single-ADE trap occurs and before a RETRY or DONE instruction is executed, `UGE_HANDLER` = 1 tells hardware that the urgent error handler is running.

Upon a multiple *async_data_error* trap, `ASI_ERROR_CONTROL.WEAK_ED` is set to 1 and the CPU starts running in the weak error detection state.

5. Set `ASI_ERROR_CONTROL.UGE_HANDLER` to 0.

Upon completion of a RETRY or DONE instruction, `ASI_ERROR_CONTROL.UGE_HANDLER` is set to 0.

P.4.3 Instruction End-Method at ADE Trap

In SPARC64 VI, upon occurrence of the ADE trap, the trapped instruction referenced by `TPC` ends by using one of the following instruction end-methods:

- Precise
- Retryable but not precise (not included in JPS1)
- Not retryable (not included in JPS1)

Upon a single-ADE trap, the trapped instruction end-method is indicated in `ASI_UGESR.INSTEND`.

TABLE P-13 defines each instruction end-method after an ADE trap.

TABLE P-13 Instruction End-Method After *async_data_error* Exception

	Precise	Retryable But Not Precise	Not Retryable
Instructions executed after the last ADE, IAE, or DAE trap and before the trapped instruction referenced by TPC.	Ended (Committed). The instructions without UGE complete as defined in the architecture. The instruction with UGE has unpredictable value at its output (destination register or, in the case of a store instruction, destination memory location).		

TABLE P-13 Instruction End-Method After *async_data_error* Exception

	Precise	Retryable But Not Precise	Not Retryable
The trapped instruction referenced by TPC	Not executed.	The output of the instruction is incomplete. Part of the output may be changed, or the invalid value may be written to the instruction output. However, the modification to the invalid target that is not defined as instruction output is not executed. The following modifications are not executed: • Store to the cacheable area including cache. • Store to the noncacheable area. • Output to the source register of the instruction (destructive overlap)	The output of the instruction is incomplete. Part of the output may be changed, or the invalid value may be written to the instruction output. However, the modification to the invalid target that is not defined as instruction output is not executed. A store to an invalid address is not executed. (Store to a valid address with uncorrected data may be executed.)
Instructions to be executed after the instruction referenced by TPC	Not executed.	Not executed.	Not executed.
The possibility of resuming the trapped program by executing the RETRY instruction to the %tpc when the trapped program is not damaged at the single-ADE trap	Possible.	Possible.	Impossible.

P.4.4 Expected Software Handling of ADE Trap

The expected software handling of an ADE trap is described by the pseudo C code below. The main purpose of this flow is to recover from the following errors as much as possible:

- An error in the CPU internal RAM or register file
- An error in the accumulator
- An error in the CPU internal temporary registers and data bus

```
void
expected_software_handling_of_ADE_trap()
{
/* Only %r0-%r7 can be used from here to Point#1 because the register window
   control registers may not have valid value until Point#1. It is
   recommended that only %r0-%r7 are used as general-purpose registers (GPR)
   in the whole single-ADE trap handler, if possible. */
ASI_SCRATCH_REGp ← %rX;
ASI_SCRATCH_REGq ← %rY;
%rX ← ASI_UGESR;

if ((%rX && 0x07) ≠ 0) {
```

```
      /*  multiple-ADE trap occurrence */
     invoke panic routine and take system dump as much as possible
     with the running environment of ASI_ERROR_CONTROL.WEAK_ED == 1;
}

if (%rX.IUG_%R == 1) {
    %r1-%r31 except %rX and %rY ← %r0;
    %y ← %r0;
    %tstate.pstate ← %r0;  /* because ccr or asi field in %tstate.pstate
                              contains the error */
}
else {
    save required %r1-%r7 to the ADE trap save area, using %rX, %rY,
       ASI_SCRATCH_REGp and ASI_SCRATCH_REGq;
    /* whole %r save and restore is required to retry the context
        with PSTATE.AG == 1 */
}

if (ASI_UGESR.IUG_PSTATE == 1) {
    %tstate.pstate ← %r0;
    %tpc ← %r0;
    %pil ← %r0;
    %wstate ← %r0;
    All general-purpose registers in the register window ← %r0;
    Set the register window control registers
        (CWP, CANSAVE, CANRESTORE, OTHERWIN, CLEANWIN) to appropriate values;
}

/*  Point#1: Program can use the general-purpose registers except %r0-%r7
   after this because the register window control registers were validated
   in the above 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)) {
    Write to each register with an error indication, to erase as many
        register errors as possible;
}

if (ASI_UGESR.IUG_DTLB == 1) {
    execute demap_all for DTLB;
    /* A locked fDTLB entry with uncorrectable error is not removed by this
      operation.  A locked fDTLB entry with UE never detects its tag match or
      causes the data_access_error trap when its tag matches at the DTLB
      reference for address translation. */
}

if (ASI_UGESR.IUG_ITLB == 1) {
    execute demap_all for ITLB;
   /* A locked fITLB entry with uncorrectable error is not removed by this
      operation.  A locked fITLB entry with UE never detects its tag  match
      or causes the data access error trap when its tag matches at the ITLB
      reference for address translation. */
}
```

```
if ((ASI_UGESR.bits22:14 == 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)
        resume the trapped context by use of the RETRY instruction;
    else
        invoke panic routine because of too many ADE trap retries;
}
else if ((ASI_UGESR.bits22:18 == 0) &&
        (ASI_UGESR.bits15:14 == 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 trapped and continue system operation;
    else
       invoke panic routine because of too may ADE trap user kill;
}
else
    invoke panic routine because of unrecoverable urgent error;
}
```

P.5 Instruction Access Errors

See Appendix , *Memory Management Unit*, for details.

P.6 Data Access Errors

See Appendix , *Memory Management Unit*, for details.

P.7 Restrainable Errors

This section describes the registers—ASI_ASYNC_FAULT_STATUS, ASI_ASYNC_FAULT_ADDR_D1, and ASI_ASYNC_FAULT_ADDR_U2—that define the restrainable errors and explains how software handles these errors.

P.7.1 ASI_ASYNC_FAULT_STATUS (ASI_AFSR)

[1]	Register name:	ASI_ASYNC_FAULT_STATUS (ASI_AFSR)
[2]	ASI:	$4C_{16}$
[3]	VA:	00_{16}
[4]	Error checking:	None
[5]	Format & function:	See TABLE P-14
[6]	Initial value at reset:	Hard POR: All fields in ASI_AFSR are set to 0.
		Other resets: Values in ASI_AFSR are unchanged.

The ASI_ASYNC_FAULT_STATUS register holds the detected restrainable error sticky bits. TABLE P-14 describes the fields of this register. In the table, the prefixes in the name field have the following meaning:

- DG_ Degradation error
- CE_ Correctable Error
- UE_ Uncorrectable Error

TABLE P-14 ASI_ASYNC_FAULT_STATUS Bit Description

Bit	Name	RW	Description
12	**DG_U2$x**	RW1C	Degradation in U2$. This bit is set when automatic way reduction is applied in U2$ due to U2$ tag errors in system.
11	**DG_U2$**	RW1C	Degradation in U2$. This bit is set when automatic way reduction is applied in U2$ due to U2$ errors in CPU or System.
10	**DG_D1$sTLB**	RW1C	Degradation in L1$ and sTLB. This bit is set when automatic way reduction is applied in I1$, D1$, sITLB, sDTLB, uITLB and uDTLB
9	Reserved	R	Always reads as 0; writes are ignored.
3	**UE_DST_BETO**	RW1C	Disrupting store JBUS bus error or time-out.
2	Reserved	R	Always reads as 0; writes are ignored.
1	**UE_RAW_L2$INSD**	RW1C	Raw UE in L2 cache inside data.
0	**UE_RAW_D1$INSD**	RW1C	Raw UE in D1 cache inside data.
Other	Reserved	R	Always reads as 0; writes are ignored.

Note – Disrupting store bus error or time-out is reported as either AFSR#UE_DST_BETO, DSFSR#BERR, or DSFSR#RTO exclusively.

P.7.2 ASI_ASYNC_FAULT_ADDR_D1

The register is always reads as 0; write to this register is ignored in SPARC64 VI.

P.7.3 ASI_ASYNC_FAULT_ADDR_U2

The register is always reads as 0; write to this register is ignored in SPARC64 VI.

P.7.4 Expected Software Handling of Restrainable Errors

Error recording and information is expected for all restrainable errors.

The expected software recovery from each type of each restrainable error is described below.

- **DG_L1$, DG_U2$, DG_U2$x** — The following status for the CPU is reported:
  - Performance is degraded by the way reduction in I1$, D1$, U2$, sITLB, or sDTLB.
  - CPU availability may be slightly down. If only one way facility is available among I1$, D1$, U2$, sITLB, and sDTLB and further way reduction is detected for this facility, the error_state transition error is detected.

    Software stops the use of the CPU, if required.
- **UE_DST_BETO** — This error is caused by either:
  - Invalid DTLB entry is specified, or
  - Invalid memory access instruction with physical address access ASI is executed in privileged software.

  This error is always caused by a mistake in privileged software. Record the error and correct the erroneous privileged software.
- **UE_RAW_L2$INSD**, and **UE_RAW_D1$INSD** — Software handles these errors as follows:
  - Correct the cache line data containing the uncorrected error by executing a block store with commit instruction, if possible. Note that the original data is deleted by this operation.
  - For UE_RAW_L2$FILL, avoid using the memory block with the UE as much as possible.
- No error indication in ASI_AFSR at *ECC_error* trap — Ignore the *ECC_error* trap.

  This situation may occur at the condition described in the TABLE P-2 on page 161 (see the third row, last column").

P.8 Internal Register Error Handling

This section describes error handling for the following registers.

- Nonprivileged and Privileged registers
- ASR registers
- ASI registers

P.8.1 Nonprivileged and Privileged Registers Error Handling

The terminology used in TABLE P-15 is defined as follows:

Column	Term	Meaning
Error Detect Condition	InstAccess	The error is detected when the instruction accesses the register.
Correction	W	The error indication is removed when an instruction performs a full write to the register
	ADE trap	The error is removed by a full write to the register in the *async_data_error* hardware trap sequence.

TABLE P-15 shows error handling for nonprivileged and privileged registers.

TABLE P-15 Nonprivileged and Privileged Registers Error Handling

Register Name	RW	Error Protect	Error Detect Condition	Error Type	Correction
%r*n*	RW	Parity	InstAccess	IUG_%R	W
%f*n*	RW	Parity	InstAccess	IUG_%F	W
PC		Parity	Always	IUG_PSTATE	ADE trap
nPC		Parity	Always	IUG_PSTATE	ADE trap
PSTATE	RW	Parity	Always	IUG_PSTATE	ADE trap
TBA	RW	Parity	PSTATE.RED = 0	error_state	W (by OBP)
PIL	RW	Parity	PSTATE.IE = 1 InstAccess	IUG_CORE 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	InstAccess	IUG_TSTATE	W

TABLE P-15 Nonprivileged and Privileged Registers Error Handling

Register Name	RW	Error Protect	Error Detect Condition	Error Type	Correction
TNPC	RW	Parity	InstAccess	IUG_TSTATE	W
TSTATE	RW	Parity	InstAccess	IUG_TSTATE	W
WSTATE	RW	Parity	Always	IUG_PSTATE	W
VER	R	None	—	—	—
FSR	RW	Parity	Always	IUG_%F	ADE trap, W
Y	RW	Parity	InstAccess	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[1]	IUG_COREERR	ADE trap[2], W
PC	R	Parity	Always	IUG_PSTATE	ADE trap
FPRS	RW	Parity	Always	IUG_%F	ADE trap, W

1.Notified as `error_state` transition error in suspended state.

2.TICK, TICK_COMPARE are set to 0x8000_0000_0000_0000 on ADE trap for correction.

P.8.2 ASR Error Handling

The terminology used in TABLE P-16 is defined as follows:

Column	Term	Meaning
Error Detect Condition	AUG always	The error is detected while (ASI_ERROR_CONTROL.UGE_HANDLER = 0) && (ASI_ERROR_CONTROL.WEAK_ED = 0)
	InstAccess	The error is detected when the instruction accesses the register.
Error Type	(I)AUG_*xxx*	The error is indicated by `ASI_UGESR.IAUG_`*xxx* = 1, and the error is an autonomous urgent error.
	I(A)UG_*xxx*	The error is indicated by `ASI_UGESR.IAUG_`*xxx* = 1, and the error is an instruction urgent error.
Correction	W	The error is removed by a full write to the register by an instruction.
	ADE trap	The error is removed by a full write to the register in the *async_data_error* hardware trap sequence.

TABLE P-16 shows the handling of ASR errors.

TABLE P-16 ASR Error Handling

ASR Number	Register Name	RW	Error Protect	Error Detect Condition	Error Type	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
				InstAccess	I(A)UG_CRE	W
25	STICK_COMPARE	RW	Parity	AUG always[1]	(I)AUG_CRE	W
				InstAccess	I(A)UG_CRE	W

1.Notified as `error_state` transition error in suspended state.

STICK Behavior upon Error

When error is occurred in %stick register, countup is stopped regardless of the error detect condition described in TABLE P-16.

P.8.3 ASI Register Error Handling

The terminology used in TABLE P-17 is defined as follows:

Column	Term	Meaning
Error Protect	Parity	Parity protected.
	ECC	ECC (double-bit error detection, single-bit error correction) protected.
	Gecc	Generated ECC.
	PP	Parity propagation. The parity error in the input registers to calculate the register value is propagated.

Column	Term	Meaning
Error Detect Condition	Always	Error is always checked.
	AUG always	Error is checked when (ASI_ERROR_CONTROL.UGE_HANDLER = 0) && (ASI_ERROR_CONTROL.WEAK_ED = 0).
	LDXA	Error is checked when the register is read by LDXA instruction.
	LDXA #I	Error is checked when the register is read by LDXA instruction. Also, the register is used for the calculation of IMMU_TSB_8KB_PTR and IMMU_TSB_64KB_PTR. When the register has a UE and the register is used for the calculation of ASI_IMMU_TSB_PTR registers, the UE is propagated to the ASI_IMMU_TSB_PTR registers. Upon execution of the LDXA instruction to read ASI_IMMU_TSB_PTR with the propagated UE, the *IUG_TSBP* error is detected.
	LDXA #D	Error is checked when the register is read by LDXA instruction. Also, the register is used for the calculation of DMMU_TSB_8KB_PTR, DMMU_TSB_64KB_PTR, and DMMU_TSB_DIRECT_PTR. When the register has a UE and the register is used for the calculation of ASI_DMMU_TSB_PTR registers, the UE is propagated to the ASI_DMMU_TSB_PTR registers. Upon execution of the LDXA instruction to read ASI_DMMU_TSB_PTR with the propagated UE, the *IUG_TSBP* error is detected.
	ITLB write	Error is checked at the ITLB update timing after completion of the STXA instruction to write or demap an ITLB entry.
	DTLB write	Error is checked at the DTLB update timing after the completion of the STXA instruction to write or demap a DTLB entry.
	Use for TLB	Error is checked when the register is used for a TLB reference.
	Enabled	Error is checked when the facility is enabled.
	intr_receive	Error is checked when the Jupiter Bus interrupt packet is received. When an uncorrectable error is detected in the received interrupt packet, the vector interrupt trap is caused but ASI_INTR_RECEIVE.BUSY = 0 is set. In this case, a new interrupt packet can be received after software writes ASI_INTR_RECEIVE.BUSY = 0.
	BV interface	Uncorrected error in the Barrier Variable transfer interface between the processor and the memory system is checked during the AUG_always period.
Error Type	error_state	`error_state` transition error.
	(I)AUG_*xxxx*	The error is indicated by ASI_UGESR.IAUG_*xxxx* = 1, and the error class is autonomous urgent error.
	I(A)UG_*xxxx*	The error is indicated by ASI_UGESR.IAUG_*xxxx* = 1, and the error class is instruction urgent error.
	Others	The name of the bit set to 1 in ASI_UGESR indicates the error type.

Column	Term	Meaning
Correction	RED trap	The whole register is updated and corrected when a RED_state trap occurs.
	W	The whole register is updated and corrected by use of an STXA instruction to write the register.
	W1AC	The whole register is updated and corrected by use of an STXA instruction to write 1 to the specified bit in the register.
	WotherI	The register is corrected by a full update of all of the following ASI registers: • ASI_IMMU_TAG_ACCESS • plus, when ASI_UGESR.IAUG_TSBCTXT = 1 is indicated in a single-ADE trap: ASI_IMMU_TSB_BASE, ASI_IMMU_TSB_PEXT, ASI_PRIMARY_CONTEXT, ASI_SECONDARY_CONTEXT IMMU_TSB_8KB_PTR and IMMU_TSB_64KB_PTR are corrected only when a fast_instruction_access_MMU_miss trap occurs.
	WotherD	The register is corrected by a full update of all of the following ASI registers: • ASI_DMMU_TAG_ACCESS • plus, when ASI_UGESR.IAUG_TSBCTXT = 1 is indicated in a single-ADE trap: ASI_DMMU_TSB_BASE, ASI_DMMU_TSB_PEXT, ASI_DMMU_TSB_SEXT, ASI_PRIMARY_CONTEXT, ASI_SECONDARY_CONTEXT DMMU_TSB_8KB_PTR and DMMU_TSB_64KB_PTR are corrected only when a fast_data_access_MMU_miss trap occurs.
	DemapAll	The error is corrected by the *demap all* operation for the TLB with the error. Note that the *demap all* operation does not remove the locked TLB entry with uncorrectable error.
	Interrupt receive	The register is corrected when the Jupiter Bus interrupt packet is received.

TABLE P-17 shows the handling of ASI register errors.

TABLE P-17 Handling of ASI Register Errors

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<16:8>	RW	Parity	Always	error_state	RED trap
		MEMORY_CONTROL<7:6>	RW	Parity	Always	error_state	RED trap
48_{16}	00_{16}	INTR_DISPATCH_STATUS	R	Gecc	LDXA	I(A)UG_CRE (UE) ignored (CE)	None
49_{16}	00_{16}	INTR_RECEIVE	RW	Gecc	LDXA	I(A)UG_CRE (UE) ignored (CE)	None
$4A_{16}$	—	JB_CONFIG_REGISTER	R	None	—	—	—
$4C_{16}$	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	—	—	—
$4D_{16}$	00_{16}	AFAR_D1	R,WAC	Parity	LDXA	I(A)UG_CRE	WAC
$4D_{16}$	08_{16}	AFAR_U2	R,WAC	Parity	LDXA	I(A)UG_CRE	WAC

TABLE P-17 Handling of ASI Register Errors

ASI	VA	Register Name	RW	Error Protect	Error Detect Condition	Error Type	Correction
50_{16}	00_{16}	IMMU_TAG_TARGET	R	Parity	LDXA #I	IUG_TSBP	WotherI
50_{16}	18_{16}	IMMU_SFSR	RW	None	—	—	—
50_{16}	28_{16}	IMMU_TSB_BASE	RW	Parity	LDXA #I	I(A)UG_TSBCTXT	W
50_{16}	30_{16}	IMMU_TAG_ACCESS	RW	Parity	LDXA #I	IUG_TSBP	W (WotherI)
50_{16}	48_{16}	IMMU_TSB_PEXT	RW	Parity	= ITSB_BASE	IAUG_TSBCTXT	W
50_{16}	58_{16}	IMMU_TSB_NEXT	R	Parity	= ITSB_BASE	IAUG_TSBCTXT	W
50_{16}	60_{16}	IMMU_TAG_ACCESS_EXT	RW	Parity	LDXA #I	IUG_TSBP	W
50_{16}	78_{16}	IMMU_SFPAR	RW	Parity	LDXA #I	I(A)UG_CRE	W
51_{16}	—	IMMU_TSB_8KB_PTR	R	PP	LDXA	IUG_TSBP	WotherI
52_{16}	—	IMMU_TSB_64KB_PTR	R	PP	LDXA	IUG_TSBP	WotherI
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 #D	IUG_TSBP	WotherD
58_{16}	08_{16}	PRIMARY_CONTEXT	RW	Parity	LDXA #I, LDXA #D	I(A)UG_TSBCTXT	W
					Use for TLB	I(A)UG_TSBCTXT	W
					AUG always	(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 #D	I(A)UG_TSBCTXT	W
58_{16}	30_{16}	DMMU_TAG_ACCESS	RW	Parity	LDXA #D	IUG_TSBP	W (WotherD)
58_{16}	38_{16}	DMMU_VA_WATCHPOINT	RW	Parity	Enabled	(I)AUG_CRE	W
					LDXA	I(A)UG_CRE	W
58_{16}	40_{16}	DMMU_PA_WATCHPOINT	RW	Parity	Enabled	(I)AUG_CRE	W
					LDXA	I(A)UG_CRE	W
58_{16}	48_{16}	DMMU_TSB_PEXT	RW	Parity	= DTSB_BASE	I(A)UG_TSBCTXT	W
58_{16}	50_{16}	DMMU_TSB_SEXT	RW	Parity	= DTSB_BASE	I(A)UG_TSBCTXT	W
58_{16}	58_{16}	DMMU_TSB_NEXT	R	Parity	= DTSB_BASE	I(A)UG_TSBCTXT	W
58_{16}	60_{16}	DMMU_TAG_ACCCESS_EXT	RW	Parity	LDXA #D	IUG_TSBP	W
58_{16}	78_{16}	DMMU_SFPAR	RW	Parity	LDXA #D	I(A)UG_CRE	W
59_{16}	—	DMMU_TSB_8KB_PTR	R	PP	LDXA	IUG_TSBP	WotherD
$5A_{16}$	—	DMMU_TSB_64KB_PTR	R	PP	LDXA	IUG_TSBP	WotherD

TABLE P-17 Handling of ASI Register Errors

ASI	VA	Register Name	RW	Error Protect	Error Detect Condition	Error Type	Correction
$5B_{16}$	—	DMMU_TSB_DIRECT_PTR	R	PP	LDXA	IUG_TSBP	WotherD
$5C_{16}$	—	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
$6E_{16}$	00_{16}	EIDR	RW	Parity	Always[1]	IAUG_CRE	W
74_{16}	addr	CACHE_INV	W	None	—	—	—
77_{16}	40_{16}–	INTR_DATA0:7_W	W	Gecc	None	—	W
	88_{16}	INTR_DISPATCH_W	W	Gecc	store	(I)AUG_CRE	W
$7F_{16}$	40_{16}–	INTR_DATA0:7_R	R	ECC	LDXA	IAUG_CRE	Interrupt Receive
	88_{16}				intr_receive	BUSY = 0	

1.Notified as `error_state` transition error in suspended state.

P.9 Cache Error Handling

In this section, handling of cache errors of the following types is specified:

- Cache tag errors
- Cache data errors in I1, D1, and U2 caches

This section concludes with the specification of automatic way reduction in the I1, D1, and U2 caches.

P.9.1 Handling of a Cache Tag Error

Error in D1 Cache Tag and I1 Cache Tag

Both the D1 cache (Data level 1) and the I1 cache (Instruction level 1) maintain a copy of their cache tags in the U2 (unified level 2) cache. The D1 cache tags, the D1 cache tags copy, the I1 cache tags, and the I1 cache tags copy are each protected by parity.

When a parity error is detected in a D1 cache tag entry or in a D1 cache tag copy entry, hardware automatically corrects the error by copying the correct tag entry from the other copy of the tag entry. If the error can be corrected in this way, program execution is unaffected.

Similarly, when a parity error is detected in an I1 cache tag entry or in a I1 cache tag copy entry, hardware automatically corrects the error by copying the correct tag entry from the other copy of the tag entry. If the error can be corrected in this way, program execution is unaffected.

When the error in the level-1 cache tag or tag copy is not corrected by the tag copying operation, the tag copying is repeated. If the error is permanent, a watchdog timeout or a FATAL error is then detected.

Error in U2 (Unified Level 2) Cache Tag

The U2 cache tag is protected by double-bit error detection and single-bit error correction ECC code.

When a correctable error is detected in a U2 cache tag, hardware automatically corrects the error by rewriting the corrected data into the U2 cache tag entry. The error is not reported to software.

When an uncorrectable error is detected in a U2 cache tag, one of following actions is taken, depending on the setting of `OPSR` (internal mode register set by the JTAG command):

1. A fatal error is detected and the CPU enters the CPU fatal error state.
2. The U2 cache tag uncorrectable error is treated as follows; however, in some cases, the fatal error is still detected.
   a. When `ASI_ERROR_CONTROL.WEAK_ED` = 0:

      The AUG_SDC is recognized during U2 cache tag error detection.

      If `ASI_ERROR_CONTROL.UGE_HANDLER` = 0, the AUG_SDC immediately generates an *async_data_error* trap with `ASI_UGESR.AUG_SDC` = 1.

      Otherwise:, if `ASI_ERROR_CONTROL.UGE_HANDLER` = 1, the AUG_SDC remains pending in hardware. At the point when `ASI_ERROR_CONTROL.UGE_HANDLER` is set to 0, an *async_data_error* exception is generated, with `ASI_UGESR.AUG_SDC` = 1.

   b. When `ASI_ERROR_CONTROL.WEAK_ED` = 1:

      Hardware ignores the U2 cache tag error if possible. However, the AUG_SDC or fatal error may still be detected.

P.9.2 Handling of an I1 Cache Data Error

I1 cache data is protected by parity attached to every doubleword.

When a parity error is detected in I1 cache data during an instruction fetch, hardware executes the following sequence:

1. Reread the I1 cache line containing the parity error from the U2 cache.

   The read data from U2 cache must contain only the doubleword without error or the doubleword with the marked UE, because error marking is applied to U2 cache outgoing data.

2. For each doubleword read from U2 cache:

   a. When the doubleword does not have a UE, save the correct data in the I1 cache doubleword without parity error and supply the data for instruction fetch if required.

      There is no direct report to software for an I1 cache error corrected by refilling data.

   b. When the doubleword has a marked UE, set the parity bit in the I1 cache doubleword to indicate a parity error and supply the parity error data for the instruction fetch if required.

3. Treat a fetched instruction with an error as follows:

   When the instruction with a parity error is fetched but not executed in any way visible to software, the fetched instruction with the error is discarded.

   Otherwise, fetch and execute the instruction with the indicated parity error. When the execution of the instruction is complete, an *instruction_access_error* exception will be generated (precise trap), and the marked UE detection and its `ERROR_MARK_ID` will be indicated in `ASI_ISFSR`.

P.9.3 Handling of a D1 Cache Data Error

D1 cache data is protected by 2-bit error detection and 1-bit error correction ECC, attached to every doubleword.

Correctable Error in D1 Cache Data

When a correctable error is detected in D1 cache data, the data is corrected automatically by hardware. There is no direct report to software for a D1 cache correctable error.

Marked Uncorrectable Error in D1 Cache Data

When a marked uncorrectable error (UE) in D1 cache data is detected during the D1 cache line writeback to the U2 cache, the D1 cache data and its ECC are written to the target U2 cache data and its ECC without modification. That is, a marked UE in D1 cache is propagated into the U2 cache. Such an error is not reported to software.

When a marked UE in D1 cache data is detected during access by a load or store (excluding doubleword store) instruction, the data access error is detected. The *data_access_error* exception is generated precisely and the marked UE detection and its ERROR_MARK_ID are indicated in ASI_DSFSR.

Raw Uncorrectable Error in D1 Cache Data During D1 Cache Line Writeback

When a raw (unmarked) UE is detected in D1 cache data during the D1 cache line writeback to the U2 cache, error marking is applied to the doubleword containing the raw UE with ERROR_MARK_ID = ASI_EIDR. Only the correct doubleword or the doubleword with marked UE is written into the target U2 cache line.

The restrainable error ASI_AFSR.UE_RAW_D1$INSD is detected.

Raw Uncorrectable Error in D1 Cache Data on Access by Load or Store Instruction

When a raw (unmarked) UE is detected in D1 cache data during access by a load or store instruction, hardware executes the following sequence:

1. Hardware writes back the D1 cache line and refills it from U2 cache. The D1 cache line containing the raw UE, whether it is clean or dirty, is always written back to the U2 cache. During this D1 cache line writeback to U2 cache, error marking is applied for the doubleword containing the raw UE with ERROR_MARK_ID = ASI_EIDR. The D1 cache line is refilled from the U2 cache and the restrainable error ASI_AFSR.UE_RAW_D1$INSD is detected.

2. Normally, hardware changes the raw UE in the D1 cache data to a marked UE. However, yet another error may introduce a raw UE into the same doubleword again. When a raw UE is detected again, step 1 is repeated until the D1 cache way reduction is applied.

3. At this point, hardware changes the raw UE in the D1 cache data to a marked UE. The load or store instruction accesses the doubleword with the marked UE. The marked UE is detected during execution of the load or store instruction, as described in *Raw Uncorrectable Error in D1 Cache Data During D1 Cache Line Writeback*, above.

P.9.4 Handling of a U2 Cache Data Error

U2 cache data is protected by 2-bit error detection and 1-bit error correction ECC, attached to every doubleword.

Correctable Error in U2 Cache Data

When a correctable error is detected in the incoming U2 cache fill data from Jupiter Bus, the data is corrected by hardware, stored into U2 cache, and the restrainable error ASI_AFSR.CE_INCOMED is detected.

When a correctable error is detected in the data from U2 cache for I1 cache fill, D1 cache fill, copyback to Jupiter Bus, or writeback to Jupiter Bus, both the transfer data and source data in U2 cache are corrected by hardware. The error is not reported to software.

Marked Uncorrectable Error in U2 Cache Data

For U2 cache data, a doubleword with marked UE is treated the same as a correct doubleword. No error is reported when the marked UE in U2 cache data is detected.

When a marked uncorrectable error (UE) is detected in incoming U2 cache fill data from Jupiter Bus, the doubleword with the marked UE is stored without modification in the target U2 cache line.

When a marked uncorrectable error is detected in incoming data from the D1 cache to writeback D1 cache line, the doubleword with the marked UE is stored without modification in target U2 cache line. Note that there is no raw UE in D1 writeback data because error marking is applied for D1 writeback data, as described in *Handling of a D1 Cache Data Error* on page 190.

When a marked UE is detected in the data read from the U2 cache for an I1 cache fill, D1 cache fill, copyback to Jupiter Bus, or writeback to Jupiter Bus, the doubleword with the marked UE is transferred without modification.

Raw Uncorrectable Error in U2 Cache Data

When a raw (unmarked) UE is detected in incoming U2 cache fill data, error marking is applied for the doubleword with the raw UE, using ERROR_MARK_ID = 0. The doubleword and its ECC are changed to the marked UE data, the changed data is stored in the target U2 cache line, and the restrainable error ASI_AFSR.UE_RAW_L2$FILL is detected.

When a raw UE is detected in data read from U2 cache, such as for I1 cache fill, D1 cache fill, copyback to Jupiter Bus, or writeback to Jupiter Bus, then error marking is applied for the doubleword with the raw UE, using ERROR_MARK_ID = ASI_EIDR. Both the doubleword and its ECC in the read data and those in the source U2 cache line are changed to marked UE data. The restrainable error ASI_AFSR.UE_RAW_L2$INSD is detected.

Implementation Note – SPARC64 VI detects ASI_AFSR.UE_FAW_L2$INSD only on writeback.

P.9.5 Automatic Way Reduction of I1 Cache, D1 Cache, and U2 Cache

When frequent errors occur in the I1, D1, or U2 cache, hardware automatically detects that condition and reduces the way, maintaining cache consistency.

Way Reduction Condition

Hardware counts the sum of the following error occurrences for each way of each cache:

- For each way of the I1 cache:
  - Parity error in I1 cache tag or I1 cache tag copy
  - I1 cache data parity error
- For each way of the D1 cache:
  - Parity error in D1 cache tag or D1 cache tag copy
  - Correctable error in D1 cache data
  - Raw UE in D1 cache data
- For each way of U2 cache:
  - Correctable error and uncorrectable error in U2 cache tag
  - Correctable error in U2 cache data
  - Raw UE in U2 cache data

If an error count per unit of time for one way of a cache exceeds a predefined threshold, hardware recognizes a cache way reduction condition and takes the actions described below.

I1 Cache Way Reduction

When way reduction condition is recognized for the I1 cache way W (W = 0 or 1), the following way reduction procedure is executed:

1. When only one way in I1 cache is active because of previous way reduction:
   - All entries in I1 cache way W are invalidated.
   - The restrainable error ASI_AFSR.DG_L1$U2$STLB is reported to software.
2. Otherwise:
   - All entries in I1 cache way W are invalidated and the way W will never be refilled.
   - The restrainable error ASI_AFSR.DG_L1$U2$STLB is reported to software.

D1 Cache Way Reduction

When a way reduction condition is recognized for the D1 cache way W (W = 0 or 1), the following way reduction procedure is executed:

1. When only one way in D1 cache is active because of previous way reduction:
   - All entries in D1 cache way W are invalidated. On invalidation of each dirty D1 cache entry, the D1 cache line is written back to its corresponding U2 cache line.
   - The restrainable error ASI_AFSR.DG_L1$U2$STLB is reported to software.
2. Otherwise:
   - All entries in D1 cache way W are invalidated and the way W will never be refilled. On invalidation of each dirty D1 cache entry, the D1 cache line is written back to its corresponding U2 cache line.
   - The restrainable error ASI_AFSR.DG_L1$U2$STLB is reported to software.

U2 Cache Way Reduction

When a way reduction condition is recognized for a U2 cache way, the U2 cache way reduction procedure is executed as follows:

1. When `ASI_L2CTL.WEAK_SPCA` = 0,

   the U2 cache way reduction procedure (below) is started immediately.

2. Otherwise, when `ASI_L2CTL.WEAK_SPCA` = 1 is set,

   the U2 cache way reduction procedure (below) becomes pending until `ASI_L2CTL.WEAK_SPCA` is changed to 0. When `ASI_L2CTL.WEAK_SPCA` is changed to 0, the U2 cache way reduction procedure will be started.

The U2 cache way W (W=0, 1, 2, or 3) reduction procedure:

1. When only one way in U2 cache is active because of previous way reductions:
   - All entries in U2 cache way W are at once invalidated (that is, all active U2 cache entries are invalidated) and U2 cache way W remains as the only available U2 cache way. The U2 cache data is invalidated to retain system consistency.
   - The restrainable error ASI_AFSR.DG_L1$U2$STLB is reported to software, even though the available U2 cache configuration is not changed as a result of the error.
2. Otherwise:
   - All entries in available U2 cache ways, including way W, are invalidated to retain system consistency.
   - Way W becomes unavailable and is never refilled.
   - The restrainable error ASI_AFSR.DG_L1$U2$STLB is reported to software.

P.10 TLB Error Handling

This section describes how TLB entry errors and sTLB way reduction are handled.

P.10.1 Handling of TLB Entry Errors

Error protection and error detection in TLB entries are described in TABLE P-18.

TABLE P-18 Error Protection and Detection of TLB Entries

TLB type	Field	Error Protection	Detectable Error
sITLB and sDTLB	tag	Parity	Parity error (Uncorrectable)
sITLB and sDTLB	data	Parity	Parity error (Uncorrectable)
fITLB and fDTLB	lock bit	Triplicated	None; the value is determined by majority
fITLB and fDTLB	tag except lock bit	Parity	Parity error (Uncorrectable)
fITLB and fDTLB	data	Parity	Parity error

Errors can occur during the following events:

- Access by LDXA instruction
- Virtual address translation (sTLB)
- Virtual address translation (fTLB)

Error in TLB Entry Detected on LDXA Instruction Access

If a parity error is detected in a DTLB entry when an LDXA instruction attempts to read ASI_DTLB_DATA_ACCESS or ASI_DTLB_TAG_ACCESS, hardware automatically demaps the entry and an instruction urgent error is indicated in ASI_UGESR.IUG_DTLB.

When a parity error is detected in an ITLB entry when an LDXA instruction attempts to read ASI_ITLB_DATA_ACCESS or ASI_ITLB_TAG_ACCESS, hardware automatically demaps the entry and an instruction urgent error is indicated in ASI_UGESR.IUG_ITLB.

Error in sTLB Entry Detected During Virtual Address Translation

When a parity error is detected in the sTLB entry during a virtual address translation, hardware automatically demaps the entry and does not report the error to software.

Error in fTLB Entry Detected During Virtual Address Translation

When an fTLB tag has a parity error, the fTLB entry never matches any virtual address. An fTLB tag error in a locked entry causes a TLB miss for the virtual address already registered as the locked TLB entry.

A parity error in fTLB entry data is detected only when the tag of the fTLB entry matches a virtual address.

When a parity error in the fITLB is detected at the time of an instruction fetch, a precise *instruction_access_error* exception is generated. The parity error in the `fITLB` entry and the `fITLB` entry index is indicated in `ASI_ISFSR`.

When a parity error in `fDTLB` is detected for the memory access of a load or store instruction, a precise *data_access_error* exception is generated. The parity error in the `fDTLB` entry and the `fDTLB` entry index is indicated in `ASI_DSFSR`.

P.10.2 Automatic Way Reduction of sTLB

When frequent errors occur in `sITLB` and `sDTLB`, hardware automatically detects that condition and reduces the way, with no adverse effects on software.

Way Reduction Condition

Hardware counts `TLB` entry parity error occurrences for each `sITLB` way and `sDTLB` way. If the error count per unit of time exceeds a predefined threshold, hardware recognizes an `sTLB` way reduction condition.

sTLB Way Reduction

When a way reduction condition is recognized for the `sTLB` way W (W = 0 or 1), hardware executes the following way reduction procedures:

1. When only one way in sTLB is active because of previous way reductions:
   - The previously reduced way is reactivated.
2. Regardless of how many ways were previously active, way reduction occurs:
   - Hardware reduces the way and invalidates all entries in sTLB way W. Way W will never be refilled.
   - The restrainable error ASI_AFSR.DG_L1$U2$STLB is reported to software.

F.APPENDIX **Q**

Performance Instrumentation

This appendix describes and specifies performance monitors that have been implemented in the SPARC64 VI processor. The appendix contains these sections:

Q.1 Performance Monitor Overview

For the definitions of performance counter registers, please refer to *Performance Control Register (PCR) (ASR 16)* on page 18 and *Performance Instrumentation Counter (PIC) Register (ASR 17)* on page 20.

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 that value. The following pseudocode procedure clears all PICs (assuming privileged access):

```
/* clear pics without altering sl/su values */
pic_init = 0x0;
pcr = rd_pcr();
pcr.ulro = 0x1;       /* don't change su/sl on write */
pcr.ovf = 0x0;        /* clear overflow bits also */
pcr.ut = 0x0;
pcr.st = 0x0;         /* disable counts for good measure */
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 `PCR.SC` and `PCR.SU`/`PCR.SL` fields. The following pseudocode selects events and enables counters (assuming privileged access):

```
pcr.ut = 0x0;         /* initially disable user counts */
pcr.st = 0x0;         /* initially disable system counts */
pcr.ulro = 0x0;       /* make sure read-only disabled */
pcr.ovro = 0x1;       /* do not modify overflow bits */
/* select the 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;       /* for not changing the last su/sl */
/* resetting of overflow bits can be done here */
wr_pcr(pcr);
```

Counter Stop and Read

The following pseudocode disables and reads counters (assuming privileged access):

```
pcr.ut = 0x0;         /* disable counts */
pcr.st = 0x0;         /* disable counts */
pcr.ulro = 0x1;       /* enable sl/su read-only */
pcr.ovro = 0x1;       /* do not modify overflow bits */
for(i=0; i<=pcr.nc; i++) {
   /* assume rest of pcr data has been preserved */
```

```
        pcr.sc = i;
        wr_pcr(pcr);
        pic = rd_pic();
        picl[i] = pic.picl;
        picu[i] = pic.picu;
    }
```

Q.2 Performance Event Description

The performance events can be divided into the following groups:

1. Instruction and Trap statistics
2. MMU and L1 cache event counters
3. L2 cache event counters
4. Jupiter Bus transaction event counters
5. Multi-thread specific event counters

There are two types of performance events, basic and extended in SPARC64 VI.

Basic performance events are documented in JPS (Joint Programmer's Specification) and verification is completed.

Extended events are not documented in JPS, and they are intended to provide information for debugging the hardware. User of these extended events should be aware of the following rules.

a. **Verification of the extended events are not necessarily completed. In other words, the counters might not work as expected.**

b. **Definition of the extended events may change without notice. Compatibility is not guaranteed between future SPARC64 generations.**

All event counters implemented in SPARC64 VI are listed in TABLE Q-1. The events in shadow are extended. The details of the performance counters is described in the following sessions. They are speculatively updated, unless specially noted.

TABLE Q-1 Events and Encoding of Performance Monitor

Encoding	Counter							
	picu0	picl0	picu1	picl1	picu2	picl2	picu3	picl3
000000	*cycle_counts*							
000001	*instruction_counts*							

TABLE Q-1 Events and Encoding of Performance Monitor *(Continued)*

Encoding	Counter							
	picu0	picl0	picu1	picl1	picu2	picl2	picu3	picl3
000010	*instruction_flow_counts*	*Reserved*			*instruction_flow_counts*	*Reserved*		
000011	*iwr_empty*	*Reserved*			*iwr_empty*	*Reserved*		
000100	*Reserved*							
000101	*op_stv_wait*							
000110	*Reserved*							
000111	*Reserved*							
001000	*load_store_instructions*							
001001	*branch_instructions*							
001010	*floating_instructions*							
001011	*impdep2_instructions*							
001100	*prefetch_instructions*							
001101	*Reserved*							
001110	*Reserved*							
001111	*Reserved*							
010000	*Reserved*							
010001	*Reserved*							
010010	*rs1*	*flush_rs*	*Reserved*					
010011	*1iid_use*	*2iid_use*	*3iid_use*	*4iid_use*	*Reserved*	*sync_intlk*	*regwin_intlk*	*Reserved*
010100	*Reserved*							
010101	*Reserved*	*toq_rsbr_phantom*	*Reserved*	*flush_rs*	*Reserved*		*rs1*	*Reserved*
010110	*trap_all*	*trap_int_vector*	*trap_int_level*	*trap_spill*	*trap_fill*	*trap_trap_inst*	*trap_IMMU_miss*	*trap_DMMU_miss*
010111	*Reserved*							
011000	*thread_switch_all*	*ts_by_sxmiss*	*ts_by_data_arrive*	*ts_by_timer*	*ts_by_intr*	*ts_by_if*	*Reserved*	*ts_by_suspend*
011001	*Reserved*							*ts_by_other*
011010	*active_cycle_count*							
011011	*Reserved*		*op_stv_wait_nc_pend*	*0iid_use*	*flush_rs*	*Reserved*		*decall_intlk*
011100	*Reserved*							
011101	*act_thread_suspend*	*op_stv_wait_sxmiss*	*op_stv_wait_sxmiss_ex*	*op_stv_wait_nc_pend*	*cse_window_empty_sp_full*	*Reserved*		
011110	*cse_window_empty*	*eu_comp_wait*	*branch_comp_wait*	*0endop*	*op_stv_wait_ex*	*fl_comp_wait*	*1endop*	*2endop*
011111	*inh_cmit_gpr_2write*	*Reserved*			*3endop*	*Reserved*	*op_stv_wait_sxmiss_ex*	*op_stv_wait_sxmiss*

TABLE Q-1 Events and Encoding of Performance Monitor *(Continued)*

Encoding	Counter							
	picu0	picl0	picu1	picl1	picu2	picl2	picu3	picl3
100000	*Reserved*		*write_if_uTLB*	*write_op_uTLB*	*if_r_iu_req_mi_go*	*op_r_iu_req_mi_go*	*if_wait_all*	*op_wait_all*
100001	*Reserved*							
100010	*Reserved*							
100011	*Reserved*							
100100	*swpf_success_all*	*swpf_fail_all*	*Reserved*		*swpf_lbs_hit*	*Reserved*		
100101	*Reserved*							
100110	*Reserved*							
100111	*Reserved*							
110000	*sx_miss_wait_dm*	*sx_miss_wait_pf*	*sx_miss_count_dm*	*sx_miss_count_pf*	*sx_read_count_dm*	*sx_read_count_pf*	*dvp_count_dm*	*dvp_count_pf*
110001	*jbus_bi_count*	*jbus_cpi_count*	*jbus_cpb_count*	*jbus_cpd_count*	*jbus_reqbus_busy*	*jbus_odrbus_busy*	*Reserved*	
110010	*Reserved*		*snres_256*	*snres_64*	*Reserved*			
110011	*Reserved*				*sx_btc_count*	*sx_miss_count_dm_if*	*sx_miss_count_dm_opsh*	*sx_miss_count_dm_opex*
110100	*lost_softpf_pfp_full*	*Reserved*	*lost_softpf_by_abort*	*Reserved*				
110101	*Reserved*							
110110	*jbus_reqbus0_busy*	*jbus_reqbus1_busy*	*jbus_reqbus2_busy*	*jbus_reqbus3_busy*	*jbus_odrbus0_busy*	*jbus_odrbus1_busy*	*jbus_odrbus2_busy*	*jbus_odrbus3_busy*
111111	*Disabled (No PIC is counted up)*							

Q.2.1 Instruction and trap Statistics

Basic events

1 *cycle_counts*

Counts the cycles when the performance monitor is enabled. This counter is similar to the `%tick` register but can separate user cycles from system cycles, based on `PCR.UT` and `PCR.ST` selection.

2 *instruction_counts* (non-speculative)

Counts the number of committed instructions. For user or system mode counts, this counter is exact. Combined with the *cycle_counts*, it provides instructions per cycle.

IPC = *instruction_counts* / *cycle_counts*

If *Instruction_counts* and *cycle_counts* are both collected for user or system mode, IPC in user or system mode can be derived.

3 *load_store_instructions* (non-speculative)

Counts the committed load/store instructions. Also counts atomic load-store instructions.

4 *branch_instructions* (non-speculative)

Counts the committed branch instructions. Also counts CALL, JMPL, and RETURN instructions.

5 *floating_instructions* (non-speculative)

Counts the committed floating-point operations (FPop1 and FPop2). Does not count Floating-Point Multiply-and-Add instructions.

6 *impdep2_instructions* (non-speculative)

Counts the committed Floating Multiply-and-Add instructions.

7 *prefetch_instructions* (non-speculative)

Counts the committed prefetch instructions.

8 *trap_all* (non-speculative)

Counts all trap events. The value is equivalent to the sum of type-specific traps counters.

9 *trap_int_vector* (non-speculative)

Counts the occurrences of *interrupt_vector_trap*.

10 *trap_int_level* (non-speculative)

Counts the occurrences of *interrupt_level_n*.

11 *trap_spill* (non-speculative)

Counts the occurrences of *spill_n_normal*, *spill_n_other*.

12 *trap_fill* (non-speculative)

Count the occurrences of *fill_n_normal*, *fill_n_other*.

13 *trap_trap_inst* (non-speculative)

Counts the occurrences of `Tcc` instructions.

14 *trap_IMMU_miss* (non-speculative)

Counts the occurrences of *fast_instruction_access_MMU_miss*.

15 *trap_DMMU_miss* (non-speculative)

Counts the occurrences of *fast_data_instruction_access_MMU_miss*.

Extended events

16 *instruction_flow_counts* (non-speculative)

Number of committed instruction flow during measuring period. In SPARC64 VI, for specific instructions, an instruction may be internally represented as a set of instructions, and executed as if it were multiple instructions. *instruction_flow_count* measures number of internal instructions during measuring period.

17 *iwr_empty*

Number of cycles that IWR (Issue Word Register) is empty. IWR is a four-entry register that holds instructions while decoder is processing. IWR empty may be caused on instruction cache miss.

18 *rs1* (non-speculative)

Number of commit re-ifetch. Example commit re-ifetch are as follows:

- trap, interrupt

- update of privilege registers
- assurance of memory order
- hardware retry

19 *flush_rs* (non-speculative)

Number of pipeline flush due to mis-prediction. Since SPARC64 VI employs speculative execution, it may execute instructions that should have not been executed due to mis-prediction. when the predict path is found to be wrong, then all instructions in the pipeline is aborted, and execution of the correct path is started. Pipeline flush is occurred at this time.

mis-prediction rate = *flush_rs* / *branch_instructions*

20 *0iid_use*

No instruction is issued in a cycle. SPARC64 VI issues up to four instruction. 0iid_use is counted up when no instruction is issued. In SPARC64 VI, for specific instructions, an instruction may be internally represented as a set of instructions. Measurement is made against this internal instructions.

21 *1iid_use*

One instruction is issued in a cycle.

22 *2iid_use*

Two instructions are issued in a cycle.

23 *3iid_use*

Three instructions are issued in a cycle.

24 *4iid_use*

Four instructions are issued in a cycle.

25 *sync_intlk*

Number of cycles that prevent issuing instructions due to pre-sync and post-sync.

26 *regwin_intlk*

Number of cycles that prevent issuing instructions due to CWR switch. CWR holds the value of window register (%r8 - %r31), and its neighbors. Replacing the contents of CWR is caused by save/restore or trap. Replacing is usually done concurrently in background, but it sometime causes an interlock such as successive save/restore.

27 decall_intlk

Number of cycles that prevent issuing instructions due to any static inter-lock conditions at the decode stage. *decall_intlk* includes *sync_intlk* and *regwin_intlk*, but it does not count stall cycles due to dynamic conditions such as reservation station full.

28 toq_rsbr_phantom

Count when an instruction predicted as taken branch is actually not a branch instruction. This may happen in SPARC64 VI since branch prediction is done prior to decode of the instruction.

29 *op_stv_wait* (non-speculative)

Number of cycles that instruction commit is not done due to data wait. SPARC64 VI has resource named CSE(Commit Stack Entry), which holds information of in-flight instructions. CSE is a fifo, and information is registered in-order. op_stv_wait is measured if the top entry of CSE(TOQ: Top of Queue) is a memory access instruction and data is not ready.

op_stv_wait does not count memory access latency for a store instruction (however, memory access latency for an atomic instruction is counted). This is due to a feature of which SPARC64 VI employs for performance improvement. SPARC64 VI commits a store instruction before data is written on L2 cache.

Caution is needed that not all of data cache miss latency is measured by op_stv_wait. When a data cache miss is occurred, and after of all instructions prior to that instruction have committed, a latency of that instruction is measured.

30 *op_stv_wait_nc_pend* (non-speculative)

op_stv_wait due to non-cache access

31 *op_stv_wait_ex* (non-speculative)

No instruction is committed waiting for an integer load instruction in TOQ to complete.

32 *op_stv_wait_sxmiss* (non-speculative)

op_stv_wait due to L2$ miss

33 *op_stv_wait_sxmiss_ex* (non-speculative)

op_stv_wait_ex due to L2$ miss

34 *cse_window_empty_sp_full* (non-speculative)

No instruction is committed because CSE is empty while the Store Port is full.

35 *cse_window_empty* (non-speculative)

No instruction is committed because CSE is empty.

36 *branch_comp_wait* (non-speculative)

No instruction is committed waiting for a branch instruction in TOQ to complete. Its priority is lower than *eu_comp_wait*.

37 *eu_comp_wait* (non-speculative)

No instruction is committed waiting for an integer and floating-point instruction in TOQ to complete. Its priority is higher than *branch_comp_wait*.

38 *fl_comp_wait* (non-speculative)

No instruction is committed waiting for a floating-point instruction in TOQ to complete.

39 *0endop* (non-speculative)

No instruction is committed.

40 *1endop* (non-speculative)

One instruction is committed.

41 *2endop* (non-speculative)

Two instructions are committed.

42 *3endop* (non-speculative)

Number of cycles three instructions are committed.

43 *inh_cmit_gpr_2write* (non-speculative)

Less than four instructions are committed due to lack of GPR write ports.

Q.2.2 MMU and L1 cache Event Counters

Basic events

1 *write_if_uTLB*

Counts the occurrences of instruction uTLB misses.

2 *write_op_uTLB*

Counts the occurrences of data uTLB misses.

Note – Occurrences of main TLB misses are counted by *trap_IMMU_miss*/*trap_DMMU_miss*.

---

3 *if_r_iu_req_mi_go*

Counts the occurrences of I1 cache misses.

4 *op_r_iu_req_mi_go*

Counts the occurrences of D1 cache misses.

5 *if_wait_all*

Counts the total latency of I1 cache misses. Sum of if_wait=xxx is shown. Caution must be taken it is not represent L1 instruction cache miss latency. Events measured in if_wait=xxx are mutually exclusive, thus, at most one of if_wait=xxx is counted up in a cycle. SPARC64 VI can process multiple cache miss in parallel since it employs non-blocking cache, but only one (TOQ) of those access is measured.

6 *op_wait_all*

Counts the total latency of D1 cache misses. Sum of op_wait=xxx is shown. Caution must be taken it is not represent L1 data cache miss latency. Events measured in if_wait=xxx are mutually exclusive, thus, at most one of op_wait=xxx is counted up in a cycle. SPARC64 VI can process multiple cache miss in parallel since it employs non-blocking cache, but only one (TOQ) of those access is measured. The condition of which access become a TOQ is so complicated to describe in this document, but prefetch instruction never be a TOQ.

Extended events

7 *swpf_success_all*

Number of prefetch instructions not lost in SU and sent to SX successfully.

8 *swpf_fail_all*

Number of prefetch instructions lost in SU.

9 *swpf_lbs_hit*

Number of prefetch instructions resulting in L1-cache hit.

The number of prefetch instructions sent to SU
= *swpf_success_all + swpf_fail_all + swpf_lbs_hit*

Q.2.3 L2 cache Event Counters

Most of L2 cache access related counters are categorized in dm (demand) and pf (prefetch), but in these counters, it does not always correspond to load/store/atomic, and prefetch instructions. This is because:

a. If load/store/atomic instruction can not be processed due to starvation of L1 cache resources, these requests are handled as if it were prefetches to L2 cache, which does not use L1 cache resources. These requests are treated as 'prefetch' in the L2 cache access related counters.

b. SPARC64 VI employs hardware to prefetch data for a sequential access. A hardware prefetch request is treated as 'prefetch' in the L2 cache access related counters.

Basic events

1 *sx_miss_wait_dm*

Counts the number of cycles from the occurrence of an L2 cache miss to data returned, caused by demand access.

2 *sx_miss_wait_pf*

Counts the number of cycles from the occurrence of an L2 cache miss to data returned, caused by both software prefetch and hardware prefetch access.

3 *sx_miss_count_dm*

Counts the occurrences of L2 cache miss by demand access. A Request to the same line of outstanding access (not yet completed) is considered to be "hit" and not counted in this counter.

4 *sx_miss_count_pf*

Counts the occurrences of L2 cache miss by both software prefetch and hardware prefetch access.

5 *sx_read_count_dm*

Counts L2 cache references by demand read access. A cache access may be aborted by many reasons, such as contention of resources. sx_read_count_dm does not measure a retry of cache accesses. It double-counts multi-flow operations. Therefore the following equation is approximately true (but not in precise):

sx_read_count_dm + sx_read_count_pf =
number of cache misses by L1I and L1D + number of non-lost hardware prefetch + number of physical address access which bypass the L1 cache (ASI:0x14, 0x1c, 0x34, 0x3c)

A request from other CPU (copyback/invalidate request) is not measured by this counter.

6 *sx_read_count_pf*

Counts L2 cache references by both software prefetch and hardware prefetch access.

7 *dvp_count_dm*

Counts the occurrences of L2 cache miss by demand, with writeback request.

8 *dvp_count_pf*

Counts the occurrences of L2 cache miss by both software prefetch and hardware prefetch, with writeback request.

Extended events

9 *sx_miss_count_dm_if*

Count of L2 cache miss by demand request for instruction fetch

10 *sx_miss_count_dm_opsh*

Count of L2 cache miss by demand request of shared type for operand access.

11 *sx_miss_count_dm_opex*

Count of L2 cache miss by demand request of exclusive type for operand access.

12 *sx_btc_count*

Number of requests of exclusive type while the line exists in SX with the S or O attributes.

13 *lost_softpf_pfp_full*

Number of software prefetch requests lost due to PF port full.

14 *lost_softpf_by_abort*

Number of software prefetch requests lost due to SX pipe abort.

Q.2.4 Jupiter Bus Event Counters

Basic events

1 *jbus_bi_count*

Counts the number of invalidation requests received.

2 *jbus_cpi_count*

Counts the number of copy and invalidate requests received.

3 *jbus_cpb_count*

Counts the number of copyback requests received.

4 *jbus_cpd_count*

Counts the number of block-load requests and reqd requests from IOs.

Extended events

5 *sn_res_64*

Count number of SC reply, which indicates 1 subline (64 byte) will be transferred to CPU.

6 *sn_res_256*

Count number of SC reply, which indicates 4 subline (256byte) will be transferred to CPU.

7 *jbus_odrbus_busy*

Count number of busy cycles of order buses, from SCs to CPU, in Jupiter Bus cycle. There are four order buses (maximum) connecting SCs and a CPU, with dedicated event counters. *jbus_odrbus_busy* summarizes these counters.

jbus_odrbus_busy = jbus_odrbus0_busy + jbus_odrbus1_busy + jbus_odrbus2_busy + jbus_odrbus3_busy

8 *jbus_reqbus_busy*

Count number of busy cycles of request buses, from CPU to SCs, in CPU cycle. There are four request buses (maximum) connecting a CPU and SCs, with dedicated event counters. *jbus_reeqbus_busy* summarizes these counters.

jbus_reqbus_busy = jbus_reqbus0_busy + jbus_reqbus1_busy + jbus_reqbus2_busy + jbus_reqbus3_busy

9 *jbus_odrbus0_busy*

Count number of busy cycles of the bus from SC0 to the CPU.

10 *jbus_reqbus0_busy*

Count number of busy cycles of the bus from the CPU to SC0.

11 *jbus_odrbus1_busy*

Count number of busy cycles of the bus from SC1 to the CPU.

12 *jbus_reqbus1_busy*

Count number of busy cycles of the bus from the CPU to SC1.

13 *jbus_odrbus2_busy*

Count number of busy cycles of the bus from SC2 to the CPU.

14 *jbus_reqbus2_busy*

Count number of busy cycles of the bus from the CPU to SC2.

15 *jbus_odrbus3_busy*

Count number of busy cycles of the bus from SC3 to the CPU.

16 *jbus_reqbus3_busy*

Count number of busy cycles of the bus from the CPU to SC3.

Q.2.5 Multi-thread specific Event Counters

Extended events

1 *active_cycle_count*

Clock cycles assigned to a given thread.

cycle_count = *active_cycle_count* (thread0) + *active_cycle_count* (thread1)

2 *active_thread_suspend*

Clock cycles when both the threads of a given core are in the suspended or sleep state by executing SUSPEND/SLEEP instructions.

3 *thread_switch_all* (non-speculative)

Total number of thread switches.

4 *ts_by_sxmiss* (non-speculative)

Number of thread switches due to L2$ misses.

5 *ts_by_data_arrive* (non-speculative)

Number of thread switches due to data return of L2$ misses.

6 *ts_by_timer* (non-speculative)

Number of thread switches due to timer.

7 *ts_by_intr* (non-speculative)

Number of thread switches due to interrupts.

8 *ts_by_if* (non-speculative)

Number of thread switches due to L2$ misses for instruction fetch.

9 *ts_by_suspend* (non-speculative)

Number of thread switches due to SUSPEND and SLEEP instructions.

10 *ts_by_other* (non-speculative)

Number of thread switches due to miscellaneous caused not listed above.

Q.3 CPI analysis

A common way to identify a performance bottleneck of SPARC64 VI is to measure the number of stall cycles and the cause of the stall for each instruction. This is called CPI (Cycle Per Instruction) analysis.

The performance events shown in TABLE Q-2 on page 214 are useful for CPI analysis of SPARC64 VI. These events are all counted at the commit stage.

TABLE Q-2 Performance events useful for CPI analysis

Number of instructions and cycles committed		**Factors to prevent the next instruction from committing**	
Inst.	**Cycle**		
4	*active_cycle_counts* *- 3endop - 2endop* *- 1endop - 0endop*	N/A (Up to 4 instructions are committed in a cycle)	
3	*3endop*	*inh_cmit_gpr_2write* + misc. misc. = *2endop + 3endop - inh_cmit_gpr_2write*	
2	*2endop*		
1	*1endop*	misc. = *1endop*	
0	*0endop*	Others	*0endop* *- op_stv_wait - cse_window_empy* *- eu_comp_wait - branch_comp_wait* *- (instruction_flow_counts* *- instruction_counts)*
		Execution	*eu_comp_wait* *+ branch_comp_wait*
		Fetch miss	*cse_window_empy*
		L1D cache miss	*op_stv_wait* *- op_stv_wait_sxmiss* *- op_stv_wait_nc_pend*
		L2 cache miss	*op_stv_wait_sxmiss* *+ op_stv_wait_nc_pend*
	cycle_counts *- active_cycle_counts*	The other thread is running.	

Q.4 Shared performance events between threads

The performance counters (PCR and PIC) are not shared between threads. This is true for performance events as well. In other words, a given performance event increments a performance counter of one and only one thread which has triggered the event.

But there are some exceptions. The following performance events are shared among all the four threads. That is, each event increments PICs of all the threads.

- *cycle_counts*
- Jupiter Bus events

F.APPENDIX **R**

Jupiter Bus Programmer's Model

This chapter describes the programmers model of the Jupiter Bus interface of the SPARC64 VI. The registers for the Jupiter Bus interface and the access method for those registers are described.

R.3 Jupiter Bus Config Register

The Jupiter Bus Config Register is an implementation-specific ASI read-only register. This register is accessible in the ASI $4A_{16}$ space from the processor.

[1]	Register Name:	ASI_JB_CONFIG_REGISTER
[2]	ASI:	$4A_{16}$
[3]	VA:	0
[4]	RW	Supervisor read, a write is ignored.
[5]	Data	

The Jupiter Bus Config Register is illustrated below and described in TABLE R-1.

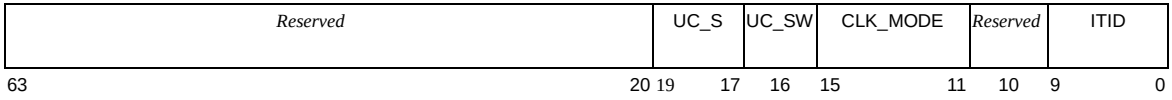


TABLE R-1 Jupiter Bus Config Register Description

Bits	Field	RW	Description
63:20	—	R	*Reserved.* Read as 0.
19:17	`UC_S`	R	U2 cache size: 100_2: 4 MB 101_2: 5 MB 110_2: 6 MB

TABLE R-1 Jupiter Bus Config Register Description *(Continued)*

Bits	Field	RW	Description
16	**UC_SW**	R	U2 cache size per way 0: 0.5 MB 1: 1 MB
15:11	**CLK_MODE**	R	Specify the ratio between CPU clock and JBUS clock. $00000_2 - 01011_2$: *Reserved* 01100_2: 3:1 01101_2: 3.25:1 01110_2: 3.5:1 ... 11110_2: 7.5:1
9:0	**ITID**	R	This field shows ITID (Interrupt Target ID) of the thread.

F.APPENDIX **S**

Summary Differences Between SPARC64 V and SPARC64 VI

The following table summarizes differences between SPARC64 V and SPARC64 VI ISA. This list is a summary, not an exhaustive list.

		SPARC64 V	SPARC64 VI	SPARC64 VI page
Chip	**Chip Architecture**	1CORE 128KB(I) + 128KB(D) L1-Cache 4MB4W L2-Cache	2CORE x 2VMT 128KB(I) + 128KB(D) L1-Cache /core 6MB12W L2-Cache	2, 45 130, 131 131
MMU	**TLB Structure** **Supported page**	2048 sTLB + 32 fTLB 8K, 64K, 512K, 4M	2048 sTLB + 32 fTLB /core 8K, 64K, 512K, 4M, 32M, 256M	94 94
Instruction	**Modified Instructions**	N/A	FMA	55
	New Added Instructions	N/A	POPC SUSPEND SLEEP	66 58 59

		SPARC64 V	SPARC64 VI	SPARC64 VI page
Register	**Newly Added Registers**	N/A	I/D_MMU_TAG_ACCESS_EXT	104
		N/A	I/D_SFPAR	115
		N/A	CACHE_INV	135
	Removed Registers	UPA_PORTID	N/A	N/A
		ASI_AFAR_D1	ASI_AFAR_D1(dummy)	180
		ASI_AFAR_U2	ASI_AFAR_U2 (dummy)	181
	Modified Registers	VER	VER	18
		MCNTL	MCNTL	99
		PRIMARY_CONTEXT	PRIMARY_CONTEXT	102
		SECONDARY_CONTEXT	SECONDARY_CONTEXT	102
		UPA_CONFIG	JB_CONFIG	217
		STCHG_ERROR_INFO	STCHG_ERROR_INFO	169
		SERIAL_ID	SERIAL_ID	48, 127
		L2_DIAG_TAG_READ_REG	L2_DIAG_TAG_READ_REG	48, 135

Index

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W