ORACLE

SPARC M5™ Supplement

to *Oracle SPARC Architecture 2011*

Draft D0.7, 29 May 2014

Privilege Levels: *Privileged and Nonprivileged*

Distribution: Public

Part No: 950-5560-00
Revision: Draft D0.7, 29 May 2014

Oracle Corporation
4150 Network Circle
Santa Clara, CA 95054
U.S.A. 650-960-1300

Contents

CHAPTER 1

SPARC M5 Basics

1.1 Background

SPARC M5 is the latest chip multi-threaded (CMT) processor in the M-series processor family. SPARC M5 utilizes the same processor core (Core S3), including private L2 cache, as the SPARC T4 processor, but implements a new SOC which includes a new crossbar, memory controllers, coherency controllers, I/O, and L3 cache shared between the processor cores.

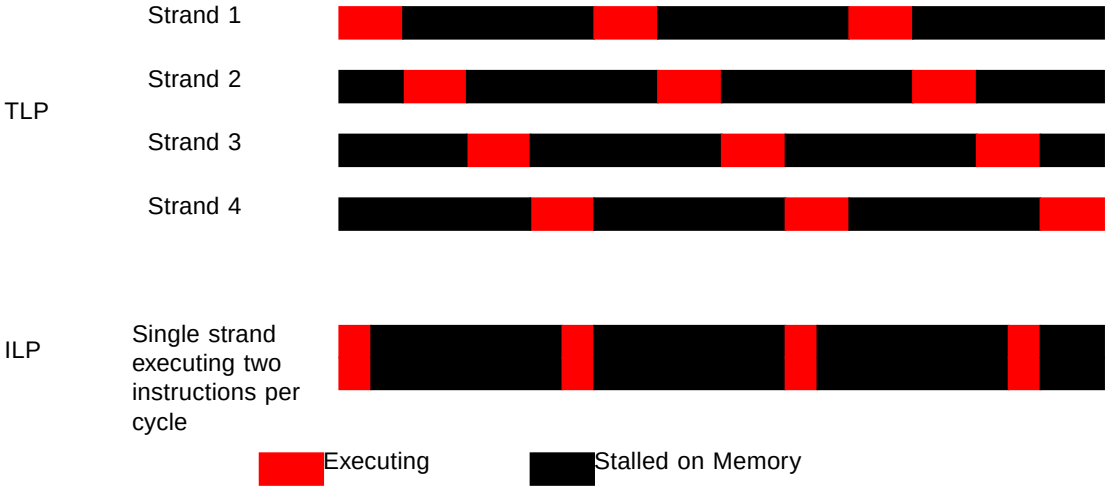
The SPARC M5 product line fulldy implements Sun's Throughput Computing initiative for the horizontal system space. Throughput Computing is a technique that takes advantage of the thread-level parallelism that is present in most commercial workloads. Unlike desktop workloads, which often have a small number of threads concurrently running, most commercial workloads achieve their scalability by employing large pools of concurrent threads.

SPARC M5 supports up to an eight way glueless (without external hub chips) coherent system using 7 coherence link channels, and up to a ninety-six way glued (with external hub chips) coherent system using 6 scalability link channels. SPARC M5 has six SPARC physical processor cores. Each core has full hardware support for eight strands, two integer execution pipelines, one floating-point execution pipeline, and one memory pipeline. The SPARC cores are connected through a crossbar to an on-chip, unified, 48 MB, 4-banked, 12 way associative L3 cache. There are two on-chip memory controllers supporting 4 cascadable BoB's. The BoB chips directly interface to DDR3/DDR4 DIMMs. In addition, there are two on-chip x8 PCI-Express 3.0 I/O interfaces.

Historically, microprocessors have been designed to target desktop workloads, and as a result have focused on running a single thread as quickly as possible. Single thread performance is achieved in these processors by a combination of extremely deep pipelines (over 20 stages in Pentium 4) and by executing multiple instructions in parallel (referred to as instruction-level parallelism or ILP). The basic tenet behind Throughput Computing is that exploiting ILP and deep pipelining has reached the point of diminishing returns, and as a result current microprocessors do not utilize their underlying hardware very efficiently. For many commercial workloads, the processor is idle most of the time waiting on memory, and even when it is executing it will often be able to only utilize a small fraction of its wide execution width. So rather than building a large and complex ILP processor that sits idle most of the time, a number of small, single-issue processors that employ multithreading are built in the same chip area. Combining multiple processors on a single chip with multiple strands per

processor provides very high performance for highly threaded commercial applications. This approach is called thread-level parallelism (TLP), and the difference between TLP and ILP is shown in the FIGURE 1-1.

FIGURE 1-1 Differences Between TLP and ILP



The memory stall time of one strand can often be overlapped with execution of other strands on the same processor, and multiple processors run their strands in parallel. In the ideal case, shown in FIGURE 1-1, memory latency can be completely overlapped with execution of other strands. In contrast, instruction-level parallelism simply shortens the time to execute instructions and does not help much in overlapping execution with memory latency.[1]

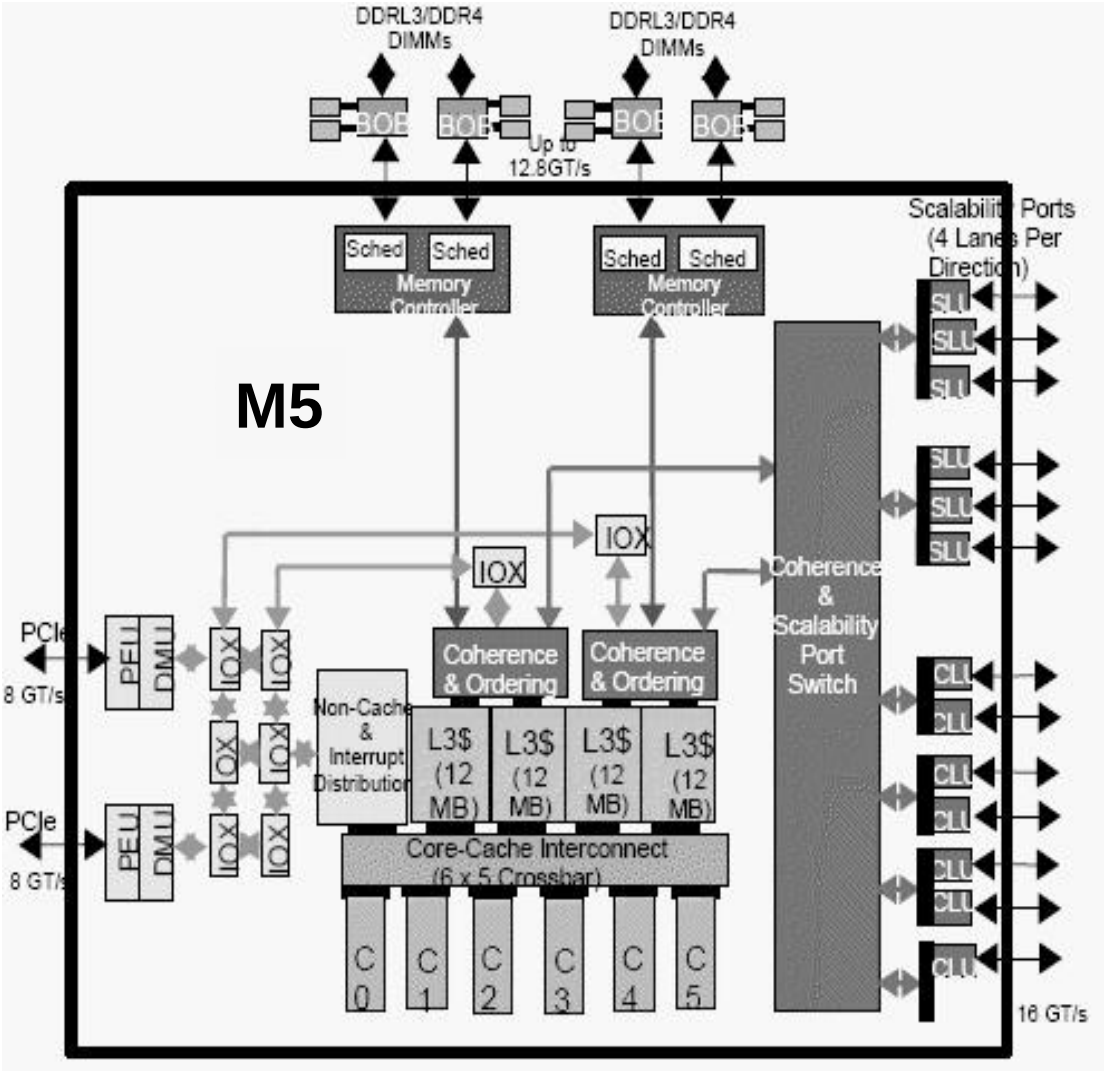
Given this ability to overlap execution with memory latency, why don't more processors utilize TLP? The answer is that designing processors is a mostly evolutionary process, and the ubiquitous deeply pipelined, wide ILP processors of today are the evolutionary outgrowth from a time when the processor was the bottleneck in delivering good performance. With processors capable of multiple GHz clocking, the performance bottleneck has shifted to the memory and I/O subsystems, and TLP has an obvious advantage over ILP for tolerating the large I/O and memory latency prevalent in commercial applications.

Unlike first-generation TLP processors, SPARC M5 seeks to provide the best of TLP and ILP processors. In particular, SPARC M5 provides a robust out-of-order, dual-issue processor core that is heavily threaded among eight strands. It has a 16-stage integer pipeline to achieve high operating frequencies, advanced branch prediction to mitigate the effect of a deep pipeline, and dynamic allocation of processor resources to threads. This allows SPARC M5 to achieve very high single-thread performance (about 5x previous CMT processors such as the UltraSPARC T2), while still scaling to very high levels of throughput.

1.2 SPARC M5 Overview

SPARC M5 is a chip multi-threaded (CMT) processor which supports cache-coherent multi-socket systems. SPARC M5 contains six SPARC physical processor cores. The SPARC physical cores are connected through a crossbar to an on-chip unified 48 Mbyte, 12-way associative L3 cache (64-byte lines). The L3 cache is banked twelve ways to provide sufficient bandwidth for the SPARC physical cores.

1. Processors that employ out-of-order ILP can overlap some memory latency with execution. However, this overlap is typically limited to shorter memory latency events such as L1 cache misses that hit in the L2 cache. Longer memory latency events such as main memory accesses are rarely overlapped to a significant degree with execution by an out-of-order processor.



1.3 SPARC M5 Components

This section describes each component in SPARC M5.

1.3.1 SPARC Physical Core

Each SPARC physical core has hardware support for eight strands. This support consists of a full integer register file with eight register windows per strand, a full floating-point register file per strand, and nearly all of the ASI, ASR, and privileged registers replicated per strand. The eight strands share the instruction and data caches.Each SPARC physical core has a 16 KB, 4-way set-associative instruction cache with 32-byte lines, a 16 KB, 4-way set-associative data cache (32-byte lines), a 128KB, 8-way set-associative L2 cache with 32B lines that are shared by the eight strands. The

L1 data cache is write-through and does not allocate on a write miss; the L2 is store-in and allocates on a write miss. All strands share a floating-point unit incorporating fused multiply-add and VIS3.0 instruction support.

The strands share a dual-issue, out-of-order pipeline, divided into two "slots". One instruction can be issued each cycle to each slot. Slot 0 contains an integer unit and a load/store unit, while slot 1 contains an integer unit, a branch unit, and a floating-point and graphics unit. Up to two instructions can complete each cycle for a peak operation rate of two instructions per cycle. The pipeline is both horizontally and vertically threaded; various segments of the pipeline handle strands differently. The instruction fetch unit fetches instructions from a given strand each cycle. Strands are selected for fetching based upon a least-recently-fetched algorithm. Once fetched, strands are then selected for decoding in a least-recently-decoded fashion and are then renamed and supplied into an out-of-order processor core. Once inside the out-of-order core, strands are picked for issue independently between slots, and in an oldest-ready-first fashion within a slot. Instructions complete out-of-order and are committed in-order within a strand, but independently between strands. Up to 128 instructions can be in flight within the processor core, in any combination across the active strands. In certain circumstances, hardware may activate heuristics to avoid starvation or performance imbalances resulting from unfair access to hardware resources. The L1 cache load-use latency is 5 cycles, the L2 cache load-use latency is 19 cycles, and the L3 load-use latency is 49 cycles.

1.3.1.1 Single-threaded and multi-threaded performance

SPARC M5 is dynamically threaded. While software can activate up to 8 strands on each core at a time, hardware dynamically and seamlessly allocates core resources such as instruction, data, and L2 caches, and out-of-order execution resources such as the 128-entry re-order buffer in the core, among the active strands.

Since the core dynamically allocates resources among the active strands, there is no explicit "single-thread mode" or "multi-thread mode" for software to activate or deactivate.

The extent to which strands compete for core resources depends upon their execution characteristics. These characteristics include cache footprints, inter-instruction dependencies in their execution streams, branch prediction effectiveness, and others. Consider one process which has a small cache footprint and a high correct branch prediction rate which, when running alone on a core, achieves 2 instructions per cycle (SPARC M5's peak rate of instruction execution). We term this a high-IPC process. If another process with similar characteristics is activated on a different strand on the same core, each of the strands will likely operate at approximately 1 instruction per cycle. In other words, the single-thread performance of each process has been cut in half. As a rule of thumb, activating N high-IPC strands will result in each strand executing at 1/N of its peak rate, assuming each strand is capable of executing close to 2 instructions per cycle.

Now consider a process which is largely memory-bound. Its native IPC will be small, perhaps 0.2. If this process runs on one strand on a core with another clone process running on a different strand, there is a good chance that both strands will suffer no noticeable performance loss, and the core throughput will improve to 0.4 IPC. If a low-IPC process runs on one strand with a high-IPC process running on another strand, it's likely that the IPC of either strand will not be greatly perturbed. The high-IPC strand may suffer a slight performance degradation (as long as the low-IPC strand does not cause a substantial increase in cache miss rates for the high-IPC strand).

The guidelines above are only general rules-of-thumb. The extent to which one strand affects another strand's performance depends upon many factors. Processes which run fine on their own but suffer from destructive cache interference when run with other strands may suffer unacceptable performance losses. Similarly, it is also possible for strands to cooperatively improve performance when run together. This may occur when the strands running on one core share code or data. In this case, one strand may prefetch instructions or data that other strands will use in the near future.

The same discussion can apply between cores running in the chip. Since the L3 cache and memory controllers are shared between the cores, activity on one core can influence the performance of strands on another core.

1.3.2 L3 Cache

The L3 cache is banked four ways. It is inclusive of all chip-local L2 caches. To provide for better partial-die recovery, SPARC M5 can also be configured in 2-bank modes (with 1/2 the total cache size). Bank selection is based on address bits 7:6 for 4 banks, and bit 6 for 2 banks. The cache is 48 Mbytes, and 12-way set associative. The line size is 64 bytes.

CHAPTER 2

Data Formats

Data formats supported by SPARC M5 are described in the Oracle SPARC Architecture 2011 specification.

CHAPTER 3

Registers

3.1 Floating-Point State Register (FSR)

Each virtual processor has a Floating-Point State register. This register follows the Oracle SPARC Architecture 2011 specification, with the ver and qne fields permanently set to 0 (SPARC M5 does not support a FQ).

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.2 Ancillary State Registers (ASRs)

This chapter discusses the SPARC M5 ancillary state registers. TABLE 3-1 summarizes and defines these registers.

TABLE 3-1 Summary of SPARC M5 Ancillary State Registers

ASR number	ASR Name	Access	priv	Description
0	Y	RW	N	Y Register
1	*Reserved*	—		Any access causes a *illegal_instruction* trap
2	CCR	RW	N	Condition Code register
3	ASI	RW	N	ASI register
4	TICK	RO	Y[1]	TICK register
5	PC	RO[2]	N	Program counter
6	FPRS	RW	N	Floating-Point Registers Status register
07 - 14	*Reserved*	-		Any access causes an *illegal_instruction* trap
15	(MEMBAR, STBAR)	—	N	Instruction opcodes only, not an actual ASR.
16 - 18	*Reserved*	—		Any access causes an *illegal_instruction* trap
19	GSR	RW	N	General Status register
20	SOFTINT_SET	W	Y[4]	Set bit in Soft Interrupt register
21	SOFTINT_CLR	W	Y[4]	Clear bit in Soft Interrupt register

TABLE 3-1 Summary of SPARC M5 Ancillary State Registers *(Continued)*

ASR number	ASR Name	Access	priv	Description
22	SOFTINT	RW	Y[3]	Soft Interrupt register
23	*Reserved*	—		Any access causes an *illegal_instruction* trap
24	STICK	RW	Y[5]	System Tick register
25	STICK_CMPR	RW	Y[3]	System TICK Compare register
26	CFR	RO[6]	Y	Compatibility Feature Register
27	PAUSE	W	N	Any read causes an *illegal_instruction* trap; PAUSE is write-only
28 - 31	*Reserved*	—		Any access causes an *illegal_instruction* trap

Notes:

1. An attempted write by nonprivileged software to this register causes a *privileged_opcode* trap.An attempted write by privileged software to this register causes an *illegal_instruction* trap. See the Oracle SPARC Architecture 2011 specification for more detail.
2. A write to this register causes an *illegal_instruction* trap.
3. An attempted access in nonprivileged mode causes a *privileged_opcode* trap.
4. Read accesses cause an *illegal_instruction* trap. An attempted write access in nonprivileged mode causes a *privileged_opcode* trap.
5. A write by privileged or user software causes an *illegal_instruction* trap. See the Oracle SPARC Architecture 2011 specification for more detail.
6. Reads are nonprivileged. A write by privileged or user software causes an *illegal_instruction* trap.

3.2.1 Tick Register (TICK)

The TICK register contains one field: counter. The counter field is shared by the eight strands on a physical core.The counter increments each processor core clock.The format of this register is shown in TABLE 3-2.

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

TABLE 3-2 TICK Register – TICK (ASR 04_{16})

Bit	Field	RW	Description
63	—	RO	*Reserved*
62:0	counter	RW	Tick counter, increments each processor core clock cycle.

3.2.2 Program Counter (PC)

Each strand has a read-only program counter register. The PC contains a 52-bit virtual address and VA{63:52} is sign-extended from VA{51}. The format of this register is shown in TABLE 3-3.

TABLE 3-3 Program Counter – PC (ASR 05_{16})

Bit	Field	R/W	Description
63:52	va_high	RO	Sign-extended from VA{51}.
51:2	va	RO	Virtual address contained in the program counter.
1:0	—	RO	The lower 2 bits of the program counter always read as 0.

3.2.3 Floating-Point Registers State Register (FPRS)

This register is described in Oracle SPARC Architecture 2011.

Implementation Note SPARC M5 sets FPRS.du or FPRS.dl when an instruction that updates the floating-point register file successfully completes, or when an FMOVcc or FMOVr instruction that does not satisfy the destination register update condition successfully completes.

3.2.4 General Status Register (GSR)

Each virtual processor has a nonprivileged general status register (GSR). When PSTATE.pef or FPRS.fef is zero, accesses to this register cause an *fp_disabled* trap.

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.2.5 Software Interrupt Register (SOFTINT)

Each virtual processor has a privileged software interrupt register. Nonprivileged accesses to this register cause a *privileged_opcode* trap. The SOFTINT register contains two fields: sm, and int_level. Note that while setting the sm (bit 16) or SOFTINT{14} bits generate *interrupt_level_14*, these bits are considered completely independent of each other. Thus an STICK compare will only set bit 16 and generate *interrupt_level_14*, not also set bit 14.

TABLE 3-4 specifies how *interrupt_level_14* is shared between SOFTINT writes and STICK compares.

TABLE 3-4 Sharing of *interrupt_level_14*

Event	SOFTINT{14}	sm	Action
STICK compare when sm = 0	Unchanged	1	*interrupt_level_14* if PSTATE.ie = 1 and PIL < 14
Set sm = 1 when sm = 0	Unchanged	1	*interrupt_level_14* if PSTATE.ie = 1 and PIL < 14
Set SOFTINT{14} = 1 when SOFTINT{14} = 0.	1	Unchanged	*interrupt_level_14* if PSTATE.ie = 1 and PIL < 14

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.2.6 System Tick Register (STICK)

Each SPARC M5 physical processor core implements an STICK register, shared by all strands of that core.

TABLE 3-5 System Tick Register – STICK (ASR 18_{16})

Bit	Field	R/W	Description
63	—	RO	*Reserved.*
62:0	stick	RW	Elapsed time value, measured in increments of 1 nS.

Privileged software can read the STICK register with the RDSTICK instruction.Privileged software cannot write the STICK register; an attempt by privileged software to execute the WRSTICK instruction results in an *illegal_instruction* exception.

Nonprivileged software can read the STICK register with RDSTICK instruction.Nonprivileged software cannot write the STICK register; an attempt by nonprivileged software to execute the WRSTICK instruction results in an *illegal_instruction* exception.

In SPARC M5, the difference of the values of two different reads of the STICK register reflects the amount of time that has passed between the reads;

(value2 - value1) * 1 = the number of nanoseconds that passed between the reads.

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.2.7 System Tick Compare Register (STICK_CMPR)

Each virtual processor has a privileged System Tick Compare (STICK_CMPR) register. Nonprivileged accesses to this register cause a *privileged_opcode* exception. STICK_CMPR contains two fields: int_dis and stick_cmpr. Only bits 62:7 of the stick_cmpr field are compared against the STICK counter field.

The int_dis bit controls whether a STICK *interrupt_level_14* interrupt is posted in the SOFTINT register when STICK_CMPR bits 62:7 match STICK bits 62:7. The format of this register is shown in TABLE 3-6.

TABLE 3-6 System Tick Compare Register – STICK_CMPR (ASR 19_{16})

Bit	Field	R/W	Description
63	int_dis	RW	stick_int interrupt disable. If 1, stick_int interrupt generation is disabled.
62:7	stick_cmpr	RW	Compare value for stick_int interrupts.
6:0	—	RO	*Reserved.*

After a power-on reset trap, STICK_CMPR.int_dis is set to 1 and STICK_CMPR.cmpr is undefined.

An *stick_match* exception occurs in the cycle in which all of the following three conditions are met:

1. STICK_CMPR.int_dis == 0.
2. A transition occurs from

   (STICK.counter)[62:7] < STICK_CMPR.cmpr[62:7]

   in one cycle, to

   (STICK.counter)[62:7] >= STICK_CMPR.cmpr[62:7]

   in the following cycle
3. This transition of state occurs due to incrementing STICK, and not due to writing STICK or STICK_CMPR

When an *stick_match* interrupt occurs, SOFTINT{16} (sm) is set to 1. This has the effect of posting an *interrupt_level_14* trap request to the virtual processor, which causes an *interrupt_level_14* trap when (PIL < 14) and (PSTATE.ie == 1). The *interrupt_level_14* trap handler must check SOFTINT{14} and SOFTINT{16} (sm) to determine the cause of the *interrupt_level_14* trap.

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.2.8 Compatibility Feature Register (CFR)

For general information on this register, see the Oracle SPARC Architecture 2011 specification.

Each virtual processor has a compatibility feature register (CFR). The CFR is read-only. The format of the CFR is shown in Table 3-7 .

TABLE 3-7 Compatibility Feature Register – CFR (ASR $1A_{16}$)

Bit	Field	R/W	Description
63:12	—	RO	*Reserved*
11	crc32c	RO	If set, the processor supports the CRC32C opcode. If not set, an attempt to execute a CRC32C instruction results in a *compatibility_feature* trap.
10	montsqr	RO	If set, the processor supports the MONTSQR opcode. If not set, an attempt to execute a MONTSQR instruction results in a *compatibility_feature* trap.
9	montmul	RO	If set, the processor supports the MONTMUL opcode. If not set, an attempt to execute a MONTMUL instruction results in a *compatibility_feature* trap.
8	mpmul	RO	If set, the processor supports the MPMUL opcode. If not set, an attempt to execute an MPMUL instruction results in a *compatibility_feature* trap.
7	sha512	RO	If set, the processor supports the SHA512 opcode. If not set, an attempt to execute a SHA512 instruction results in a *compatibility_feature* trap.
6	sha256	RO	If set, the processor supports the SHA256 opcode. If not set, an attempt to execute a SHA256 instruction results in a *compatibility_feature* trap.
5	sha1	RO	If set, the processor supports the SHA1 opcode. If not set, an attempt to execute a SHA1 instruction results in a *compatibility_feature* trap.
4	md5	RO	If set, the processor supports the MD5 opcode. If not set, an attempt to execute an MD5 instruction results in a *compatibility_feature* trap.
3	camellia	RO	If set, the processor supports Camellia opcodes (CAMELLIA_F, CAMELLIA_FL, and CAMELLIA_FLI). If not set, an attempt to execute a Camellia instruction results in a *compatibility_feature* trap.
2	kasumi	RO	If set, the processor supports Kasumi opcodes (KASUMI_FL_XOR, KASUMI_FI_XOR, and KASUMI_FI_FI). If not set, an attempt to execute a Kasumi instruction results in a *compatibility_feature* trap.
1	des	RO	If set, the processor supports DES opcodes (DES_ROUND, DES_IP, DES_IIP, and DES_KEXPAND). If not set, an attempt to execute a DES instruction results in a *compatibility_feature* trap.
0	aes	RO	If set, the processor supports AES opcodes (AES_EROUND01, AES_EROUND23, AES_DROUND01, AES_DROUND23, AES_EROUND_01_LAST, AES_EROUND_23_LAST, AES_DROUND_01_LAST, AES_DROUND_23_LAST, AES_KEXPAND0, AES_KEXPAND1, and AES_KEXPAND2). If not set, an attempt to execute an AES instruction results in a *compatibility_feature* trap.

The CFR enumerates the capabilities that SPARC M5 supports. While the current definition of the CFR only relates to cryptographic capability, additional capabilities may be added in future processors. Software can use the CFR to determine whether a set of cryptographic opcodes associated with a cryptographic function can be executed on an instance of SPARC M5. Hardware also uses the CFR to determine whether a cryptographic capability associated with an opcode is present. When SPARC M5 executes a cryptographic opcode, it associates a bit in the CFR with each opcode; the bit must be set, otherwise a *compatibility_feature* trap occurs.

Programming Note | For optimal performance, prior to using instruction-level cryptographic functions, applications and libraries should first check the CFR to ensure that the desired algorithm is supported by the hardware.

The CFR allows software to construct an architecture that enables opcode reuse. A complete discussion is outside the scope of this document; however, a brief overview follows.

Consider the situation where a processor is introduced that supports three cryptographic opcodes: opA, opB, and opC. Cryptographic requirements could be such that opA=AES, opB=DES, and opC=Kasumi. Traditionally, for any derivative or next-generation processor for which different ciphers were of interest, it would be necessary to expend additional opcodes to achieve the necessary support: e.g. opD=Camellia, opE=MD5. OpA, OpB, and OpC would still be consumed in these follow-on processors, even if there was no longer any interest in the AES, DES, and Kasumi algorithms.

In conjunction with appropriate software architecture and infrastructure, the CFR enables opcode reuse by future processor generations when cryptographic algorithms become obsolete. Potential aliasing problems are disambiguated using the CFR. Each bit in the CFR is permanently assigned to a different cryptographic operation. For instance, bits 0, 1, and 2 are assigned to AES, DES, and Kasumi family opcodes, as shown above. The mapping in the CFR is fixed for all future and derivative processors. When an application wishes to perform an AES operation, it registers that request using the appropriate software architectural means, and uses opA in its binary. Prior to executing, system software or the application checks to make sure that the target processor binds the AES function to opA. It does so by examining the CFR to see if bit 0 is set. If so, the program executes using native AES instructions (opA); if not, system software and/or the application must support a non-native AES instruction implementation using standard instructions. It is expected that cryptographic libraries will contain the necessary checking, so hardware cryptographic support will be transparent to applications that perform cryptographic operations using cryptographic library calls. If the application does not use cryptographic libraries, it should check the CFR to make sure that hardware supports the appropriate function, otherwise it should emulate the function using standard instructions. Alternatively, if performance is not critical, it may rely on trap-and-emulate support provided by higher-level system software.

When the first generation of processor (G1) executes an AES opcode it checks that CFR bit 0 is set. If so, the hardware performs the requested AES operation. Accordingly, on G1, an application is free to perform AES operations using opA. Similar enforcement is applied to DES and Kasumi, respectively.

Now consider what happens if the application is moved to a future processor (G2) which has re-used opA to provide support for Camellia; i.e. opA=Camellia. When system software checks the capabilities for the program, or the program checks, it will see that G2 does not support AES using opA (CFR bit 0 will be 0). This allows system software or the application to emulate AES support using standard instructions. Note that if the application somehow runs without this check having been performed and issues opA, the G2 processor will examine the CFR bit for Camellia, and if set, the application will execute, and get erroneous results (Camellia instead of AES). A similar problem exists if the application is developed for G2 hardware, but somehow runs on a G1 processor. Thus it is vital that system software and/or the application appropriately register their intent and check hardware capability prior to executing cryptographic opcodes.

As a result, given appropriate software infrastructure, instruction set designers may reuse opcodes to perform a variety of different operations and applications will continue to see the expected results on different generation platforms.

3.2.9 Pause (PAUSE)

SPARC M5 physically implements a 12-bit PAUSE register in bits 11:0. The value written to the PAUSE register via the WRPAUSE instruction is an unsigned 15-bit value that is then right-shifted by 3 bits (divided by 8) since hardware decrements the PAUSE register once every 8 cycles. Thus the unsigned 12-bit value represents a cycle count from 0 to a maximum of 32760 cycles. Writing to the non-privileged PAUSE register stalls a thread for the number of cycles specified by the XOR of the source operands, except as follows:

1. Writing 0 to the PAUSE register stalls the thread for the minimum number of cycles (greater than zero since there is a minimum stall time due to internal pipeline delays).
2. Writing a value larger than 2^{15} - 1 causes hardware to saturate the 12-bit PAUSE register; hardware sets PAUSE to FFF_{16} prior to decrementing it.

While the PAUSE register is nonzero, no instructions are selected from the strand issuing a WRPAUSE. An unmasked disrupting exception terminates the PAUSE.

For more information on this instruction, see the Oracle SPARC Architecture 2011 specification or Section 5.4, *WRPAUSE*, on page 36.

3.3 Privileged PR State Registers

TABLE 3-8 lists the privileged registers.

TABLE 3-8 Privileged Registers

Register	Register Name	Access	Description
0	TPC	RW	Trap PC[1]
1	TNPC	RW	Trap Next PC[1]
2	TSTATE	RW	Trap State
3	TT	RW	Trap Type
4	TICK	RW	Tick
5	TBA	RW	Trap Base Address[1]
6	PSTATE	RW	Process State
7	TL	RW	Trap Level
8	PIL	RW	Processor Interrupt Level
9	CWP	RW	Current Window Pointer
10	CANSAVE	RW	Savable Windows
11	CANRESTORE	RW	Restorable Windows
12	CLEANWIN	RW	Clean Windows
13	OTHERWIN	RW	Other Windows
14	WSTATE	RW	Window State
16	GL	RW	Global Level

1. SPARC M5 only implements bits 51:0 of the TPC, TNPC, and TBA registers. Bits 63:52 are always sign-extended from bit 51.

3.3.1 Trap State Register (TSTATE)

Each virtual processor has *MAXPTL* (2) Trap State registers. These registers hold the state values from the previous trap level. The format of one element the TSTATE register array (corresponding to one trap level) is shown in TABLE 3-9.

TABLE 3-9 Trap State Register

Bit	Field	R/W	Description
63:42	—	RO	*Reserved*.
41:40	gl	RW	Global level at previous trap level
39:32	ccr	RW	CCR at previous trap level
31:24	asi	RW	ASI at previous trap level
23:21	—	RO	*Reserved*
20	pstate tct	RW	PSTATE.tct at previous trap level
19:18	—	RO	*Reserved* (corresponds to bits 11:10 of PSTATE)
17	pstate cle	RW	PSTATE.cle at previous trap level
16	pstate tle	RW	PSTATE.tle at previous trap level
15:13	—	RO	*Reserved* (corresponds to bits 7:5 of PSTATE)
12	pstate pef	RW	PSTATE.pef at previous trap level
11	pstate am	RW	PSTATE.am at previous trap level
10	pstate priv	RW	PSTATE.priv at previous trap level
9	pstate ie	RW	PSTATE.ie at previous trap level
8	—	RO	*Reserved* (corresponds to bit 0 of PSTATE)
7:3	—	RO	*Reserved*
2:0	cwp	RW	CWP from previous trap level

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.3.2 Processor State Register (PSTATE)

Each virtual processor has a Processor State register. More details on PSTATE can be found in the Oracle SPARC Architecture 2011 specification. The format of this register is shown in TABLE 3-10; note that the memory model selection field (mm) mentioned in Oracle SPARC Architecture 2011 is not implemented in SPARC M5.

TABLE 3-10 Processor State Register

Bit	Field	R/W	Description
63:13	—	RO	*Reserved*
12	tct	RW	Trap on control transfer
11:10	—	RO	*Reserved*
9	cle	RW	Current little endian
8	tle	RW	Trap little endian
7:6	—	RO	*Reserved* (mm; not implemented in SPARC M5)
5	—	RO	*Reserved*
4	pef	RW	Enable floating-point
3	am	RW	Address mask
2	priv	RW	Privileged mode
1	ie	RW	Interrupt enable
0	—	RO	*Reserved* (was ag)

Programming Note | Hyperprivileged changes to translation in delay slots of delayed control transfer instructions should be avoided.

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.3.3 Trap Level Register (TL)

Each virtual processor has a Trap Level register. Writes to this register saturate at *MAXPTL* (2). This saturation is based on bits 2:0 of the write data; bits 63:3 of the write data are ignored.

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.3.4 Current Window Pointer (CWP) Register

Since *N_REG_WINDOWS* = 8 on SPARC M5, the CWP register in each virtual processor is implemented as a 3-bit register.

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

3.3.5 Global Level Register (GL)

Each virtual processor has a Global Level register, which controls which set of global register windows is in use. The maximum global level (*MAXPGL*) for SPARC M5 is 2, so GL is implemented as a 2-bit register on SPARC M5. On a trap, GL is set to **min**(GL + 1,*MAXPTL*).

Writes to the GL register saturate at *MAXPTL*. This saturation is based on bits 1:0 of the write data; bits 63:2 of the write data are ignored.

The format of the GL register is shown in TABLE 3-11.

TABLE 3-11 Global Level Register

Bit	Field	R/W	Description
63:2	—	RO	*Reserved*
1:0	gl	RW	Global level.

For more information on this register, see the Oracle SPARC Architecture 2011 specification.

CHAPTER 4

Instruction Formats

Instruction formats are described in the Oracle SPARC Architecture 2011 specification.

CHAPTER 5

Instruction Definitions

5.1 Instruction Set Summary

The SPARC M5 CPU implements the Oracle SPARC Architecture 2011 instruction set.

TABLE 5-1 lists the complete SPARC M5 instruction set supported in hardware. All instructions that are part of Oracle SPARC Architecture 2011 are documented in the Oracle SPARC Architecture 2011 specification; any instructions that are extensions to OSA 2011 are documented in this chapter.

TABLE 5-1 Complete SPARC M5 Hardware-Supported Instruction Set *(1 of 6)*

Opcode	Description
ADD (ADDcc)	Add (and modify condition codes)
ADDC (ADDCcc)	Add with carry (and modify condition codes)
ADDXC (ADDXCcc)	Add extended with carry (and modify condition codes)
AES_DROUND01	AES decrypt round, columns 0 & 1
AES_DROUND23	AES decrypt round, columns 2 & 3
AES_DROUND01_LAST	AES decrypt last round, columns 0 & 1
AES_DROUND23_LAST	AES decrypto last round, columns 2 & 3
AES_EROUND01	AES encrypt round, columns 0 & 1
AES_EROUND23	AES encrypt round, columns 2 & 3
AES_EROUND01_LAST	AES encrypt last round, columns 0 & 1
AES_EROUND23_LAST	AES encrypt last round, columns 2 & 3
AES_KEXPAND0	AES key expansion without round constant
AES_KEXPAND1	AES key expansion with round constant
AES_KEXPAND2	AES key expansion without SBOX
ALIGNADDRESS	Calculate address for misaligned data access
ALIGNADDRESSL	Calculate address for misaligned data access (little-endian)
ALLCLEAN	Mark all windows as clean
AND (ANDcc)	And (and modify condition codes)
ANDN (ANDNcc)	And not (and modify condition codes)
ARRAY{8,16,32}	3-D address to blocked byte address conversion
Bicc	Branch on integer condition codes
BMASK	Writes the GSR.mask field
BPcc	Branch on integer condition codes with prediction
BPr	Branch on contents of integer register with prediction
BSHUFFLE	Permutes bytes as specified by the GSR.mask field
CALL[1]	Call and link
CAMELLIA_F	Camellia F operation
CAMELLIA_FL	Camellia FL operation

TABLE 5-1 Complete SPARC M5 Hardware-Supported Instruction Set *(2 of 6)*

Opcode	Description
CAMELLIA_FLI	Camellia FLI operation
CASA	Compare and swap word in alternate space
CASXA	Compare and swap doubleword in alternate space
C{W,X}Bcond	Fused 32 or 64 bit compare and conditional branch
CMASK{8,16,32}	Create GSR.maskfrom SIMD operation result
CRC32C	CRC32C polynomial instruction
DES_IP	DES initial permutation
DES_IIP	DES inverse initial permutation
DES_KEXPAND	DES key expansion
DES_ROUND	DES round
DONE	Return from trap
EDGE{8,16,32}{L}{N}	Edge boundary processing {little-endian} {non-condition-code altering}
FABS(s,d)	Floating-point absolute value
FADD(s,d)	Floating-point add
FALIGNDATA	Perform data alignment for misaligned data
FANDNOT1{s}	Negated *src1* **and** *src2* (single precision)
FANDNOT2{s}	*Src1* **and** negated *src2* (single precision)
FAND{s}	Logical **and** (single precision)
FBPfcc	Branch on floating-point condition codes with prediction
FBfcc	Branch on floating-point condition codes
FCHKSM16	16-bit partitioned checksum
FCMP(s,d)	Floating-point compare
FCMPE(s,d)	Floating-point compare (exception if unordered)
FCMPEQ{16,32}	Four 16-bit / two 32-bit compare: set integer dest if *src1* = *src2*
FCMPGT{16,32}	Four 16-bit / two 32-bit compare: set integer dest if *src1* > *src2*
FCMPLE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if *src1* ≤ *src2*
FCMPNE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if *src1* ≠ *src2*
FDIV(s,d)	Floating-point divide
FEXPAND	Four 8-bit to 16-bit expand
FHADD{s,d}	Floating-point add and halve
FHSUB{s,d}	Floating-point subtract and halve
FiTO(s,d)	Convert integer to floating-point
FLUSH	Flush instruction memory
FLUSHW	Flush register windows
FLCMP{s,d}	Lexicographic compare
FMADD{s,d}	Floating-point multiply-add single/double (fused)
FMEAN16	16-bit partitioned average
FMOV(s,d)	Floating-point move
FMOV(s,d)cc	Move floating-point register if condition is satisfied
FMOV(s,d)R	Move floating-point register if integer register contents satisfy condition
FMSUB{s,d}	Floating-point multiply-subtract single/double (fused)
FMUL(s,d)	Floating-point multiply
FMUL8SUX16	Signed upper 8- x 16-bit partitioned product of corresponding components
FMUL8ULX16	Unsigned lower 8- x 16-bit partitioned product of corresponding components
FMUL8X16	8- x 16-bit partitioned product of corresponding components
FMUL8X16AL	Signed lower 8- x 16-bit lower α partitioned product of four components
FMUL8X16AU	Signed upper 8- x 16-bit lower α partitioned product of four components

TABLE 5-1 Complete SPARC M5 Hardware-Supported Instruction Set *(3 of 6)*

Opcode	Description
FMULD8SUX16	Signed upper 8- x 16-bit multiply → 32-bit partitioned product of components
FMULD8ULX16	Unsigned lower 8- x 16-bit multiply → 32-bit partitioned product of components
FNADD(s,d)	Floating-point add and negate
FNAND{s}	Logical **nand** (single precision)
FNEG(s,d)	Floating-point negate
FNHADD{s,d}	Floating-point add and halve, then negate
FNMADD{s,d}	Floating-point multiply-add and negate
FNMSUB{s,d}	Floating-point negative multiply-subtract single/double (fused)
FNMUL{s,d}	Floating-point multiply and negate
FNsMULd	Floating-point multiply and negate
FNOR{s}	Logical **nor** (single precision)
FNOT1{s}	Negate (1's complement) *src1* (single precision)
FNOT2{s}	Negate (1's complement) *src2* (single precision)
FONE{s}	One fill (single precision)
FORNOT1{s}	Negated *src1* **or** *src2* (single precision)
FORNOT2{s}	*src1* **or** negated *src2* (single precision)
FOR{s}	Logical **or** (single precision)
FPACKFIX	Two 32-bit to 16-bit fixed pack
FPACK{16,32}	Four 16-bit/two 32-bit pixel pack
FPADD{16,32}{s}	Four 16-bit/two 32-bit partitioned add (single precision)
FPADD64	Fixed-point partitioned add
FPADDS{16,32}{s}	Fixed-point partitioned add
FPMADDX	Unsigned integer multiply-add
FPMADDXHI	Unsigned integer multiply-add, return high-order 64 bits of result
FPMERGE	Two 32-bit to 64-bit fixed merge
FPSUB{16,32}{s}	Four 16-bit/two 32-bit partitioned subtract (single precision)
FPSUB64	Fixed-point partitioned subtract, 64-bit
FPSUBS{16,32}{s}	Fixed-point partitioned subtract
FSLL{16,32}	16- or 32-bit partitioned shift, left (old mnemonic FSHL)
FSLAS{16,32}	16- or 32-bit partitioned shift, left or right (old mnemonic FSHLAS)
FSRA{16,32}	16- or 32-bit partitioned shift, left or right (old mnemonic FSHRA)
FSRL{16,32}	16- or 32-bit partitioned shift, left or right (old mnemonic FSHRL)
FsMULd	Floating-point multiply single to double
FSQRT(s,d)	Floating-point square root
FSRC1{s}	Copy *src1* (single precision)
FSRC2{s}	Copy *src2* (single precision)
F(s,d)TO(s,d)	Convert between floating-point formats
F(s,d)TOi	Convert floating point to integer
F(s,d)TOx	Convert floating point to 64-bit integer
FSUB(s,d)	Floating-point subtract
FUCMP{GT,LE,NE,EQ}8	Compare 8-bit unsigned fixed-point values
FXNOR{s}	Logical **xnor** (single precision)
FXOR{s}	Logical **xor** (single precision)
FxTO(s,d)	Convert 64-bit integer to floating-point
FZERO{s}	Zero fill (single precision)
ILLTRAP	Illegal instruction
INVALW	Mark all windows as CANSAVE

TABLE 5-1 Complete SPARC M5 Hardware-Supported Instruction Set *(4 of 6)*

Opcode	Description
JMPL	Jump and link
KASUMI_FI_XOR	Kasumi FI followed by XOR
KASUMI_FI_FI	Kasumi FI followed by FI
KASUMI_FL_XOR	Kasumi FL followed by XOR
LDBLOCKF	64-byte block load
LDDF	Load double floating-point
LDDFA	Load double floating-point from alternate space
LDF	Load floating-point
LDFA	Load floating-point from alternate space
LDFSR	Load floating-point state register lower
LDSB	Load signed byte
LDSBA	Load signed byte from alternate space
LDSH	Load signed halfword
LDSHA	Load signed halfword from alternate space
LDSTUB	Load-store unsigned byte
LDSTUBA	Load-store unsigned byte in alternate space
LDSW	Load signed word
LDSWA	Load signed word from alternate space
LDTW	Load twin words
LDTWA	Load twin words from alternate space
LDUB	Load unsigned byte
LDUBA	Load unsigned byte from alternate space
LDUH	Load unsigned halfword
LDUHA	Load unsigned halfword from alternate space
LDUW	Load unsigned word
LDUWA	Load unsigned word from alternate space
LDX	Load extended
LDXA	Load extended from alternate space
LDXFSR	Load extended floating-point state register
LDXEFSR	Load extended floating-point state register
LZD	Leading zero detect on 64-bit integer register
MD5	MD5 hash
MEMBAR	Memory barrier
MONTMUL	Montgomery multiplication
MONTSQR	Montgomery squaring
MOVcc	Move integer register if condition is satisfied
MOVr	Move integer register on contents of integer register
MOVdTOx	Move floating-point register to integer register
MOVsTO{d,s}w	Move floating-point register to integer register
MOV{x,w}TO{d,s}	Move integer register to floating-point register
MPMUL	Multiple-precision multiplication
MULScc	Multiply step (and modify condition codes)
MULX	Multiply 64-bit integers
NOP	No operation
NORMALW	Mark other windows as restorable
OR (ORcc)	Inclusive-or (and modify condition codes)
ORN (ORNcc)	Inclusive-or not (and modify condition codes)

TABLE 5-1 Complete SPARC M5 Hardware-Supported Instruction Set *(5 of 6)*

Opcode	Description
OTHERW	Mark restorable windows as other
PDIST	Distance between 8 8-bit components
PDISTN	Pixel component distance
POPC	Population count
PREFETCH	Prefetch data
PREFETCHA	Prefetch data from alternate space
PST	Eight 8-bit/4 16-bit/2 32-bit partial stores
RDASI	Read ASI register
RDASR	Read ancillary state register
RDCCR	Read condition codes register
RDCFR	Read compatibility feature register
RDFPRS	Read floating-point registers state register
RDPC	Read program counter
RDPR	Read privileged register
RDTICK	Read TICK register
RDY	Read Y register
RESTORE	Restore caller's window
RESTORED	Window has been restored
RETRY	Return from trap and retry
RETURN	Return
SAVE	Save caller's window
SAVED	Window has been saved
SDIV (SDIVcc)	32-bit signed integer divide (and modify condition codes)
SDIVX	64-bit signed integer divide
SETHI	Set high 22 bits of low word of integer register
SHA1	SHA-1 hash
SHA256	SHA-256 hash
SHA512	SHA-512 hash
SIAM	Set interval arithmetic mode
SLL	Shift left logical
SLLX	Shift left logical, extended
SMUL (SMULcc)	Signed integer multiply (and modify condition codes)
SRA	Shift right arithmetic
SRAX	Shift right arithmetic, extended
SRL	Shift right logical
SRLX	Shift right logical, extended
STB	Store byte
STBA	Store byte into alternate space
STBAR	Store barrier
STBLOCKF	64-byte block store
STDF	Store double floating-point
STDFA	Store double floating-point into alternate space
STF	Store floating-point
STFA	Store floating-point into alternate space
STFSR	Store floating-point state register
STH	Store halfword
STHA	Store halfword into alternate space

TABLE 5-1 Complete SPARC M5 Hardware-Supported Instruction Set *(6 of 6)*

Opcode	Description
STTW	Store twin words
STTWA	Store twin words into alternate space
STW	Store word
STWA	Store word into alternate space
STX	Store extended
STXA	Store extended into alternate space
STXFSR	Store extended floating-point state register
SUB (SUBcc)	Subtract (and modify condition codes)
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)
SWAP	Swap integer register with memory
SWAPA	Swap integer register with memory in alternate space
TADDcc (TADDccTV)	Tagged add and modify condition codes (trap on overflow)
TSUBcc (TSUBccTV)	Tagged subtract and modify condition codes (trap on overflow)
Tcc	Trap on integer condition codes (with 8-bit sw_trap_number, if bit 7 is set trap to hyperprivileged)
UDIV (UDIVcc)	Unsigned integer divide (and modify condition codes)
UDIVX	64-bit unsigned integer divide
UMUL (UMULcc)	Unsigned integer multiply (and modify condition codes)
UMULXHI	Unsigned 64 x 64 multiply, returning upper 64 product bits
WRASI	Write ASI register
WRASR	Write ancillary state register
WRCCR	Write condition codes register
WRFPRS	Write floating-point registers state register
WRPR	Write privileged register
WRY	Write Y register
XMULX{HI}	XOR multiply
XNOR (XNORcc)	Exclusive-nor (and modify condition codes)
XOR (XORcc)	Exclusive-or (and modify condition codes)

1. The PC format saved by the CALL instruction is the same as the format of the PC register specified in Section 3.2.2, *Program Counter (PC)*, on page 18.

TABLE 5-1 lists the SPARC V9 and sun4v instructions that are not directly implemented in hardware by SPARC M5, and the exception that occurs when an attempt is made to execute them.

TABLE 5-2 Oracle SPARC Architecture 2011 Instructions Not Directly Implemented by SPARC M5 Hardware *(1 of 2)*

Opcode	Description	Exception
FABSq	Floating-point absolute value quad	*illegal_instruction*
FADDq	Floating-point add quad	*illegal_instruction*
FCMPq	Floating-point compare quad	*illegal_instruction*
FCMPEq	Floating-point compare quad (exception if unordered)	*illegal_instruction*
FDIVq	Floating-point divide quad	*illegal_instruction*
FdMULq	Floating-point multiply double to quad	*illegal_instruction*
FiTOq	Convert integer to quad floating-point	*illegal_instruction*
FMOVq	Floating-point move quad	*illegal_instruction*
FMOVqcc	Move quad floating-point register if condition is satisfied	*illegal_instruction*

TABLE 5-2 Oracle SPARC Architecture 2011 Instructions Not Directly Implemented by SPARC M5 Hardware *(2 of 2)*

Opcode	Description	Exception
FMOVqr	Move quad floating-point register if integer register contents satisfy condition	*illegal_instruction*
FMULq	Floating-point multiply quad	*illegal_instruction*
FNEGq	Floating-point negate quad	*illegal_instruction*
FSQRTq	Floating-point square root quad	*illegal_instruction*
F(s,d,q)TO(q)	Convert between floating-point formats to quad	*illegal_instruction*
FQTOI	Convert quad floating point to integer	*illegal_instruction*
FQTOX	Convert quad floating point to 64-bit integer	*illegal_instruction*
FSUBq	Floating-point subtract quad	*illegal_instruction*
FxTOq	Convert 64-bit integer to floating-point	*illegal_instruction*
IMPDEP1 (not listed in TABLE 5-1)	Implementation-dependent instruction	*illegal_instruction*
IMPDEP2 (not listed in TABLE 5-1)	Implementation-dependent instruction	*illegal_instruction*
LDQF	Load quad floating-point	*illegal_instruction*
LDQFA	Load quad floating-point into alternate space	*illegal_instruction*
STQF	Store quad floating-point	*illegal_instruction*
STQFA	Store quad floating-point into alternate space	*illegal_instruction*

5.2 SPARC M5-Specific Instructions

5.3 PREFETCH/PREFETCHA

See the PREFETCH and PREFETCHA instruction descriptions in the Oracle SPARC Architecture 2011 specification for the standard definitions of these instructions. This section describes how SPARC M5 handles PREFETCH instructions.

SPARC M5 interprets the function codes for prefetch variants as follows:

TABLE 5-3 SPARC M5 interpretation of prefetch variants

fcn	Prefetch Variant	Action
0	Weak prefetch for several reads	Prefetch to L1 data cache and L2 cache
1	Weak prefetch for one read	Prefetch to L2 cache
2	Weak prefetch for several writes	Prefetch to L2 cache (exclusive)
3	Weak prefetch for one write	Prefetch to L2 cache (exclusive)
4	Prefetch Page	NOP - no action taken
5 - 15	*Reserved*	*illegal_instruction* trap
16	NOP	NOP - no action taken
17	Strong prefetch to nearest unified cache	Prefetch to L2 cache
18 - 19	NOP	NOP - no action taken
20	Strong prefetch for several reads	Prefetch to L1 data cache and L2 cache
21	Strong prefetch for one read	Prefetch to L2 cache

TABLE 5-3 SPARC M5 interpretation of prefetch variants *(Continued)*

fcn	Prefetch Variant	Action
22	Strong prefetch for several writes	Prefetch to L2 cache (exclusive)
23	Strong prefetch for one write	Prefetch to L2 cache (exclusive)
24-31	NOP	NOP - no action taken

Programming Note SPARC M5 does not implement any prefetch functions that prefetch solely to the L3 cache.

Implementation Note On SPARC M5, prefetches can be dropped either at the L1 data cache, the L2 cache, or the L3 cache. Prefetches may be dropped regardless of whether they are strong or weak. Weak prefetches are dropped if they miss the DTLB, whereas strong prefetches are dropped if hardware tablewalk returns an error or is not enabled; otherwise, the following conditions apply to either type. Prefetches are dropped when:

1. The prefetch is to an I/O page, or a page marked as non-cacheable or with side-effects.
2. The miss buffer in the L1 data cache fills beyond a high-water mark (this only applies when more than one thread is unparked).
3. The prefetch is for a data cache miss which is already outstanding.
4. The prefetch is a read prefetch that hits in the L1 cache.
5. The prefetch is a read prefetch to L2 which hits in the L2 cache.
6. The prefetch is a write prefetch which exists in the L2 cache in the exclusive state.
7. The prefetch misses in the L2 cache, and the L2 miss buffer fills beyond a high water mark.
8. The prefetch misses in the L3 cache, and the L3 miss buffer fills beyond a high water mark.

5.4 WRPAUSE

WRPAUSE is a mnemonic for a WRASR to ASR 27, the PAUSE register.

Writing to the PAUSE register suspends a strand for a specified number of processor cycles. The PAUSE register is write-only; the PAUSE register cannot be read. SPARC M5 implements a 15-bit PAUSE register as described below:

TABLE 5-4 PAUSE Register

Bit	Field	R/W	Description
63:15	—	WO	*Reserved*.
14:3	pause	WO	Pause value from 0..32767 cycles
2:0	—	WO	*Ignored*.

When WRPAUSE is executed, the following sequence of events occurs in SPARC M5:

1. Hardware places the strand in a paused state[1]. While in this state, execution of any subsequent instructions to WRPAUSE for this strand is temporarily suspended. Instruction fetch continues for this strand until the strand's instruction buffer is filled.

2. Hardware checks the value that will be written to the PAUSE register. Hardware updates the strand's PAUSE register with the value of ((min (2^{15} - 1, (R[rs1] **xor** simm13)) >> 3)) or the value ((min (2^{15} - 1, (R[rs1] **xor** R[rs2])) >> 3)), depending upon the instruction format. If the value written to PAUSE is 0, hardware will pause the strand for 1 cycle. The value placed in the PAUSE register is divided by 8 since each strand's PAUSE register is processed every 8th processor clock cycle. Thus the actual duration of a WRPAUSE ranges from 1 to a maximum of 32760 cycles.

3. Hardware decrements the PAUSE register every 8th processor clock cycle. The strand remains in the paused state until any of the following events occur; any one of which immediately forces the PAUSE register to be reset to 0:

   a. the PAUSE register decrements to zero

   b. an unmasked disrupting trap request is received

   c. ,a deferred trap request is received

   d. certain hyperprivileged events occur

   e. Also see Oracle SPARC Architecture 2011 for details on what events can terminate a WRPAUSE operation.

4. When the PAUSE register becomes 0, SPARC M5 resumes instruction fetch and execution at the NPC of the WRPAUSE[2].

A masked trap request does not affect the PAUSE register or suspension of the strand.

Any disrupting trap request that is posted after WRPAUSE has updated the PAUSE register and the strand has suspended forward progress does not result in a trap being taken on the WRPAUSE instruction; the trap is taken on a later instruction. This ensures forward progress when the trap handler retries the instruction on which the trap was taken.

Programming Note | WRPAUSE is intended to be used as part of a progressive (exponential) backoff algorithm.

Programming Note | In SPARC M5, care must be taken with the frequency of branch instructions when coding loops using WRPAUSE. See Section A.4 for details.

1. SPARC M5 post-sync's the WRPAUSE instruction; hardware prevents any subsequent instruction from this strand from entering the pipeline at the Select stage.

2. Hardware releases the post-sync at the Select stage, enabling subsequent instructions to enter the pipeline.

5.5 Block Load and Store Instructions

See the LDBLOCKF and STBLOCKF instruction descriptions in the Oracle SPARC Architecture 2011 specification for the standard definitions of these instructions.

Block store commits in SPARC M5 do NOT force the data to be written to memory as specified in the Oracle SPARC Architecture 2011 specification. Block store commits are implemented the same as block stores in SPARC M5. As with all stores, block stores and block store commits maintain coherency with all I-caches, but will not flush any modified instructions executing down a pipeline. Flushing those instructions requires the pipeline to execute a FLUSH instruction.

Notes

If LDBLOCKF is used with an `ASI_BLK_COMMIT_{P,S}` and a destination register number rd is specified which is not a multiple of 8 (a misaligned rd), SPARC M5 generates an *illegal_instruction* exception (impl. dep. #255-U3-Cs10).

If LDBLOCKF is used with an `ASI_BLK_COMMIT_{P,S}` and a memory address is specified with less than 64-byte alignment, SPARC M5 generates a *DAE_invalid_ASI* exception (impl. dep. #256-U3)

These instructions are used for transferring large blocks of data (more than 256 bytes); for example, `memcpy()` and `memset()`. On SPARC M5, a block load forces a miss in the primary cache and will not allocate a line in the primary cache, but does allocate in L2.

SPARC M5 breaks block load and store instructions into 8 individual "helper" instructions. Each helper is translated as an independent instruction. Thus, it is possible that any individual helper or set of helpers translates to a different memory page from other helpers from the same instruction, if the underlying memory mapping is changed by another process during the execution of the block instruction. Any individual helper or set of helpers may also trap if memory mapping attributes are changed by another process in the midst of a series of helper translations. In the event multiple helpers have exceptions, SPARC M5 commits the helpers in program order from the lowest virtual address to the highest virtual address. Thus, the helper with the lowest virtual address which experiences an exception determines which trap will be taken. SPARC M5 makes no guarantee about the atomicity of address translation for block operations.

Block stores execute differently on SPARC M5 than on prior UltraSPARC processors. On previous processors, such as UltraSPARC T2, UltraSPARC T2+, and SPARC T3, block stores fetched the data from memory prior to updating the line with the store data. On SPARC M5, the processor first establishes the line in the L2 cache and zeroes the data, prior to updating the line with the store source data. The block store is helperized into 8 individual block init stores. The first helper establishes the line in the L2 cache, zeroes the line out, then updates the first 8 bytes of the line with the first 8 bytes of the store source data. The remaining seven helpers collectively update the remaining 56 bytes with the remaining 56 bytes of store source data. As a result, it is possible for another process to see the old data, the new data, or a value of zero while the block store is being executed.

SPARC M5 treats LDBLOCKF as interlocked with respect to following instructions. All later instructions see the effect of the newly loaded values.

STBLOCKF source data registers are interlocked against completion of previous instructions, including block load instructions; STBLOCKF instructions don't commit until all previous instructions commit. Thus STBLOCKF instructions read the most recent value of the floating-point source register(s) when committing to memory. STBLOCKF instructions may or may not initialize the target memory locations to 0 prior to updating them with the source data. Thus another strand may observe these intermediate zero values prior to observing the final source data value.

LDBLOCKF does not follow memory model ordering with respect to stores. In particular, a read-after-write hazard to overlapping addresses is not detected. The side-effect bit associated with the access is ignored (see *Translation Table Entry (TTE)* on page 83). If ordering with respect to earlier stores is important (for example, a block load that overlaps previous stores), then there must be an intervening MEMBAR `#StoreLoad` (or stronger MEMBAR). If the LDBLOCKF overlaps a previous store and there is no intervening MEMBAR or data reference, the LDBLOCKF may return data from before or after the store.

STBLOCKF instructions do not conform to TSO store-store ordering with respect to older non-overlapping stores. A subsequent load to the same address as a STBLOCKF may not read the results of the STBLOCKF. The side-effects bit associated with the access is ignored. If ordering with respect to later loads is important then there must be an intervening MEMBAR instruction. If the STBLOCKF overlaps a later load and there is no intervening MEMBAR #StoreLoad instruction, the result of the load is undefined.

Compatibility Notes

Block load and store operations do not obey the ordering restrictions of the currently selected processor memory model (TSO, PSO, or RMO); block operations always execute under an RMO memory ordering model. In general, explicit MEMBAR instructions are required to order block memory operations among themselves or with respect to normal loads and stores. In addition, block operations do not generally conform to dependence order on the issuing virtual processor; that is, no read-after-write or write-after-read checking occurs between block loads and stores. Explicit MEMBARs are required to enforce dependence ordering between block operations that reference the same address. However, SPARC M5 partially orders some block operations.

TABLE 5-5 describes the synchronization primitives required in SPARC M5, if any, to guarantee TSO ordering between various sequences of memory reference operations. The first column contains the reference type of the first or earlier instruction; the second column contains the reference type of the second or the later instruction. SPARC M5 orders loads and block loads against all subsequent instructions.

TABLE 5-5 SPARC M5 Synchronization Requirements for Memory Reference Operations

First reference	Second reference	Synchronization Required
Load	Load	—
	Block load	MEMBAR #LoadLoad, #StoreLoad, #MemIssue, or #Sync
	Store	—
	Block store	—
Block load	Load	—
	Block load	MEMBAR #LoadLoad, #StoreLoad, #MemIssue, or #Sync
	Store	—
	Block store	—
Store	Load	—
	Block load	MEMBAR #StoreLoad or #Sync
	Store	—
	Block store	MEMBAR #StoreStore or stronger, if to non-overlapping addresses

TABLE 5-5 SPARC M5 Synchronization Requirements for Memory Reference Operations

First reference	Second reference	Synchronization Required
Block store	Load	MEMBAR #`StoreLoad` or #`Sync`
	Block load	MEMBAR #`StoreLoad` or #`Sync`
	Store	MEMBAR #`StoreStore` or stronger, if to non-overlapping addresses
	Block store	MEMBAR #`StoreStore` or stronger, if to non-overlapping addresses

5.6 Integer Multiply-Add

SPARC M5 provides nonprivilged unsigned integer multiply-add instructions, FPMADDX and FPMADDXHI. More details regarding these instructions can be found in the Oracle SPARC Architecture 2011 specification.

5.7 Compare and Branch

SPARC M5 provides nonprivilged compare-and-branch instructions, as follows:

Opcode	Operation	Test
	32-bit Compare and Branch Operations	
CWBNE	Compare and Branch if Not Equal	**not** Z
CWBE	Compare and Branch if Equal	Z
CWBG	Compare and Branch if Greater	**not** (Z **or** (N **xor** V))
CWBLE	Compare and Branch if Less or Equal	Z **or** (N **xor** V)
CWBGE	Compare and Branch if Greater or Equal	**not** (N **xor** V)
CWBL	Compare and Branch if Less	N **xor** V
CWBGU	Compare and Branch if Greater Unsigned	**not** (C **or** Z)
CWBLEU	Compare and Branch if Less or Equal Unsigned	C **or** Z
CWBCC	Compare and Branch if Carry Clear (Greater Than or Equal, Unsigned)	**not** C
CWBCS	Compare and Branch if Carry Set (Less Than, Unsigned)	C
CWBPOS	Compare and Branch if Positive	**not** N
CWBNEG	Compare and Branch if Negative	N
CWBVC	Compare and Branch if Overflow Clear	**not** V
CWBVS	Compare and Branch if Overflow Set	V
	64-bit Compare and Branch Operations	
CXBNE	Compare and Branch if Not Equal	**not** Z
CXBE	Compare and Branch if Equal	Z
CXBG	Compare and Branch if Greater	**not** (Z **or** (N **xor** V))
CXBLE	Compare and Branch if Less or Equal	Z **or** (N **xor** V)
CXBGE	Compare and Branch if Greater or Equal	**not** (N **xor** V)
CXBL	Compare and Branch if Less	N **xor** V
CXBGU	Compare and Branch if Greater Unsigned	**not** (C **or** Z)
CXBLEU	Compare and Branch if Less or Equal Unsigned	C **or** Z
CXBCC	Compare and Branch if Carry Clear (Greater Than or Equal, Unsigned)	**not** C
CXBCS	Compare and Branch if Carry Set (Less Than, Unsigned)	C
CXBPOS	Compare and Branch if Positive	**not** N
CXBNEG	Compare and Branch if Negative	N
CXBVC	Compare and Branch if Overflow Clear	**not** V
CXBVS	Compare and Branch if Overflow Set	V

† *synonym:* `cbnz{x}` ‡ *synonym:* `cbz{x}` ◊ *synonym:* `cbgeu{x}` ∇ *synonym:* `cblu{x}`

More details regarding the vompare-and-branch instructions can be found in the Oracle SPARC Architecture 2011 specification.

5.8 AES Operations (4 operand) **Crypto**

SPARC M5 provides nine nonprivilged, 4-operand instructions to support the AES cryptographic algorithm, as follows:

Instruction	Operation
AES_EROUND01	AES Encrypt columns 0&1
AES_EROUND23	AES Encrypt columns 2&3
AES_DROUND01	AES Decrypt columns 0&1
AES_DROUND23	AES Decrypt columns 2&3
AES_EROUND01_LAST	AES Encrypt columns 0&1 last round
AES_EROUND23_LAST	AES Encrypt columns 2&3 last round
AES_DROUND01_LAST	AES Decrypt columns 0&1 last round
AES_DROUND23_LAST	AES Decrypt columns 2&3 last round
AES_KEXPAND1	AES Key expansion with RCON

More details regarding these instructions can be found in the Oracle SPARC Architecture 2011 specification.

5.9 AES Operations (3 operand) **Crypto**

SPARC M5 provides two nonprivilged, 3-operand instructions to support the AES cryptographic algorithm, as follows:

Instruction	Operation
AES_KEXPAND0	AES Key expansion without RCON
AES_KEXPAND2	AES Key expansion without SBOX

More details regarding these instructions can be found in the Oracle SPARC Architecture 2011 specification.

5.10 DES Operations (4 operand) **Crypto**

SPARC M5 provides one nonprivilged, 4-operand instruction to support the DES cryptographic algorithm, as follows:

Instruction	Operation
DES_ROUND	two DES round operations

More details regarding this instruction can be found in the Oracle SPARC Architecture 2011 specification.

5.11 DES Operations (2 operand) Crypto

SPARC M5 provides three nonprivilged, 2-operand instructions to support the DES cryptographic algorithm, as follows:

Instruction	Operation
DES_IP	DES Initial Permutation
DES_IIP	DES Inverse Initial Permutation
DES_KEXPAND	DES Key Expand

More details regarding these instructions can be found in the Oracle SPARC Architecture 2011 specification.

5.12 Camellia Operations (4 operand) Crypto

SPARC M5 provides one nonprivilged, 4-operand instruction to support the Camellia cryptographic algorithm, as follows:

Instruction	Operation
CAMELLIA_F	Camellia F operation

More details regarding this instruction can be found in the Oracle SPARC Architecture 2011 specification.

5.13 Camellia Operations (3 Operand) Crypto

SPARC M5 provides two nonprivilged, 3-operand instructions to support the Camellia cryptographic algorithm, as follows:

Instruction	Operation
CAMELLIA_FL	Camellia FL operation
CAMELLIA_FLI	Camellia FLI operation

More details regarding these instructions can be found in the Oracle SPARC Architecture 2011 specification.

5.14 Hash Operations Crypto

SPARC M5 provides four nonprivilged instructions to support cryptographic digest (hash) algorithms, as follows:

Instruction	Operation
MD5	MD5 operation on a single block
SHA1	SHA1 operation on a single block
SHA256	SHA256 operation on a single block
SHA512	SHA512 operation on a single block

More details regarding the hash instructions can be found in the Oracle SPARC Architecture 2011 specification.

5.15 CRC32C Operation (3 operand) Crypto

SPARC M5 provides one nonprivilged instruction to support the CRC32c checksum operation, as follows:

Instruction	Operation
CRC32C	two CRC32c operations

More details regarding the CRC32c instructions can be found in the Oracle SPARC Architecture 2011 specification.

5.16 MPMUL Crypto

SPARC M5 provides a nonprivilged instruction to support the multiple-precision multiplication operation, as follows:

Instruction	Operation
MPMUL	Multiple Precision Multiply

More details regarding this instruction can be found in the Oracle SPARC Architecture 2011 specification.

5.17 MONTMUL Crypto

SPARC M5 provides a nonprivilged instruction to support the Montgomery multiplication operation, as follows:

Instruction	Operation
MONTMUL	Montgomery Multiplication

More details regarding this instruction can be found in the Oracle SPARC Architecture 2011 specification.

5.18 MONTSQR Crypto

SPARC M5 provides a nonprivilged instruction to support the Montgomery squaring operation, as follows:

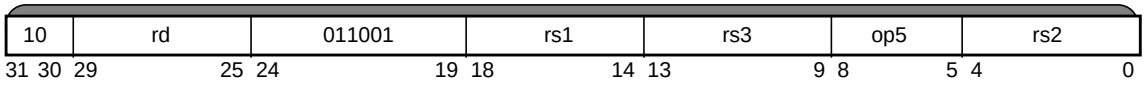
Instruction	Operation
MONTSQR	Montgomery Squaring

More details regarding this instruction can be found in the Oracle SPARC Architecture 2011 specification.

5.19 Kasumi Operations (4 operand) **Crypto**

The Kasumi instructions are new and not expected to be implemented on all Oracle SPARC Architecture implementations. Therefore, they should only be used in platform-specific dynamically-linked libraries or in software created by a runtime code generator that is aware of the specific virtual processor implementation on which it is executing.

Instruction	op5	Operation	Assembly Language Syntax	Class
KASUMI_FL_XOR	1010	Kasumi FL followed by xor	`kasumi_fl_xor` $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**N1**
KASUMI_FI_XOR	1011	Kasumi FI followed by xor	`kasumi_fi_xor` $freg_{rs1}$, $freg_{rs2}$, $freg_{rs3}$, $freg_{rd}$	**N1**



Description Kasumi is a block cipher that produces a 64-bit output from a 64-bit input under the control of a 128-bit key. The Kasumi cipher has eight rounds. Each round consists of an FL function, an FO function, and an XOR operation. Each FO is composed of three FI functions with xors above and below each FI. Odd rounds apply the functions FL, FO, XOR whereas even rounds order the functions FO, FL and XOR. A number of temporary variables are used in the functional descriptions below. For example, data_fl is the result of applying the FL operation to the rs1 data using the rs2 key. The Kasumi instructions operate on 64-bit floating-point registers.

CFR.kasumi must be set; otherwise a *compatibility_feature* trap results.

KASUMI_FL_XOR:

data_fl{31:0} ← kasumi FL(data=F_D[rs1]{31:0} , key=F_D[rs2]{31:0});
F_D[rd]{31:0} ← data_fl[31:0} **xor** F_D[rs3]{31:0};
F_D[rd]{63:32} ← $0000\ 0000_{16}$

where F_D[rs2]{63:0} = (32 unused **::** KL(i1) **::** KL(i2))

KASUMI_FI_XOR:

data_x1{15:0} ← F_D[rs1]{31:16} **xor** F_D[rs2]{63:48};
data_fi{15:0} ← kasumi FI(data_x1[15:0}, F_D[rs2]{47:32});
data_x2{15:0} ← data_fi[15:0} **xor** F_D[rs1]{15:0};
data_x2{31:16} ← F_D[rs1]{15:0};
F_D[rd]{31:0} ← data_x2[31:0} **xor** F_D[rs3]{31:0};
F_D[rd]{63:32} ← $0000\ 0000_{16}$;

where F_D[rs2]{63:0} = (KO(i3) **::** KI(i3) **::** 32 unused)

Programming Note

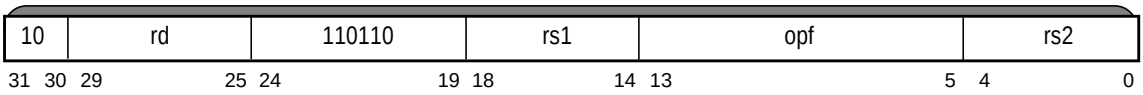
The Kasumi instructions are components of the overall Kasumi algorithm. To perform an encryption or decryption, software must first expand the key. Key expansion is done only once per session key. The expanded keys are then applied to all blocks for that session. The following example has the expanded keys in $F_D[0]$ thru $F_D[46]$. The initial 64-bit data is split into left and right halves and loaded into the lower half of $F_D[52]$ and $F_D[54]$ respectively. $F_D[56]$ must be initialized to 0. For each block, the following instruction sequence can be applied :

```
kasumi_fl_xor   %f52, %f0 , %f56, %f58       !# Round 1
kasumi_fi_fi    %f58, %f2 ,        %f58
kasumi_fi_xor   %f58, %f4 , %f54, %f54
kasumi_fi_fi    %f54, %f6 ,       %f58       !# Round 2
kasumi_fi_xor   %f58, %f8 , %f56, %f58
kasumi_fl_xor   %f58, %f10, %f52, %f52
kasumi_fl_xor   %f52, %f12, %f56, %f58       !# Round 3
kasumi_fi_fi    %f58, %f14,        %f58
kasumi_fi_xor   %f58, %f16, %f54, %f54
kasumi_fi_fi    %f54, %f18,       %f58       !# Round 4
kasumi_fi_xor   %f58, %f20, %f56, %f58
kasumi_fl_xor   %f58, %f22, %f52, %f52
kasumi_fl_xor   %f52, %f24, %f56, %f58       !# Round 5
kasumi_fi_fi    %f58, %f26,        %f58
kasumi_fi_xor   %f58, %f28, %f54, %f54
kasumi_fi_fi    %f54, %f30,       %f58       !# Round 6
kasumi_fi_xor   %f58, %f32, %f56, %f58
kasumi_fl_xor   %f58, %f34, %f52, %f52
kasumi_fl_xor   %f52, %f36, %f56, %f58       !# Round 7
kasumi_fi_fi    %f58, %f38,        %f58
kasumi_fi_xor   %f58, %f40, %f54, %f54
kasumi_fi_fi    %f54, %f42,       %f58       !# Round 8
kasumi_fi_xor   %f58, %f44, %f56, %f58
kasumi_fl_xor   %f58, %f46, %f52, %f52
```

Exceptions *fp_disabled*

5.20 Kasumi Operations (3 operand) **Crypto**

Instruction	opf	Operation	Assembly Language Syntax	Class
KASUMI_FI_FI	1 0011 1000	two Kasumi FI operations	`kasumi_fi_fi` *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$	**N1**



Description The KASUMI_FI_FI instruction is one component of the Kasumi cipher. It operates on 64-bit floatin-point registers.

CFR.kasumi must be set; otherwise a *compatibility_feature* trap results.

KASUMI_FI_FI:

data_x1{31:16} ← F_D[rs1]{31:16} **xor** F_D[rs2]{63:48};
data_x1{15:0} ← F_D[rs1]{15:0} **xor** F_D[rs2]{31:16};
data_fi{31:16} ← kasumi FI (data_x1{31:16}, F_D[rs2]{47:32});
data_fi{15:0} ← kasumi FI (data_x1{15:0}, F_D[rs2]{15:0});
F_D[rd]{31:16} ← data_fi{31:16} **xor** F_D[rs1]{15:0};
F_D[rd]{15:0} ← data_fi{31:16} **xor** F_D[rs1]{15:0} **xor** data_fi{15:0};
F_D[rd]{63:32} ← $0000\ 0000_{16}$;

where F_D[rs2]{63:0} = (KO(i1) **::** KI(i1) **::** KO(i2) **::** KI(i2))

Exceptions *fp_disabled*

CHAPTER 6

Traps

6.1 Trap Levels

Only SPARC M5 specific behavior is described in this chapter; refer to Oracle SPARC Architecture 2011 for more detail on trap handling.

Each virtual processor supports two trap levels (*MAXPTL* = 2).

6.2 Trap Behavior

TABLE 6-1 specifies the codes used in the tables below.

TABLE 6-1 Table Codes

Code	Meaning
H	Trap is taken in Hyperprivileged mode
P	Trap is taken via the Privileged trap table, in Privileged mode (PSTATE.priv = 1)
-x-	Not possible. Hardware cannot generate this trap in the indicated running mode. For example, all privileged instructions can be executed in privileged mode, therefore a *privileged_opcode* trap cannot occur in privileged mode.
—	This trap can only legitimately be generated by hyperprivileged software, not by the CPU hardware. So, for the purposes of sun4v, the trap vector has to be correct, but for a hardware CPU implementation these trap types are not generated by the hardware, therefore the resultant running mode is irrelevant.

Programming Note – TABLE 6-2 only contains those traps in which SPARC M5 differs from Oracle SPARC Architecture 2011; not all traps are listed. Refer to Oracle SPARC Architecture 2011 for more detail.

CHAPTER 7

Interrupt Handling

The chapter describes the hardware interrupt delivery mechanism for the SPARC M5 chip.

Hyperprivileged code notifies privileged code about *interrupt_vector*, *sw_recoverable_error*, and *hw_corrected_error* traps (and precise error traps) through the *cpu_mondo*, *dev_mondo*, and *resumable_error* traps as described in *Interrupt Queue Registers* on page 52. Software interrupts are delivered to each virtual processor using the *interrupt_level_n* traps. Software interrupts are described in the Oracle SPARC Architecture 2011 specification.

Details of I/O interrupt processing are given in Chapter 22, *PCI-Express*. A high-level summary is given here. Any event generated by an I/O device which requires a CPU thread to be interrupted is passed through the DMU sitting at the root of the tree containing that I/O device. Note that all clusters hosted by a DMU (PCIe root ports, NIU Ethernet ports, and third-party IP) have a PCI-Express programming model. The implication is that an event requiring an interrupt be sent arrives at the DMU in a PCIe compatible way: as a PCIe Messages, MSI (Message Signalled Interrupt), MSI-X (expanded MSI), or INTx (legacy wire-based interrupt used by older PCIe or PCI devices). The DMU manages a number of Event Queues (EQs) in main memory. The DMU keeps track of tail pointers, which it updates when writing the EQs, and head pointers, which software updates when reading the EQs. When a message, MSI/X, or INTx arrives at the DMU it maps from the message, MSI/X, or INTx ID to a specific Event Queue. The DMU contains a set of CSRs to do this mapping. The DMU writes an entry to the given EQ which contains information useful to software about the event. If the EQ transitions from empty to non-empty as a result of the new entry written, the DMU sends a mondo Interrupt Request (IREQ) transaction over the IOX to the local NCU. The IREQ contains the destination thread for the interrupt, which may be either a local thread or a thread on a remote SPARC M5 node.[1] The NCU maintains a state machine for each of the local threads, keeping track of whether a mondo is already pending to each respective thread. If the IREQ specifies a local thread and that thread already has a mondo pending, the IREQ is dropped because a thread can handle only one mondo outstanding and Hypervisor will eventually see the new EQ entry during the course of its processing the prior mondo. If the IREQ specifies a local thread and that thread does not have a mondo pending, the NCU generates a mondo to the given thread. Since no useful data is sent, mondo data is no longer available in SPARC M5 as is was in some earlier UltraSPARC processors before SPARC T3. If the IREQ specifies a remote thread, the NCU passes it back to the local IOX fabric, and it is routed to the destination SPARC M5 node where the NCU there receives it and processes it as described above.

A mondo is processed by Hypervisor, which reads EQ or EQs mapped to the given mondo it has received, takes the appropriate actions, calling device drivers as necessary to clear interrupt status in the originating device, and updates the header pointer(s) for the given EQ or EQs. Note that multiple EQs managed by a given DMU may map to the same thread, and multiple EQs managed by different DMUs (even DMUs on different SPARC M5 nodes) may map to the same thread. The software flow for mondo handling is covered in more detail in the PCIe chapter.

1. SPARC T3 and SPARC T4 did not support remote interrupts, meaning that a mondo could be directed only to a local thread. Because SPARC M5 supports core off-lining for power reduction, it is necessary to support directing interrupts to remote threads where cores are powered. The mapping of EQ to threads can be changed dynamically through programming DMU CSRs.

7.1 Interrupt Flow

7.1.1 Sources

CPU cross-call interrupts can be generated by writing the Interrupt Vector Dispatch register described in *Interrupt Vector Dispatch Register* on page 145. Dispatching inter-CPU interrupts is described in *Dispatching* on page 139.

JTAG TAP interrupts can be generated by writing the NCU Interrupt Vector/Trap Dispatch Register described in *CPU Interrupt Registers* on page 52, via the JTAG port.

SSI interrupts (device ID = 2) are caused by an assertion (edge trigger) on the EXT_INT_L pin.

I/O interrupts arrive to the CPU thread as mondos, as described in the previous section.

7.2 CPU Interrupt Registers

7.2.1 Interrupt Queue Registers

Each virtual processor has eight ASI_QUEUE registers at ASI = 25_{16}, VA{63:0} = $3C0_{16}$-$3F8_{16}$ that are used for communicating interrupts to the operating system. These registers contain the head and tail pointers for four supervisor interrupt queues: *cpu_mondo*, *dev_mondo*, *resumable_error*, *nonresumable_error*. The tail registers are read-only by supervisor, and read/write by hypervisor. Writes to the tail registers by the supervisor generate a *DAE_invalid_ASI* trap. The head registers are read/write by both supervisor and hypervisor.

Whenever the CPU_MONDO_HEAD register does not equal the CPU_MONDO_TAIL register, a *cpu_mondo* trap is generated. Whenever the DEV_MONDO_HEAD register does not equal the DEV_MONDO_TAIL register, a *dev_mondo* trap is generated. Whenever the RESUMABLE_ERROR_HEAD register does not equal the RESUMABLE_ERROR_TAIL register, a *resumable_error* trap is generated. Unlike the other queue register pairs, the *nonresumable_error* trap is *not* automatically generated whenever the NONRESUMABLE_ERROR_HEAD register does not equal the NONRESUMABLE_ERROR_TAIL register; instead, the hypervisor will need to generate the *nonresumable_error* trap.TABLE 7-1 through TABLE 7-8 define the format of the eight ASI_QUEUE registers.

TABLE 7-1 CPU Mondo Head Pointer – ASI_QUEUE_CPU_MONDO_HEAD (ASI 25_{16}, VA $3C0_{16}$)

Bit	Field	Initial Value	Access	Description
63:31	—	0	RO	*Reserved*
30:6	head	X	RW	Head pointer for CPU mondo interrupt queue.
5:0	—	0	RO	*Reserved*

TABLE 7-2 CPU Mondo Tail Pointer – ASI_QUEUE_CPU_MONDO_TAIL (ASI 25_{16}, VA $3C8_{16}$)

Bit	Field	Initial Value	Access	Description
63:31	—	0	RO	*Reserved*
30:6	tail	X	RW	Tail pointer for CPU mondo interrupt queue.
5:0	—	0	RO	*Reserved*

TABLE 7-3 Device Mondo Head Pointer – ASI_QUEUE_DEV_MONDO_HEAD (ASI 25_{16}, VA $3D0_{16}$)

Bit	Field	Initial Value	Access	Description
63:31	—	0	RO	*Reserved*
30:6	head	X	RW	Head pointer for device mondo interrupt queue.
5:0	—	0	RO	*Reserved*

TABLE 7-4 Device Mondo Tail Pointer – ASI_QUEUE_DEV_MONDO_TAIL (ASI 25_{16}, VA $3D8_{16}$)

Bit	Field	Initial Value	Access	Description
63:31	—	0	RO	*Reserved*
30:6	tail	X	RW	Tail pointer for device mondo interrupt queue.
5:0	—	0	RO	*Reserved*

TABLE 7-5 Resumable Error Head Pointer – ASI_QUEUE_RESUMABLE_HEAD (ASI 25_{16}, VA $3E0_{16}$)

Bit	Field	Initial Value	Access	Description
63:31	—	0	RO	Reserved.
30:6	head	X	RW	Head pointer for resumable error queue.
5:0	—	0	RO	*Reserved*

TABLE 7-6 Resumable Error Tail Pointer – ASI_QUEUE_RESUMABLE_TAIL (ASI 25_{16}, VA $3E8_{16}$)

Bit	Field	Initial Value	Access	Description
63:31	—	0	RO	*Reserved*
30:6	tail	X	RW	Tail pointer for resumable error queue.
5:0	—	0	RO	*Reserved*

TABLE 7-7 Nonresumable Error Head Pointer – ASI_QUEUE_NONRESUMABLE_HEAD (ASI 25_{16}, VA $3F0_{16}$)

Bit	Field	Initial Value	Access	Description
63:31	—	0	RO	*Reserved*
30:6	head	X	RW	Head pointer for nonresumable error queue.
5:0	—	0	RO	*Reserved*

TABLE 7-8 Nonresumable Error Tail Pointer – ASI_QUEUE_NONRESUMABLE_TAIL (ASI 25_{16}, VA $3F8_{16}$)

Bit	Field	Initial Value	Access	Description
63:31	—	0	RO	*Reserved*
30:6	tail	X	RW	Tail pointer for nonresumable error queue.
5:0	—	0	RO	*Reserved*

CHAPTER 8

Memory Models

SPARC V9 defines the semantics of memory operations for three memory models. From strongest to weakest, they are Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory Order (RMO). The differences in these models lie in the freedom an implementation is allowed in order to obtain higher performance during program execution. The purpose of the memory models is to specify any constraints placed on the ordering of memory operations in uniprocessor and shared-memory multiprocessor environments. SPARC M5 supports only TSO, with the exception that certain ASI accesses (such as block loads and stores) may operate under RMO.

Although a program written for a weaker memory model potentially benefits from higher execution rates, it may require explicit memory synchronization instructions to function correctly if data is shared. MEMBAR is a SPARC V9 memory synchronization primitive that enables a programmer to control explicitly the ordering in a sequence of memory operations. Processor consistency is guaranteed in all memory models.

The current memory model is indicated in the PSTATE.mm field. It is unaffected by normal traps. SPARC M5 ignores the value set in this field and always operates under TSO.

A memory location is identified by an 8-bit address space identifier (ASI) and a 64-bit virtual address. The 8-bit ASI may be obtained from a ASI register or included in a memory access instruction. The ASI is used to distinguish between and provide an attribute for different 64-bit address spaces. For example, the ASI is used by the SPARC M5 MMU to control access to implementation-dependent control and data registers and for access protection. Attempts by nonprivileged software (PSTATE.priv = 0) to access restricted ASIs (ASI{7} = 0) cause a *privileged_action* trap.

Real memory spaces can be accessed without side effects. For example, a read from real memory space returns the information most recently written. In addition, an access to real memory space does not result in program-visible side effects.

8.1 Supported Memory Models

The following sections contain brief descriptions of the two memory models supported by SPARC M5. These definitions are for general illustration. Detailed definitions of these models can be found in *The SPARC Architecture Manual-Version 9*. The definitions in the following sections apply to system behavior as seen by the programmer.

Notes

Stores to SPARC M5 internal ASIs, block loads, and block stores and block initializing stores are outside the memory model; that is, they need MEMBARs to control ordering.

Atomic load-stores are treated as both a load and a store and can only be applied to cacheable address spaces.

8.1.1 TSO

SPARC M5 implements the following programmer-visible properties in Total Store Order (TSO) mode:

- Loads are processed in program order; that is, there is an implicit MEMBAR #`LoadLoad` between them.
- Loads may bypass earlier stores. Any such load that bypasses such earlier stores must check (snoop) the store buffer for the most recent store to that address. A MEMBAR #`Lookaside` is not needed between a store and a subsequent load at the same noncacheable address.
- A MEMBAR #`StoreLoad` must be used to prevent a load from bypassing a prior store if Strong Sequential Order is desired.
- Stores are processed in program order.
- Stores cannot bypass earlier loads.
- Accesses to I/O space are all strongly ordered with respect to each other.
- An L2 cache update is delayed on a store hit until all outstanding stores reach global visibility. For example, a cacheable store following a noncacheable store is not globally visible until the noncacheable store has reached global visibility; there is an implicit MEMBAR #`MemIssue` between them.

8.1.2 RMO

SPARC M5 implements the following programmer-visible properties for special ASI accesses that operate under Relaxed Memory Order (RMO) mode:

- There is no implicit order between any two memory references, either cacheable or noncacheable, except that noncacheable accesses to I/O space) are all strongly ordered with respect to each other.
- A MEMBAR must be used between cacheable memory references if stronger order is desired. A MEMBAR #`MemIssue` is needed for ordering of cacheable after noncacheable accesses.

CHAPTER 9

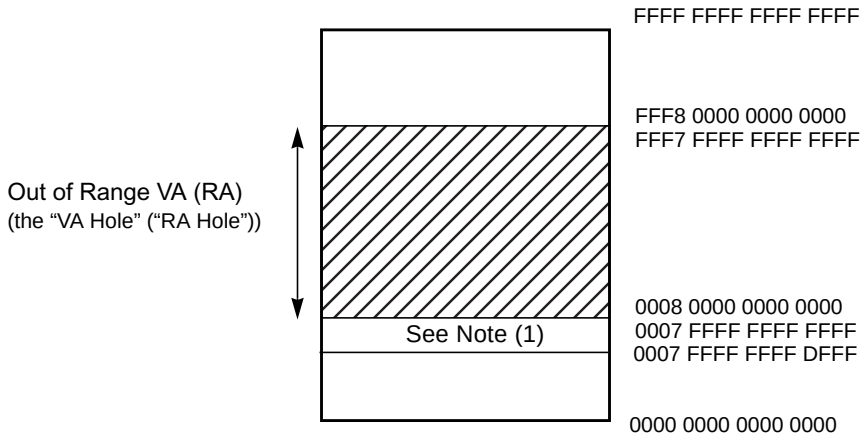
Address Spaces and ASIs

9.1 Address Spaces

SPARC M5 supports a 52-bit virtual address space.

9.1.1 52-bit Virtual and Real Address Spaces

SPARC M5 supports a 52-bit subset of the full 64-bit virtual and real address spaces. Although the full 64 bits are generated and stored in integer registers, legal addresses are restricted to two equal halves at the extreme lower and upper portions of the full virtual (real) address space. Virtual (real) addresses between 0008 0000 0000 0000_{16} and FFF7 FFFF FFFF $FFFF_{16}$ inclusive lie within a "VA hole" ("RA hole"), are termed "out-of-range"[1], and are illegal. Prior UltraSPARC implementations introduced the additional restriction on software to not use pages within 4 Gbytes of the VA (RA) hole as instruction pages to avoid problems with prefetching into the VA (RA) hole. SPARC M5 implements a hardware check for instruction fetching near the VA (RA) hole and generates a trap when instructions are executed from a location in the address range 0007 FFFF FFFF $FFE0_{16}$ to 0007 FFFF FFFF $FFFF_{16}$, inclusive. However, even though SPARC M5 provides this hardware checking, it is still recommended that software should *not* use the 8-Kbyte page before the VA (RA) hole for instructions. Address translation and MMU related descriptions can be found in *Translation* on page 89.



Note (1): Use of this region restricted to data only.

FIGURE 9-1 SPARC M5's 52-bit Virtual and Real Address Spaces, With Hole

Throughout this document, when virtual (real) address fields are specified as 64-bit quantities, they are assumed to be sign-extended based on VA{51} (RA{51}).

1. Another way to view an out-of-range address is as any address where bits {63:52} are not all equal to bit {51}.

A number of state registers are affected by the reduced virtual and real address spaces. The PC register is 52 bits, sign-extended to 64-bits on read accesses. The TBA, TPC, and TNPC registers are 52-bits and their values are *not* sign-extended when read. No checks are done when these registers are written by software. It is the responsibility of privileged software to properly update these registers.

An out-of-range virtual (real) address during an instruction access, caused by execution into the VA (RA) hole or into 0007 FFFF FFFF $FFE0_{16}$ to 0007 FFFF FFFF $FFFF_{16}$ inclusive, results in a trap if PSTATE.am = 0.

If the target virtual (real) address of a JMPL, RETURN, branch, or CALL instruction is an out-of-range address and PSTATE.am = 0, a trap is generated with TPC equal to the address of the JMPL, RETURN, branch, or CALL instruction.

An out-of-range virtual (real) address during a data access results in a trap if PSTATE.am = 0.

9.2 Alternate Address Spaces

TABLE 9-4 summarizes the ASI usage in SPARC M5. The Section/Page column contains a reference to the detailed explanation of the ASI (the page number refers to this chapter). For internal ASIs, the legal VAs are listed (or the field contains "Any" if all VAs are legal). Only bits 51:0 are checked when determining the legal VA range. An access outside the legal VA range generates a *DAE_invalid_asi* trap.

Notes

All internal, nontranslating ASIs in SPARC M5 can only be accessed using LDXA and STXA.

ASIs 80_{16}–FF_{16} are unrestricted (access allowed in all modes -- nonprivileged, privileged). ASIs 00_{16}–$2F_{16}$ are restricted to privileged and hyperprivileged modes.

TABLE 9-1 SPARC M5 ASI Usage *(1 of 6)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
00_{16}–03_{16}			Any	—	*DAE_invalid_asi*	
04_{16}	ASI_NUCLEUS	RW	Any	—	Implicit address space, nucleus context, TL > 0	(See OSA 2011)
05_{16}–$0B_{16}$			Any	—	*DAE_invalid_asi*	
$0C_{16}$	ASI NUCLEUS_LITTLE	RW	Any	—	Implicit address space, nucleus context, TL > 0 (LE)	(See OSA 2011)
$0D_{16}$–$0F_{16}$			Any	—	*DAE_invalid_asi*	
10_{16}	ASI_AS_IF_USER_PRIMARY	RW	Any	—	Primary address space, user privilege	(See OSA 2011)
11_{16}	ASI_AS_IF_USER_SECONDARY	RW	Any	—	Secondary address space, user privilege	(See OSA 2011)
12_{16}–13_{16}			Any	—	*DAE_invalid_asi*	
14_{16}	ASI_REAL	RW	Any	—	Real address (normally used as cacheable)	Section 9.2.1

TABLE 9-1 SPARC M5 ASI Usage *(2 of 6)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
15_{16}	ASI_REAL_IO	RW	Any	—	Real address (normally used as noncacheable, with side effect)	Section 9.2.1
16_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY	RW	Any	—	64-byte block load/store, primary address space, user privilege	5.5
17_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY	RW	Any	—	64-byte block load/store, secondary address space, user privilege	5.5
18_{16}	ASI_AS_IF_USER_PRIMARY_LITTLE	RW	Any	—	Primary address space, user privilege (LE)	(See OSA 2011)
19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE	RW	Any	—	Secondary address space, user privilege (LE)	(See OSA 2011)
$1A_{16}$–$1B_{16}$			Any	—	*DAE_invalid_asi*	
$1C_{16}$	ASI_REAL_LITTLE	RW	Any	—	Real address (normally used as cacheable) (LE)	Section 9.2.1
$1D_{16}$	ASI_REAL_IO_LITTLE	RW	Any	—	Real address (normally used as noncacheable, with side effect) (LE)	Section 9.2.1
$1E_{16}$	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE	RW	Any	—	64-byte block load/store, primary address space, user privilege (LE)	5.5
$1F_{16}$	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE	RW	Any	—	64-byte block load/store, secondary address space, user privilege (LE)	5.5
20_{16}	ASI_SCRATCHPAD	RW	0_{16}–18_{16}	Y	Scratchpad registers	Section 9.2.2
20_{16}	ASI_SCRATCHPAD		20_{16}–28_{16}	—	*DAE_invalid_asi*	
20_{16}	ASI_SCRATCHPAD	RW	30_{16}–38_{16}	Y	Scratchpad registers	Section 9.2.2
21_{16}	ASI_MMU	RW	8_{16}	Y	I/DMMU Primary Context register 0	13.7.2
21_{16}	ASI_MMU	RW	10_{16}	Y	DMMU Secondary Context register 0	13.7.2
21_{16}	ASI_MMU	RW	108_{16}	Y	I/DMMU Primary Context register 1	13.7.2
21_{16}	ASI_MMU	RW	110_{16}	Y	DMMU Secondary Context register 1	13.7.2
22_{16}	ASI_TWINX_AIUP, ASI_STBI_AIUP	RW	Any	—	Load: 128-bit atomic load twin extended word, primary address space, user privilege Store: Block initializing store, primary address space, user privilege	5.7.4
23_{16}	ASI_TWINX_AIUS, ASI_STBI_AIUS	RW	Any	—	Load: 128-bit atomic load twin extended word, secondary address space, user privilege Store: Block initializing store	(See OSA 2011)
24_{16}			Any	—	*DAE_invalid_asi*	

TABLE 9-1 SPARC M5 ASI Usage *(3 of 6)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
25_{16}	ASI_QUEUE	RW	$3C0_{16}$	Y	CPU Mondo Queue head pointer	7.2.1
25_{16}	ASI_QUEUE	RW (hyperpriv) RO (priv)	3C8	Y	CPU Mondo Queue tail pointer	7.2.1
25_{16}	ASI_QUEUE	RW	$3D0_{16}$	Y	Device Mondo Queue head pointer	7.2.1
25_{16}	ASI_QUEUE	RW (hyperpriv) RO (priv)	$3D8_{16}$	Y	Device Mondo Queue tail pointer	7.2.1
25_{16}	ASI_QUEUE	RW	$3E0_{16}$	Y	Resumable Error Queue head pointer	7.2.1
25_{16}	ASI_QUEUE	RW (hyperpriv) RO (priv)	$3E8_{16}$	Y	Resumable Error Queue tail pointer	7.2.1
25_{16}	ASI_QUEUE	RW	$3F0_{16}$	Y	Nonresumable Error Queue head pointer	7.2.1
25_{16}	ASI_QUEUE	RW (hyper-priv) RO (priv)	$3F8_{16}$	Y	Nonresumable Error Queue tail pointer	7.2.1
26_{16}	ASI_TWINX_REAL, ASI_STBI_REAL	RW	Any	—	Load:128-bit atomic LDDA, real address Store: Block initializing store, real address	(See OSA 2011)
27_{16}	ASI_TWINX_NUCLEUS, ASI_STBI_N	RW	Any	—	Load: 128-bit atomic load twin extended word from nucleus context Store: Block initializing store from nucleus context	(See OSA 2011)
28_{16}–29_{16}			Any	—	*DAE_invalid_asi*	
$2A_{16}$	ASI_TWINX_AIUPL, ASI_STBI_AIUPL	RW	Any	—	Load: 128-bit atomic load twin extended word, primary address space, user privilege, little endian Store: Block initializing store, primary address space, user privilege, little endian	(See OSA 2011)
$2B_{16}$	ASI_TWINX_AIUSL, ASI_STBI_AIUSL	RW	Any	—	Load: 128-bit atomic load twin extended word, secondary address space, user privilege, little endian Store: Block initializing store, secondary address space, user privilege, little endian	((See OSA 2011)
$2C_{16}$			Any	—	*DAE_invalid_asi*	
$2D_{16}$			Any	—	*DAE_invalid_asi*	
$2E_{16}$	ASI_TWINX_REAL_LITTLE, ASI_STBI_REAL_LITTLE	RW	Any	—	Load: 128-bit atomic LDDA, real address (LE) Store: Block initializing store, real address (LE)	(See OSA 2011)

TABLE 9-1 SPARC M5 ASI Usage *(4 of 6)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
$2F_{16}$	ASI_TWINX_NL, ASI_STBI_NL	RW	Any	—	Load: 128-bit atomic load twin extended word from nucleus context, little endian Store: Block initializing store from nucleus context, little endian	(See OSA 2011)
$4C_{16}$	ASI_CHDER	RW	20_{16}	N	Core Hang Detection Enable Register	Ch17
$4C_{16}$	ASI_DO_STATUS	RO	38_{16}	Y	Disable Overlap Status Register	Ch 21
$4E_{16}$	ASI_SPARC_HW_CONFIG	RW	8_{16}	N	SPARC hardware configuration register	20.1
80_{16}	ASI_PRIMARY	RW	Any	—	Implicit primary address space	(See OSA 2011)
81_{16}	ASI_SECONDARY	RW	Any	—	Implicit secondary address space	(See OSA 2011)
82_{16}	ASI_PRIMARY_NO_FAULT	RO	Any	—	Primary address space, no fault	(See OSA 2011)
83_{16}	ASI_SECONDARY_NO_FAULT	RO	Any	—	Secondary address space, no fault	(See OSA 2011)
84_{16}–87_{16}			Any	—	*DAE_invalid_asi*	
88_{16}	ASI_PRIMARY_LITTLE	RW	Any	—	Implicit primary address space (LE)	(See OSA 2011)
89_{16}	ASI_SECONDARY_LITTLE	RW	Any	—	Implicit secondary address space (LE)	((See OSA 2011)
$8A_{16}$	ASI_PRIMARY_NO_FAULT_LITTLE	RO	Any	—	Primary address space, no fault (LE)	(See OSA 2011)
$8B_{16}$	ASI_SECONDARY_NO_FAULT_LITTLE	RO	Any	—	Secondary address space, no fault (LE)	(See OSA 2011)
$8C_{16}$–AF_{16}			Any	—	*DAE_invalid_asi*	
$B0_{16}$	ASI_PIC	RW	0_{16}		Performance Instrumentation Counter 0	10.3
$B0_{16}$	ASI_PIC	RW	8_{16}		Performance Instrumentation Counter 1	10.3
$B0_{16}$	ASI_PIC	RW	10_{16}		Performance Instrumentation Counter 2	10.3
$B0_{16}$	ASI_PIC	RW	18_{16}		Performance Instrumentation Counter 3	10.3
$B1_{16}$–BF_{16}			Any	—	*DAE_invalid_asi*	
$C0_{16}$	ASI_PST8_P	WO	Any	—	Eight 8-bit conditional stores, primary address	(See OSA 2011)
$C1_{16}$	ASI_PST8_S	WO	Any	—	Eight 8-bit conditional stores, secondary address	(See OSA 2011)
$C2_{16}$	ASI_PST16_P	WO	Any	—	Four 16-bit conditional stores, primary address	(See OSA 2011)
$C3_{16}$	ASI_PST16_S	WO	Any	—	Four 16-bit conditional stores, secondary address	(See OSA 2011)

TABLE 9-1 SPARC M5 ASI Usage *(5 of 6)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
$C4_{16}$	ASI_PST32_P	WO	Any	—	Two 32-bit conditional stores, primary address	(See OSA 2011)
$C5_{16}$	ASI_PST32_S	WO	Any	—	Two 32-bit conditional stores, secondary address	(See OSA 2011)
$C6_{16}$–$C7_{16}$			Any	—	*DAE_invalid_asi*	
$C8_{16}$	ASI_PST8_PL	WO	Any	—	Eight 8-bit conditional stores, primary address, little endian	((See OSA 2011)
$C9_{16}$	ASI_PST8_SL	WO	Any	—	Eight 8-bit conditional stores, secondary address, little endian	(See OSA 2011)
CA_{16}	ASI_PST16_PL	WO	Any	—	Four 16-bit conditional stores, primary address, little endian	(See OSA 2011)
CB_{16}	ASI_PST16_SL	WO	Any	—	Four 16-bit conditional stores, secondary address, little endian	(See OSA 2011)
CC_{16}	ASI_PST32_PL	WO	Any	—	Two 32-bit conditional stores, primary address, little endian	(See OSA 2011)
CD_{16}	ASI_PST32_SL	WO	Any	—	Two 32-bit conditional stores, secondary address, little endian	(See OSA 2011)
CE_{16}–CF_{16}			Any	—	*DAE_invalid_asi*	
$D0_{16}$	ASI_FL8_P	RW	Any	—	8-bit load/store, primary address	(See OSA 2011)
$D1_{16}$	ASI_FL8_S	RW	Any	—	8-bit load/store, secondary address	(See OSA 2011)
$D2_{16}$	ASI_FL16_P	RW	Any	—	16-bit load/store, primary address	(See OSA 2011)
$D3_{16}$	ASI_FL16_S	RW	Any	—	16-bit load/store, secondary address	(See UA 2007)
$D4_{16}$–$D7_{16}$			Any	—	*DAE_invalid_asi*	
$D8_{16}$	ASI_FL8_PL	RW	Any	—	8-bit load/store, primary address, little endian	(See OSA 2011)
$D9_{16}$	ASI_FL8_SL	RW	Any	—	8-bit load/store, secondary address, little endian	(See OSA 2011)
DA_{16}	ASI_FL16_PL	RW	Any	—	16-bit load/store, primary address, little endian	(See OSA 2011)
DB_{16}	ASI_FL16_SL	RW	Any	—	16-bit load/store, secondary address, little endian	(See OSA 2011)
DC_{16}–DF_{16}			Any	—	*DAE_invalid_asi*	
$E0_{16}$	ASI_BLK_COMMIT_PRIMARY	RW	Any	—	64-byte block commit store, primary address	5.5
$E1_{16}$	ASI_BLK_COMMIT_SECONDARY	RW	Any	—	64-byte block commit store, secondary address	5.5

TABLE 9-1 SPARC M5 ASI Usage *(6 of 6)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
$E2_{16}$	ASI_TWINX_P, ASI_STBI_P	RW	Any	—	Load: 128-bit atomic load twin extended word, primary address space Store: Block initializing store, primary address space	(See OSA 2011)
$E3_{16}$	ASI_TWINX_S, ASI_STBI_S	RW	Any	—	Load: 128-bit atomic load twin extended word, secondary address space Store: Block initializing store, secondary address space	(See OSA 2011)
$E4_{16}$–$E9_{16}$			Any	—	*DAE_invalid_asi*	
EA_{16}	ASI_TWINX_PL, ASI_STBI_PL	RW	Any	—	Load: 128-bit atomic load twin extended word, primary address space, little endian Store: Block initializing store, primary address space, little endian	(See OSA 2011)
EB_{16}	ASI_TWINX_PL, ASI_STBI_PL	RW	Any	—	Load: 128-bit atomic load twin extended word, secondary address space, little endian Store: Block initializing store, secondary address space, little endian	(See OSA 2011)
EC_{16}–EF_{16}			Any	—	*DAE_invalid_asi*	
$F0_{16}$	ASI_BLK_P	RW	Any	—	64-byte block load/store, primary address	5.5
$F1_{16}$	ASI_BLK_S	RW	Any	—	64-byte block load/store, secondary address	5.5
$F2_{16}$	ASI_STBIMRU_PRIMARY	RW	Any		Block initializing store to primary, install as MRU in L2 cache	5.8.1
$F3_{16}$	ASI_STBIMRU_SECONDARY	RW	Any		Block initializing store to secondary, install as MRU in L2 cache	5.8.1
$F4_{16}$–$F7_{16}$			Any	—	*DAE_invalid_asi*	
$F8_{16}$	ASI_BLK_PL	RW	Any	—	64-byte block load/store, primary address (LE)	5.5
$F9_{16}$	ASI_BLK_SL	RW	Any	—	64-byte block load/store, secondary address (LE)	5.5
FA_{16}	ASI_STBIMRU_PRIMARY_LITTLE	WO	Any		Block initializing store to primary little-endian, install as MRU in L2 cache	5.8.1
FB_{16}	ASI_STBIMRU_SECONDARY_LITTLE	WO	Any		Block initializing store to secondary little-endian, install as MRU in L2 cache	5.8.1
FC_{16}–FF_{16}			Any	—	*DAE_invalid_asi*	

9.2.1 ASI_REAL, ASI_REAL_LITTLE, ASI_REAL_IO, and ASI_REAL_IO_LITTLE

These ASIs are used to bypass the VA-to-RA translation. For these ASIs, the real address is set equal to the truncated virtual address (that is, RA{51:0} ← VA{51:0}), and the attributes used are those present in the matching TTE. The hypervisor will normally set the TTE attributes for ASI_REAL and ASI_REAL_LITTLE to cacheable (cp = 1) and for ASI_REAL_IO and ASI_REAL_IO_LITTLE to noncacheable, with side effect (cp = 0, e = 1). The hardware, however, does not require this, i.e. it allows an ASI_REAL/ASI_REAL_LITTLE to be issued to a noncacheable address (PA{47} = 1) or an ASI_REAL_IO/ASI_REAL_IO_LITTLE to be issued to a cacheable address (PA{47} = 0); no error is flagged in this case.

9.2.2 ASI_SCRATCHPAD

Each virtual processor has a set of privileged ASI_SCRATCHPAD registers at ASI 20_{16} with VA{63:0} = 0_{16}–18_{16}, 30_{16}–38_{16}. These registers are for scratchpad use by privileged software.

M5 Implementation Note Accesses to VA 20_{16} and 28_{16} are much slower than to the other six scratchpad registers.

9.2.3 ASI Accessible Shared Registers

There are a number of ASI addressable registers which are shared by all cores. These registers are located outside the cores, and are mapped to the CMT region of IO space (PA[31:28] = 4'b1111). Please refer to Section 16.2.1 for details.

9.2.4 Block Initializing Store ASIs

Instruction	imm_asi	ASI Value	Operation
ST[B,H,W,TW,X]A	ASI_ST_BLKINIT_AS_IF_USER_PRIMARY (ASI_STBI_AIUP)	22_{16}	64-byte block initializing store to primary address space, user privilege
	ASI_ST_BLKINIT_AS_IF_USER_SECONDARY (ASI_STBI_AIUS)	23_{16}	64-byte block initializing store to secondary address space, user privilege
	ASI_ST_BLKINIT_REAL (ASI_STBI_R)	26_{16}	64-byte block initializing store to real address
	ASI_ST_BLKINIT_NUCLEUS (ASI_STBI_N)	27_{16}	64-byte block initializing store to nucleus address space
	ASI_ST_BLKINIT_AS_IF_USER_PRIMARY_LITTLE (ASI_STBI_AIUPL)	$2A_{16}$	64-byte block initializing store to primary address space, user privilege, little-endian
	ASI_ST_BLKINIT_AS_IF_USER_SECONDARY_LITTLE (ASI_STBI_AIUS_L)	$2B_{16}$	64-byte block initializing store to secondary address space, user privilege, little-endian
	ASI_ST_BLKINIT_REAL_LITTLE (ASI_STBI_RL)	$2E_{16}$	64-byte block initializing store to real address, little-endian
	ASI_ST_BLKINIT_NUCLEUS_LITTLE (ASI_STBI_NL)	$2F_{16}$	64-byte block initializing store to nucleus address space, little-endian
	ASI_ST_BLKINIT_PRIMARY (ASI_STBI_P)	$E2_{16}$	64-byte block initializing store to primary address space
	ASI_ST_BLKINIT_SECONDARY (ASI_STBI_S)	$E3_{16}$	64-byte block initializing store to secondary address space
	ASI_ST_BLKINIT_PRIMARY_LITTLE (ASI_STBI_PL)	EA_{16}	64-byte block initializing store to primary address space, little-endian
	ASI_ST_BLKINIT_SECONDARY_LITTLE (ASI_STBI_SL)	EB_{16}	64-byte block initializing store to secondary address space, little-endian
	ASI_ST_BLKINIT_MRU_PRIMARY (ASI_STBIMRU_P)	$F2_{16}$	64-byte block initializing store to primary address space, install as MRU in L2 cache
	ASI_ST_BLKINIT_MRU_SECONDARY (ASI_STBIMRU_S)	$F3_{16}$	64-byte block initializing store to secondary address space, install as MRU in L2 cache
	ASI_ST_BLKINIT_MRU_PRIMARY_LITTLE (ASI_STBIMRU_PL)	FA_{16}	64-byte block initializing store to primary address space, little-endian, install as MRU in L2 cache
	ASI_ST_BLKINIT_MRU_SECONDARY_LITTLE (ASI_STBIMRU_SL)	FB_{16}	64-byte block initializing store to secondary address space, little-endian, install as MRU in L2 cache

Description

Block initializing store ASIs can be selected for use in integer or floating-point store instructions. These ASIs allow block initializing stores to be performed to the same address spaces as normal stores. Little-endian ASIs access data in little-endian format, otherwise the access is assumed to be big-endian.

Integer and floating-point stores of all sizes (to alternate space) are allowed to use these ASIs.

All stores to these ASIs operate under relaxed memory ordering (RMO). To ensure ordering with respect to subsequent stores and loads, software must follow a sequence of these stores with a MEMBAR #StoreStore or #StoreLoad, respectively. To ensure ordering with respect to prior stores, software must precede these stores with a MEMBAR #StoreStore.

Stores to these ASIs where the least-significant 5 bits of the address are non-zero (that is, not the first word in the L2 cache line) behave the same as a normal RMO store. A store to these ASIs where the least-significant 5 bits are zero will load a line in the L2 cache with all zeros, and then update that line with the new store data. A store to these ASIs where the least-significant 6 bits are zero will load the first line (bit 4 equal to 0) in the L2 cache with all zeros, and then update that line with the new store data. The second 32B line may or may not be initialized to 0 prior to being established in the L2 cache. If the second 32B line is not initialized to 0, it is copied into the L2 cache using the current value from the L3 cache or memory. This special store will make sure the 32B lines maintain coherency when they are loaded into the L2 cache, but will not generally fetch the line from L3 cache or memory (initializing it with zeros instead), except as noted above. Stores using these ASIs to a noncacheable address behave the same as a normal store.

The ASIs $F2_{16}$, $F3_{16}$, FA_{16}, and FB_{16} operate as described above, but establish the line in the L2 cache as most-recently-used (MRU), thereby helping to ensure they are not replaced shortly after being established. This can aid in cases where the newly-established line is expected to be referenced in the near future from a process running on the same physical core.

Note These instructions are used for transferring large blocks of data (more than 256 bytes); for example, `memcpy()` and `memset()`. On SPARC M5, a twin load forces a miss in the primary cache and will not allocate a line in the primary cache, but does allocate in L2.

The following pseudocode shows how these ASIs can be used to do a quadword-aligned (on both source and destination) copy of N quadwords from *A* to *B* (where N > 3). Note that the final 64 bytes of the copy is performed using normal stores, guaranteeing that all initial zeros in a cache line are overwritten with copy data. This pseudocode may not be optimal for SPARC M5; it is provided as an example only.

```
%l0 ← [A]
%l1 ← [B]
prefetch [%l0]
for (i = 0; i < N-4; i++) {
   if ((i mod 4) ≠ 0) {
      prefetch [%l0+64]
   }
   ldtxa [%l0] #ASI_TWINX_P, %l2
   add %l0, 16, %l0
   stxa %l2, [%l1] #ASI_ST_BLKINIT_PRIMARY
   add %l1, 8, %l1
   stxa %l3, [%l1] #ASI_ST_BLKINIT_PRIMARY
   add %l1, 8, %l1
}
for (i = 0; i < 4; i++) {
   ldtxa [%l0] #ASI_TWINX_P, %l2
   add %l0, 16, %l0
   stx %l2, [%l1]
   stx %l3,d [%l1+8]
   add %l1, 16, %l1
}
membar #Sync
```

Programming Notes

The Block Initializing Store ASIs are of Class "N" and are only allowed in dynamically linked, platform-specific, OS-enabled libraries.

CHAPTER 10

Performance Instrumentation

10.1 Introduction

As in previous UltraSPARC CMT processors such as UltraSPARC T1, UltraSPARC T2, UltraSPARC T2+, and SPARC T3, SPARC M5 supports monitoring processor performance by virtue of a set of performance counters. SPARC M5 expands on the capabilities of previous UltraSPARC CMT processors by adding more counters per virtual processor and by being able to measure additional processor and pipeline events. Significant differences from SPARC T3 are as follows:

1. SPARC M5 supports 4 counters (PICs) per virtual processor instead of two.
2. Each PIC is controlled via a dedicated PCR. Each PCR controls only one PIC.
3. The format of the PCR has changed significantly.
4. Access to the PCRs is via hyperprivileged ASIs only, instead of ASRs. The hypervisor can then permit privileged and user access only to the PICs via PCR.picnht and PCR.picnpt, respectively. The PCRs can thus be allocated to hypervisor, supervisor, or user code in any combination. This also enables virtualization of performance counter and measurement infrastructure to ease future development as processor architecture evolves.
5. Access to the PICs is via non-privileged ASIs only, instead of ASRs. Access is only granted based upon the settings of PCR.picnht and PCR.picnpt as described above.
6. The pic_overflow trap no longer exists. Instead, a PIC which overflows due to a precise performance event generates a precise_performance_event trap, and a PIC which overflows due to an asynchronous performance event generate a disrupting_performance_event trap. Neither trap sets SOFTINT{15}.
7. Precise performance counter overflows have no skid.

10.2 SPARC Performance Control Registers

Each virtual processor has four hyperprivileged, read/write Performance Control registers: PCR0, PCR1, PCR2, and PCR3. Each PCR controls its corresponding PIC: PCR0 controls PIC0, PCR1 controls PIC1, PCR2 controls PIC2, and PCR3 controls PIC3. Each Performance Control register contains ten fields: ntc, picnht, picnpt, sl, mask, ht, ut, st, toe, and ov. All bits except ntc and ov are always updated on a Performance Control register write. ov is a state bit associated with PIC overflow traps and is provided to allow software to determine whether a PIC counter has overflowed. ntc is also a state bit associated with PIC overflow traps that allows software to handle a special case on a *precise_performance_event* trap: TPC and TNPC point to the instruction which caused the overflow, but hardware already executed the instruction at TPC. In this case software must execute a DONE

instead of a RETRY. ntc and ov can be reset by software but can never be written to 1. sl controls which events are counted in a PIC. mask is used in conjunction with sl to determine which set of subevents are counted in a PIC. toe controls whether a trap is generated when the PIC counter overflows. ut controls whether user-level events are counted. st controls whether supervisor-level events are counted. ht controls whether hypervisor level events are counted.The format of this register is shown in TABLE 10-1. Note that changing a field in the PCR does not directly affect a PIC value. To reliably change the events being monitored, software should perform the following sequence:

1. Disable counting by writing zeroes to PCR.sl and clearing PCR.ut, PCR.ht, and PCR.st.
2. Reset the PIC.
3. Enable the new event via writing a non-zero value to PCR.sl and setting PCR.ut, PCR.ht, or PCR.st, as appropriate.

TABLE 10-1 Performance Control Registers – PCR0-3 (ASI 64_{16}, VA 00_{16}, 08_{16}, 10_{16}, 18_{16})

Bit	Field	Initial Value	R/W	Description
63:19	—	0	RO	*Reserved*
18	ntc	0	RW	Set to 1 when PIC wraps from 2^{32} –1 to 0 on a next-to-commit (ntc) instruction[1]. Once set, ntc remains set until reset by software. Hardware sets ntc whenever it sets ov on a next-to-commit instruction.
17	picnht	0	RW	PIC non-hyperprivileged trap. Privileged software can access the PIC only if picnht = 0, otherwise a *privileged_action* trap occurs. Non-privileged software can access PIC only when picnht = 0 and picnpt = 0, otherwise a *privileged_action* trap occurs.
16	picnpt	0	RW	PIC non-privileged trap. Non-privileged software can access PIC only when picnht = 0 and picnpt = 0, otherwise a *privileged_action* trap occurs.
15:11	sl	0	RW	Selects one of 32 events to be counted for PIC as per the following table.
10:5	mask	0	RW	Mask event for PIC as listed in TABLE 10-2.
4	ht	0	RW	If ht = 1, count events in hyperprivileged mode; otherwise, ignore hyperprivileged mode events.
3	st	0	RW	If st = 1, count events in privileged mode; otherwise, ignore privileged mode events.
2	ut	0	RW	If ut = 1, count events in user mode; otherwise, ignore user mode events.
1	toe	0	RW	Trap-on-Event: This field controls whether a precise trap (*precise_performance_event*) or disrupting trap (*disrupting_performance_event*) to hyperprivileged software occurs if the corresponding PIC counter overflows. Hardware **AND**s the value of toe with ov to produce a trap. Events in certain event groups (those marked as Precise in TABLE 10-2) generate a precise *precise_performance_event* trap, assuming that PCR.toe = 1 and PCR.ht = 0 — TPC will contain the address of an instruction that generated the counter overflow event[2]. Events in other event groups are not directly related to the instruction stream and an overflow for one of the asynchronous events generates a *disrupting_performance_event* trap; therefore, the TPC may be some number of instructions later than when the overflow event occurred.
0	ov	0	RW	Set to 1 when PIC wraps from 2^{32} –1 to 0. Once set, ov remains set until reset by software.

1. The following instructions are next-to-commit instructions: MD5, SHA1, SHA256, SHA512, MPMUL, MONTMUL, MONTSQR, loads and stores to I/O space, CAS{X}A, LDSTUB, SWAP, WRHPR, WRASR, WRPR, RDHPR, RDPR, RDASR instructions, and any non-translating load or store alternate instruction as defined in Table 9-3, "UltraSPARC {YF(VT40)} ASI Usage," on page 163. When hardware takes a *precise_performance_event* trap on a next-to-commit instruction, the instruction has already been executed. Therefore, trap handler software should execute a DONE instruction; it must not execute a RETRY instruction. Software can examine the ntc bit to determine whether to execute a DONE or a RETRY instruction.
2.

TABLE 10-2 describes the settings of the sl field. Most sl fields have a mask associated with them. Setting multiple mask bits at the same time can lead to multiple events being counted as one event. Some sl groups do not use all of the mask bits; setting unused mask bits has no effect. More details are described in TABLE 10-2.

10.3 SPARC Performance Instrumentation Counter

Each virtual processor has four Performance Instrumentation Counter registers: PIC0, PIC1, PIC2, and PIC3. PCR0 controls PIC0, PCR1 controls PIC1, PCR2 controls PIC2, and PCR3 controls PCR3. Access privilege is controlled by the settings of PCR.picnht and PCR.picnpt. When PCR.picnht = 1, an attempt to access a PIC register in privileged or nonprivileged mode will cause a *privileged_action* trap. When PCR.picnpt = 1 an attempt to access this register in nonprivileged mode causes a *privileged_action* trap.

The PIC counter contains a single 32-bit counter field. The field counts the event selected by PCR.sl. The ut, st, and ht fields for PCR control which combination of user, supervisor, and/or hypervisor events are counted.

Performance counter overflows a) set PCR.ov, and b) generate a hyperprivileged trap if PCR.toe is set. Which trap is generated depends upon whether the event being counted is synchronous or asynchronous, as denoted in TABLE 10-2 above. If the event is asynchronous, a *disrupting_performance_event* trap is generated; otherwise, a *precise_performance_event* trap is generated. For precise traps, the instruction that caused the overflow will not have been executed, and the PC and NPC of the instruction will be captured on the trap stack, with the following caveat. The *precise_performance_event* trap is delivered precisely to hypervisor.

The format of the PIC registers is shown in TABLE 10-2.

TABLE 10-2 Performance Instrumentation Counter Register – PIC0-3 (ASI $B0_{16}$, VA 00_{16}, 08_{16}, 10_{16}, 18_{16})

Bit	Field	Initial Value	R/W	Description
63:32	—	0	RW	*Reserved*
31:0	counter	0	RW	Programmable event counter, event controlled by PCR.sl.

CHAPTER 11

Implementation Dependencies

11.1 SPARC V9 General Information

11.1.1 Level-2 Compliance (Impdep #1)

SPARC M5 is designed to meet Level-2 SPARC V9 compliance. It

- Correctly interprets all nonprivileged operations, and
- Correctly interprets all privileged elements of the architecture.

Note | System emulation routines (for example, quad-precision floating-point operations) shipped with SPARC M5 also must be Level-2 compliant.

11.1.2 Unimplemented Opcodes, ASIs, and ILLTRAP

SPARC V9 unimplemented, *reserved*, ILLTRAP opcodes, and instructions with invalid values in *reserved* fields (other than *reserved* FPops) encountered during execution cause an *illegal_instruction* trap. Unimplemented and *reserved* ASI values cause a *DAE_invalid_ASI* trap.

11.1.3 Trap Levels (Impdep #37, 38, 39, 40, 114, 115)

SPARC M5 supports two trap levels; that is, *MAXPTL* = 2. Normal execution is at TL = 0.

A virtual processor normally executes at trap level 0 (`execute_state`, TL = 0). Per SPARC V9, a trap causes the virtual processor to enter the next higher trap level, which is a very fast and efficient process because there is one set of trap state registers for each trap level. After saving the most important machine states (PC, NPC, PSTATE) on the trap stack at this level, the trap (or error) condition is processed.

11.1.4 Trap Handling (Impdep #16, 32, 33, 35, 36, 44)

SPARC M5 supports precise trap handling for all operations except for deferred and disrupting traps from hardware failures and interrupts. SPARC M5 implements precise traps, interrupts, and exceptions for all instructions, including long-latency floating-point operations. Multiple traps levels are supported, allowing graceful recovery from faults. SPARC M5 can efficiently execute kernel code even in the event of multiple nested traps, promoting strand efficiency while dramatically reducing the system overhead needed for trap handling.

Multiple sets of global registers are provided. This further increases OS performance, providing fast trap execution by avoiding the need to save and restore registers while processing exceptions.

All traps supported in SPARC M5 are listed in TABLE 6-2 on page 49.

11.1.5 Secure Software

To establish an enhanced security environment, it may be necessary to initialize certain virtual processor states between contexts. Examples of such states are the contents of integer and floating-point register files, condition codes, and state registers. See also *Clean Window Handling (Impdep #102)*.

11.1.6 Address Masking (Impdep #125)

SPARC M5 follows Oracle SPARC Architecture 2011 for PSTATE.am masking. Addresses to non-translating ASIs, *REAL* ASIs, and accesses that bypass translation are never masked.

11.2 SPARC V9 Integer Operations

11.2.1 Integer Register File and Window Control Registers (Impdep #2)

SPARC M5 implements an eight-window 64-bit integer register file; that is, *N_REG_WINDOWS* = 8. SPARC M5 truncates values stored in the CWP, CANSAVE, CANRESTORE, CLEANWIN, and OTHERWIN registers to three bits. This includes implicit updates to these registers by SAVE, SAVED, RESTORE, and RESTORED instructions. The most significant two bits of these registers read as zero.

11.2.2 Clean Window Handling (Impdep #102)

SPARC V9 introduced the concept of "clean window" to enhance security and integrity during program execution. A clean window is defined to be a register window that contains either all zeroes or addresses and data that belong to the current context. The CLEANWIN register records the number of available clean windows.

When a SAVE instruction requests a window and there are no more clean windows, a *clean_window* trap is generated. System software needs to clean one or more windows before returning to the requesting context.

11.2.3 Integer Multiply and Divide

Integer multiplications (MULScc, SMUL{cc}, MULX) and divisions (SDIV{cc}, UDIV{cc}, UDIVX) are executed directly in hardware.

11.2.4 MULScc

SPARC V9 does not define the value of xcc and rd{63:32] for MULScc. SPARC M5 sets xcc.n to 0, xcc.z to 1 if rd{63:0} is zero and to 0 if rd{63:0} is not zero, xcc.v to 0, and xcc.c to 0. SPARC M5 sets rd{63:33} to zeros, and sets rd{32} to icc.c (that is, rd{32} is set if there is a carry-out of rd{31}; otherwise, it is cleared).

11.3 SPARC V9 Floating-Point Operations

11.3.1 Overflow, Underflow, and Inexact Traps (Impdep #3, 55)

SPARC M5 implements precise floating-point exception handling. Tininess, as it pertains to underflow is detected before rounding.

11.3.2 Quad-Precision Floating-Point Operations (Impdep #3)

All quad-precision floating-point instructions, listed in TABLE 11-1, cause an *illegal_instruction* trap. These operations are then emulated by system software.

TABLE 11-1 Unimplemented Quad-Precision Floating-Point Instructions

Instruction	Description
F<s|d>TOq	Convert single-/double- to quad-precision floating-point.
F<i|x>TOq	Convert 32-/64-bit integer to quad-precision floating-point.
FqTO<s|d>	Convert quad- to single-/double-precision floating-point.
FqTO<i|x>	Convert quad-precision floating-point to 32-/64-bit integer.
FCMP<E>q	Quad-precision floating-point compares.
FMOVq	Quad-precision floating-point move.
FMOVqcc	Quad-precision floating-point move if condition is satisfied.
FMOVqr	Quad-precision floating-point move if register match condition.
FABSq	Quad-precision floating-point absolute value.
FADDq	Quad-precision floating-point addition.
FDIVq	Quad-precision floating-point division.
FdMULq	Double- to quad-precision floating-point multiply.
FMULq	Quad-precision floating-point multiply.
FNEGq	Quad-precision floating-point negation.
FSQRTq	Quad-precision floating-point square root.
FSUBq	Quad-precision floating-point subtraction.

11.3.3 Floating-Point Upper and Lower Dirty Bits in FPRS Register

The FPRS_dirty_upper (du) and FPRS_dirty_lower (dl) bits in the Floating-Point Registers State (FPRS) register are set when an instruction that modifies the corresponding upper or lower half of the floating-point register file is issued. Floating-point register file modifying instructions include floating-point operate, graphics, floating-point loads and block load instructions.

SPARC V9 allows FPRS.du and FPRS.dl to be set pessimistically. SPARC M5 sets FPRS.du or FPRS.dl either when an instruction that updates the floating-point register file successfully completes, or when an FMOVcc or FMOVr that does not meet the condition successfully completes.

11.3.4 Floating-Point Status Register (FSR) (Impdep #13, 19, 22, 23, 24)

SPARC M5 supports precise-traps and implements all three exception fields (tem, cexc, and aexc) conforming to IEEE Standard 754-1985.

SPARC M5 implements the FSR register according to the definition in Oracle SPARC Architecture 2011, with the following implementation-specific clarifications:

- SPARC M5 does not contain an FQ, therefore FSR.qne always reads as 0 and an attempt to read the FQ with an RDPR instruction causes an *illegal_instruction* trap.
- SPARC M5 does not detect the unimplemented_FPop, unfinished_FPop, sequence_error, hardware_error, or invalid_fp_register floating-point trap types directly in hardware, therefore does not generate a trap when those conditions occur.

TABLE 11-2 documents the fields of the FSR.

TABLE 11-2 Floating-Point Status Register Format

Bits	Field	RW	Description
63:38	—	RO	*Reserved*
37:36	fcc3	RW	Floating-point condition code (set 3). One of four sets of 2-bit floating-point condition codes, which are modified by the FCMP{E} (and LD{X}FSR) instructions. The FBfcc, FMOVcc, and MOVcc instructions use one of these condition code sets to determine conditional control transfers and conditional register moves. **Note:** fcc0 is the same as the FCC in SPARC V8.
35:34	fcc2	RW	Floating-point condition code (set 2). See fcc3 description.
33:32	fcc1	RW	Floating-point condition code (set 1) See fcc3 description.
31:30	**rd**	RW	IEEE Std. 754-1985 rounding direction, as follows:
29:28	—	RO	*Reserved*
27:23	tem	RW	IEEE-754 trap enable mask. Five-bit trap enable mask for the IEEE-754 floating-point exceptions. If a floating-point operate instruction produces one or more exceptions, the corresponding *cexc/aexc* bits are set and an *fp_exception_ieee_754* (with FSR.ftt = 1, *IEEE_754_exception*) exception is generated.

rd	Round Toward
0	Nearest (even if tie)
1	0
2	+∞
3	−∞

TABLE 11-2 Floating-Point Status Register Format *(Continued)*

Bits	Field	RW	Description
22	ns	RO	Nonstandard floating-point results. SPARC M5 does not implement a non-standard floating-point mode. FSR.ns always reads as 0, and writes to it are ignored.
21:20	—	RO	*Reserved*
19:17	ver	RO	FPU version number. This field identifies a particular implementation of the SPARC M5 FPU architecture.
16:14	ftt	RW	Floating-point trap type. Set whenever a floating-point instruction causes the *fp_exception_ieee_754* or *fp_exception_other* traps. Values are as follows:

ftt	Floating-Point Trap Type	Trap Signalled
0	None	—
1	IEEE_754_exception	*fp_exception_ieee_754*
2	reserved	—
3	reserved	—
4	reserved	—
5	reserved	—
6	invalid_fp_register	*fp_exception_other*
7	reserved	—

Note: SPARC M5 neither detects nor generates the unimplemented_FPop, unfinished_FPop, sequence_error, hardware_error or invalid_fp_register trap types directly in hardware.
Note: SPARC M5 does not contain an FQ. An attempt to read the FQ with an RDPR instruction causes an *illegal_instruction* trap.

13:	qne	RW	Floating-point deferred-trap queue (FQ) not empty. Not used, because SPARC M5 implements precise floating-point exceptions.
12	—	RO	*Reserved*
11:10	fcc0	RW	Floating-point condition code (set 0). See fcc3 description.
9:5	aexc	RW	Accumulated outstanding exceptions. Accumulates IEEE 754 exceptions while floating-point exception traps are disabled (that is, while corresponding bit in FSR.tem is zero)
4:0	cexc	RW	Current outstanding exceptions. Indicates the most recently generated IEEE 754 exceptions.

11.4 SPARC V9 Memory-Related Operations

11.4.1 Load/Store Alternate Address Space (Impdep #5, 29, 30)

Supported ASI accesses are listed in Section 9.3.

11.4.2 Read/Write ASR (Impdep #6, 7, 8, 9, 47, 48)

Supported ASRs are listed in Chapter 3, *Registers*.

11.4.3 MMU Implementation (Impdep #41)

SPARC M5 memory management is based on in-memory Translation Storage Buffers (TSBs) backed by a Software Translation Table. See Chapter 13, *Memory Management Unit* for more details.

11.4.4 FLUSH and Self-Modifying Code (Impdep #122)

FLUSH is needed to synchronize code and data spaces after code space is modified during program execution. FLUSH is described in Section D.3.4. On SPARC M5, the FLUSH effective address is ignored, and as a result, FLUSH cannot cause a *DAE_invalid_ASI* trap.

Note | SPARC V9 specifies that the FLUSH instruction has no latency on the issuing virtual processor. In other words, a store to instruction space prior to the FLUSH instruction is visible immediately after the completion of FLUSH. When a flush is performed, SPARC M5 guarantees that earlier code modifications will be visible across the whole system.

11.4.5 PREFETCH{A} (Impdep #103, 117)

For SPARC M5, PREFETCH{A} instructions follow TABLE 11-3 based on the fcn value. See Section 5.3, *PREFETCH/PREFETCHA*, on page 35 for more detail.

TABLE 11-3 PREFETCH{A} Variants in SPARC M5

fcn	Prefetch Function	Action
0_{16}	Weak prefetch for several reads	Prefetch to L1 data cache and Level 2 cache.
1_{16}	Weak prefetch for one read	Prefetch to L2 cache.
2_{16}	Weak prefetch for several writes	Prefetch to L2 cache (exclusive)
3_{16}	Weak prefetch for one write	Prefetch to L2 cache (exclusive)
4_{16}	Prefetch Page	No operation.
5_{16}–F_{16}	—	*Illegal_instruction* trap.
10_{16}	NOP	NOP - no action taken.
11_{16}	Strong prefetch to nearest unified cache	Prefetch into Level 2 cache.
12_{16}–13_{16}	NOP	NOP - no action taken.
14_{16}	Strong prefetch for several reads	Prefetch to L1 data cache and Level 2 cache.
15_{16}	Strong prefetch for one read	Prefetch to L2 cache.
16_{16}	Strong prefetch for several writes	Prefetch to L2 cache (exclusive)
17_{16}	Strong prefetch for one write	Prefetch to L2 cache (exclusive)
18_{16}–$1F_{16}$	NOP	No operation

11.4.6 LDD/STD Handling (Impdep #107, 108)

LDD and STD instructions are directly executed in hardware.

Note | LDD/STD are deprecated in SPARC V9. In SPARC M5 it is more efficient to use LDX/STX for accessing 64-bit data. LDD/STD take longer to execute than two 32- or 64-bit loads/stores.

11.4.7 FP mem_address_not_aligned (Impdep #109, 110, 111, 112)

LDDF{A}/STDF{A} cause an *LDDF_/STDF_ mem_address_not_aligned* trap if the effective address is 32-bit aligned but not 64-bit (doubleword) aligned.

LDQF{A}/STQF{A} are not directly executed in hardware; they cause an *illegal_instruction* trap.

11.4.8 Supported Memory Models (Impdep #113, 121)

SPARC M5 supports only the TSO memory model, although certain specific operations such as block loads and stores operate under the RMO memory model. See Chapter 8, Section 8.2. Supported Memory Models.".

11.4.9 Implicit ASI When TL > 0 (Impdep #124)

SPARC M5 matches all Oracle SPARC Architecture implementations and makes the implicit ASI for instruction fetching `ASI_NUCLEUS` when TL > 0, while the implicit ASI for loads and stores when TL > 0 is `ASI_NUCLEUS` if PSTATE.cle=0 or `ASI_NUCLEUS_LITTLE` if PSTATE.cle=1.

11.5 Non-SPARC V9 Extensions

11.5.1 Cache Subsystem

SPARC M5 contains one or more levels of cache. The cache subsystem architecture is described in Appendix D, *Cache Coherency and Ordering*.

11.5.2 Block Memory Operations

SPARC M5 supports 64-byte block memory operations utilizing a block of eight double-precision floating point registers as a temporary buffer. See Section 5.5.

11.5.3 Partial Stores

SPARC M5 supports 8-/16-/32-bit partial stores to memory. See Section 5.5.

11.5.4 Short Floating-Point Loads and Stores

SPARC M5 supports 8-/16-bit loads and stores to the floating-point registers.

11.5.5 Load Twin Extended Word

SPARC M5 supports 128-bit atomic load operations to a pair of integer registers.

11.5.6 SPARC M5 Instruction Set Extensions (Impdep #106)

The SPARC M5 processor supports VIS 3.0. VIS instructions are designed to enhance graphics functionality and improve the efficiency of memory accesses.

Unimplemented IMPDEP1 and IMPDEP2 opcodes encountered during execution cause an *illegal_instruction* trap.

Other instruction extensions are described in Chapter 3, *Registers*.

11.5.7 Performance Instrumentation

SPARC M5 performance instrumentation is described in Chapter 10, *Performance Instrumentation*.

CHAPTER 12

Cryptographic Extensions

Oracle SPARC Architecture CMT processors have always provided hardware support for a range of cryptographic operations. In UltraSPARC T1, UltraSPARC T2, and SPARC T3 contained discrete, hyperprivileged, per-core accelerators. However, the software overheads associated with using these accelerators can be somewhat problematic for small cryptographic operations.

SPARC M5 dispenses with per-core discrete cryptographic accelerators, and provides cryptographic support via non-privileged instructions. The instructions accelerate bulk ciphers, secure hashes, and public-key algorithms. Since these instructions are non-privileged, they can be used directly by applications, or by commonly used open source cryptographic libraries such as OpenSSL. In doing so, SPARC M5 eliminates software overhead associated with discrete cryptographic accelerators.

In SPARC M5, symmetric ciphers are implemented such that a single instruction is capable of performing a significant portion of a round. Secure hashes are implemented such that a single instruction performs a single block of the hash operation (i.e. multiple rounds). Public-key operations are accelerated via instructions that perform large (up to 2048-bit) Montgomery multiplication operations. More details on these instructions can be found in Chapter 5, *Instruction Definitions*.

The SPARC M5 cryptographic extensions have been designed such that future UltraSPARC processors can drop support for older, deprecated ciphers (and introduce support for new ones) by reclaiming opcodes previously reserved for old ciphers. This is achieved by the introduction of the Compatibility Feature Register (CFR).

12.1 CFR Register

The CFR is described in Chapter 3, *Registers*.

12.2 Cryptographic Instructions

SPARC M5 introduces a number of new cryptographic opcodes, which are detailed in Chapter 5, *Instruction Definitions*.

12.3 Cryptographic performance

For a single-thread executing on a core, the basic low-level performance on SPARC M5 is detailed in the following tables.

TABLE 12-1 Symmetric-key performance

Algorithm	Block Size (Bytes)	Block Latency (Cycles)
DES-ECB	8	
3DES-ECB	8	
AES-128-ECB	16	
AES-192-ECB	16	
AES-256-ECB	16	
Kasumi		
Camellia		

TABLE 12-2 Secure hash performance

Algorithm	Block Size (Bytes)	Block Latency (Cycles)
MD5	64	186
SHA-1	64	220
SHA-256	64	188
SHA-512	128	236

TABLE 12-3 Public-key performance

Algorithm	Operation Latency (cycles)
RSA1024(sign)	TBD
RSA2048(sign)	TBD

12.4 Core S3 Crypto Coding Guidance

It is anticipated that the SPARC M5 cryptographic instructions will be widely deployed - not only in Solaris libraries, but also in Open Source libraries like OpenSSL. Implementation of key cryptographic algorithms using these instructions is very straight-forward, and example use is provided in the instructions chapter. It is important that software use the CFR as detailed in Section 3.2.8, *Compatibility Feature Register (CFR)*, on page 21, or software may perform sub-optimally on future processors.

Compatibility Note | *(-- need to fill in with text about compatibility with legacy software, or expected compatibility going forward ... use of the CFR register, etc ---)*

CHAPTER 13

Memory Management Unit

This chapter provides detailed information about the SPARC M5 Memory Management Unit. It describes the internal architecture of the MMU and how to program it.

13.1 Translation Table Entry (TTE)

The Translation Table Entry holds information for a single page mapping. The TTE is broken into two 64-bit words, representing the tag and data of the translation. Just as in a hardware cache, the tag is used to determine whether there is a hit in the TSB.

TABLE 13-1 shows the Oracle SPARC Architecture 2011 TTE tag format, modified to support 5 page sizes, as interpreted by SPARC M5.

TABLE 13-1 TTE Tag Format

Bit	Field	Description
63:48	context	The 16-bit context identifier associated with the TTE.
47:42	0	Must be 0
41:0	va	Virtual Address Tag{63:22}. The virtual page number. Bits 21 through 13 are not maintained in the tag, since these bits are used to index the smallest TSB (512 entries). **NOTE:** SPARC Core S3 hardware only supports a 52-bit VA.

The sun4v TTE data format is shown in TABLE 13-2.

TABLE 13-2 TTE Data Format

Bit	Field	Description
63	v	Valid. If the Valid bit is set, the remaining fields of the TTE are meaningful.
62	nfo	No-fault-only. If this bit is set, loads with `ASI_PRIMARY_NO_FAULT{_LITTLE}`, `ASI_SECONDARY_NO_FAULT{_LITTLE}` are translated. Any other DMMU access will trap with a *DAE_nfo_page* trap. For the IMMU, if the nfo bit is set, an *iae_nfo_page* trap will be taken.
61:56	soft2	soft2 and soft are software-defined fields, provided for use by the operating system. Software fields are not implemented in the SPARC M5 TLB. soft and soft2 fields may be written with any value; they read from the TLB as zero, with the exception of soft{61}, which contains the TLB data parity bit.
55:13	ra	The real page[1] number. For SPARC M5, a 48-bit real address range is supported by the hardware tablewalker, and bits {55:48} should always be zero.

TABLE 13-2 TTE Data Format *(Continued)*

Bit	Field	Description
12	ie	Invert endianess. If this bit is set, accesses to the associated page are processed with inverse endianness from what is specified by the instruction (big-for-little and little-for-big). For the IMMU, the ie bit in the TTE is written into the ITLB but ignored during ITLB operation. The value of the ie bit written into the ITLB will be read out on an ITLB Data Access read. **Note:** This bit is intended to be set primarily for noncacheable accesses.
11	e	Side effect. If this bit is set, noncacheable memory accesses other than block loads and stores are strongly ordered against other e bit accesses, and noncacheable stores are not merged. This bit should be set for pages that map I/O devices having side effects. Note, however, that the e bit does not prevent normal instruction prefetching. For the IMMU, the e bit in the TTE is written into the ITLB, but ignored during ITLB operation. The value of the e bit written into the ITLB will be read out on an ITLB Data Access read. **NOTE:** The e bit does not force an uncacheable access. It is expected, but not required, that the cp and cv bits will be set to zero when the e bit is set.
10:9	cp, cv	The cacheable-in-physically-indexed-cache and cacheable-in-virtually-indexed-cache (cp, cv) bits determine the placement of data in SPARC M5 caches, according to TABLE 13-3. The MMU does not operate on the cacheable bits, but merely passes them through to the cache subsystem. The cv bit is ignored by SPARC M5, is not written into the TLBs, and returns zero on a Data Access read.
8	p	Privileged. If the p bit is set, only privileged software can access the page mapped by the TTE. If the p bit is set and an access to the page is attempted when PSTATE.priv = 0, the MMU will signal an *IAE_privilege_violation* or *DAE_privilege_violation* trap.
7	ep	Executable. If the ep bit is set, the page mapped by this TTE has execute permission granted. Otherwise, execute permission is not granted and the hardware table-walker will not load the ITLB with a TTE with ep = 0. For the IMMU and DMMU, the ep bit in the TTE is not written into the TLB. It returns one on a Data Access read for the ITLB and zero on a Data Access read for the DTLB.

TABLE 13-3 Cacheable Field Encoding (from TSB)

Cacheable (cp:cv)	Meaning of TTE When Placed in: iTLB (I-cache PA-Indexed)	dTLB (D-cache PA-Indexed)
0x	Cacheable in L2 and L3 caches only	Cacheable in L2 and L3 caches only
1x	Cacheable in L3 cache, L2 cache, and I-cache	Cacheable in L3 cache, L2 cache, and D-cache

TABLE 13-2 TTE Data Format *(Continued)*

Bit	Field	Description
6	w	Writable. If the w bit is set, the page mapped by this TTE has write permission granted. Otherwise, write permission is not granted and the MMU will cause a trap if a write is attempted. For the IMMU, the w bit in the TTE is written into the ITLB, but ignored during ITLB operation. The value of the w bit written into the ITLB will be read out on an ITLB Data Access read.
5:4	soft	(see soft2, above)
3:0	size	The page size of this entry, encoded as shown in TABLE 13-4.

TABLE 13-4 Size Field Encoding (from TTE)

Size{3:0}	Page Size
0000	8 KB
0001	64 KB
0010	Reserved
0011	4 MB
0100	Reserved
0101	256 MB
0110	2 GB
0111-1111	Reserved

1. sun4v supports translation from virtual addresses (VA) to real addresses (RA) to physical addresses (PA). Privileged code manages the VA-to-RA translations.

13.2 Translation Storage Buffer (TSB)

A TSB is an array of TTEs managed entirely by software. It serves as a cache of the Software Translation table

A TSB is arranged as a direct-mapped cache of TTEs.

The TSB exists as a normal data structure in memory and therefore may be cached. This policy may result in some conflicts with normal instruction and data accesses, but the dynamic sharing of the level-2 cache resource should provide a better overall solution than that provided by a fixed partitioning.

FIGURE 13-1 shows the TSB organization. The constant *N* is determined by the size field in the TSB register; it may range from 512 entries to 16 M entries.

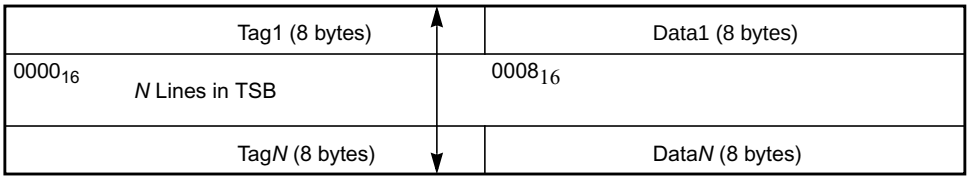


FIGURE 13-1 TSB Organization

13.3 MMU-Related Faults and Traps

13.3.1 *IAE_privilege_violation* Trap

The I-MMU detects a privilege violation for an instruction fetch; that is, an attempted access to a privileged page when PSTATE.priv = 0.

13.3.2 *IAE_nfo_page* Trap

During a hardware tablewalk, the I-MMU matches a TTE entry whose nfo (no-fault-only) bit is set.

Implementation Note | The nfo bit is only checked on I-MMU translations. It is not checked on hardware tablewalks.

13.3.3 *DAE_privilege_violation* Trap

This trap occurs whn the D-MMU detects a privilege violation for a data access; that is, a load or store instruction attempts access to a privileged page when PSTATE.priv = 0.

13.3.4 *DAE_side_effect_page* Trap

This trap occurs when a speculative (nonfaulting) load instruction is issued to a page marked with the side-effect (e) bit = 1.

13.3.5 *DAE_nc_page* Trap

This trap occurs when an atomic instruction (including a 128-bit atomic load) is issued to a memory address marked uncacheable; for example,, with cp = 0.

Implementation Note | For SPARC M5, cp only controls cacheability in the L1 cache, not the private L2 caches or the the shared L3. SPARC M5 performs atomic operations in the L2 cache and supports the ability to complete an atomic operation for pages with the cp bit = 0 even if the L2 cache is disabled. However, to keep SPARC M5 compliant with the Oracle SPARC Architecture 2011 specification, a *DAE_nc_page* trap is generated when an atomic is issued to a memory address marked with cp = 0.

13.3.6 *DAE_invalid_asi* Trap

This trap occurs when an invalid LDA/STA ASI value, invalid virtual address, read to write-only register, or write to read-only register occurs, but not for an attempted user access to a restricted ASI (see the *privileged_action* trap described below).

13.3.7 *DAE_nfo_page* Trap

This trap occurs when an access occurs with an ASI other than ASI_{PRIMARY,SECONDARY}_NO_FAULT{_LITTLE} to a page marked with the nfo (no-fault-only) bit.

13.3.8 *privileged_action* Trap

13.3.9 **_mem_address_not_aligned* Traps

This trap occurs when an access is attempted using a *restricted* ASI while in non-privileged mode (PSTATE.priv = 0).

The *lddf_mem_address_not_aligned*, *stdf_mem_address_not_aligned*, and *mem_address_not_aligned* traps occur when a load, store, atomic, or JMPL/RETURN instruction with a misaligned address is executed.

13.4 MMU Operation Summary

TABLE 13-7 summarizes the behavior of the D-MMU for noninternal ASIs using tabulated abbreviations. TABLE 13-8 summarizes the behavior of the I-MMU. In each case, and for all conditions, the behavior of the MMU is given by one of the abbreviations in TABLE 13-5. TABLE 13-6 lists abbreviations for ASI types.

TABLE 13-5 Abbreviations for MMU Behavior

Abbreviation	Meaning
ok	Normal translation
dasi	*DAE_invalid_asi* trap
dpriv	*DAE_privilege_violation* trap
dse	*DAE_side_effect_page* trap
ipriv	*IAE_privilege_violation* trap

TABLE 13-6 Abbreviations for ASI Types

Abbreviation	Meaning
NUC	`ASI_NUCLEUS*`
PRIM	Any ASI with PRIMARY translation, except `*NO_FAULT`
SEC	Any ASI with SECONDARY translation, except `*NO_FAULT`
PRIM_NF	`ASI_PRIMARY_NO_FAULT*`
SEC_NF	`ASI_SECONDARY_NO_FAULT*`
U_PRIM	`ASI_*_AS_IF_USER_PRIMARY*`
U_SEC	`ASI_*_AS_IF_USER_SECONDARY*`
U_PRIV	`ASI_*_AS_IF_PRIV_*`
REAL	`ASI_*REAL*`

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

Other abbreviations include “`w`” for the writable bit, “`e`” for the side-effect bit, and “`p`” for the privileged bit.

TABLE 13-7 and TABLE 13-8 do not cover the following cases:

- Invalid ASIs, ASIs that have no meaning for the opcodes listed, or nonexistent ASIs; for example, `ASI_PRIMARY_NO_FAULT` for a store or atomic; also, access to SPARC M5 internal registers other than LDXA, LDFA, STDFA or STXA; the MMU signals a *DAE_invalid_asi* trap for this case.

- Attempted access using a restricted ASI in nonprivileged mode; the MMU signals a *privileged_action* trap for this case. Attempted use of a hyperprivileged ASI in privileged mode; the MMU also signals *privileged_action* trap for this case.
- An atomic instruction (including 128-bit atomic load) issued to a memory address marked uncacheable in a physical cache (that is, with cp = 0 or pa{47} = 1); the MMU signals a *DAE_nc_page* trap for this case.
- A data access with an ASI other than ASI_{PRIMARY,SECONDARY}_NO_FAULT{_LITTLE} to a page marked nfo; the MMU signals a *DAE_nfo_page* for this case.
- An instruction access to a page marked with the nfo (no-fault-only) bit. The MMU signals an *IAE_nfo_page* trap for this case.
- An instruction fetch to a memory address marked non-executable (ep = 0). This is checked when Hardware Tablewalk attempts to load the I-MMU, and an *IAE_unauth_access* trap is taken instead.
- Real address out of range; the MMU signals an *instruction_real_range* trap for this case.
- Virtual address out of range and PSTATE.am is not set; the MMU signals an *instruction_address_range* trap for this case.

TABLE 13-7 D-MMU Operations for Normal ASIs

Condition					Behavior			
Opcode	priv Mode	ASI	w		e = 0 p = 0	e = 0 p = 1	e = 1 p = 0	e = 1 p = 1
Load	non-privileged	PRIM, SEC	—		ok	dpriv	ok	dpriv
		PRIM_NF, SEC_NF	—		ok	dpriv	dse	dpriv
	privileged	PRIM, SEC, NUC	—		ok			
		PRIM_NF, SEC_NF	—		ok		dse	
		U_PRIM, U_SEC	—		ok	dpriv	ok	dpriv
		REAL	—		ok			
FLUSH	non-privileged		—		ok			
	privileged		—		ok			
Store or Atomic	non-privileged	PRIM, SEC	0		dprot	dpriv	dprot	dpriv
			1		ok	dpriv	ok	dpriv
	privileged	PRIM, SEC, NUC	0		dprot			
			1		ok			
		U_PRIM, U_SEC	0		dprot	dpriv	dprot	dpriv
			1		ok	dpriv	ok	dpriv
		REAL	0		dprot			
			1		ok			

TABLE 13-8 I-MMU Operations

Condition	Behavior	
privilege Mode	P = 0	P = 1
nonprivileged	ok	ipriv
privileged	ok	

See summary of the SPARC M5 ASI map.

13.5 Translation

13.5.1 Instruction Translation

13.5.1.1 Instruction Prefetching

SPARC M5 fetches instructions sequentially (including delay slots). SPARC M5 fetches delay slots before the branch is resolved (before whether the delay slot will be annulled is known). SPARC M5 also fetches the target of a DCTI before the delay slot executes.

13.5.2 Data Translation

TABLE 13-9 DMMU Translation *(1 of 3)*

ASI Value (hex)	ASI NAME	Translation		
		Nonprivileged	Privileged	Hypervisor
00_{16}–03_{16}	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
04_{16}	ASI_NUCLEUS	*privileged_action*	VA → PA	
05_{16}–$0B_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$0C_{16}$	ASI_NUCLEUS_LITTLE	*privileged_action*	VA → PA	
$0D_{16}$–$0F_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
10_{16}	ASI_AS_IF_USER_PRIMARY	*privileged_action*	VA → PA	
11_{16}	ASI_AS_IF_USER_SECONDARY	*privileged_action*	VA → PA	
12_{16}–13_{16}	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
14_{16}	ASI_REAL	*privileged_action*	RA → PA	
15_{16}	ASI_REAL_IO	*privileged_action*	RA → PA	
16_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY	*privileged_action*	VA → PA	
17_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY	*privileged_action*	VA → PA	
18_{16}	ASI_AS_IF_USER_PRIMARY_LITTLE	*privileged_action*	VA → PA	
19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE	*privileged_action*	VA → PA	
$1A_{16}$–$1B_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$1C_{161}$	ASI_REAL_LITTLE	*privileged_action*	RA → PA	
$1D_{16}$	ASI_REAL_IO_LITTLE	*privileged_action*	RA → PA	
$1E_{16}$	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE	*privileged_action*	VA → PA	
$1F_{16}$	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE	*privileged_action*	VA → PA	
20_{16}	ASI_SCRATCHPAD	*privileged_action*	nontranslating	
21_{16}	ASI_PRIMARY_CONTEXT_0_REG, ASI_PRIMARY_CONTEXT_1_REG, ASI_SECONDARY_CONTEXT_0_REG, ASI_SECONDARY_CONTEXT_1_REG	*privileged_action*	nontranslating	

TABLE 13-9 DMMU Translation *(2 of 3)*

ASI Value (hex)	ASI NAME	Translation		
		Nonprivileged	Privileged	Hypervisor
22_{16}	ASI_TWINX_AIUP, ASI_STBI_AIUP	*privileged_action*	VA → PA	
23_{16}	ASI_TWINX_AIUS, ASI_STBI_AIUS	*privileged_action*	VA → PA	
24_{16}	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
25_{16}	ASI_QUEUE	*privileged_action*	nontranslating	
26_{16}	ASI_TWINX_REAL, ASI_STBI_REAL	*privileged_action*	RA → PA	
27_{16}	ASI_TWINX_NUCLEUS, ASI_STBI_N	*privileged_action*	VA → PA	
28_{16}–29_{16}	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$2A_{16}$	ASI_TWINX_AIUPL, ASI_STBI_AIUPL	*privileged_action*	VA → PA	
$2B_{16}$	ASI_TWINX_AIUSL, ASI_STBI_AIUSL	*privileged_action*	VA → PA	
$2C_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$2D_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$2E_{16}$	ASI_TWINX_REAL_LITTLE, ASI_STBI_REAL_LITTLE	*privileged_action*	RA → PA	
$2F_{16}$	ASI_TWINX_NL, ASI_STBI_NL	*privileged_action*	VA → PA	
80_{16}	ASI_PRIMARY	VA → PA	VA → PA	
81_{16}	ASI_SECONDARY	VA → PA	VA → PA	
82_{16}	ASI_PRIMARY_NO_FAULT	VA → PA	VA → PA	
83_{16}	ASI_SECONDARY_NO_FAULT	VA → PA	VA → PA	
84_{16}–87_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
88_{16}	ASI_PRIMARY_LITTLE	VA → PA	VA → PA	
89_{16}	ASI_SECONDARY_LITTLE	VA → PA	VA → PA	
$8A_{16}$	ASI_PRIMARY_NO_FAULT_LITTLE	VA → PA	VA → PA	
$8B_{16}$	ASI_SECONDARY_NO_FAULT_ LITTLE	VA → PA	VA → PA	
$8C_{16}$–AF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$B0_{16}$	ASI_PIC0, ASI_PIC1, ASI_PIC2, ASI_PIC3	nontranslating	nontranslating	
$B1_{16}$–BF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$C0_{16}$	ASI_PST8_P	VA → PA	VA → PA	
$C1_{16}$	ASI_PST8_S	VA → PA	VA → PA	
$C2_{16}$	ASI_PST16_P	VA → PA	VA → PA	
$C3_{16}$	ASI_PST16_S	VA → PA	VA → PA	
$C4_{16}$	ASI_PST32_P	VA → PA	VA → PA	
$C5_{16}$	ASI_PST32_S	VA → PA	VA → PA	
$C6_{16}$–$C7_{16}$	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	

TABLE 13-9 DMMU Translation *(3 of 3)*

ASI Value (hex)	ASI NAME	Translation		
		Nonprivileged	Privileged	Hypervisor
$C8_{16}$	ASI_PST8_PL	VA → PA	VA → PA	
$C9_{16}$	ASI_PST8_SL	VA → PA	VA → PA	
CA_{16}	ASI_PST16_PL	VA → PA	VA → PA	
CB_{16}	ASI_PST16_SL	VA → PA	VA → PA	
CC_{16}	ASI_PST32_PL	VA → PA	VA → PA	
CD_{16}	ASI_PST32_SL	VA → PA	VA → PA	
CE_{16}–CF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$D0_{16}$	ASI_FL8_P	VA → PA	VA → PA	
$D1_{16}$	ASI_FL8_S	VA → PA	VA → PA	
$D2_{16}$	ASI_FL16_P	VA → PA	VA → PA	
$D3_{16}$	ASI_FL16_S	VA → PA	VA → PA	
$D4_{16}$–$D7_{16}$		*DAE_invalid_asi*	*DAE_invalid_asi*	
$D8_{16}$	ASI_FL8_PL	VA → PA	VA → PA	
$D9_{16}$	ASI_FL8_SL	VA → PA	VA → PA	
DA_{16}	ASI_FL16_PL	VA → PA	VA → PA	
DB_{16}	ASI_FL16_SL	VA → PA	VA → PA	
DC_{16}–DF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$E0_{16}$	ASI_BLK_COMMIT_PRIMARY	VA → PA	VA → PA	
$E1_{16}$	ASI_BLK_COMMIT_SECONDARY	VA → PA	VA → PA	
$E2_{16}$	ASI_TWINX_P, ASI_STBI_P	VA → PA	VA → PA	
$E3_{16}$	ASI_TWINX_S, ASI_STBI_S	VA → PA	VA → PA	
$E4_{16}$–$E9_{16}$	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
EA_{16}	ASI_TWINX_PL, ASI_STBI_PL	VA → PA	VA → PA	
EB_{16}	ASI_TWINX_SL, ASI_STBI_SL	VA → PA	VA → PA	
EC_{16}-EF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$F0_{16}$	ASI_BLK_PRIMARY	VA → PA	VA → PA	
$F1_{16}$	ASI_BLK_SECONDARY	VA → PA	VA → PA	
$F2_{16}$	ASI_STBI_MRU_PRIMARY	VA → PA	VA → PA	
$F3_{16}$	ASI_STBI_MRU_SECONDARY	VA → PA	VA → PA	
$F4_{16}$-$F7_{16}$	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$F8_{16}$	ASI_BLK_PL	VA → PA	VA → PA	
$F9_{16}$	ASI_BLK_SL	VA → PA	VA → PA	
FA_{16}	ASI_STBI_MRU_PRIMARY_LITTLE	VA → PA	VA → PA	
FB_{16}	ASI_STBI_MRU_SECONDARY_LITTLE	VA → PA	VA → PA	
FC_{16}-FF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	

13.6 Compliance With the SPARC V9 Annex F

The SPARC M5 MMU complies completely with the SPARC V9 MMU Requirements described in Annex F of the *The SPARC Architecture Manual, Version 9*. TABLE 13-10 shows how various protection modes can be achieved, if necessary, through the presence or absence of a translation in the I- or D-MMU.

TABLE 13-10 MMU Compliance With SPARC V9 Annex F Protection Mode

Condition			
TTE in D-MMU	**TTE in I-MMU**	**Writable Attribute Bit**	**Resultant Protection Mode**
Yes	No	0	Read-only
No	Yes	Don't Care	Execute-only
Yes	No	1	Read/Write
Yes	Yes	0	Read-only/Execute
Yes	Yes	1	Read/Write/Execute

13.7 MMU Internal Registers and ASI Operations

13.7.1 Accessing MMU Registers

All internal MMU registers can be accessed directly by the virtual processor through ASIs defined by SPARC M5.

See Section 13.5 for details on the behavior of the MMU during all other SPARC M5 ASI accesses.

Note | STXA to an MMU register *does not* require any subsequent instructions such as a MEMBAR #Sync, FLUSH, DONE, or RETRY before the register effect will be visible to load / store / atomic accesses. SPARC M5 resolves all MMU register hazards via an automatic synchronization on all MMU register writes.

If the low order three bits of the VA are non-zero in an LDXA/STXA to/from these registers, a *mem_address_not_aligned* trap occurs. Writes to read-only, reads to write-only, illegal ASI values, or illegal VA for a given ASI may cause a *DAE_invalid_asi* trap.

Caution | SPARC M5 does not check for out-of-range virtual addresses during an STXA to any internal register; it simply sign-extends the virtual address based on VA{51}. Software must guarantee that the VA is within range.

TABLE 13-11 SPARC M5 MMU Internal Registers and ASI Operations

I-MMU ASI	D-MMU ASI	VA{63:0}	Access	Register or Operation Name
21_{16}		8_{16}	Read/Write	Primary Context 0 register
—	21_{16}	10_{16}	Read/Write	Secondary Context 0 register
21_{16}		108_{16}	Read/Write	Primary Context 1 register
—	21_{16}	110_{16}	Read/Write	Secondary Context 1 register

13.7.2 Context Registers

SPARC M5 supports a pair of primary and a pair of secondary context registers per strand, which are shared by the I- and D-MMUs. Primary Context 0 and Primary Context 1 are the primary context registers, and a TLB entry for a translating primary ASI can match the context field with either Primary Context 0 or Primary Context 1 to produce a TLB hit. Secondary Context 0 and Secondary Context 1 are the secondary context registers, and a TLB entry for a translating secondary ASI can match the context field with either Secondary Context 0 or Secondary Context 1 to produce a TLB hit.

Compatibility Note | To maintain backward compatibility with software designed for a single primary and single secondary context register, writes to Primary (Secondary) Context 0 Register also update Primary (Secondary) Context 1 Register.

The Primary Context 0 and Primary Context 1 registers are defined as shown in FIGURE 13-2, where pcontext is the context value for the primary address space.

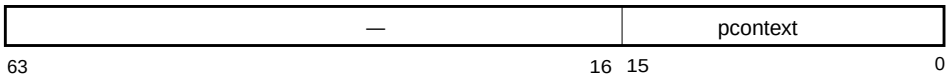


FIGURE 13-2 Primary Context 0/1 registers, ASI 21_{16}, VA 8_{16} and ASI 21_{16}, VA 108_{16}

The Secondary Context 0 and Secondary Context 1 Registers are defined in FIGURE 13-3, where scontext is the context value for the secondary address space.

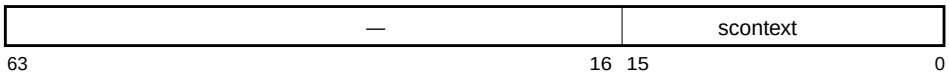


FIGURE 13-3 Secondary Context 0/1 Registers, ASI 21_{16}, VA 10_{16} and 21_{16}, VA 110_{16}

The contents of the Nucleus Context register are hardwired to the value zero:

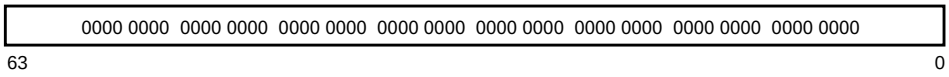


FIGURE 13-4 Nucleus Context Register

APPENDIX **A**

Programming Guidelines

A.1 Multithreading

In SPARC M5, each physical core contains eight strands. Each strand has a full set of architected state registers and appears to software as a complete processor[1]. In general, each of the 8 strands share the execution pipeline including the instruction, data, and L2 caches, branch predictor, out-of-order scheduling, execution pipelines, and retirement mechanisms. The pipeline is both horizontally and vertically threaded. It is vertically threaded since instructions from different strands can be in adjacent pipeline stages. It is horizontally threaded where parellelism allows. For example, each cycle the Pick unit may pick one instruction from one thread, and another instruction from a different thread, to be issued to independent execution units. SPARC M5 utilizes advanced branch prediction, dual instruction issue, out-of-order execution with up to 128 instructions in flight using a reorder buffer, hardware prefetching of instruction and data cache misses, and seamless hardware thread switching to provide high per-thread performance as well as high throughput. The pipeline is partitioned into several major subsections: instruction fetch, select/decode/rename, pick/issue/execute, and commit, each of which are mostly independent of one another.

A.1.1 Instruction fetch

Each cycle an arbiter chooses one strand for instruction fetching. The least-recently-fetched strand among the strands which are ready for fetching is the one chosen. A strand may not be ready for fetching due to instruction cache misses, instruction buffer full conditions, or other reasons. Once selected for fetch, up to four instructions may be fetched from the instruction cache and placed in per-strand instruction buffers. Instruction fetching occupies the first few stages of the pipeline. Instruction fetching is decoupled from the rest of the pipeline by the Select stage.

A.1.2 Select/Decode/Rename

In the same fashion that instruction fetch chooses a strand for fetching, Select chooses a strand for decoding, renaming, and transfer to the Pick unit. Each cycle, in parallel with and independent of instruction fetch, Select determines which strand, among the ready strands, is the least-recently selected. A strand may not be ready due to per-strand wait conditions, such as an empty instruction buffer or a post-synchronizing[2] instruction pending, or due to pipeline-wide resource constraints, such as a lack of reorder buffer entries.

Select then reads up to 2 instructions per cycle from that strand's instruction buffers, and decodes and renames the instructions. As it decodes the instructions it identifies any intra-strand dependencies upon prior instructions, and enforces these dependencies until the instructions are sent to the Pick

1. Certain state registers are shared across strands to conserve hardware resources. These shared registers will (eventually) be listed in this Appendix.

2. A post-synchronizing instruction stalls instruction issue for the strand after issuing the post-synchronizing instruction until the instruction commits. Instructions which are post-synchronizing are listed in Section TABLE A-1, *SPARC M5 Instruction Latencies*, on page 97 below.

unit and written into the pick queue. Decode also assigns instructions to “slots”. There are 2 primary slots. Slot 0 is reserved for integer and load/store instructions. Slot 1 is reserved for integer, floating-point, graphics, cryptographic, and control transfer instructions. There is a third auxiliary slot, slot 2, which is reserved for store data operations.

A.1.3 Pick/Issue/Execute

Pick selects up to 2 instructions per cycle (an additional store data operation may also be picked) without regard to strand ID from a 36-entry out-of-order scheduler termed the pick queue. Instructions are written with a relative age in mind, so the pick queue picks the oldest ready instruction within a slot. An instruction is ready when all of its source data is available. Only one instruction can be picked for each slot each cycle. There are never any inter-strand instruction dependencies. As Pick issues instructions, pick queue entries are reclaimed, and made available for use by subsequent instructions coming from Select/decode/rename.

As Pick issues instructions to the execution units, the instructions execute in one of several functional units. There are 2 integer units, a floating-point and graphics unit, and a load/store unit. Each of these units has independent pipelines and operates in parallel with other execution units. When instructions finish execution, they report their completion status to the Commit unit.

A.1.4 Commit

Commit utilizes a 128-entry reorder buffer to hold completion status and other per-instruction information. Instructions commit once their completion status is available. Instructions which cause an exception complete, but do not commit. Instead, they trap, and the thread begins fetching instructions from the trap handler. Similarly, if a branch misprediction occurs, instruction fetching resumes from the correct path once the branch predictor has been updated, and execution resumes once all instructions prior to the mispredicted branch commit.

Commit is threaded and each cycle attempts to commit instructions from the least-recently-committed thread among the threads which are ready-to-commit.

A.1.5 Context Switching Between Strands

Since context switching is built into the SPARC M5 pipeline (via the instruction fetch, select/decode/rename, pick/issue/execute, and commit blocks), strands can be switched each cycle with no pipeline stall penalty.

A.1.6 Synchronization

Certain instructions require the pipeline to synchronize. One type of synchronization, post-synchronizing or post-sync’ing, puts the strand in a wait state at Select. The strand remains in a wait state, and subsequent instructions are not selected, decoded, or renamed until the post-sync clears. This is resolved by the commit of the post-sync’ing instruction.

A.2 Optimizing for Single-Threaded Performance or Throughput

Section 1.3.1.1, *Single-threaded and multi-threaded performance*, on page 31 describes some aspects of optimizing for single-threaded and/or multi-threaded performance.

A.3 Instruction Latency

TABLE A-1 lists the minimum single-strand instruction latencies for SPARC M5. When multiple strands are executing, some or much of the additional latency for multicycle instructions will be overlapped with execution of the additional strands.

A pre-sync'ing instruction waits at Pick for all prior instructions from the strand to commit before being picked; therefore these instructions have a variable latency, whose minimum is listed in TABLE A-1. A post-sync'ing instruction causes a flush after the instruction commits. Loads have a 5-cycle load-use delay (4 cycles need to be filled but out-of-order execution covers much of this latency in many cases). Branch instructions have a 2 cycle latency inthe branch unit but are fully pipelined.

TABLE A-1 SPARC M5 Instruction Latencies *(1 of 9)*

Opcode	Description	Latency	Post-sync	Notes
ADD (ADDcc)	Add (and modify condition codes)	1		
ADDC (ADDCcc)	Add with carry (and modify condition codes)	1		
ADDXC (ADDXCcc)	Add extended with carry (and modify condition codes)	1		
AES_DROUND01	AES decrypt round, columns 0 & 1	3		
AES_DROUND23	AES decrypt round, columns 2 & 3	3		
AES_DROUND01_LAST	AES decrypt last round, columns 0 & 1	3		
AES_DROUND23_LAST	AES decrypto last round, columns 2 & 3	3		
AES_EROUND01	AES encrypt round, columns 0 & 1	3		
AES_EROUND23	AES encrypt round, columns 2 & 3	3		
AES_EROUND01_LAST	AES encrypt last round, columns 0 & 1	3		
AES_EROUND23_LAST	AES encrypt last round, columns 2 & 3	3		
AES_KEXPAND0	AES key expansion without round constant	3		
AES_KEXPAND1	AES key expansion with round constant	3		
AES_KEXPAND2	AES key expansion without SBOX	3		
ALIGNADDRESS	Calculate address for misaligned data access	12		
ALIGNADDRESSL	Calculate address for misaligned data access (little-endian)	12		
ALLCLEAN	Mark all windows as clean	1		breaks decode group

TABLE A-1 SPARC M5 Instruction Latencies *(2 of 9)*

Opcode	Description	Latency	Post-sync	Notes
AND (ANDcc)	Logical **and** (and modify condition codes)	1		
ANDN (ANDNcc)	Logical **and not** (and modify condition codes)	1		
ARRAY{8,16,32}	3-D address to blocked byte address conversion	12		
Bicc	Branch on integer condition codes	2		
BMASK	Write the GSR.mask field	12		
BPcc	Branch on integer condition codes with prediction	2		
BPr	Branch on contents of integer register with prediction	2		
BSHUFFLE	Permute bytes as specified by the GSR.mask field	11		
CALL	Call and link	2		
CAMELLIA_F	Camellia F operation	3		
CAMELLIA_FL	Camellia FL operation	3		
CAMELLIA_FLI	Camellia FLI operation	3		
CASA	Compare and swap word in alternate space	20-30		Done in L2 cache
CASXA	Compare and swap doubleword in alternate space	20-30		Done in L2 cache
CBcond	Compare-and-Branch instructions	2		
CMASK{8,16,32}	Create GSR.mask from SIMD operation result	12		
DES_IP	DES initial permutation	3		
DES_IIP	DES inverse initial permutation	3		
DES_KEXPAND	DES key expansion	3		
DES_ROUND	DES round	3		
DONE	Return from trap	23		Causes flush and redirect to TNPC (23 cycle bubble)
EDGE{8,16,32}{L}{N}	Edge boundary processing {little-endian} {non-condition-code altering}	12		
FABS(s,d)	Floating-point absolute value	11		
FADD(s,d)	Floating-point add	11		
FALIGNDATA	Perform data alignment for misaligned data	11		
FANDNOT1{s}	Negated src1 **and** src2 (single precision)	11		
FANDNOT2{s}	src1 **and** negated src2 (single precision)	11		
FAND{s}	Logical **and** (single precision)	11		
FBPfcc	Branch on floating-point condition codes with prediction	1		
FBfcc	Branch on floating-point condition codes	1		
FCHKSM16	16-bit partitioned checksum	11		
FCMP(s,d)	Floating-point compare	11		
FCMPE(s,d)	Floating-point compare (exception if unordered)	11		
FCMPEQ{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 = src2	12		
FCMPGT{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 > src2	12		
FCMPLE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 ≤ src2	12		

TABLE A-1 SPARC M5 Instruction Latencies *(3 of 9)*

Opcode	Description	Latency	Post-sync	Notes
FCMPNE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 ≠ src2	12		
FDIV(s,d)	Floating-point divide	24 SP, 37 DP		
FEXPAND	Four 8-bit to 16-bit expand	11		
FHADD{s,d}	Floating-point add and halve	11		
FHSUB{s,d}	Floating-point subtract and halve	11		
FiTO(s,d)	Convert integer to floating-point	11		
FLUSH	Flush instruction memory	27	Y	Flushes pipeline, 27 cycle bubble minimum
FLUSHW	Flush register windows	1		breaks decode group
FLCMP{s,d}	Lexicographic compare	11		
FMADD{s,d}	Floating-point multiply-add single/double (fused)	11		
FMEAN16	16-bit partitioned average	11		
FMOV(s,d)	Floating-point move	11		
FMOV(s,d)cc	Move floating-point register if condition is satisfied	11		
FMOV(s,d)R	Move floating-point register if integer register contents satisfy condition	11		Cracked into 2 ops, breaks decode group
FMSUB{s,d}	Floating-point multiply-subtract single/double (fused)	11		
FMUL(s,d)	Floating-point multiply	11		
FMUL8SUx16	Signed upper 8- x 16-bit partitioned product of corresponding components	11		
FMUL8ULx16	Unsigned lower 8- x 16-bit partitioned product of corresponding components	11		
FMUL8x16	8- x 16-bit partitioned product of corresponding components	11		
FMUL8x16AL	Signed lower 8- x 16-bit lower α partitioned product of 4 components	11		
FMUL8x16AU	Signed upper 8- x 16-bit lower α partitioned product of 4 components	11		
FMULD8SUx16	Signed upper 8- x 16-bit multiply → 32-bit partitioned product of components	11		
FMULD8ULx16	Unsigned lower 8- x 16-bit multiply → 32-bit partitioned product of components	11		
FNADD(s,d)	Floating-point add and negate	11		
FNAND{s}	Logical **nand** (single precision)	11		
FNEG(s,d)	Floating-point negate	11		
FNHADD{s,d}	Floating-point add and halve, then negate	11		
FNMADD{s,d}	Floating-point add and negate	11		
FNMSUB{s,d}	Floating-point negative multiply-subtract single/double (fused)	11		
FNMUL{s,d}	Floating-point multiply and negate	11		

TABLE A-1 SPARC M5 Instruction Latencies *(4 of 9)*

Opcode	Description	Latency	Post-sync	Notes
FNsMULd	Floating-point multiply and negate	11		
FNOR{s}	Logical **nor** (single precision)	11		
FNOT1{s}	Negate (1's complement) src1 (single precision)	11		
FNOT2{s}	Negate (1's complement) src2 (single precision)	11		
FONE{s}	One fill (single precision)	11		
FORNOT1{s}	Negated src1 **or** src2 (single precision)	11		
FORNOT2{s}	src1 **or** negated src2 (single precision)	11		
FOR{s}	Logical **or** (single precision)	11		
FPACKFIX	Two 32-bit to 16-bit fixed pack	11		
FPACK{16,32}	Four 16-bit/two 32-bit pixel pack	11		
FPADD{16,32}{s}	Four 16-bit/two 32-bit partitioned add (single precision)	11		
FPADD64	Fixed-point partitioned add	11		
FPADDS{16,32}{s}	Fixed-point partitioned add	11		
FPMADDX	Unsigned integer multiply-add	11		
FPMADDXHI	Unsigned integer multiply-add, return high-order 64 bits of result	11		
FPMERGE	Two 32-bit to 64-bit fixed merge	11		
FPSUB{16,32}{s}	Four 16-bit/two 32-bit partitioned subtract (single precision)	11		
FPSUB64	Fixed-point partitioned subtract, 64-bit	11		
FPSUBS{16,32}{s}	Fixed-point partitioned subtract	11		
FSLL{16,32}	16- or 32-bit partitioned shift, left (old mnemonic FSHL)	11		
FSLAS{16,32}	16- or 32-bit partitioned shift, left or right (old mnemonic FSHLAS)	11		
FSRA{16,32}	16- or 32-bit partitioned shift, left or right (old mnemonic FSHRA)	11		
FSRL{16,32}	16- or 32-bit partitioned shift, left or right (old mnemonic FSHRL)	11		
FsMULd	Floating-point multiply single to double	11		
FSQRT(s,d)	Floating-point square root	24 SP, 37 DP		
FSRC1	Copy src1	11		
FSRC2{s}	Copy src1 (single precision)	11		
FSRC2	Copy src2	1		
FSRC2{s}	Copy src2 (single precision)	11		
F(s,d)TO(s,d)	Convert between floating-point formats	11		
F(s,d)TOi	Convert floating point to integer	11		
F(s,d)TOx	Convert floating point to 64-bit integer	11		
FSUB(s,d)	Floating-point subtract	11		
FUCMP{GT,LE,NE,EQ}8	Compare 8-bit unsigned fixed-point values	12		

TABLE A-1 SPARC M5 Instruction Latencies *(5 of 9)*

Opcode	Description	Latency	Post-sync	Notes
FXNOR{s}	Logical **xnor** (single precision)	11		
FXOR{s}	Logical **xor** (single precision)	11		
FxTO(s,d)	Convert 64-bit integer to floating-point	11		
FZERO{s}	Zero fill (single precision)	11		
ILLTRAP	Illegal instruction	23		
INVALW	Mark all windows as CANSAVE	1		breaks decode group
JMPL	Jump and link	2		
KASUMI_FI_XOR	Kasumi FI followed by XOR	3		
KASUMI_FI_FI	Kasumi FI followed by FI	3		
KASUMI_FL_XOR	Kasumi FL followed by XOR	3		
LDBLOCKF	64-byte block load	26		Cracked into 8 helper loads that reference L2
LDD	Load doubleword	20		
LDDA	Load doubleword from alternate space	20		Latency can be larger depending on ASI value
LDDF	Load double floating-point	5		
LDDFA	Load double floating-point from alternate space	5		
LDF	Load floating-point	5		
LDFA	Load floating-point from alternate space	5		
LDFSR	Load floating-point state register lower	variable	Y	
LDSB	Load signed byte	5		
LDSBA	Load signed byte from alternate space	5		
LDSH	Load signed halfword	5		
LDSHA	Load signed halfword from alternate space	5		
LDSTUB	Load-store unsigned byte	20-30		Done in L2 cache
LDSTUBA	Load-store unsigned byte in alternate space	20-30		Done in L2 cache
LDSW	Load signed word	5		
LDSWA	Load signed word from alternate space	5		
LDTW	Load twin word	20		breaks decode group
LDTWA	Load twin extended word	20		breaks decode group
LDTX	Load twin extended word	20		breaks decode group
LDTXA	Load twin extended word from alternate space	20		breaks decode group
LDUB	Load unsigned byte	5		
LDUBA	Load unsigned byte from alternate space	5		

TABLE A-1 SPARC M5 Instruction Latencies *(6 of 9)*

Opcode	Description	Latency	Post-sync	Notes
LDUH	Load unsigned halfword	5		
LDUHA	Load unsigned halfword from alternate space	5		
LDUW	Load unsigned word	5		
LDUWA	Load unsigned word from alternate space	5		
LDX	Load extended	5		
LDXA	Load extended from alternate space	variable if from nontrans lating ASI, else 5		
LDXFSR	Load extended floating-point state register	variable	Y	
LDXEFSR	Load extended floating-point state register	variable	Y	
LZD	Leading zero detect on 64-bit integer register	12		
MD5	MD5 hash	192	Y	
MEMBAR	Memory barrier	variable	membar #sync is post-sync'ing; other membar forms are not	
MOVcc	Move integer register if condition is satisfied	1		
MOVr	Move integer register on contents of integer register	1		breaks decode group
MOVdTOx	Move floating-point register to integer register	1		
MOVsTO{u,s}w	Move floating-point register to integer register	12		
MOVxTOd	Move integer register to floating-point register	1		
MOVwTOs	Move integer register to floating-point register	12		
MPMUL	Multiple-precision multiplication	variable	Y	pre-sync
MONTMUL	Montgomery multiplication	variable	Y	pre-sync
MONTSQR	Montgomery squaring	variable	Y	pre-sync
MULScc	Multiply step (and modify condition codes)	12		pre-sync
MULX	Multiply 64-bit integers	12		
NOP	No operation	1		
NORMALW	Mark other windows as restorable	1		breaks decode group
OR (ORcc)	Inclusive-**or** (and modify condition codes)	1		
ORN (ORNcc)	Inclusive-**or not** (and modify condition codes)	1		
OTHERW	Mark restorable windows as other	1		breaks decode group
PDIST	Distance between eight 8-bit components	11		
PDISTN	Pixel component distance	12		
POPC	Population count	12		

TABLE A-1 SPARC M5 Instruction Latencies *(7 of 9)*

Opcode	Description	Latency	Post-sync	Notes
PREFETCH	Prefetch data	1		
PREFETCHA	Prefetch data from alternate space	1		
RDASI	Read ASI register	variable	Y	
RDASR	Read ancillary state register	variable	Y	
RDCCR	Read condition codes register	variable	Y	
RDCFR	Read compatibility feature register	variable		
RDFPRS	Read floating-point registers state register	variable	Y	
RDPC	Read program counter (PC)	2		
RDPR	Read privileged register	variable	Y	
RDTICK	Read TICK register	variable	Y	
RDY	Read Y register	variable	Y	
RESTORE	Restore caller's window	1		breaks decode group
RESTORED	Window has been restored	1		breaks decode group
RETRY	Return from trap and retry	23		Causes flush and redirect to TPC (23 cycle bubble)
RETURN	Return	1		breaks decode group
SAVE	Save caller's window	1		breaks decode group
SAVED	Window has been saved	1		breaks decode group
SDIVcc	32-bit signed integer divide (and modify condition codes)	41-60		pre-sync
SDIVX{i}	64-bit signed integer divide	26-44		
SETHI	Set high 22 bits of low word of integer register	1		
SHA1	SHA-1 hash	226	Y	pre-sync
SHA256	SHA-256 hash	194	Y	pre-sync
SHA512	SHA-512 hash	242	Y	pre-sync
SHUTDOWN	(deprecated)	1		
SIAM	Set interval arithmetic mode	1		
SLL	Shift left logical	1		
SLLX	Shift left logical, extended	1		
SMUL (SMULcc)	Signed integer multiply (and modify condition codes)	12		
SRA	Shift right arithmetic	1		
SRAX	Shift right arithmetic, extended	1		
SRL	Shift right logical	1		
SRLX	Shift right logical, extended	1		
STB	Store byte	1		
STBA	Store byte into alternate space	1		

TABLE A-1 SPARC M5 Instruction Latencies *(8 of 9)*

Opcode	Description	Latency	Post-sync	Notes
STBAR	Store barrier	variable		
STBLOCKF	64-byte block store	8		
STD	Store doubleword	1		
STDA	Store doubleword into alternate space	1		
STDF	Store double floating-point	1		
STDFA	Store double floating-point into alternate space	1		
STF	Store floating-point	1		
STFA	Store floating-point into alternate space	1		
STFSR	Store floating-point state register	variable	Y	
STH	Store halfword	1		
STHA	Store halfword into alternate space	1		
STPARTIALF	Eight 8-bit/4 16-bit/2 32-bit partial stores	1		
STW	Store word	1		
STWA	Store word into alternate space	1		
STX	Store extended	1		
STXA	Store extended into alternate space	variable if from nontranslating ASI, else 1	depends upon ASI	
STXFSR	Store extended floating-point state register	variable	Y	pre-sync
SUB (SUBcc)	Subtract (and modify condition codes)	1		
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)	1		
SWAP	Swap integer register with memory	20-30		Done in L2 cache
SWAPA	Swap integer register with memory in alternate space	20-30		Done in L2 cache
TADDcc (TADDccTV)	Tagged add and modify condition codes (trap on overflow)	1		
TSUBcc (TSUBccTV)	Tagged subtract and modify condition codes (trap on overflow)	1		
Tcc	Trap on integer condition codes (with 8-bit sw_trap_number, if bit 7 is set, trap to hyperprivileged)	1 if no trap or 23 if trap taken		
UDIVcc	Unsigned integer divide (and modify condition codes)	41-60		pre-sync
UDIVX{i}	64-bit unsigned integer divide	26-44		
UMUL (UMULcc)	Unsigned integer multiply (and modify condition codes)	12		
UMULXHI	Unsigned 64 x 64 multiply, returning upper 64 product bits	12		
WRASI	Write ASI register	variable	Y	
WRASR	Write ancillary state register	variable	Y	
WRCCR	Write condition codes register	variable	Y	
WRFPRS	Write floating-point registers state register	variable	Y	

TABLE A-1 SPARC M5 Instruction Latencies *(9 of 9)*

Opcode	Description	Latency	Post-sync	Notes
WRPAUSE	Pause instruction	variable	Y	
WRPR	Write privileged register	variable	Y	
WRY	Write Y register	variable	Y	
XMULX{HI}	XOR multiply	12		
XNOR (XNORcc)	Exclusive-**nor** (and modify condition codes)	1		
XOR (XORcc)	Exclusive-**or** (and modify condition codes)	1		

A.4 Coding PAUSE loops

The WRPAUSE instruction can be used to place a strand in a paused state to temporarily suspend instruction execution on that strand. This is useful while implementing exponential backoff algorithms, for example.

In SPARC M5, care needs to be taken when coding WRPAUSE in loops. Dynamic branch instruction density[1] should not exceed 10% for a pause loop. That is, in the dynamic instruction count executed during the pause loop, a maximum of 10% of the instructions should be branches (taken or not taken). The loop may be padded with NOPs to achieve this density. This is a limitation of SPARC M5 processors, which will be addressed in future SPARC processors.

1. "Branch instruction", in this context, refers to any control transfer instruction (CTI) *except* DONE, RETRY, or Tcc

APPENDIX **B**

IEEE 754 Floating-Point Support

SPARC M5 conforms to Oracle SPARC Architecture 2011 and the corresponding IEEE Std 754-1985 Requirements chapter.

Note | SPARC M5 detects tininess before rounding.

B.1 Special Operand and Result Handling

The SPARC M5 FGU provides full hardware support for subnormal operands and results for all instructions. SPARC M5 never generates an unfinished_FPop trap type. SPARC M5 does not implement a non-standard floating-point mode. The NS bit of the FSR is always read as 0, and writes to it are ignored.

APPENDIX C

Differences Between SPARC T4 and SPARC M5

This chapter describes the differences between the earlier SPARC T4 and SPARC M5. A summary of the differences is provided in the table below.

Area	vs SPARC T4	Description
Architecture and Microarchitecture	Different	Section C.1
Data Format	Same	
Registers	Same	
Instruction Format	Same	
Instruction Definitions	Same	
Traps	Same	
Interrupt Handling	Different	Section C.2
Memory Models	Same	
Address Spaces & ASIs	Different	Section C.3
Performance Mesurement	Different	Section C.4
Crypto	Same	
MMU	Same	

C.1 Architectural and Microarchitectural Differences

SPARC M5 reuses the SPARC core from SPARC T4, the unified L3 cache is shared among six cores (vs eight in SPARC T4), and all the SPARC M5 SOC components are either re-designed or modified from SPARC T4.

SPARC M5 is capable of supporting up to 8 processors in a glue-less fashion and provides scalability ports for scaling beyond 8 processors.

For details, refer to the following chapters:

- For details of overall architectural and microarchitectural differences, see Chapter 1, *SPARC M5 Basics*.
- For details of all supported system configuration, see Chapter 34, *System Configurations*.
- For details of coherency and ordering protocol differences, see Chapter 30, *Coherency and Ordering Unit (COU)*.

C.2 Interrupt Handling Differences

Handling of I/O mondo interrupts is different in SPARC M5.

For details, refer to the following chapters:

- Interrupt Handling Chapter
- PCIE Chapter
- Non-cacheable Unit (NCU) Chapter

C.3 Address Spaces and ASIs Differences

C.3.1 ASIs

Addressing of all ASIs in SPARC core (including L2) does not change from SPARC T4 to SPARC M5.

See Address Spaces and ASIs Chapter for details.

C.3.2 CSRs

Addressing of all CSRs in SPARC core (including L2) does not change from SPARC T4 to SPARC M5.

See Address Spaces and ASIs Chapter for details.

APPENDIX **D**

Cache Coherency and Ordering

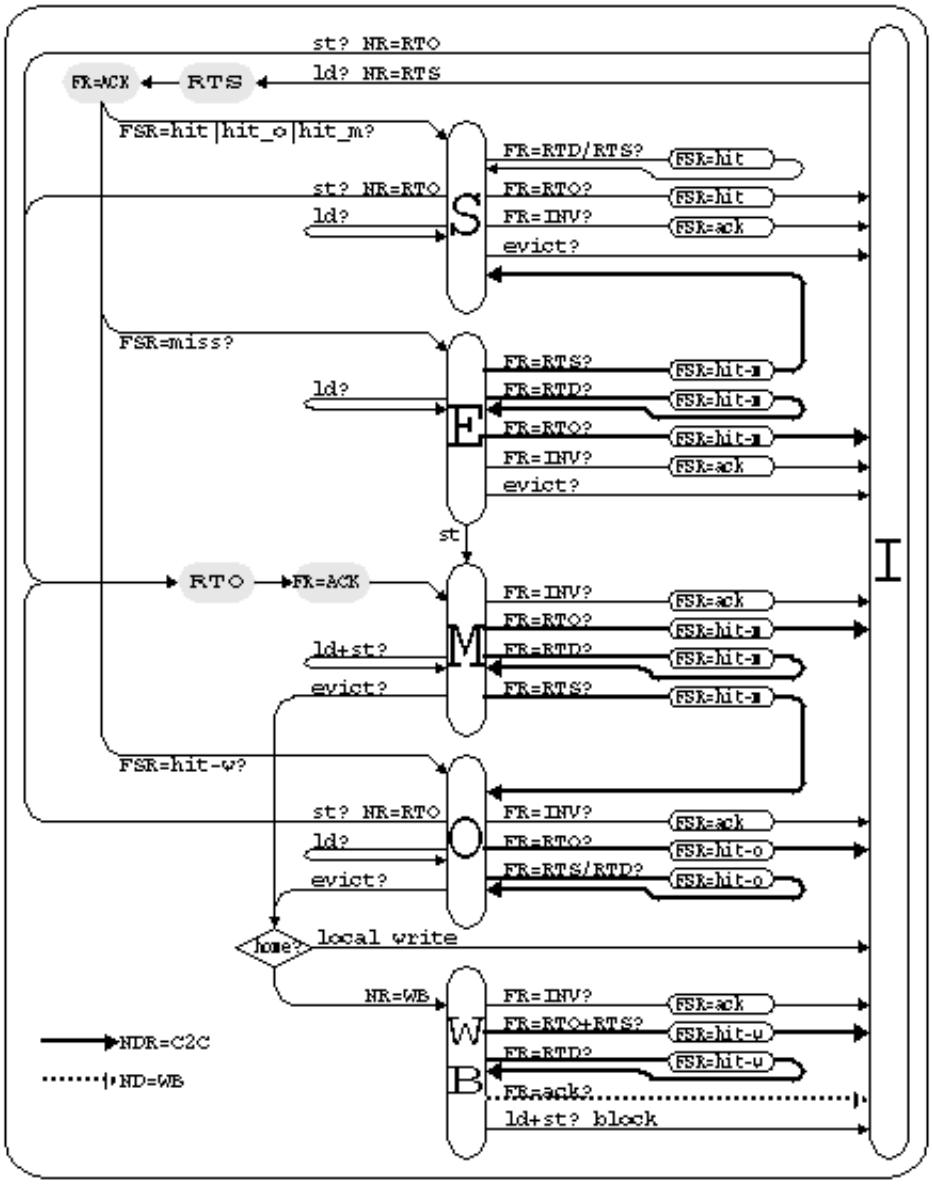
D.1 Cache and Memory Interactions

This appendix describes various interactions between the caches and memory, and the management processes that an operating system must perform to maintain data integrity in these cases. In particular, it discusses the following:

- Invalidation of one or more cache entries—when and how to do it
- Differences between cacheable and noncacheable accesses
- Ordering and synchronization of memory accesses
- Accesses to addresses that cause side effects (I/O accesses)
- Nonfaulting loads
- Cache sizes, associativity, replacement policy, etc.

D.2 Coherency Overview

Please refer to “VF Link ERS, Coherency Chapter”.



D.3 Cache Flushing

Data in the level-1 (read-only or writethrough) caches can be flushed by invalidating the entry in the cache (in a way that also leaves the L2 directory in a consistent state). Modified data in the level-2 (writeback) cache must be written back to memory when flushed.

Cache flushing is required in the following cases:

- **I-cache:** Flush is needed before executing code that is modified by a local store instruction. This is done with the FLUSH instruction, which just forces previous stores to complete to all affected caches.. Flushing the I-cache with ASI accesses (Section 23.5, *L1 I-Cache Diagnostic Access*, on page 1489) also works, because the L2 directory correctly handles the cases where the directory thinks the line is in the L1, but the L1 doesn't.
- **D-cache:** Flush is needed when a physical page is changed from (physically) cacheable to (physically) noncacheable. This is done with a displacement flush (*Displacement Flushing*, below), or with ASI accesses (see Section 20.10, *L1 I-Cache Diagnostic Access*, on page 90), which work for similar reasons as for the I-cache.
- **L2 cache:** Flush is needed for stable storage. Examples of stable storage include battery-backed memory and transaction logs. The recommended way to perform this is by using the PrefetchICE instruction (see Section 20.19, *L3 Index and Bank Hashing*, on page 109). Alternatively, this can be done by a displacement flush (see the next section). Flushing the L2 cache flushes the corresponding blocks from the I- and D-caches, because SPARC M5 maintains inclusion between the L2 and L1 caches.
- **Errors:** Flush is needed for error processing. Examples include (1) forcing UE data from a cache to memory, in order to convert it to NotData, or (2) using flushes to force memory (not cache) reads and writes, to diagnose a memory error, or (3) writing a line of good data and flushing it to memory, to overwrite a memory soft error.

D.3.1 Displacement Flushing

Cache flushing of the L2 cache or the D-cache can be accomplished by a displacement flush. This is done by placing the cache in direct-map mode, and reading a range of read-only addresses that map to the corresponding cache line being flushed, forcing out modified entries in the local cache. Care must be taken to ensure that the range of read-only addresses is mapped in the MMU before starting a displacement flush; otherwise, the TLB miss handler may put new data into the caches. In addition, the range of addresses used to force lines out of the cache must not be present in the cache when starting the displacement flush. (If any of the displacing lines are present before starting the displacement flush, fetching the already present line will *not* cause the proper way in the direct-mapped mode L2 to be loaded; instead, the already present line will stay at its current location in the cache.)

Note | Diagnostic accesses to the L2 cache can be used to invalidate a line, but they are not an alternative to PrefetchICE or displacement flushing. L2 diagnostic accesses do not cause invalidation of L1 lines (breaking L1 inclusion) and modified data in the L2 cache will not be written back to memory using these ASI accesses. See Section 20.22, *L2 Cache Diagnostic Access*, on page 111.

Architectural Note – Does direct-mapped mode still work if any L2 lines are mapped out? If so, displacement flushing may still be possible, but involves clearing all U bits, then forcing at least 6 MB of misses.

D.3.2 Memory Accesses and Cacheability

Note | Atomic load-store instructions are treated as both a load and a store; they can be performed only in cacheable address spaces.

In SPARC M5, all memory accesses are cached in the L2 cache (as long as the L2 cache is enabled). The cp bit in the TTE corresponding to the access controls whether the memory access will be cached in the primary caches (if cp = 1, the access is cached in the primary caches; if cp = 0 the access is not cached in the primary caches). Atomic operations are always performed at the L2 cache.

D.3.3 Coherence Domains

Two types of memory operations are supported in SPARC M5: cacheable and noncacheable accesses, as indicated by the page translation. Cacheable accesses are inside the coherence domain; noncacheable accesses are outside the coherence domain.

SPARC V9 does not specify memory ordering between cacheable and noncacheable accesses. SPARC M5 maintains TSO ordering, regardless of the cacheability of the accesses, relative to other access by processors.

Programming Note – Ordering of processor accesses relative to DMA accesses roughly follows PCI ordering rules for PCI devices.

See the *The SPARC Architecture Manual-Version 9* for more information about the SPARC V9 memory models.

On SPARC M5, a MEMBAR #Lookaside is effectively a NOP and is not needed for forcing order of stores vs. loads to noncacheable addresses.

D.3.3.1 Cacheable Accesses

Accesses that fall within the coherence domain are called cacheable accesses. They are implemented in SPARC M5 with the following properties:

- Data resides in real memory locations.
- They observe the supported cache coherence protocol.
- The unit of coherence is 64 bytes at the system level (coherence between the virtual processors and I/O), enforced by the L2 cache.
- The unit of coherence for the primary caches (coherence between multiple virtual processors) is the primary cache line size (16 bytes for the data cache, 32 bytes for the instruction cache), enforced by the L2 cache directories.

D.3.3.2 Noncacheable and Side-Effect Accesses

Accesses that are outside the coherence domain are called noncacheable accesses. Accesses of some of these memory (or memory mapped) locations may result in side effects. Noncacheable accesses are implemented in SPARC M5 with the following properties:

- Data may or may not reside in real memory locations.
- Accesses may result in program-visible side effects; for example, memory-mapped I/O control registers in a UART may change state when read.
- Accesses may not observe supported cache coherence protocol.
- The smallest unit in each transaction is a single byte.

Noncacheable accesses are all strongly ordered with respect to other noncacheable accesses (regardless of the e bit). Speculative loads with the e bit set cause a *DAE_so_page* trap.

Note | The side-effect attribute does not imply noncacheability.

D.3.3.3 Global Visibility and Memory Ordering

To ensure the correct ordering between the cacheable and noncacheable domains, explicit memory synchronization is needed in the form of MEMBARs or atomic instructions. CODE EXAMPLE D-1 illustrates the issues involved in mixing cacheable and noncacheable accesses.

CODE EXAMPLE D-1 Memory Ordering and MEMBAR Examples

Assume that all accesses go to non-side-effect memory locations.

```
Process A:
While (1)
{


   Store D1:data produced
1  MEMBAR #StoreStore (needed in PSO, RMO)
   Store F1:set flag
   While F1 is set (spin on flag)
   Load F1
2 MEMBAR #LoadLoad | #LoadStore (needed in RMO)

  Load D2
}

Process B:
While (1)
{

  While F1 is cleared (spin on flag)

      Load F1
2 MEMBAR #LoadLoad | #LoadStore (needed in RMO)

  Load D1

  Store D2
1 MEMBAR #StoreStore (needed in PSO, RMO)

Store F1:clear flag
}
```

Note | A MEMBAR #MemIssue or MEMBAR #Sync is needed if ordering of cacheable accesses following noncacheable accesses must be maintained for RMO cacheable accesses.

Due to load and store buffers implemented in SPARC M5, CODE EXAMPLE D-1 may not work for RMO accesses without the MEMBARs shown in the program segment.

Under TSO, loads and stores (except block stores) cannot pass earlier loads, and stores cannot pass earlier stores; therefore, no MEMBAR is needed.

Under RMO, there is no implicit ordering between memory accesses; therefore, the MEMBARs at both #1 and #2 are needed.

D.3.4 Memory Synchronization: MEMBAR and FLUSH

The MEMBAR (STBAR in SPARC V8) and FLUSH instructions provide for explicit control of memory ordering in program execution. MEMBAR has several variations; their implementations in SPARC M5 are described below. See the references to "Memory Barrier," "The MEMBAR Instruction," and "Programming With the Memory Models," in *The The SPARC Architecture Manual-Version 9* for more information.

D.3.4.1 MEMBAR #LoadLoad

All loads on SPARC M5 switch a strand out until the load completes. Thus, MEMBAR #LoadLoad is treated as a NOP on SPARC M5.

D.3.4.2 MEMBAR #StoreLoad

MEMBAR #StoreLoad forces all loads after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility. MEMBAR #StoreLoad behaves the same as MEMBAR #Sync on SPARC M5.

D.3.4.3 MEMBAR #LoadStore

All loads on SPARC M5 switch a strand out until the load completes. Thus, MEMBAR #LoadStore is treated as a NOP on SPARC M5

D.3.4.4 MEMBAR #StoreStore and STBAR

Stores on SPARC M5 maintain order in the store buffer. Thus Membar #StoreStore is treated as a NOP on SPARC M5.

Notes

STBAR has the same semantics as MEMBAR #StoreStore; it is included for SPARC-V8 compatibility.

SPARC M5 block stores and block-init stores are RMO. If a program needs to maintain order between RMO stores to different L2 cache lines, it should use a MEMBAR #Sync.

D.3.4.5 MEMBAR #Lookaside

Loads and stores to noncacheable addresses are "self-synchronizing" on SPARC M5. Thus MEMBAR #Lookaside is treated as a NOP on SPARC M5.

Note

For SPARC V9 compatibility, this variation should be used before issuing a load to an address space that cannot be snooped,

D.3.4.6 MEMBAR #MemIssue

MEMBAR #MemIssue forces all outstanding memory accesses to be *completed* before any memory access instruction after the MEMBAR is issued. It must be used to guarantee ordering of cacheable accesses following noncacheable accesses. For example, I/O accesses must be followed by a MEMBAR #MemIssue before subsequent cacheable stores; this ensures that the I/O accesses reach global visibility (as viewed by other strands) before the cacheable stores after the MEMBAR.

Since loads are already self-synchronizing, Membar #MemIssue just needs to drain the store buffer (and receive all the store ACKs) before allowing memory operations to issue again. This is the same operation as SPARC M5's Membar #Sync.

D.3.4.7 MEMBAR #Sync (Issue Barrier)

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

Note | MEMBAR #Sync is a costly instruction; unnecessary usage may result in substantial performance degradation.

D.3.4.8 Self-Modifying Code (FLUSH)

The SPARC V9 instruction set architecture does not guarantee consistency between code and data spaces. A problem arises when code space is dynamically modified by a program writing to memory locations containing instructions. Dynamic optimizers, LISP programs, and dynamic linking require this behavior. SPARC V9 provides the FLUSH instruction to synchronize instruction and data memory after code space has been modified.

In SPARC M5, FLUSH behaves like a store instruction for the purpose of memory ordering. In addition, all instruction fetch (or prefetch) buffers are invalidated. The issue of the FLUSH instruction is delayed until previous (cacheable) stores are completed. Instruction fetch (or prefetch) resumes at the instruction immediately after the FLUSH.

D.3.5 Atomic Operations

SPARC V9 provides three atomic instructions to support mutual exclusion. These instructions behave like both a load and a store but the operations are carried out indivisibly. Atomic instructions may be used only in the cacheable domain.

An atomic access with a restricted ASI in unprivileged mode (PSTATE.priv = 0) causes a *privileged_action* trap. An atomic access with a noncacheable address causes a *data_access_exception* trap (with SFSR.ft = 4, atomic to page marked noncacheable). An atomic access with an unsupported ASI causes a *DAE_invalid_ASI* trap. TABLE D-1 lists the ASIs that support atomic accesses.

TABLE D-1 ASIs That Support SWAP, LDSTUB, and CAS

ASI Name
ASI_NUCLEUS{_LITTLE}
ASI_AS_IF_USER_PRIMARY{_LITTLE}
ASI_AS_IF_USER_SECONDARY{_LITTLE}
ASI_PRIMARY{_LITTLE}
ASI_SECONDARY{_LITTLE}
ASI_REAL{_LITTLE}

Notes | Atomic accesses with nonfaulting ASIs are not allowed, because these ASIs have the load-only attribute.

For all atomics, allocation is done to the L2 cache only and will invalidate the L1s.

D.3.5.1 SWAP Instruction

SWAP atomically exchanges the lower 32 bits in an integer register with a word in memory. This instruction is issued only after store buffers are empty. Subsequent loads interlock on earlier SWAPs.

D.3.5.2 LDSTUB Instruction

LDSTUB behaves like SWAP, except that it loads a byte from memory into an integer register and atomically writes all 1's (FF_{16}) into the addressed byte.

D.3.5.3 Compare and Swap (CASX) Instruction

Compare-and-swap combines a load, compare, and store into a single atomic instruction. It compares the value in an integer register to a value in memory; if they are equal, the value in memory is swapped with the contents of a second integer register. All of these operations are carried out atomically; in other words, no other memory operation may be applied to the addressed memory location until the entire compare-and-swap sequence is completed.

D.3.6 Nonfaulting Load

A nonfaulting load behaves like a normal load, except that

- It does not allow side-effect access. An access with the e bit set causes a *DAE_so_page* trap.
- It can be applied to a page with the nfo bit set; other types of accesses will cause a *DAE_NFO_page* trap.

Nonfaulting loads are issued with `ASI_PRIMARY_NO_FAULT{_LITTLE}` or `ASI_SECONDARY_NO_FAULT{_LITTLE}`. A store with a `NO_FAULT` ASI causes a *DAE_invalid_ASI* trap.

When a nonfaulting load encounters a TLB miss, the operating system should attempt to translate the page. If the translation results in an error (for example, address out of range), a 0 is returned and the load completes silently.

Typically, optimizers use nonfaulting loads to move loads before conditional control structures that guard their use. This technique potentially increases the distance between a load of data and the first use of that data, to hide latency; it allows for more flexibility in code scheduling. It also allows for improved performance in certain algorithms by removing address checking from the critical code path.

For example, when following a linked list, nonfaulting loads allow the null pointer to be accessed safely in a read-ahead fashion if the operating system can ensure that the page at virtual address 0_{16} is accessed with no penalty. The nfo (nonfault access only) bit in the MMU marks pages that are mapped for safe access by nonfaulting loads but can still cause a trap by other, normal accesses. This allows programmers to trap on wild pointer references (many programmers count on an exception being generated when accessing address 0_{16} to debug code) while benefitting from the acceleration of nonfaulting access in debugged library routines.

D.4 L1 I-Cache

The L1 Instruction cache is 16 Kbytes, physically tagged and indexed, with 32-byte lines, and 8-way associative with pseudo-random replacement. The format used to index the cache is shown in TABLE D-2.

TABLE D-2 L1 Instruction Cache Addressing

Bit	Field	Description
39:11	tag	Tag for cache line.
10:5	set	Selects cache set containing the cache line.
4:2	instr	Selects 32-bit instruction in cache line.
1:0	—	Always 0 for access to 32-bit instructions.

D.4.1 LFSR Replacement Algorithm

Details TBD.

D.4.2 Direct-Mapped Mode

The I-cache direct-mapped mode (see Section 20.9.1, *ASI_DC_DIRECT_MAP_REG*, on page 89) works by forcing all replacements to the "way" identified by bits [13:11] of the virtual address. Since lines already present are not affected but only new lines brought into the cache are affected, it is safe to turn on (or off) the direct-mapped mode at any time.

D.4.3 I-Cache Disable

Clearing the I-cache enable bit (see Section 5., *ASI changes for SPARC VT core*, on page 6) stops all accesses to the I-cache for that strand. All fetches will miss, and the returned data will not fill the I-cache. Invalidates will still be serviced while the
I-cache is disabled.

D.5 L1 D-Cache

The L1 Data cache is 8 Kbytes, writethrough, physically tagged and indexed, with 16-byte lines, and 4-way associative with true LRU replacement. The format used to index the cache is shown in TABLE D-3.

TABLE D-3 L1 Data Cache Addressing

Bit	Field	Description
39:11	tag	Tag for cache line.
10:4	set	Selects cache set containing the cache line.
3:0	data	Selects data byte(s) in cache line.

D.5.1 LRU Replacement Algorithm

The D-cache replacement algorithm is true least-recently-used (LRU). Six bits are maintained for each cache index.

D.5.2 Direct-Mapped Mode

The D-cache direct-mapped mode (see Section 20.9.1, *ASI_DC_DIRECT_MAP_REG*, on page 89) works by changing the replacement algorithm from LRU to instead use two bits of index (address[12:11]) to select the "way." Since lines already present are not affected but only new lines brought into the cache are affected, it is safe to turn on (or off) the direct-mapped mode at any time.

Note that if the D-cache is in direct-mapped mode, and a parity error occurs, the way replaced will be the way which experienced the parity error. This overrides the index selected by the address in direct-mapped mode.

D.5.3 D-Cache Disable

The D-cache enable bit (see Section 5., *ASI changes for SPARC VT core*, on page 6) works by modifying the replacement algorithm, by forcing all D-cache misses to be nonallocating. Thus, dc = 0 has no effect if a line is already in the cache (it hits anyway), but only affects D-cache misses. Stores that hit in the L1 will be performed in the L2, then update the L1 (as normal).

To get the D-cache fully disabled, the dc bit must be off on all strands in the virtual processor, and the D-cache must be flushed in a way that doesn't bring new lines back in. This can be done by storing (from a different core) to each line that is in the D-cache, or by displacement flushing the L2 cache so that inclusion will force all
D-cache lines to be invalidated.

D.6 L2 Cache

APPENDIX **E**

Glossary

This chapter defines concepts and terminology unique to the SPARC M5 implementation. Definitions of terms common to all Oracle SPARC Architecture implementations may be found in the Definitions chapter of *Oracle SPARC Architecture 2011*.

ALU Arithmetic Logical Unit

architectural state Software-visible registers and memory (including caches).

ARF Architectural register file.

blocking ASI An ASI access that accesses its ASI register or array location once all older instructions in that strand have retired, no instructions in the other strand can issue, and the store queue, TSW, and LMB are all empty.

branch outcome A reference as to whether or not a branch instruction will alter the flow of execution from the sequential path. A taken branch outcome results in execution proceeding with the instruction at the branch target; a not-taken branch outcome results in execution proceeding with the instruction along the sequential path after the branch.

branch resolution A branch is said to be resolved when the result (that is, the branch outcome and branch target address) has been computed and is known for certain. Branch resolution can take place late in the pipeline.

branch target address The address of the instruction to be executed if the branch is taken.

commit An instruction commits when it modifies architectural state.

complex instruction A complex instruction is an instruction that requires the creation of secondary "helper" instructions for normal operation, excluding trap conditions such as spill/fill traps (which use helpers). Refer to*Instruction Latency* on page 97 for a complete list of all complex instructions and their helper sequences.

consistency *See* **coherence**.

CPU Central Processing Unit. A synonym for **virtual processor**.

CSR Control Status register.

FP Floating point.

L2C (or L2$) Level 2 cache.

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

nonblocking ASI A nonblocking ASI access will access its ASI register/array location once all older instructions in that strand have retired, and there are no instructions in the other strand which can issue.

older instruction Refers to the relative fetch order of instructions. Instruction *i* is older than instruction *j* if instruction *i* was fetched before instruction *j*. Data dependencies flow from older instructions to younger instructions, and an instruction can only be dependent upon older instructions.

one hot An n-bit binary signal is one hot if and only if $n - 1$ of the bits are each zero and a single bit is a 1.

quadlet

SIAM Set interval arithmetic mode instruction.

younger instruction *See* **older instruction**.

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

APPENDIX **F**

Bibliography

[contents of this appendix are TBD]

Index

D

E

F

G

H

I

J