The SPARC™ Architecture Manual

Version 7

SOLBOURNE COMPUTER, Inc.
2190 Miller Drive
Longmont, Colorado 80501
303 772-3400

For Solbourne Support Call: 1-800-447-2861

Part Number: 101482-00

September 1988

Reprinted by permission.

TABLE OF CONTENTS

SECTION 1: INTRODUCTION

1.1. Introduction

This manual describes version 7 of the SPARC architecture, Sun Microsystems' 32-bit RISC architecture. This architecture makes possible implementations that can execute instructions for high-level language programs at rates approaching 1 instruction per processor clock. It supports a floating-point coprocessor with multiple arithmetic units and a second, implementation-definable coprocessor.

1.2. Architecture and Implementation

This document provides a specification for the SPARC architecture; it describes the major aspects of that architecture. Any design which conforms to this specification is an implementation; aspects of the design that are not specified in this document are implementation-dependent. For example, the SPARC architecture defines a set of instructions, a set of registers, how the registers work, and how traps and interrupts work. It does **not** define details such as the size and timing of data and address busses, caches, or memory management units.

Specific information about Sun Microsystems' implementations of the SPARC architecture appear in companion manuals.

1.3. Features

The SPARC architecture provides the following features:

- Simple instructions — Most instructions require only a single arithmetic operation.
- Few and simple instruction formats — All instructions are 32 bits wide, and are aligned on 32-bit boundaries in memory. There are only three basic instruction formats, and they feature uniform placement of opcode and register address fields.
- Register-intensive architecture — Most instructions operate on either two registers or one register and a constant, and place the result in a third register. Only load and store instructions access storage.
- A large "windowed" register file — The processor has access to a large number of registers configured into several overlapping sets. This scheme allows compilers to cache local values across subroutine calls, and provides a register-based parameter passing mechanism.
- Delayed control transfer — The processor always fetches the next instruction after a control transfer, and either executes it or annuls it, depending on the transfer's "annul" bit. Compilers can rearrange code to place a useful instruction after a delayed control transfer and thereby take better advantage of the processor's pipeline.
- One-cycle execution — To take maximum advantage of the SPARC architecture, the memory system should be able to fetch instructions at an average rate of one per processor cycle. This allows most instructions to execute in one cycle.
- Concurrent floating point — Floating-point operate instructions can execute concurrently with each other and with other non-floating-point instructions.

- Coprocessor interface — The architecture supports a simple coprocessor interface. The coprocessor instruction set is analogous to the floating-point instruction set.

1.4. Using This Manual

This section provides information to help you use this manual. It includes an overview of the manual, a definition of the intended audience, a description of the fonts used and what they mean, a glossary, and a list of references.

1.4.1. Contents

The section after this contains an overview of the SPARC architecture. This is followed by sections that describe the registers, then the instructions, and finally, trapping and exceptions.

A series of appendices follow the sections. The most important is Appendix B, *Instruction Descriptions.* This contains a complete description of every instruction that the architecture supports, and includes tables showing the recommended assembly language syntax for each instruction. Another appendix contains tables detailing all the opcodes and condition codes, and another contains ISP description language for all the instructions plus other architecture functions.

1.4.2. Fonts in Text

In this manual, we use the following fonts to make things clearer:

- Roman font is the normal font used for text.
- *Italic font* represents either a register class or a field name. For example:

  "The *rs1* field contains the address of the *r register.*"

  It is also used for regular notes, and for references to sections, sections or appendices in this manual, or to other documents.
- *Typewriter font* is used for the names of certain signals that are defined in the section *SPARC Architecture Overview*, and for literals in the appendix *Suggested Assembly Language Syntax.* These signal names appear in *typewriter font*, and contain underbar characters in the spaces between the words in the name. For example:

  The signal *bp_reset_in* indicates that the system is requesting a reset.
- **Bold font** indicates that a word or phrase requires emphasis. For example:

  "The delay instruction occurs immediately **after** a control transfer".
- UPPER CASE items may be either acronyms or instruction names. The most common acronyms appear in the glossary in this section, and the instructions are all listed by name in *Appendix B.* Note that names of some instructions contain both upper case and lower case letters.
- Underbar characters between two or more words mean that the words represent an identifier, which may be a trap, or some other condition. These appear in ordinary text as well as in the pseudocode examples in the appendices. For example:

  "The IU acknowledges the exception by taking an fp_exception trap."

1.4.3. Notes

This manual provides three types of notes: ordinary notes, programming notes, and implementation notes.

- Ordinary notes contain incidental information about the current subject; they appear in *italic font.*
- Programming notes contain incidental information about programming using the SPARC architecture; they appear in reduced pitch Roman font.
- Implementation notes contain information which may be specific to an implementation or which may differ in different implementations. They also appear in reduced pitch Roman font.

1.4.4. Glossary

The following paragraphs list and describe some of the most important words and acronyms used in this manual:

- Architecture/implementation — The architecture is the set of operating principles defined in this manual. An implementation is any specific design that conforms to the architecture defined here.
- Current window — The block of 24 *r registers* currently pointed to by the CWP.
- Current Window Pointer (CWP) — Selects the current register window.
- Delay instruction — The instruction immediately following a control transfer. This instruction is always fetched, and is either executed or annulled before the control transfer takes place.
- Floating-Point Unit (FPU) — The coprocessor that performs floating-point calculations.
- Floating-Point Arithmetic Unit (FAU) — A subsection of the FPU that executes floating-point operate instructions.
- Floating-Point Operate (FPop) instruction — An instruction that performs a floating-point calculation. They do **not** include loads and stores between memory and the FPU.
- Floating-Point Queue (FQ) — The queue where information about floating-point operate instructions is held while they are being executed by the FPU.
- *f register* — One of the 32 FPU working registers.
- Global registers — A block of 8 registers that are available regardless of the value of the current window pointer.
- Integer Unit (IU) — The main computing engine. It fetches all instructions, and executes all but FPop and CPop instructions.
- Next Program Counter (nPC) — Contains the address of the instruction to be executed next (assuming a trap does not occur).
- Processor — The combination of the IU and FPU.
- Processor State Register (PSR) — The IU's status register.
- Program Counter (PC) — Contains the address of the current instruction being executed by the IU.
- *r register* — A global register or a register in the IU's current window.
- *rd, rs1* and *rs2* — Fields in instructions. These specify the register operands of an instruction. *rd* is the destination register and *rs1* and *rs2* are the source registers.

- r[rd], r[rs1] and r[rs2] — The *r registers* specified by *rd, rs1* and *rs2*.
- Word — A word is 32 bits.

1.4.5. References

For additional information about RISC architecture, see:

- "Reduced Instruction Set Computers", Communications of the ACM, Volume 28, Number 1, January, 1985 by Dave Patterson.

SECTION 2: SPARC ARCHITECTURE OVERVIEW

2.1. Introduction

The SPARC architecture is used in 32-bit Reduced Instruction Set Computers (RISCs). It provides an Integer Unit (IU) to perform basic processing and a Floating-Point Unit (FPU) to perform floating-point calculations concurrently with the IU. It also provides instruction set support for an optional coprocessor. The details of the coprocessor itself are implementation-specific.

A typical system that uses the SPARC architecture is organized around a 32-bit virtual address bus and a 32-bit instruction/data bus. Its storage subsystem consists of a memory management unit (MMU) and a large cache for both instructions and data. The cache is virtual-address-based. Depending on the storage subsystem's interpretation of the processor's address space identifier (*asi*) bits, I/O registers are either addressed directly, bypassing the MMU, or they are mapped by the MMU into virtual addresses.

2.2. IU, FPU, and CP

The IU is the basic processing engine of the SPARC architecture. It executes all the instruction set except floating-point operate instructions and coprocessor instructions. A block diagram of the IU appears in Figure 2-1.

The FPU performs floating-point arithmetic using several floating-point arithmetic units (FAUs) to perform the actual calculations. The number of these units, which is implementation-dependent, determines the minimum number of floating-point operate instructions that can be executed at the same time.

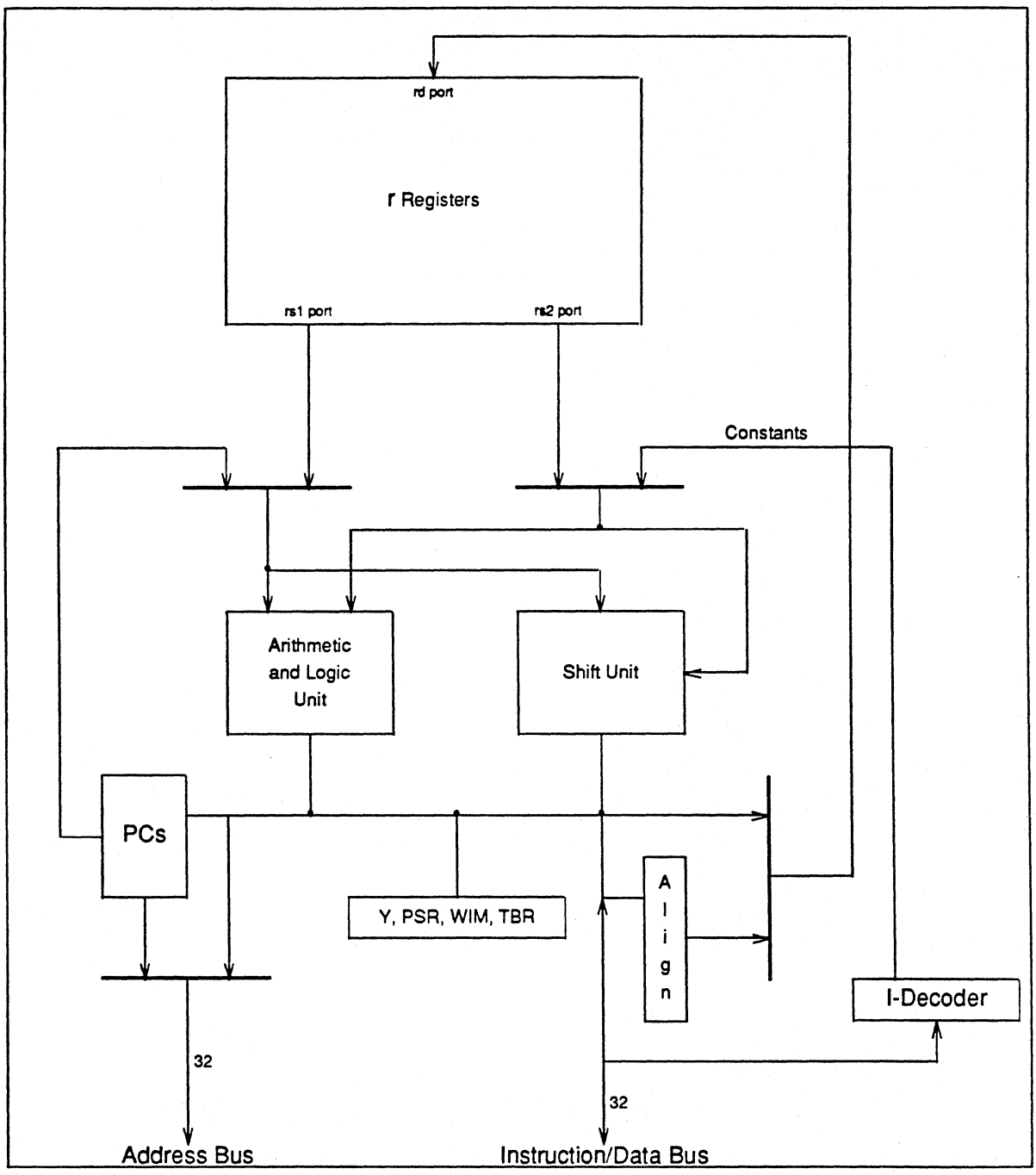
The FPU and the IU operate concurrently. The FPU recognizes floating-point operate instructions and places them into a queue. Meanwhile, the IU continues to execute instructions. Floating-point operate instructions are executed from the queue when the specified floating-point registers are free and the required FAU is available. If the FPU encounters a floating-point operate instruction that doesn't fit in the queue, the IU stalls until the required FPU resource becomes available.

Floating-point load/store instructions are used to move data between the FPU and memory. The IU generates a memory address and the FPU either sources or sinks the data. **Note that floating-point loads and stores are not floating-point operate instructions.**

The architecture hides floating-point concurrency from the programmer, so the implementation must provide the appropriate register interlocks. **A program including floating-point computations generates the same results as if all instructions were executed sequentially.**

The architecture supports an optional coprocessor. Like the FPU, the coprocessor recognizes coprocessor arithmetic instructions, and executes them concurrently with instructions executed by the IU.

Likewise, coprocessor load/store instructions are used to move data between the coprocessor and memory. For each floating-point load/store instruction, there is an analogous coprocessor load/store instruction.


rd port
r Registers
rs1 port
rs2 port
Constants
Arithmetic and Logic Unit
Shift Unit
PCs
Y, PSR, WIM, TBR
A l i g n
I-Decoder
32
32
Address Bus
Instruction/Data Bus

2.3. Registers

The register structure forms an important part of the overall architecture. The IU's working registers are divided into several windows, each with twenty-four 32-bit working registers, and each having access to the same eight 32-bit global registers. The current window pointer (CWP) field in the processor state register (PSR) keeps track of which window is currently "active".

In addition to the window registers and global registers, the SPARC architecture provides several control and status registers, and a non-windowed working register file for the FPU.

2.4. Multitasking Support

The SPARC architecture supports a multitasking operating system by providing user and supervisor modes. Some instructions are privileged, and can only be executed while the processor is in supervisor mode. Changing from user to supervisor mode requires taking a hardware trap, or using a trap instruction.

2.5. Instruction Categories

Instructions fall into six basic categories:

1 Load and store

2 Arithmetic/logical/shift

3 Control-transfer

4 Read/write control register

5 Floating-point operate

6 Coprocessor operate

The following sections describe each briefly; for more detail, see the section *Instructions*.

2.5.1. Load and Store Instructions

Load and store instructions are the only instructions that access memory. They use two IU registers or an IU register and a signed immediate value to calculate the memory address. The instruction's destination field specifies either an IU register, FPU register, or coprocessor register; this register supplies the data for a store, or receives the data from a load.

Integer load and store instructions support byte, halfword (16-bit), word (32-bit), and doubleword (64-bit) accesses. Floating-point and coprocessor load and store instructions support word and doubleword memory accesses. Halfword accesses must be aligned on a 2-byte boundary, word accesses must be aligned on a 4-byte boundary, and doubleword accesses must be aligned on an 8-byte boundary. Improperly aligned addresses cause load or store instructions to trap.

The order of bytes, halfwords, and words appears in Figure 4-2.

2.5.2. Arithmetic/Logical/Shift

These instructions (with one exception) compute a result that is a function of two source operands; they either write the result into a destination register or discard it. They perform arithmetic, tagged arithmetic, logical, or shift operations. The exception is a specialized instruction used to create 32-bit constants in two instructions.

Shift instructions can be used to shift the contents of a register left or right, by a distance specified by the instruction or by an IU register.

The tagged arithmetic instructions assume that the least-significant two bits of the operands are tags and set a condition code bit if they are not zero.

2.5.3. Control-Transfer Instructions

Control-transfer instructions include jumps, calls, traps, and branches. Control transfer is usually delayed so that the instruction immediately following the control transfer is executed before control actually transfers to the target address. The instruction following the control-transfer instruction is called a **delay instruction**. The delay instruction is always fetched, even when the control transfer is an unconditional branch. However, a bit in the control-transfer instruction can cause the delay instruction to be annulled (i.e. to have no effect) if the branch is not taken (or in one case, if the branch is taken).

Branch and call instructions use PC-relative displacements. The jump and link (JMPL) instruction uses a register-indirect displacement: it computes its target address as either the sum of two registers, or the sum of a register and a 13-bit signed immediate. The branch instruction provides a displacement of ± 8 Mbytes, while the call instruction's 30-bit word displacement allows a transfer to an arbitrary address.

2.5.4. Read/Write Control Register

The SPARC architecture provides instructions to read and write the contents of the various control registers. For reads and writes, the source and destination (respectively) are implied by the instruction itself.

2.5.5. Floating-point and Coprocessor Operate Instructions

Floating-point operate instructions perform all floating-point calculations. These are register-to-register instructions that use the floating-point registers. Like arithmetic/logical/shift instructions, these also compute some result that is a function of two source operands. However, they always write the result into a destination register.

Floating-point operate instructions execute concurrently with IU instructions and possibly with other floating-point instructions. A particular floating-point operate instruction is specified by a subfield of the FPop instructions.

Coprocessor arithmetic instructions are defined by the implemented coprocessor, if any. They are specified by the CPop instruction. The architecture supports 1024 distinct coprocessor arithmetic instructions.

Floating-point loads and stores are NOT floating-point operate instructions (FPops), and coprocessor loads and stores are NOT coprocessor operate instructions. Floating-point and coprocessor loads and stores fall in the category "loads and stores".

Because the IU and the FPU can execute instructions concurrently, when a floating-point exception occurs, the program counter usually does not contain the address of the floating-point instruction that caused the exception. However, the first element of the floating-point queue points to the instruction that caused the exception, and the remaining elements point to floating-point operate instructions that have not yet completed. These can be re-executed or emulated.

Likewise, if the coprocessor executes instructions concurrently with the IU, the coprocessor can support a queue that, at the time of a coprocessor exception, will contain the instruction that

generated the exception and remaining, unexecuted coprocessor instructions.

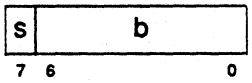
2.6. Processor Data Types

The architecture defines nine data types; these appear in Figure 2-2. The integer types include **byte, unsigned byte, halfword, unsigned halfword, word** and **unsigned word**. The ANSI/IEEE 754-1985 floating-point types include **single, double,** and **extended**. A byte is 8 bits wide, a halfword is 16 bits, a word is 32 bits, a double is 64 bits, and an extended is 128 bits.

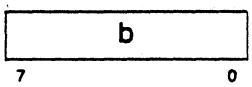
The floating-point double type includes two subfields: 1) the **double-e**, which contains the sign, exponent, and high-order fraction, and 2) the **double-f**, which includes the low-order fraction. The floating-point extended type includes 4 subfields: 1) the **extended-e**, which contains the sign and exponent, 2) the **extended-f**, which contains the integer part of the mantissa, and the high-order part of the fraction, 3) the **extended-f-low**, which contains the low-order fraction, and 4) the **extended-u** which is unused.

The following tables show a) the double and extended types in memory, b) the single-, double-, and extended-precision formats, and c) the processor data types:

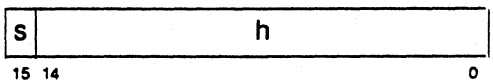
Byte



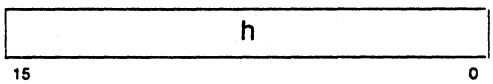
Unsigned Byte



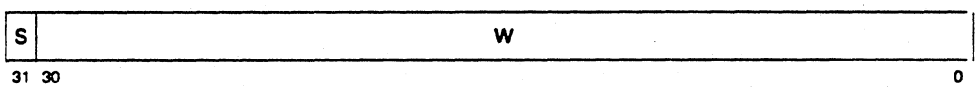
Halfword



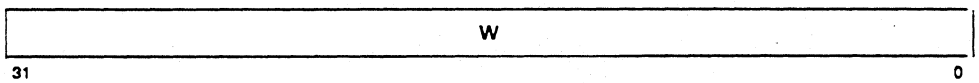
Unsigned Halfword



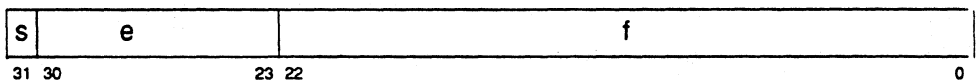
Word



Unsigned Word

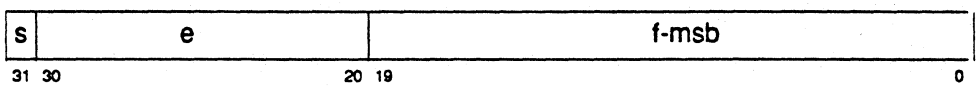


Single

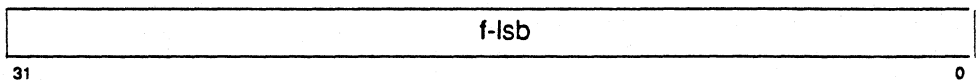


Double

Double -e

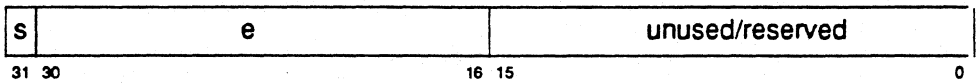


Double -f

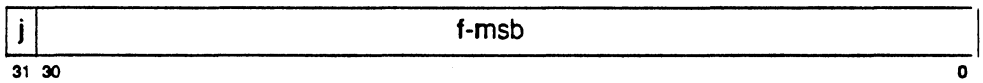


Extended Precision

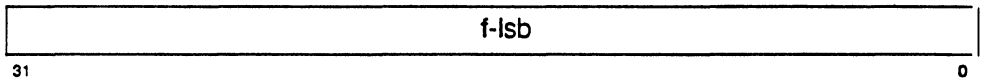
Extended -e



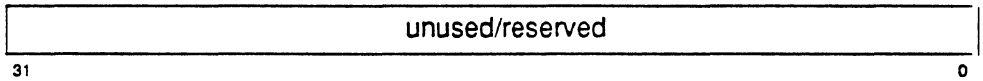
Extended -f



Extended -f low



Extended -u



subfield	address
double-e	n
double-f	n+4
extended-e	n
extended-f	n+4
extended-f-low	n+8
extended-u	n+12

s = sign (1) e = biased exponent (8) f = fraction (23)	
normalized number (0 < e < 255):	$(-1)^s * 2^{e-127} * 1.f$
subnormal number (e = 0):	$(-1)^s * 2^{-126} * 0.f$
zero (e = 0):	$(-1)^s * 0$
signaling NaN:	s = *u*; e = 255 (max); f = .0*uuu*— *uu* (at least one bit must be nonzero)
quiet NaN:	s = *u*; e = 255 (max); f = .1*uuu*— *uu*
infinity:	s = *u*; e = 255 (max); f = .000— 00 (all zeroes)

s = sign (1) e = biased exponent (11) f-msb — f-lsb = f = fraction (52)	
normalized number (0 < e < 2047):	$(-1)^s * 2^{e-1023} * 1.f$
subnormal number (e = 0):	$(-1)^s * 2^{-1022} * 0.f$
zero (e = 0):	$(-1)^s * 0$
signaling NaN:	s = *u*; e = 2047 (max); f = .0*uuu*— *uu* (at least one bit must be nonzero)
quiet NaN:	s = *u*; e = 2047 (max); f = .1*uuu*— *uu*
infinity:	s = *u*; e = 2047 (max); f = .000— 00 (all zeroes)

s = sign (1) e = biased exponent (15) j = integer part (1) f-msb — f-lsb = f = fraction (63)	
normalized number ($0 \le e < 32767$; j = 1):†	$(-1)^s * 2^{e-16383} * j.f$
subnormal number (e = 0; j = 0):	$(-1)^s * 2^{-16383} * j.f$
zero (s = 0; e = 0):	$(-1)^s * 0$
signaling NaN:	s = *u*; e = 32767 (max); j = *u*; f = .0*uuu*— *uu* (at least one bit must be nonzero)
quiet NaN:	s = *u*; e = 32767 (max); j = *u*; f = .1*uuu*— *uu*
infinity:	s = *u*; e = 32767 (max);j = *u*; f = .000— 00 (all zeroes)

2.7. Traps and Exceptions

SPARC supports three types of traps: **synchronous**, **floating-point/coprocessor** and **asynchronous** (asynchronous traps are also called **interrupts**).

> Synchronous traps are caused by an instruction, and occur before the instruction is completed.
>
> Floating-point/coprocessor traps are caused by a floating-point operate (FPop) or coprocessor (CPop) instruction, and occur before the instruction is completed. However, due to the concurrent operation of the IU and the FPU, other non-floating-point instructions may have executed in the meantime.
>
> Asynchronous traps occur when an external event interrupts the processor; they are not related to any particular instruction and occur between the execution of instructions.

Synchronous and floating-point/coprocessor traps are generally taken before the instruction changes any processor or system state visible to a programmer; they happen "between" instructions. Instructions which access memory twice (double loads and stores and atomic instructions) are the only exceptions.

Traps transfer control to an offset within a table. The base address is specified in the trap base register (TBR), and the offset depends on the type of trap. Reset traps, however, cause the processor to transfer control to address 0. Because the program counters are not updated until after an instruction completes, the trap hardware captures both program counters and guarantees that the PC points to either the instruction that caused a synchronous trap, or to the instruction that was about to execute when a floating-point/coprocessor or asynchronous trap occurred. For floating-point/coprocessor traps, the instruction that caused the trap is in the floating-point queue (FQ) or the coprocessor queue (CP), and the PC will usually not point to it.

Traps are described in the section *Traps, Exceptions, and Error Handling.*

2.8. System Interface

The SPARC architecture does not define many of the standard signals, such as bus grant and request lines, or acknowledges; these may differ among implementations. However, it does define the following signals, which are used by the instruction set:

† The architecture does not define or create results with $0 < e < 32767$, j = 0.

bp_IRL<3:0>
This external signal presents an asynchronous interrupt request to the processor. Level 0 indicates that no interrupt is being requested, and levels 1 through 15 request interrupts, with level 15 having the highest priority. Level 15 is non-maskable unless all traps are disabled. The interrupt acknowledge signal is implementation-dependent.

bp_reset_in
This signal indicates that the external system is requesting a reset. The processor responds by entering reset_mode and clearing *pb_error*.

pb_error
The processor asserts this signal when it is in error_mode.

pb_retain_bus
The processor asserts this signal to ensure that the memory bus logic will not relinquish the bus.

bp_FPU_present
This signal indicates that the FPU is present.

bp_CP_present
This signal indicates that a coprocessor is present.

bp_I_cache_present
This signal indicates that there is an external instruction cache present. The IFLUSH instruction uses this signal.

bp_CP_exception
The coprocessor asserts this signal in order to cause a cp_exception trap. An implementation may delay the taking of the trap to the next CPop instruction.

bp_CP_cc(1:0)
The coprocessor supplies these condition codes for the coprocessor branch instruction (CBccc).

bp_memory_access_exception
The memory system asserts this signal when the memory system is unable to provide the data at the requested address. The assertion of this signal will cause either an instruction_access_exception or a data_access_exception trap.

SECTION 3: REGISTERS

3.1. Introduction

The integer unit has two types of registers associated with it; working registers (*r registers*) and control/status registers. Working registers are used for normal operations, and control/status registers keep track of and control the state of the IU. The FPU has 32 working registers (called *f registers*), and two control/status registers: the Floating-point State Register (FSR), and the Floating-point Queue (FQ).

3.2. Integer Unit r Registers

All *r registers* are 32 bits wide. They are divided into 8 global registers and a number of blocks called windows. Each window contains 24 *r registers*.

The number of windows (NWINDOWS) ranges from 2 to 32 depending on the implementation. Implemented windows must be contiguously numbered from 0 to NWINDOWS -1.

3.2.1. Programming Note

At most NWINDOWS -1 windows are available to user code since one window must be available for trap handlers.

The windows are addressed by the CWP, a field of the Processor State Register (PSR). The CWP is incremented by a RESTORE or RETT instruction and decremented by a SAVE instruction. The **active** window is defined as the window currently pointed to by the CWP.

The Window Invalid Mask (WIM) is a register which, under software control, detects the occurrence of IU register file overflows and underflows.

The registers in each window are divided into *ins*, *outs*, and *locals*. Note that the *globals*, while not really part of any particular window, can be addressed when any window is **active**. When any particular window is *active*, the registers are addressed as follows:

Register numbers	Name
r[24] to r[31]	ins
r[16] to r[23]	locals
r[8] to r[15]	outs
r[0] to r[7]	globals

Each window shares its *ins* and *outs* with adjacent windows. The *outs* from a previous window (CWP +1) are the *ins* of the current window, and the *outs* from the current window are the *ins* for the next window (CWP -1). The *globals* are equally available from all windows, and the *locals* are unique to each window.

The register addresses overlap such that, given a register with address *o* where $8 \le o \le 15$, *o* refers to exactly the same register as $(o + 16)$ after the CWP is decremented by 1 modulo NWINDOWS (points to the **next** window). Likewise, given a register with address *i* where $24 \le i \le 31$, *i* refers to exactly the same register as address $(i - 16)$ after the CWP is

incremented by 1 modulo NWINDOWS (points to the **previous** window).

The windows are joined together in a circular stack, where the highest numbered window is adjacent to the lowest. If NWINDOWS = 8, the *outs* of window 7 are the *ins* of window 0. Figures 3-1 and 3-2 show the relationships.

3.3. Special r registers

The utilization of two *r registers* is partially fixed by the instruction set:

- If global register r[0] is addressed as a source operand (rs1 or rs2 = 0), the operand value 0 is returned. If r[0] is addressed as a destination operand (rd = 0), **no** register is modified.
- The CALL instruction writes its own address into *out* register r[15].

Also note that traps save the program counters (PC and nPC) into two locals of the *next* window. This is described in the section *Traps, Exceptions, and Error Handling.*

3.3.1. Programming Notes

Because the processor logically provides new *locals* and *outs* after every procedure call, register local values need not be saved and restored across calls. The overlap registers also minimize the overhead of passing and returning values. They can be used as follows:

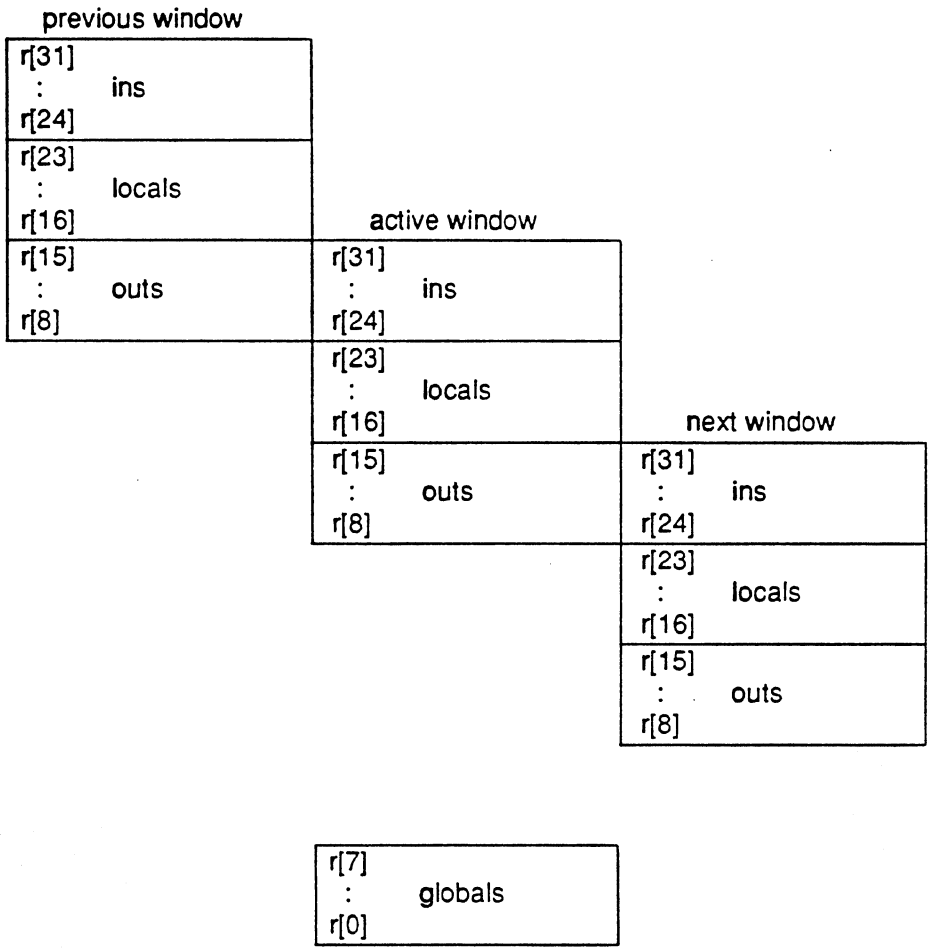
- In preparation for a procedure call, a routine generally moves the parameters into its *out* registers. After the CALL, the CWP is decremented with the SAVE instruction, what was the next window becomes the active window, and the parameters are directly accessible by the callee, since the caller's *outs* are the callee's *ins.*
- Likewise, in preparing for a procedure return, a routine generally moves its result(s) into its *in* registers. After the CWP is incremented via the RESTORE instruction, what was the previous window becomes the active window, and the return values are accessible by the returnee, because the returner's *ins* are the returnee's *outs.* Note that the terms *ins* and *outs* are defined relative to calling, not returning.

Since any implementation has only a finite number of windows, the register file becomes full after the number of procedure calls exceeds the number of returns by NWINDOWS – 1. A subsequent call causes the operating system to move one or more (*in* and *local* sets of) windows from the register file into memory. The SAVE instruction automatically checks for the window_overflow condition.

Similarly, the register file can become empty when the number of procedure returns exceeds the number of calls by NWINDOWS -1. A subsequent return causes one or more previously saved windows to be moved from memory into the register file. The RESTORE instruction automatically checks for the window_underflow condition. The architecture works best with efficient window_overflow and window_underflow handlers.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

By software convention, you can provide additional locals (and consequently, fewer ins and outs). For example, software can assume that the boundary is actually between r[26] and r[27], providing 6 outs, 10 locals, and 6 ins.



In this figure, NWINDOWS = 8. It does not show the 8 *globals*. If the procedure corresponding to the window labeled w0 does a procedure call (executes a SAVE instruction), a window_overflow trap will occur. The overflow trap handler uses the *locals* of w7:

CWP=0 active window = 0
CWP+1 = 1 previous window = 1
CWP-1 = 7 next window = 7
WIM=10000000_2 trap window = 7

3.4. Integer Unit Control/Status Registers

The IU's control/status registers are all 32-bit read/write registers unless specified otherwise. They include the program counters (PC and nPC), the Processor State Register (PSR), the Window Invalid Mask register (WIM), the Trap Base Register (TBR), and the multiply-step (Y) register.

> ☆ ☆ ☆ **NOTE** ☆ ☆ ☆
>
> Control/status registers contain two types of fields, mode and status. Mode fields are set by the programmer; they appear in UPPER CASE (for example, PIL). *Status fields appear in lower case italic font (for example, ver).*

3.4.1. Integer Program Counters (PC and nPC)

The Program Counter (PC) contains the address of the instruction currently being executed by the IU, and the nPC holds the address of the next instruction to be executed (assuming a trap does not occur).

In delayed control transfers, the instruction that immediately follows a control transfer may be executed before control is transferred to the target. The nPC is necessary to implement this feature.

3.4.2. Processor State Register (PSR)

This 32-bit register contains various fields describing the state of the IU. It can be modified by the SAVE, RESTORE, Ticc and RETT instructions, or by instructions that modify the condition codes. The (privileged) instructions RDPSR and WRPSR read and write it directly.

The PSR provides the following fields:

impl	*ver*	*icc*	reserved	EC	EF	PIL	S	PS	ET	CWP
31:28	27:24	23:20	19:14	13	12	11:8	7	6	5	4:0

impl
Bits 31 through 28 identify the implementation number of the processor. The WRPSR instruction does not modify this field.

ver Bits 27 through 24 contain a constant: the meaning of this constant depends on the value of the *impl* field. The WRPSR instruction does not modify this field.

icc Bits 23 through 20 contains the integer unit's condition codes. These bits are modified by the WRPSR instruction, and by arithmetic and logical instructions whose names end with the letters **cc** (for example, ANDcc). The Bicc and Ticc instructions base their control transfer on these bits, which are defined as follows:

n	*z*	*v*	*c*
23	22	21	20

Negative (*n*)
Bit 23 indicates whether the ALU result was negative for the last instruction that modified the *icc* field. 1 = negative, 0 = not negative.

Zero (*z*)
Bit 22 indicates whether the ALU result was zero for the last instruction that modified the *icc*

field. 1 = result was zero, and 0 = result was nonzero.

Overflow (*v*)
If bit 21 is 1, it indicates that an arithmetic overflow occurred during the last instruction that modified the *icc* field. If bit 21 is 0, this indicates that an arithmetic overflow did not occur. Logical instructions that modify the *icc* field always set the overflow bit to 0.

Carry (*c*)
If bit 20 is 1, it indicates that either an arithmetic carry out of bit 31 occurred as the result of the last addition that modified the *icc*, or that a borrow into bit 31 occurred as the result of the last subtraction that modified the *icc*. If bit 20 is 0, this indicates that a carry did not occur. Logical instructions that modify the *icc* field always set the carry bit to 0.

reserved
Bits 19 through 14 are reserved. This field should only be written to 0 by the WRPSR instruction.

EC This bit determines whether the coprocessor is enabled or disabled.
1 = enabled, 0 = disabled.

EF This bit determines whether the FPU is enabled or disabled.
1 = enabled, 0 = disabled.

3.4.3. Programming Note

If the FPU is either disabled, or enabled and not present, an FPop, FBfcc, or floating-point load/store instruction causes an fp_disabled trap. Similarly, if the coprocessor is either disabled, or enabled and not present, a CPop, CBccc, or coprocessor load/store instruction causes a cp_disabled trap.

When the FPU (or CP) is disabled, it retains its state until it is reenabled or reset. When disabled, the FPU can continue to execute instructions in its queue. The CP can also, if it has a queue.

When the FPU is present, software can use the EF bit to determine whether a particular process uses the FPU. If a process does not use the FPU, the FPU's registers need not be saved and restored across context switches. Also, if the FPU is not present, (as indicated by the *bp_FPU_present* signal), the fp_disabled trap can be used to emulate the floating-point instruction set. (This also applies to the coprocessor.)

PIL Bits 11 through 8 identify the processor interrupt level. The processor only accepts interrupts whose interrupt level is greater than the value in PIL. Bit 11 is the MSB and bit 8 is the LSB.

S Bit 7 determines whether the processor is in supervisor mode: when S = 1, the processor is in supervisor mode. Note that because the instructions to write the PSR are only available in supervisor mode, supervisor mode can only be entered by a software or hardware trap.

PS Bit 6 contains the value of the S bit at the time of the most recent trap.

ET Bit 5 is the Trap Enable bit. When ET = 1, traps are enabled. When ET = 0, traps are disabled, and all asynchronous traps are ignored. Synchronous traps and floating-point/coprocessor traps cause the IU to halt and enter error_mode. (See *Appendix C* for a definition of error_mode.)

3.4.4. Programming Note

If traps are enabled (ET=1), some care must be taken when you disable them (ET=0). Since the "RDPSR, WRPSR" instruction sequence is interruptible, it may not be appropriate in some situations. Here are two alternatives: 1) generate a "trap_instruction" trap instead (this disables traps); or 2) use the "RDPSR, WRPSR" sequence and write the interrupt trap handlers so that before they return to the supervisor, they restore the PSR to the value it had when the interrupt handler was entered. Note that the PS bit cannot be restored. In alternative (1), the trap handler should verify that it was called from the supervisor state before returning to the supervisor.

CWP

Bits 4 through 0 comprise the Current Window Pointer, which points to the current active *r register* window. It is decremented by traps and the SAVE instruction, and incremented by RESTORE and RETT instructions.

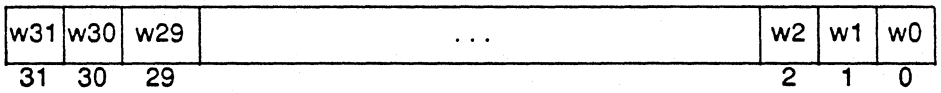
The CWP cannot point to an unimplemented window; therefore arithmetic on the CWP is done modulo the number of implemented windows (NWINDOWS).

3.4.5. Window Invalid Mask Register (WIM)

This register is used to determine whether a window_overflow or window_underflow trap should be generated by a SAVE, RESTORE, or RETT instruction. Each bit in the WIM register corresponds to a window. For example, bit 0 corresponds to window 0 (CWP = 0), bit 1 corresponds to window 1 (CWP = 1), and so on. If a SAVE, RESTORE, or RETT would cause the CWP to point to a window whose corresponding WIM bit equals 1, it causes a window_overflow (SAVE) or window_underflow (RESTORE, RETT) trap.

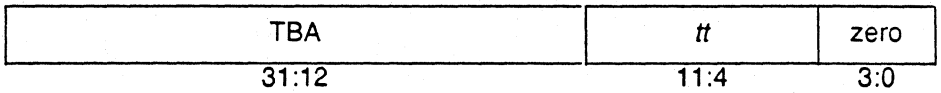
This register can be read by the RDWIM instruction, and written by the WRWIM instruction. Bits corresponding to unimplemented windows read as zeroes and values written to unimplemented bits are ignored.

The WIM provides the following fields:



3.4.6. Trap Base Register (TBR)

The trap base register contains three fields that generate the address of the trap handler when a trap occurs. These are:



TBA

Bits 31 through 12 comprise the Trap Base Address (TBA), which is controlled by software. It contains the most-significant 20 bits of the trap table address. (Note that the reset trap is an exception; it traps to address 0). The TBA field can be written by the WRTBR instruction.

tt Bits 11 through 4 comprise the Trap Type (*tt*) field. This is an 8-bit field that is written by the processor at the time of a trap, and retains its value until the next trap. It provides an offset into the trap table. The WRTBR instruction does not affect the *tt* field.

zero

Bits 3 through 0 are zeroes. The WRTBR instruction does not affect this field.

For additional information, see the section *Traps, Exceptions, and Error Handling.*

3.4.7. Y Register

The multiply step instruction (MULScc) uses the 32-bit Y register to create 64-bit products. An example algorithm is described in *Appendix B.*

This register can be read and written using the RDY and WRY instructions.

3.5. Floating-Point Registers

The floating-point unit has 32 working registers called *f registers*, a Floating-Point State Register (FSR) that contains mode and status information about the FPU, and a Floating-Point Queue (FQ) that holds one or more 64-bit instruction/address pairs. Software uses the FQ to recover from floating-point exceptions.

3.5.1. Floating-Point f registers

The 32-bit *f registers* are numbered from f[0] to f[31]. These can be read and written by floating-point operate (FPop and FPcmp) instructions, or by load/store single/double floating-point instructions (LDF, LDDF, STF, STDF). They are addressable at all times.

A single *f register* can hold one single-precision operand. Double-precision operands require an *f register pair*, where the double-e datum occupies an even-numbered register, and the double-f datum occupies the following odd-numbered register. Extended-precision operands require an *f register quad*, with extended-e, extended-f, extended-f low, and extended-u in register addresses 0, 1, 2, and 3 modulo 4, respectively. Thus, the *f register* file can hold 8 extended, 16 double, or 32 single-precision operands.

3.5.2. Floating-Point State Register (FSR)

The FSR register fields contain FPU mode and status information. The fields are:

RD	RP	TEM	AU	reserved	*ftt*	*qne*	res	*fcc*	*aexc*	*cexc*
31:30	29:28	27:23	22	21:17	16:14	13	12	11:10	9:5	4:0

Rounding Direction (RD)

Bits 31 and 30 select the rounding direction for floating-point results, according to the ANSI/IEEE 754-1985 Standard:

RD	Round Toward:
0	Nearest (even, if a tie)
1	0
2	+ ∞
3	- ∞

Extended Rounding Precision (RP)

Bits 28 and 29 determine the precision to which extended results are rounded, according to the ANSI/IEEE 754-1985 Standard:

RP	Round to:
0	Extended
1	Single
2	Double
3	(Unused)

Trap Enable Mask (TEM)

Bits 27 to 23 are enable bits for each of the five floating-point exceptions that can be indicated in the current_exception field (*cexc*). (See definition of *cexc* below.) If a floating-point operate instruction generates one or more exceptions and the TEM bit corresponding to one or more of the exceptions is set (1), an fp_exception trap is caused. A reset (0) TEM bit prevents that exception type from generating a trap. (See below.) The TEM field may be read and written by the STFSR and LDFSR instructions.

NVM	OFM	UFM	DZM	NXM
27	26	25	24	23

The TEM field may be read and written by the STFSR and LDFSR instructions.

An implementation need not implement all of the TEM bits as defined above, except NXM, which must be implemented as described above. If a particular bit of the TEM field is not implemented according to the above definition, then it is implemented as a state bit instead. That is, if the particular bit is written to a value by a LDFSR instruction, that same value will be read by a subsequent STFSR instruction.

Abrupt Underflow (AU)

Bit 22, when set to 1, causes denormalized floating-point operands and/or results to be rounded to zero. The definition of AU mode is implementation-dependent and is not defined by the ANSI/IEEE 754-1985 Standard.

Reserved

Bits 21 through 17 and bit 12 are reserved. When read by an STFSR instruction, this field delivers all zeroes. This field should only be written to zero by the LDFSR instruction.

Floating-Point Trap Type (*ftt*)

Bits 16 through 14 identify fp_exception traps. After a floating-point exception trap occurs, the *ftt* field encodes the type of exception. *ftt* remains valid until the next FPop instruction completes. (Note that the exception-causing FPop and its address are in the first entry of the Floating-point Queue — see below.)

The *ftt* field can be read by the STFSR instruction. An LDFSR instruction does not affect *ftt*. This field encodes the exception types as follows:

ftt	Trap Type
0	None
1	IEEE_exception
2	unfinished_FPop
3	unimplemented_FPop
4	sequence_error

An IEEE_exception indicates that an ANSI/IEEE 754-1985 exception occurred for the FPop identified by the front entry of the FQ. The exception type(s) is indicated in the *cexc* field. If the IEEE_exception results in a fp_exception trap (as determined by the TEM) then the destination *f register*, *fcc*, and *aexc* fields remain unchanged. However, if the IEEE_exception does **not** result in a trap; then the *f register*, *fcc*, and *aexc* fields are updated to their new values

An unfinished_FPop indicates that an implementation's FPU was unable to generate correct results or exceptions, as defined by the ANSI/IEEE 754-1985 Standard. In this case, the *cexc* field is undefined. (However, the *aexc* and *fcc* fields, and the destination *f register* are not affected by the exception.)

An unimplemented_FPop indicates that an implementation's FPU decoded an FPop that it did not implement. In this case, the *cexc* field is undefined. (However, the *aexc* and *fcc* fields, and the destination *f register* are not affected by the exception.)

3.5.3. Programming Note

In the case of an unfinished_FPop or unimplemented_FPop, the software should emulate or reexecute the instructions in the FQ, and update the FSR and destination *f register(s)*.

A sequence_error indicates that an FPop or a load floating-point instruction is fetched while the FPU is in FPU_exception_mode, waiting for the FQ to be emptied by software. (See *Appendix C*).

Queue Not Empty (*qne*)

Bit 13 indicates whether the Floating-point Queue (FQ) is empty after an fp_exception trap or after a Store Double Floating-point Queue (STDFQ) instruction is executed. If *qne* = 0, the queue is empty; if *qne* = 1, the queue is not empty.

The *qne* bit can be read by the STFSR instruction. The LDFSR instruction does not affect *qne*. However, executing successive STDFQ instructions will (eventually) cause the FQ to become empty (*qne* = 0).

Floating-point Condition Codes (*fcc*)

Bits 11 and 10 contain the FPU condition codes. These bits are updated by floating-point compare instructions (FCMP and FCMPE) and are read and written by the STFSR and LDFSR instructions, respectively. Note that *fcc* is updated even if FCMPE generates an IEEE_exception trap.

In the following table, *fs1* and *fs2* correspond to the values in the *f registers* specified by an instruction's *rs1* and *rs2* fields. The question mark (?) indicates an unordered relation, which is true if either *fs1* or *fs2* is a signaling or quiet NaN (see the section *Processor Data Types* in the section *SPARC Architecture Overview*).

The FBfcc instruction bases its control transfer on this field, which is interpreted as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

Accrued Exception Bits (*aexc*)

Bits 9 through 5 accumulate IEEE floating-point exceptions while fp_exception traps are disabled. After an FPop completes, the TEM and *cexc* fields are logically *and*'d together. If the result is nonzero, an FP_exception trap is generated; otherwise, the new *cexc* field is *or*'d into the *aexc* field. Thus, while traps are masked, exceptions are accumulated in the *aexc* field. (See below).

nva	*ofa*	*ufa*	*dza*	*nxa*
9	8	7	6	5

The *aexc* field is read and written by the STFSR and LDFSR instructions.

An implementation need not implement all of the *aexc* bits as defined above, except *nxa*, which must be implemented as described above. If a particular bit of the *aexc* field is not implemented according to the above definition, then it is implemented as a state bit instead. That is, if the particular bit is written to a value by a LDFSR instruction, that same value will be read by a subsequent STFSR instruction.

Current Exception Bits (*cexc*)

Bits 4 through 0 indicate one or more IEEE exceptions that were generated by the most recently executed FPop instruction. The absence of an exception causes the corresponding bit to be cleared.

nvc	*ofc*	*ufc*	*dzc*	*nxc*
4	3	2	1	0

The *cexc* field is read and written by the STFSR and LDFSR instructions.

An implementation need not implement all of the *cexc* bits as defined above, except *nxc*, which must be implemented as described above. If a particular bit of the *cexc* field is not implemented according to the above definition, then it is implemented as a state bit instead. That is, if the particular bit is written to a value by a LDFSR instruction, that same value will be read by a subsequent STFSR instruction.

The *cexc* bits are not defined following an FPop that causes an unimplemented_FPop or unfinished_FPop fp_exception trap. Following an FPop that does not generate an fp_exception trap or that generates an IEEE_exception trap, the *cexc* bits are set as follows:

nvc = 1 indicates invalid: an operand is improper for the operation to be performed. For example, 0/0, and $\infty - \infty$ are invalid.

ofc = 1 indicates overflow: the rounded result would be larger in magnitude than the

largest normalized number in the specified format.

ufc = 1 indicates underflow: the rounded result is inexact, and would be smaller in magnitude than the smallest normalized number in the indicated format.

dzc = 1 indicates division-by-zero: X/0, where X is subnormal or normalized. Note that 0/0 does **not** set the *dzc* bit.

nxc = 1 indicates inexact: The rounded result differs from the infinitely precise correct result.

The following illustration summarizes the handling of IEEE_exception traps. Note that the *aexc* and *ftt* fields can normally only be cleared by software.

```
FPop generates an IEEE exception;
cexc ← IEEE exceptions generated by this FPop;
if ( cexc and TEM ) = 0
then ( aexc ← aexc or cexc; f[] ← result; fcc ← fcc_result)
else ( ftt ← IEEE_exception; cause fp_exception trap )
```

3.5.4. Programming Note

Since the operating system must be capable of simulating the entire FPU in order to properly handle the unimplemented_FPop and unfinished_FPop floating-point exceptions, a user process always "sees" a fully implemented FSR as defined above. In other words, a user process always "sees" *cexc*, *aexc*, and TEM fields that conform to the ANSI/IEEE 754-1985 Standard.

3.5.5. Floating-Point Queue (FQ)

The Floating-point Queue keeps track of FPops that are pending completion by the FPU when an fp_exception trap occurs. When an fp_exception trap occurs, the first entry in the queue gives the address of the FPop that caused the exception and the instruction itself. Any remaining entries in the queue contain FPop instructions (and their addresses) that had not finished when the exception occurred.

3.5.6. Implementation Note

If an implementation provides *n* entries in the queue, at most *n* FPops can execute simultaneously in the FPU. For example, if the FPU provides one adder and one multiplier that can operate independently, then the FQ has no fewer than two entries.

SECTION 4: INSTRUCTIONS

4.1. Introduction

Functionally, SPARC architecture instructions fall into six categories: 1) load and store 2) arithmetic/logical/shift, 3) control transfer, 4) read/write control register, 5) floating-point operate, and 6) coprocessor operate. Instructions may also be classified into three major formats, two of which include subformats.

4.2. Instruction Formats

The three instruction formats are called format 1, format 2, and format 3. Figure 4-1 shows each instruction format, with its fields and bit positions. It also lists the types of instructions that use that format:

The fields in these instructions have the following meanings:

op This field places the instruction into one of the 3 major formats:

Format	op value	Instruction
1	1	Call
2	0	Bicc, FBfcc, CBccc, SETHI
3	2 or 3	other

op2 This field comprises bits 24 through 22 of format 2 instructions. It selects the instruction as follows:

op2 value	Instruction
0	UNIMP
2	Bicc
4	SETHI
6	FBfcc
7	CBccc

rd For store instructions, this register selects an *r register* (or an *f register pair*), or an *f register* (or an *f register pair*) to be the source. For all other instructions, this field selects an *r register* (or an *f register pair*), or an *f register* (or an *f register pair*) to be the destination.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

Reading r[0] produces the result 0, and writing it causes the result to be discarded.

For more information on *r registers*, see the section *Registers*.

a The "a" bit means "annul" in format 2 instructions. This bit changes the behavior of the instruction encountered immediately after a control transfer, as described later in this section.

cond
This field selects the condition code for format 2 instructions.

imm22
This field is a 22-bit constant value used by the SETHI instruction.

disp22 and *disp30*
These fields are 30-bit and 22-bit sign-extended word displacements, for PC-relative calls and branches, respectively.

op3 The *op3* field selects one of the format 3 opcodes.

i The *i* bit selects the type of the second ALU operand for non-FPop instructions. If *i* = 0, the second operand is r[rs2]. If *i* = 1, the second operand is sign-extended *simm13*.

asi This 8-bit field is the *address space identifier* generated by load/store alternate instructions. See discussion below.

rs1 This 5-bit field selects the first source operand from either the *r registers* or the *f registers*.

rs2 This 5-bit field selects the second source operand from either the *r registers* or the *f registers*.

simm13
This field is a sign-extended 13-bit immediate value used as the second ALU operand when *i* = 1.

opf
This 9-bit field identifies a floating-point operate (FPop) instruction or a coprocessor operate (CPop) instruction. Note that it uses the synonym *opc* for coprocessor operate instructions (see the coprocessor operate instructions in *Appendix B*). A table in *Appendix F* shows the relationship between the *opf* field and FPop instructions.

4.3. Load/Store Instructions

Load and store instructions are the only instructions that access memory and registers external to the processor. They generate a 32-bit byte address. In addition to the address, the processor always generates an *address space identifier*, or *asi*.

4.3.1. Address Space Identifier

The address space identifier generated by the processor is made available to the external system to distinguish up to 256 address spaces. These spaces can include system control registers, main memory, etc. The number of defined spaces is implementation-dependent.

The SPARC architecture defines four address spaces and their *asi* values; these appear in Table 4-3. They indicate to the external system whether the processor is in **user** or **supervisor** mode (as indicated by the PSR), and whether the access is an **instruction** or a **data** reference.

asi	Assignment
0 - 7	Implementation-definable
8	User instruction space
9	Supervisor instruction space
10	User data space
11	Supervisor data space
12 - 255	Implementation-definable

Load/store instructions normally generate an *asi* of either 10 or 11 for the data access, depending on whether the processor is in user or supervisor mode. However, the load from alternate space and store into alternate space instructions use the *asi* field supplied by the instruction itself.

Note that the load/store alternate instructions are privileged; they can only be executed in supervisor mode.

4.3.2. Addressing Conventions

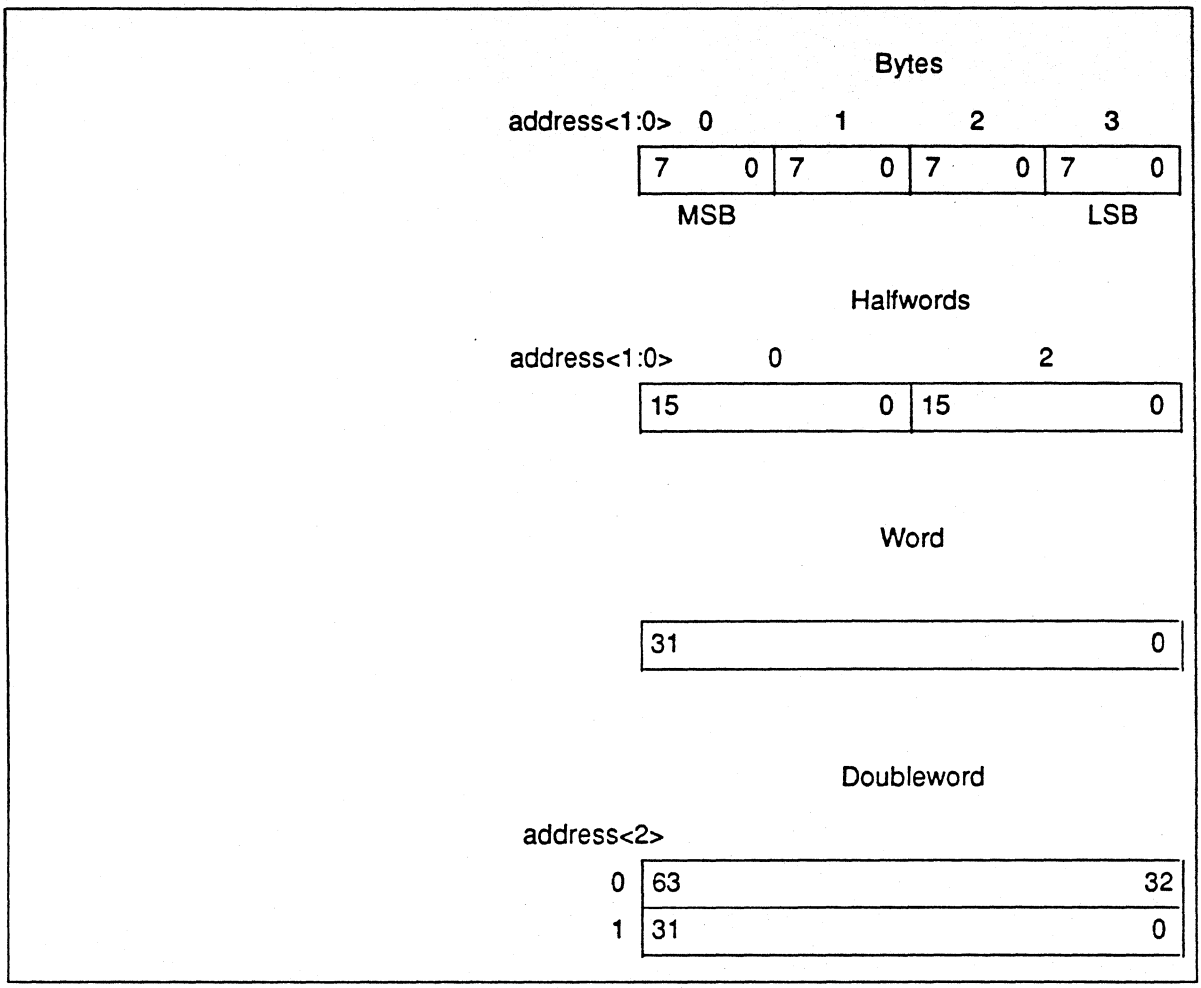
The load and store instructions use the following addressing conventions:

Bytes
: For load and store byte instructions, increasing the address generally means decreasing the significance of the byte within a word: the most significant byte (MSB) of a word is accessed when address bits <1:0> are 0 and the least significant byte (LSB) is accessed when address<1:0> = 3.

Halfwords
: For load and store halfword instructions, when address bit 1 = 1, the least significant halfword of a word is accessed, and when address bit 1 = 0, the most significant halfword is accessed.

Doublewords
: For load and store double instructions, the most significant word is accessed when address bit 2 = 0, and the least significant word is accessed when address bit 2 = 1.

In general, the address of a doubleword, word, or halfword is the address of its most significant byte. These conventions are illustrated in the following figure:



A doubleword-aligned datum is located at a doubleword address, which must be evenly divisible by 8. A word-aligned datum is located at a word address, which must be evenly divisible by 4. A halfword-aligned datum is located at a halfword address, which must be divisible by 2.

If a doubleword, word, or halfword load or store instruction generates an improperly aligned address, a memory_address_not_aligned trap occurs.

4.4. Arithmetic, Logical, and Shift Instructions

All of these instructions compute some result that is a function of two source operands, and either write the result into a destination *r register (r[rd])* or discard it. One of the operands is always *r[rs1]*. The other operand depends on the *i* bit in the instruction: if *i* = 0, the operand is *r[rs2]*, but if *i* = 1, the operand is the sign-extended constant *sign_extend(simm13)*.

Reading r[0] produces the value zero. If the destination field indicates a write into r[0], no *r register* is modified and the result is discarded.

Most of these instructions have dual versions which modify the *integer condition codes (icc)* as a side effect.

4.4.1. Programming Note

r[0] can be used to implement a register-to-register move in one of several ways: ADD with 0, OR with 0, etc. Subtract and set condition codes (SUBcc) can be used as an integer COMPARE instruction.

The tagged add and subtract instructions (TADDcc, TSUBcc, TADDccTV and TSUBccTV) operate on tagged data where the tag is the low-order two bits of the data. If either of the instruction's two operands has a nonzero tag, the overflow bit of the PSR is set. The "trap on overflow" versions, TADDccTV and TSUBccTV, in addition to writing the condition codes, also cause an overflow trap.

4.4.2. Programming Note

One possible model for tagging is to use 0 to tag integers and 3 for pointers to doublewords, i.e. list cells.

If trapping overhead is insignificant, then TADDccTV or TSUBccTV is faster than the non-trapping versions, which would need to be followed by 'branch on overflow' instructions.

Suppose *p* contains a tagged pointer to a list cell, i.e. has 3 in its low-order two bits. Since the load and store instructions execute successfully only with properly aligned addresses, a load or store word with an address specifier of "*p - 3*" or "*p + 1*" will succeed, accessing the first or second word of the list cell, respectively; if, on the other hand, *p* contains a tag value other than 3, they will trap.

Shift instructions shift an *r register* left or right by a constant or variable amount, as described in *Appendix B*. None of the shift instructions changes the condition codes.

The "set high 22 bits of r" (SETHI) instruction writes a 22-bit constant from the instruction into the high-order bits of the destination register. It clears the low-order 10 bits, and does not change the condition codes.

4.4.3. Programming Note

SETHI can be used to construct a 32-bit constant using two instructions.

4.5. Control Transfer Instructions

Control-transfer instructions change the values of PC and nPC. There are five types of control transfer instructions:

1) Conditional branch (Bicc, FBfcc, CBccc)
2) Jump and Link (JMPL)
3) Call (CALL)
4) Trap (Ticc)
5) Return from trap (RETT)

Each of these can be further categorized according to whether it is 1) PC-relative or register-indirect, or 2) delayed or non-delayed. The following matrix shows these characteristics:

Instruction	PC-relative or Register-indirect	Delayed
Bicc, FBfcc, CBccc, CALL	PC-Relative	Yes
JMPL, RETT	Reg-Indirect	Yes
Ticc	Reg-Indirect	No

The following paragraphs describe each of the characteristics:

PC-relative
: A PC-relative control transfer computes its target address by adding the (shifted) sign-extended immediate displacement to the program counter (PC).

Register-indirect
: A register-indirect instruction computes its target address as either "r[rs1] + r[rs2]" if *i* = 0, or "r[rs1] + sign_ext(simm13)" if *i* = 1.

Delayed
: A control transfer instruction is delayed if it transfers control to the target address after a one-instruction delay. Delayed control transfers are described in the next section.

4.5.1. Delayed Control Transfers

Traditional architectures usually execute the target of a control transfer instruction immediately after the control-transfer instruction. The SPARC architecture delays by one instruction the execution of the target of a delayed control-transfer instruction. The instruction encountered immediately after a delayed control transfer is called the *delay instruction.*

4.5.2. PC and nPC

In general, the PC points to the instruction being executed by the IU, and the nPC points to the instruction to be executed next. Most instructions complete by copying the contents of the nPC into the PC, then either increment nPC by 4, or, if the instruction implies a control transfer, write the computed target address into nPC. The PC now points to the instruction that will be executed next, and the nPC points to the instruction that will be executed **after** the next one; in other words, **two** instructions hence.

The sequence is:

```
PC ← nPC
nPC ← nPC + 4    or  target address
```

4.5.3. Delay Instruction

The instruction pointed to by the nPC when a delayed control-transfer instruction is encountered is called the delay instruction. Normally, this is the next sequential instruction in the code space. However, if the instruction that preceded the delayed control transfer was itself a delayed control transfer, the address of the delay instruction is the target of the (first) control-transfer instruction, since that is where the nPC will point. This behavior is explained further in the section *Back-to-Back Delayed Control Transfers* below.

The following example shows the order of execution for a simple (not back-to-back) delayed control transfer. The order of execution is 8, 12, 16, 40. If the delayed control transfer-instruction were not taken, the order would be 8, 12, 16, 20.

PC before instruction	nPC before instruction	Instruction
8	12	Non-control transfer
12	16	Control transfer (target = 40)
16	40	Non-control transfer (delay instruction)
		Transfers control to 40
40	44	...

4.5.4. Annul Bit

The *a* (annul) bit changes the behavior of the delay instruction. This bit is only available on conditional branch instructions (Bicc, FBfcc and CBccc). If *a* is set on a conditional branch (except BA, FBA and CBA) and the branch is **not** taken, the delay instruction is "annulled" (not executed). An annulled instruction has no effect on the state of the IU nor can a trap occur during an annulled instruction. If the branch is taken, the *a* bit is ignored and the delay instruction is executed. For example:

PC	nPC	Instruction	Action
8	12	Non-control transfer	Executed
12	16	Bicc (a=1) 40	Not taken
16	40	Non-control transfer	Annulled (not executed)
20	24	...	Executed

PC	nPC	Instruction	Action
8	12	Non-control transfer	Executed
12	16	Bicc (a=0) 40	Not taken
16	40	...	Executed
40	44	...	Executed

BA, FBA and CBA instructions are a special case; if the *a* bit is set in these instructions the delay instruction is **not** executed if the branch is taken, but it is executed if the branch is not taken.

The following display shows the effect of the *a* bit on the delay instruction after various kinds of branches:

a bit	Type of branch	Delay instr. executed?
a = 1	Always	No
	Conditional, taken	Yes
	Conditional, not taken	No
a = 0	Always	Yes
	Conditional, taken	Yes
	Conditional, non taken	Yes

4.5.5. Programming Notes

The annul bit increases the likelihood that a compiler or optimizer can place a useful instruction in the delay slot after a branch. Refer to the following table:

Address	Instruction	Target
L	non-control transfer instruction	
L'		
:		
:		
	Bicc	L
D	NOP	
:	:	

If the Bicc has a = 0, a code optimizer may be able to move a non-control-transfer instruction from within the loop into location D. If the Bicc has a = 1, then the compiler can copy the non-control-transfer instruction at location L into location D, and change the branch to Bicc L'.

The annul bit can also be used to optimize "if-then-else" statements. Since the conditional branch instructions provide both true and false tests for all the conditions, an optimizer can arrange the code so that a non-control-transfer instruction from either the "else" branch or the "then" branch can be moved into the delay position after the branch instruction. For example:

Address	Instruction	Address	Instruction
	Bicc(cond, a=1) THEN		Bicc(cond, a=1) ELSE
Delay:	then-phrase-instr-1	Delay:	else-phrase-instr-1
	else-phrase-instr-1		then-phrase-instr-1
	else-phrase-instr-2		then-phrase-instr-2
	goto ...		goto ...
	:		:
THEN:	then-phrase-instr-2	ELSE:	else-phrase-instr-2
	then-phrase-instr-3		else-phrase-instr-3
	:		:

When set in a branch always instruction (BA, BFA), the annul bit implements a "traditional," non-delayed branch instruction. This can also be used to dynamically replace unimplemented instructions with branches to software emulation routines as this requires less overhead than a trap.

4.5.6. Calls and Returns

A procedure that requires a register window is invoked by executing both a CALL (or a JMPL) and a SAVE instruction. A procedure that does not need a register window, a so-called "leaf" routine, is invoked by executing only a CALL (or a JMPL). Leaf routines can use only the *out* registers.

The CALL instruction stores PC, which points to the CALL itself, into register r[15] (an *out* register). JMPL stores PC, which points to the JMPL instruction, into the specified *r register*. These instructions then cause a transfer of control to a target that can be arbitrarily distant.

The SAVE instruction is similar to an ADD instruction, except that it also decrements the CWP by one, causing the active window to become the previous window, thereby "saving" the caller's window. Also, the source registers for the addition are from the previous window while the result is written into the new window.

A procedure that uses a register window returns by executing both a RESTORE and a JMPL instruction. A leaf procedure returns by executing a JMPL only. The JMPL instruction typically returns to the instruction following the CALL's or JMPL's delay instruction; in other words, the typical return address is 8 plus the address saved by the CALL.

The RESTORE instruction, also like an ADD instruction, increments the CWP by one, causing the previous window to become the active window, thereby "restoring" the caller's window. Also, the source registers for the addition are from the current window while the result is written into the previous window.

Both SAVE and RESTORE compare the new CWP against the Window Invalid Mask (WIM) to check for window overflow or underflow.

4.5.7. Programming Note

The SAVE and RESTORE instructions can be used to atomically update the CWP while establishing a new memory stack pointer in an *r register*.

4.5.8. Trap (Ticc) Instruction

The Ticc instruction evaluates the condition codes specified by its *cond* field, and if the result is true, it causes a trap with no delay instruction. If the condition codes evaluate to false, it executes as a NOP.

A taken Ticc identifies the software trap by writing "trap_number + 128" into the *tt* field of the TBR. The processor enters supervisor mode, disables traps, decrements the CWP, and saves PC and nPC into the locals r[17] and r[18] (respectively) of the new window.

4.5.9. Programming Note

Ticc can be used to implement kernel calls, breakpointing, and tracing. It can also be used for run-time checks, such as out-of-range array indices, integer overflow, etc.

4.5.10. Delayed Control Transfers Couples

When a delayed control transfer is encountered immediately after another delayed control transfer, this creates what is called a *delayed control-transfer couple*, which the processor handles differently from a simple control transfer.

The following tables show, first, a sequence of instructions that includes a delayed control-transfer couple, and second, a table that illustrates the order of execution depending on the nature of the control-transfer instructions.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

In the following tables, 'delayed control-transfer instruction' is abbreviated to 'DCTI'. Note that a "non-DCTI" may be either a non-control-transfer instruction, or a control-transfer instruction which is not delayed (i.e. a Ticc).

address:	instruction	target
8:	non-DCTI	
12:	DCTI	40
16:	DCTI	60
20:	non-DCTI	
24:	...	
:	:	
40:	non-DCTI	
44:	...	
:	:	
60:	non-DCTI	
64:	...	
:	:	

Case	12: DCTI 40	16: any DCTI 60	Order of Execution:
1	DCTI unconditional	DCTI taken	12, 16, 40, 60, 64, ...
2	DCTI unconditional	B*cc(a=0) untaken	12, 16, 40, 44, ...
3	DCTI unconditional	B*cc(a=1) untaken	12, 16, 44, 48, ... (40 annulled)
4	DCTI unconditional	B*A(a=1)	12, 16, 60, 64, ... (40 annulled)
5	B*A(a=1)	any CTI	12, 40, 44, ... (16 annulled)
6	B*cc	DCTI	not supported (see text)

Where the annul bit is not indicated, it may be either 0 or 1. Abbreviations are as follows:

B*A	BA, FBA or CBA
B*cc	Bicc, FBfcc, or CBccc (except B*A)
DCTI unconditional	CALL, JMPL, RETT, or B*A(a=0)
DCTI taken	CALL, JMPL, RETT, B*cc taken, or B*A(a=0)

When the first instruction of a delayed control-transfer couple is a conditional branch, the transfer of control is undefined (case 6). If such a couple is executed, the location where execution continues is within the same address space but otherwise undefined. This sequence does not change any other aspect of the processor state.

Case 1 of the above table includes the "JMPL, RETT" couple. RETT must always be preceded by a JMPL instruction. (If it is not, the location where execution continues is not necessarily within the address space implied by the PS bit of the PSR.)

4.5.11. Programming Note

Trap handlers complete execution by executing the "JMPL, RETT" couple.

4.6. Read and Write Control Registers

These instructions read or write the contents of the programmer-visible control registers. This category includes instructions to read and write the PSR, the WIM, the TBR, the Y register, the FSR, and the CSR. These instructions are all privileged (available in supervisor state only), except those that read and write the Y register, the FSR, and the CSR.

4.7. Floating Point Operate (FPop) Instructions

Floating-point operate instructions (FPops) are generally three-register instructions that compute some result that is a function of one or two source operands, and place the result in a destination *f register*. The exception is floating-point compare operations, which update the *fcc* field of the FSR.

The term "FPop" does NOT include the load/store floating-point instructions.

Multiple-precision instructions assume that their operands are in multiple contiguous *f registers*. The operands must be aligned in the *f registers* according to their size: the number of the first *f register* of a multiprecision operand must be a multiple of the operand size in words.

All FPops except move instructions can modify the status fields of the FSR.

FPops execute concurrently with IU instructions and other FPops. Concurrent operation is described in the section *SPARC architecture Overview* and in *Appendix C*.

There are no direct IU-to-FPU or FPU-to-IU move instructions.

4.8. Coprocessor Operate (CPop) Instructions

The coprocessor operate instructions are executed by the attached coprocessor. If there is no attached coprocessor, a CPop instruction generates a cp_disabled trap.

The instruction fields of a CPop instruction, except for *op* and *op3*, are interpreted only by the coprocessor. A CPop takes all operands from and returns all results to coprocessor registers.

Note that the term "CPop" does NOT include the load/store coprocessor instructions.

SECTION 5: TRAPS, EXCEPTIONS, AND ERROR HANDLING

5.1. Introduction

SPARC supports three types of traps: **synchronous**, **floating-point/coprocessor** and **asynchronous** (asynchronous traps are also called **interrupts**). Synchronous traps are caused by an instruction, and occur before the instruction is completed. Floating-point/coprocessor traps are caused by a Floating-Point Operate (FPop) or coprocessor (CPop) instruction, and occur before the instruction is completed. However, due to the concurrent operation of the IU and the FPU, other non-floating-point instructions may have executed in the meantime. Asynchronous traps occur when an external event interrupts the processor. They are not related to any particular instruction and occur between the execution of instructions.

Synchronous and floating-point/coprocessor traps are generally taken before the instruction changes any processor or system state visible to a programmer; they happen "between" instructions. Instructions which access memory twice (double loads and stores and atomic instructions) are the only exceptions.

An instruction is defined to be **trapped** if any trap occurs during the course of its execution. If multiple traps occur during one instruction, the highest priority trap is taken. Lower priority traps are ignored because the traps are arranged under the assumption that the lower priority traps persist, recur, or are meaningless due to the presence of the higher priority trap. For example, if a mem_address_not_aligned trap is detected during an instruction fetch, the potential unimplemented_instruction trap is meaningless because the address is invalid. Pending interrupts persist; therefore, they have the lowest priority.

The ET bit in the PSR must be set for traps to occur normally. If a synchronous trap occurs while traps are disabled the processor halts and enters an error state. In most implementations, this causes a reset trap.

5.1.1. Implementation Note

Since interrupts are ignored while traps are disabled, they should persist until they are acknowledged.

Load/store instructions generally trap before the instruction changes the state of the processor. However, those instructions that do more than one memory access (namely the load and store doubles and the atomic load and store instructions) may trap on a data_access_exception after the first memory access, causing a trap after the processor state has been partially modified. This can only occur for non-resumable exceptions, such as uncorrectable memory errors. (See *Appendix B* for instruction descriptions.)

5.2. Trap Addressing

The Trap Base Register (TBR) generates the exact address of a trap handling routine. When a trap (other than some types of reset trap) occurs, the hardware writes a value into the trap type (*tt*) field of the TBR. This uniquely identifies the trap and serves as an offset into the table whose starting address is given by the TBA field of the TBR.

The 8-bit wide *tt* field allows for 256 distinct types of traps. Half of these (0 - 127) are dedicated to hardware traps, and half (128-255) are dedicated to programmer-initiated traps (see the Ticc instruction). The *tt* field remains valid until another trap occurs.

5.3. Trap Priorities

The following table shows the trap types, priorities, and assignments.

Trap	Priority	*tt*
reset	1	—
instruction_access_exception	2	1
illegal_instruction	3	2
privileged_instruction	4	3
fp_disabled	5	4
cp_disabled	5	36
window_overflow	6	5
window_underflow	7	6
mem_address_not_aligned	8	7
fp_exception	9	8
cp_exception	9	40
data_access_exception	10	9
tag_overflow	11	10
trap_instruction (Ticc)	12	128-255
interrupt_level_15	13	31
interrupt_level_14	14	30
interrupt_level_13	15	29
interrupt_level_12	16	28
interrupt_level_11	17	27
interrupt_level_10	18	26
interrupt_level_9	19	25
interrupt_level_8	20	24
interrupt_level_7	21	23
interrupt_level_6	22	22
interrupt_level_5	23	21
interrupt_level_4	24	20
interrupt_level_3	25	19
interrupt_level_2	26	18
interrupt_level_1	27	17

5.4. Trap Definition

A trap causes the following action:

- It disables traps (ET ← 0).
- It copies the S field of the PSR into the PS field and then sets the S field to 1.
- It decrements the CWP by 1, modulo the number of implemented windows.
- It saves the PC and nPC into r[17] and r[18], respectively, of the new window.
- It sets the *tt* field of the TBR to the appropriate value.
- If the trap is not a reset, it writes the PC with the contents of TBR, and the nPC with the contents of TBR + 4. If the trap is a reset, it loads the PC with 0 and the nPC with 4.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

Unlike many other processors, the SPARC architecture does not automatically save the PSR into memory during a trap. Instead, it saves the volatile S field into the PSR itself and the remaining fields are either altered in a reversible way (ET and CWP), or should not be altered in a trap handler until the PSR has been saved into memory.

The last two instructions of a trap handler should be a JMPL followed by a RETT. This restores the PC, the nPC and the S bit of the PSR.

Because the FPU and IU operate concurrently, the address that is saved from the PC as a result of a floating-point exception may not be the address of the FPop that caused the exception. If a floating-point exception occurs, the first element in the FQ points to the FPop that caused the exception, and the remaining elements point to FPops that have been started by the FPU but have not yet completed. These can be re-executed or emulated.

For additional information on trap handlers, see *Appendix C.*

5.5. Interrupt Detection

As long as ET = 1, the IU checks for interrupts. It compares the external interrupt level (*bp_IRL*) against the PIL field of the PSR, and if *bp_IRL* is greater than the PIL, or if *bp_IRL* is 15 (unmaskable), then a trap occurs at the level requested by *bp_IRL*.

5.5.1. Implementation Note

Processor implementations may ignore interrupts for multiple cycles even though ET=1.

5.6. Floating-point/Coprocessor Exception Traps

Floating-point/coprocessor exception traps are considered a separate class of traps because they are both synchronous and asynchronous. They are asynchronous because they occur sometime after the floating-point or coprocessor instruction that caused the exception. However, they are synchronous because a floating-point or coprocessor instruction must be encountered in the instruction stream before the trap is taken.

When the FPU or CP recognizes an exception condition, it enters an "exception_pending_mode" state, and remains in this state until the IU takes the fp_exception trap. When the IU takes the exception trap, the FPU leaves "exception_pending" state, and enters "exception_mode" state. The FPU or coprocessor remains in the exception_mode state until the floating-point or coprocessor queue has been emptied by execution of one or more STDFQ or STDCQ instructions.

The PC that corresponds to a floating-point or coprocessor exception always points to a floating-point or coprocessor instruction. However, the exception itself is always due to a previously executed floating-point or coprocessor instruction. The instruction and the value of the PC from which it was fetched are in the floating-point (or coprocessor) queue.

5.7. Trap Descriptions

The following paragraphs describe the various traps, and the conditions that cause them.

reset
: A reset trap occurs when the IU leaves reset_mode and enters execute_mode. This is controlled by the *bp_reset_in* signal. The IU enters reset_mode when *bp_reset_in* = 1, and enters execute mode when *bp_reset_in* = 0. Except in one situation, reset does **not** change the value of the *tt* field of the TBR; the exception is when a return from trap instruction is executed while traps are not enabled and the processor is not in supervisor mode (see description of return from trap instruction in *Appendix B*). Also, a reset trap causes the IU to begin execution at location 0, regardless of the value of the TBR.

  Reset traps set the PSR S bit to 1 and the ET bit to 0. All other PSR fields, and all other registers retain their values from the last execute_mode, except that on power-up they are undefined.

instruction_access_exception
: This trap occurs when *bp_memory_access_exception* = 1 for a memory address used in an instruction fetch.

illegal_instruction
: This trap occurs 1) when the UNIMP instruction is encountered, 2) when an unimplemented instruction which is not an FPop or a CPop is encountered, or 3) when an instruction is fetched which, if executed, would result in an illegal processor state (e.g. writing an illegal CWP into the PSR). Unimplemented floating point operate and unimplemented coprocessor operate instructions generate fp_exceptions and cp_exception traps, respectively.

privileged_instruction
: This trap occurs when a privileged instruction is encountered while the S bit in the PSR = 0.

fp_disabled
: This trap occurs when a FPop, FBfcc, or a floating-point load or store is encountered while the EF bit in the PSR = 0 or no FPU is present.

cp_disabled
: This trap occurs when a CPop, CBccc, or a coprocessor load or store instruction is decoded while the EC bit in the PSR = 0 or no coprocessor is present.

window_overflow
: This trap occurs when a SAVE instruction would, if executed, cause the CWP to point to a window marked invalid in the WIM.

window_underflow
: This trap occurs when a RESTORE instruction would, if executed, cause the CWP to point to a window marked invalid in the WIM.

mem_address_not_aligned
: This trap occurs when a load, store or JMPL instruction would, if executed, generate a memory address or a new PC value that is not properly aligned.

fp_exception
: This trap occurs when the FPU is in exception_pending state and a floating-point instuction (FP operate, floating-point load/store, FBfcc) is encountered in the instruction stream. The type of exception is encoded in the *tt* field of the FSR as described in the section *Registers*.

cp_exception
: This trap occurs when the CP is in exception_pending state and a coprocessor instuction (CP operate, coprocessor load/store, CBccc) is encountered in the instruction stream.

data_access_exception
: This trap occurs when bp_memory_exception=1 for a memory address that corresponds to a data movement by a load or store instruction.

tag_overflow
: This trap occurs when a TADDccTV or TSUBccTV instruction is executed which causes the overflow bit of the integer condition codes to be set.

trap_instruction
: This trap occurs when a taken Ticc instruction is executed.

interrupt_level<3:0>
: External interrupts are controlled by the value of *bp_IRL*. A value of 0 indicates that no interrupt is requested. Level 1 is the lowest priority interrupt and 15 is the highest. Interrupt level 15 cannot be masked by the Processor Interrupt Level (PIL) field of the PSR. When ET = 1, an external interrupt is recognized if *bp_IRL* = 15 or *bp_IRL* > PIL. When ET = 0 or (*bp_IRL* ≠ 15 and *bp_IRL* ≤ PIL), no external interrupt is recognized.

APPENDIX A: SUGGESTED ASSEMBLY LANGUAGE

A.1. Introduction

This appendix supports *Appendix B, Instruction Descriptions*. Every instruction description in *Appendix B* includes a table that describes the suggested assembly language format for that instruction. This appendix describes the notation used in the assembly language syntax descriptions.

Understanding the use of type fonts is crucial to understanding the syntax descriptions in *Appendix B*. Items in *typewriter font* are literals, to be entered exactly as they appear. Items in *italic font* are metasymbols which are to be replaced by numeric or symbolic values when actual SPARC assembly-language code is written. For example, "*asi*" would be replaced by a number in the range of 0 to 255 (the value of the asi bits in the binary instruction), or by a symbol which had been bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, reg_{rs2} is a *reg* (i.e. register name) whose binary value will end up in the *rs2* field of the resulting instruction.

Register Names

reg A *reg* is an Integer Unit register. It can have a value of:

%0 through *%31*	all integer registers
%g0 through *%g7*	*global* registers — same as *%0* through *%7*
%o0 through *%o7*	*out* registers — same as *%8* through *%15*
%l0 through *%l7*	*local* registers — same as *%16* through *%23*
%i0 through *%i7*	*in* registers — same as *%24* through *%31*

Subscripts further identify the placement of the operand in the binary instruction as one of:

reg_{rs1} — *rs1* field
reg_{rs2} — *rs2* field
reg_{rd} — *rd* field

freg An *freg* is a floating-point register. It can have a value from *%f0* through *%f31*. Subscripts further identify the placement of the operand in the binary instruction as one of:

$freg_{rs1}$ — *rs1* field
$freg_{rs2}$ — *rs2* field
$freg_{rd}$ — *rd* field

creg

A *creg* is a coprocessor register. It can have a value from *%c0* through *%c31*. Subscripts further identify the placement of the operand in the binary instruction as one of:

$creg_{rs1}$ — *rs1* field
$creg_{rs2}$ — *rs2* field
$creg_{rd}$ — *rd* field

Special Symbol Names

Certain special symbols need to be written exactly as they appear in the syntax table. These appear in *typewriter font*, and include a percent sign (%), also in typewriter font. The percent sign is part of the symbol name; it must appear as part of the literal value.

The symbol names are:

%psr	Processor State Register
%wim	Window Invalid Mask register
%tbr	Trap Base Register
%y	Y register
%fsr	Floating-point State Register
%csr	Coprocessor State Register
%fq	Floating-point Queue
%cq	Coprocessor Queue
%hi	Unary operator that extracts high 22 bits of its operand
%lo	Unary operation that extracts low 10 bits of its operand

Values

Some instructions use operands comprising values as follows:

simm13 — A signed immediate constant that fits in 13 bits

const22 — A constant that fits in 22 bits

asi — An alternate address space identifier (0 to 255)

Label

A sequence of characters, comprised of alphabetic letters (a-z, A-Z [upper and lower case distinct]), underscore (_), dollar sign ($), period (.), and decimal digits (0-9), which does not begin with a decimal digit.

Some instructions offer a choice of operands. These are grouped as follows:

regaddr:

reg_{rs1}
$reg_{rs1} + reg_{rs2}$

address:

reg_{rs1}
$reg_{rs1} + reg_{rs2}$
$reg_{rs1} + simm13$
$reg_{rs1} - simm13$
simm13
$simm13 + reg_{rs1}$

reg_or_imm

reg_{rs2}
simm13

APPENDIX B: INSTRUCTION DEFINITIONS

B.1. Introduction

This appendix describes the SPARC architecture's instruction set. A more detailed, algorithmic definition of the instruction set appears in *Appendix C.*

Related instructions are grouped into subsections. Each subsection consists of five parts:

(1) A **table** of the opcodes defined in the subsection with the values of the field(s) which uniquely identify the instruction(s).

(2) An illustration of the applicable instruction **format(s)**.

(3) A table of the suggested **assembly language syntax**. (The syntax notation is described in Appendix A.)

(4) A **description** of the salient features, restrictions, and trap conditions.

(6) A list of the synchronous or floating-point/coprocessor traps which can occur as a consequence of executing the instruction(s).

This section does not include any timing information (in either cycles or absolute time) since timing is strictly implementation-dependent.

The following table lists all the instructions:

Opcode	Name
LDSB (LDSBA†)	Load Signed Byte (from Alternate space)
LDSH (LDSHA†)	Load Signed Halfword (from Alternate space)
LDUB (LDUBA†)	Load Unsigned Byte (from Alternate space)
LDUH (LDUHA†)	Load Unsigned Halfword (from Alternate space)
LD (LDA†)	Load Word (from Alternate space)
LDD (LDDA)†	Load Doubleword (from Alternate space)
LDF	Load Floating-point
LDDF	Load Double Floating-point
LDFSR	Load Floating-point State Register
LDC	Load Coprocessor
LDDC	Load Double Coprocessor
LDCSR	Load Coprocessor State Register
STB (STBA†)	Store Byte (into Alternate space)
STH (STHA†)	Store Halfword (into Alternate space)
ST (STA†)	Store Word (into Alternate space)
STD (STDA†)	Store Doubleword (into Alternate space)
STF	Store Floating-point
STDF	Store Double Floating-point
STFSR	Store Floating-point State Register
STDFQ†	Store Double Floating-point Queue
STC	Store Coprocessor
STDC	Store Double Coprocessor
STCSR	Store Coprocessor State Register
STDCQ†	Store Double Coprocessor Queue
LDSTUB (LDSTUBA†)	Atomic Load-Store Unsigned Byte (in Alternate space)
SWAP (SWAPA†)	Swap r Register with Memory (in Alternate space)
ADD (ADDcc)	Add (and modify icc)
ADDX (ADDXcc)	Add with Carry (and modify icc)
TADDcc (TADDccTV)	Tagged Add and modify icc (and Trap on overflow)
SUB (SUBcc)	Subtract (and modify icc)
SUBX (SUBXcc)	Subtract with Carry (and modify icc)
TSUBcc (TSUBccTV)	Tagged Subtract and modify icc (and Trap on overflow)
MULScc	Multiply Step and modify icc
AND (ANDcc)	And (and modify icc)
ANDN (ANDNcc)	And Not (and modify icc)
OR (ORcc)	Inclusive-Or (and modify icc)
ORN (ORNcc)	Inclusive-Or Not (and modify icc)
XOR (XORcc)	Exclusive-Or (and modify icc)
XNOR (XNORcc)	Exclusive-Nor (and modify icc)
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SETHI	Set High 22 bits of *r register*
SAVE	Save caller's window
RESTORE	Restore caller's window

Opcode	Name
Bicc	Branch on integer condition codes
FBfcc	Branch on floating-point condition codes
CBccc	Branch on coprocessor condition codes
CALL	Call
JMPL	Jump and Link
RETT†	Return from Trap
Ticc	Trap on integer condition codes
RDY	Read Y register
RDPSR†	Read Processor State Register
RDWIM†	Read Window Invalid Mask register
RDTBR†	Read Trap Base Register
WRY	Write Y register
WRPSR†	Write Processor State Register
WRWIM†	Write Window Invalid Mask register
WRTBR†	Write Trap Base Register
UNIMP	Unimplemented instruction
IFLUSH	Instruction cache Flush
FPop	Floating-point Operate: FiTO(s,d,x), F(s,d,x)TOi FsTOd, FsTOx, FdTOs, FdTOx, FxTOs, FxTOd, FMOVs, FNEGs, FABSs, FSQRT(s,d,x), FADD(s,d,x), FSUB(s,d,x), FMUL(s,d,x), FDIV(s,d,x), FCMP(s,d,x), FCMPE(s,d,x)
CPop	Coprocessor operate

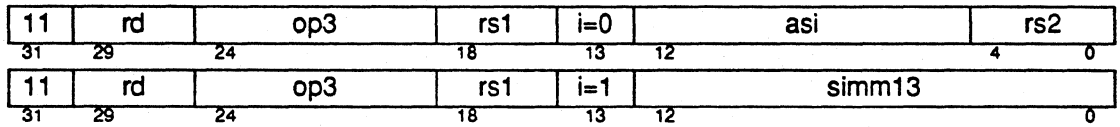
† privileged instruction

B.2. Load Integer Instructions

opcode	op3	operation
LDSB	001001	Load Signed Byte
LDSBA†	011001	Load Signed Byte from Alternate space
LDSH	001010	Load Signed Halfword
LDSHA†	011010	Load Signed Halfword from Alternate space
LDUB	000001	Load Unsigned Byte
LDUBA†	010001	Load Unsigned Byte from Alternate space
LDUH	000010	Load Unsigned Halfword
LDUHA†	010010	Load Unsigned Halfword from Alternate space
LD	000000	Load Word
LDA†	010000	Load Word from Alternate space
LDD	000011	Load Doubleword
LDDA†	010011	Load Doubleword from Alternate space

† privileged instruction

Format (3):



Suggested Assembly Language Syntax	
ldsb	[*address*], *reg*$_{rd}$
ldsba	[*regaddr*] *asi*, *reg*$_{rd}$
ldsh	[*address*], *reg*$_{rd}$
ldsha	[*regaddr*] *asi*, *reg*$_{rd}$
ldub	[*address*], *reg*$_{rd}$
lduba	[*regaddr*] *asi*, *reg*$_{rd}$
lduh	[*address*], *reg*$_{rd}$
lduha	[*regaddr*] *asi*, *reg*$_{rd}$
ld	[*address*], *reg*$_{rd}$
lda	[*regaddr*] *asi*, *reg*$_{rd}$
ldd	[*address*], *reg*$_{rd}$
ldda	[*regaddr*] *asi*, *reg*$_{rd}$

Description:

The load single integer instructions move either a byte, halfword, or word from memory into the *r register* defined by the *rd* field. A fetched byte or halfword is right-justified in *rd* and may be either zero-filled or sign-extended.

The load double integer instructions (LDD, LDDA) move a doubleword from memory into an *r register pair*. The most significant word at the effective memory address is moved into the even *r register*. The least significant word at the effective memory address + 4 is moved into the odd *r register*. The least significant bit of the *rd* field is ignored. (Note that a load double with rd = 0 modifies only r[1].)

The effective address for a load instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. Instructions which load from an alternate

address space must have zero in the *i* field and the address space identifier to be used for the load in the *asi* field. Otherwise the address space indicates either a user or system data space access, according to the S bit of the PSR.

LD and LDA cause a mem_address_not_aligned trap if the effective address is not word-aligned; LDUH, LDSH, LDUHA, and LDSHA trap if the address is not halfword-aligned; and LDD and LDDA trap if the address is not doubleword-aligned.

If a load single instruction traps, the destination register remains unchanged.

If a load double instruction is trapped with a data access exception during the effective address memory access, the destination registers remain unchanged. However a specific implementation might cause a data_access_exception trap during the effective address + 4 memory access, but not during the effective address access. Thus, the *even* destination *r register* can be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.)

B.2.1. Implementation Note:

On effective address + 4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

B.2.2. Programming Note

The execution time of a load integer instruction may increase if the next instruction uses the register specified by the *rd* field of the load instruction as a source operand (*rs1* or *rs2*). In the case of load doubleword instructions, this applies to both destination registers. Whether the time increase occurs or not is implementation-dependent.

B.2.3. Programming Note

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

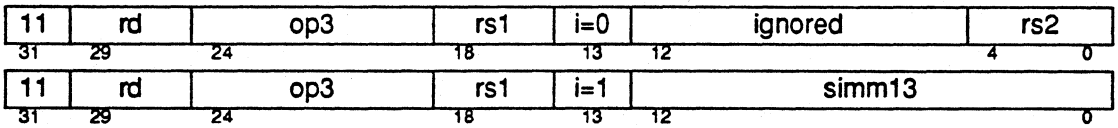
Traps:

illegal_instruction (load alternate space with i = 1)
privileged_instruction (load alternate space only)
mem_address_not_aligned (excluding LDSB, LDSBA, LDUB, and LDUBA)
data_access_exception

B.3. Load Floating-point Instructions

opcode	op3	operation
LDF	100000	Load Floating-point register
LDDF	100011	Load Double Floating-point register
LDFSR	100001	Load Floating-point State Register

Format (3):



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

Description:

The load single floating-point instruction (LDF) moves a word from memory into the *f register* identified by the *rd* field.

The load double floating-point instruction (LDDF) moves a doubleword from memory into an *f register pair*. The most significant word at the effective memory address is moved into the even *f register*. The least significant word at the effective memory address + 4 is moved into the odd *f register*. The least significant bit of the *rd* field is ignored.

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

The effective address for the load instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

LDF and LDFSR cause a mem_address_not_aligned trap if the effective address is not word-aligned; and LDDF traps if the address is not doubleword-aligned. A load floating-point instruction causes an fp_disabled trap if the EF field of the PSR is 0 or if no FPU is present.

If a load single floating-point instruction is trapped with a data access exception, the destination *f register* either remains unchanged or is set to an implementation-defined constant value.

If a load double floating-point instruction is trapped with a data access exception, either the destination *f registers* remain unchanged or one or both are set to an implementation-defined constant value.

B.3.1. Programming Note

The execution time of a load floating-point instruction may increase if the next instruction uses the register specified by the *rd* field of the load instruction as a source operand (*rs1* or *rs2*). In the case of load double floating-point instructions, this applies to both destination registers. Whether the time increases or not is implementation-dependent.

B.3.2. Programming Note

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

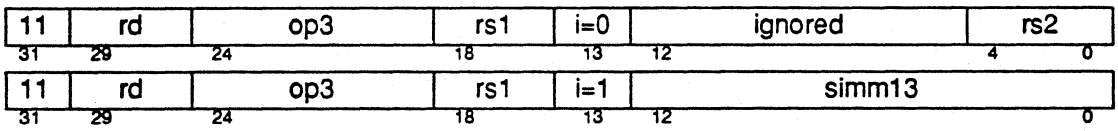
Traps:

fp_disabled
fp_exception
mem_address_not_aligned
data_access_exception

B.4. Load Coprocessor Instructions

opcode	op3	operation
LDC	110000	Load Coprocessor register
LDDC	110011	Load Double Coprocessor register
LDCSR	110001	Load Coprocessor State Register

Format (3):



Suggested Assembly Language Syntax	
ld	[*address*], $creg_{rd}$
ldd	[*address*], $creg_{rd}$
ld	[*address*], *%csr*

Description:

The load single coprocessor instruction (LDC) moves a word from memory into a coprocessor register. The load double coprocessor instruction (LDDC) moves a doubleword from memory into a coprocessor register pair. The load coprocessor state register instruction (LDCSR) moves a word from memory into the Coprocessor State Register. The semantics of these instructions depend on the implementation of the attached coprocessor.

The effective address for the load instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

LDC and LDCSR cause a mem_address_not_aligned trap if the effective address is not word-aligned; and LDDC traps if the address is not doubleword-aligned. A load coprocessor instruction causes a cp_disabled trap if the EC field of the PSR is 0 or if no coprocessor is present.

If a load coprocessor instruction traps, the state of the coprocessor depends on its implementation.

B.4.1. Implementation Note:

On effective address + 4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

B.4.2. Programming Note

The execution time of a load coprocessor instruction may increase if the next instruction uses the register specified by the *rd* field of the load instruction as a source operand (*rs1* or *rs2*). In the case of load double coprocessor instructions, this applies to both destination registers. Whether the time increases or not is implementation-dependent.

B.4.3. Programming Note

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

Traps:

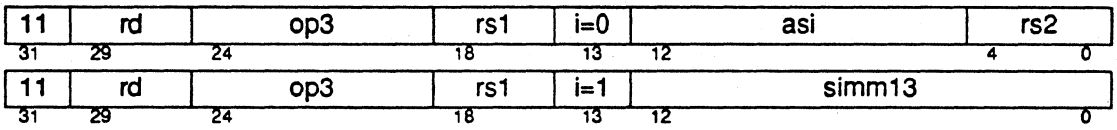
cp_disabled
cp_exception
mem_address_not_aligned
data_access_exception

B.5. Store Integer Instructions

opcode	op3	operation
STB	000101	Store Byte
STBA†	010101	Store Byte into Alternate space
STH	000110	Store Halfword
STHA†	010110	Store Halfword into Alternate space
ST	000100	Store Word
STA†	010100	Store Word into Alternate space
STD	000111	Store Doubleword
STDA†	010111	Store Doubleword into Alternate space

† privileged instruction

Format (3):



Suggested Assembly Language Syntax			
stb	reg_{rd}, [*address*]		*synonyms:stub, stsb*
stba	reg_{rd}, [*regaddr*]	*asi*	*synonyms:stuba, stsba*
sth	reg_{rd}, [*address*]		*synonyms:stuh, stsba*
stha	reg_{rd}, [*regaddr*]	*asi*	*synonyms:stuha, stsha*
st	reg_{rd}, [*address*]		
sta	reg_{rd}, [*regaddr*]	*asi*	
std	reg_{rd}, [*address*]		
stda	reg_{rd}, [*regaddr*]	*asi*	

Description:

The store single integer instructions move the word, the least significant halfword, or the least significant byte from the *r register* specified by the *rd* field into memory.

The store double integer instructions (STD, STA) move a doubleword from an *r register pair* into memory. The most significant word in the even *r register* is written into memory at the effective address and the least significant word in the following odd *r register* is written into memory at the effective address + 4.

The effective address for a store instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. Instructions which store to an alternate address space must have zero in the *i* field and the address space identifier to be used for the store in the *asi* field. Otherwise the address space indicates either a user or system data space access, according to the S bit in the PSR.

ST and STA cause a mem_address_not_aligned trap if the effective address is not word-aligned; STH and STHA trap if the address is not halfword-aligned; and STD and STDA trap if the address is not doubleword-aligned.

If a store single instruction traps, memory remains unchanged. However, in the case of a store double, an implementation might cause a data_access_exception trap during the effective address + 4 memory access, but not during the effective address access. Thus,

data at the effective memory address can be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.)

B.5.1. Implementation Note:

On effective address + 4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

B.5.2. Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of memory can be written without using a register.

Traps:

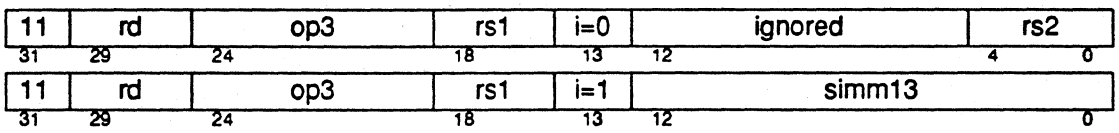
illegal_instruction (store alternate with i = 1)
privileged_instruction (store alternate only)
mem_address_not_aligned (excluding STB and STBA)
data_access_exception

B.6. Store Floating-point Instructions

opcode	op3	operation
STF	100100	Store Floating-point
STDF	100111	Store Double Floating-point
STFSR	100101	Store Floating-point State Register
STDFQ†	100110	Store Double Floating-point Queue

† privileged instruction

Format (3):



Suggested Assembly Language Syntax	
st	*freg$_{rd}$, [address]*
std	*freg$_{rd}$, [address]*
st	*%fsr, [address]*
std	*%fq, [address]*

Description:

The store single floating-point instruction (STF) moves the contents of the *f register* specified by the *rd* field into memory.

The store double floating-point instruction (STDF) moves a doubleword from an *f register pair* into memory. The most significant word in the even *f register* is written into memory at the effective address and the least significant word in the odd *f register* is written into memory at the effective address + 4.

The store floating-point queue instruction (STDFQ) stores the front entry of the Floating-point Queue (FQ) into memory. The address part of the front entry is stored into memory at the effective address, and the instruction part of the front entry at the effective address + 4. If the FPU is in exception_mode, the queue is then advanced to the next entry, or it becomes empty (as indicated by the *qne* bit in the FSR).

The store floating-point state register instruction (STFSR) waits for all FPops that have not finished execution to complete and then writes the FSR into memory.

The effective address for a store instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

STF and STFSR cause a mem_address_not_aligned trap if the address is not word-aligned and STDF and STDFQ trap if the address is not doubleword-aligned. A store floating-point instruction causes an fp_disabled trap if the EF field of the PSR is 0 or if the FPU is not present.

If a store single floating-point instruction traps, memory remains unchanged. However, in the case of a store double, an implementation may cause a data_access_exception trap during the effective address + 4 memory access, but not during the effective address access. Data at the effective memory address can be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.)

B.6.1. Implementation Note:

On effective address + 4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

Traps:

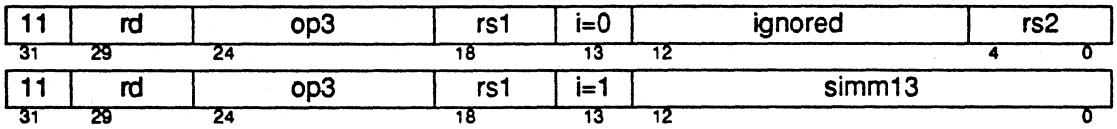
fp_disabled
fp_exception
privileged_instruction (STDFQ only)
mem_address_not_aligned
data_access_exception

B.7. Store Coprocessor Instructions

opcode	op3	operation
STC	110100	Store Coprocessor
STDC	110111	Store Double Coprocessor
STCSR	110101	Store Coprocessor State Register
STDCQ†	110110	Store Double Coprocessor Queue

† privileged instruction

Format (3):



Suggested Assembly Language Syntax	
st	$creg_{rd}$, [*address*]
std	$creg_{rd}$, [*address*]
st	*%csr, [address]*
std	*%cq, [address]*

Description:

The store single coprocessor instruction (STC) moves the contents of a coprocessor register into memory. The store double coprocessor instruction (STDC) moves the contents of a coprocessor register pair into memory. The store coprocessor state register instruction (STCSR) moves the contents of the coprocessor state register into memory. The store double coprocessor queue instruction (STDCQ) moves the front entry of the coprocessor queue into memory. The semantics of these instructions depend on the implementation of the attached coprocessor, if any.

The effective address for a store instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

STC and STCSR cause a mem_address_not_aligned trap if the address is not word-aligned and STDC and STDCQ trap if the address is not doubleword-aligned. A store coprocessor instruction causes a cp_disabled trap if the EC field of the PSR is 0 or if no coprocessor is present.

If a store single coprocessor instruction traps, memory remains unchanged. However, in the case of a store double, an implementation might cause a data_access_exception trap during the effective address + 4 memory access, but not during the effective address access. Thus, data at the effective memory address can be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.)

B.7.1. Implementation Note:

On effective address + 4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

Traps:

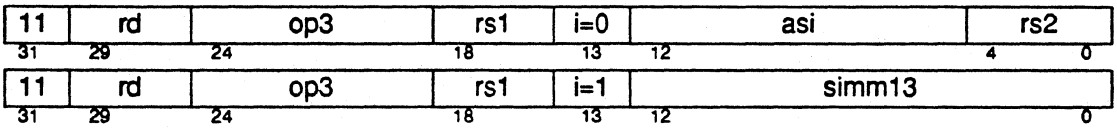
cp_disabled
cp_exception
privileged_instruction (STDCQ only)
mem_address_not_aligned
data_access_exception

B.8. Atomic Load-Store Unsigned Byte Instructions

opcode	op3	operation
LDSTUB	001101	Atomic Load-Store Unsigned Byte
LDSTUBA†	011101	Atomic Load-Store Unsigned Byte into Alternate space

† privileged instruction

Format (3):



Suggested Assembly Language Syntax	
ldstub	[*address*], reg_{rd}
ldstuba	[*regaddr*] *asi*, reg_{rd}

Description:

The atomic load-store instructions move a byte from memory into an *r register* identified by the *rd* field and then rewrite the same byte in memory to all ones without allowing intervening asynchronous traps. In a multiprocessor system, two or more processors executing atomic load-store instructions addressing the same byte simultaneously are guaranteed to execute them in some serial order.

The effective address of an atomic load-store is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. LDSTUBA must have zero in the *i* field, or an illegal_instruction trap occurs. The address space identifier used for the memory accesses is taken from the *asi* field. For LDSTUB, the address space indicates either a user or system data space access, according to the S bit in the PSR.

If an atomic load-store instruction traps, memory remains unchanged. However, an implementation may cause a data_access_exception trap during the store memory access, but not during the load access. In this case, the destination register can be changed.

B.8.1. Implementation Note:

The system should limit data_access_exceptions on the store access to non-restartable errors, such as protection violation or uncorrectable memory errors.

B.8.2. Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of memory can be accessed without using a register.

Traps:

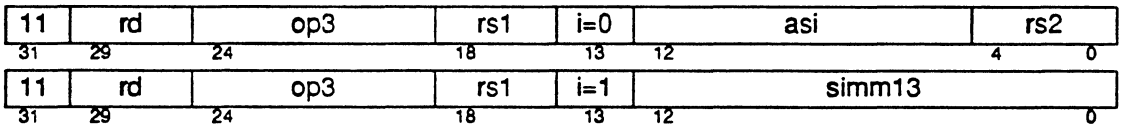
illegal_instruction (LDSTUBA with i = 1 only)
privileged_instruction (LDSTUBA only)
data_access_exception

B.9. SWAP r Register with Memory

opcode	op3	operation
SWAP	001111	SWAP *r register* with memory
SWAPA†	011111	SWAP *r register* with Alternate space memory

† privileged instruction

Format (3):



Suggested Assembly Language Syntax	
swap	[*source*], reg_{rd}
swapa	[*regsource*] *asi*, reg_{rd}

Description:

The swap instructions exchange the *r register* identified by the *rd* field with the contents of the addressed memory location. This is performed atomically without allowing asynchronous traps. In a multiprocessor system, two or more processors issuing swap instructions simultaneously are guaranteed to get results corresponding to the executing the instructions serially, in some order.

The effective address of the swap instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. SWAPA must have zero in the *i* field or an illegal_instruction trap occurs. The address space identifier used for the memory accesses is taken from the *asi* field. For SWAP, the address space indicates either a user or a system data space access, according to the S bit in the PSR.

These instructions cause a mem_address_not_aligned trap if the effective address is not word-aligned.

If a swap instruction traps, memory remains unchanged.

B.9.1. Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of memory can be written without using a register.

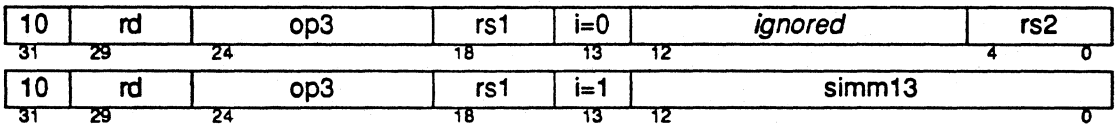
Traps:

illegal instruction (i = 1 and SWAPA only)
privileged_instruction (SWAPA only)
data_access_exception

B.10. Add Instructions

opcode	op3	operation
ADD	000000	Add
ADDcc	010000	Add and modify icc
ADDX	001000	Add with Carry
ADDXcc	011000	Add with Carry and modify icc

Format (3):



Suggested Assembly Language Syntax	
add	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
addcc	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
addx	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
addxcc	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description:

ADD and ADDcc compute either"r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one, and place the result in the *r register* specified in the *rd* field.

ADDX and ADDXcc add the PSR's carry (*c*) bit also; that is, they compute "r[rs1] + r[rs2] + *c*" or "r[rs1] + sign_ext(simm13) + *c*" and place the result in the *r register* specified in the *rd* field.

ADDcc and ADDXcc modify **all** the integer condition codes.

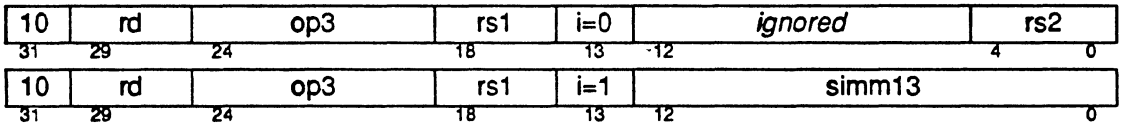
Traps:

(none)

B.11. Tagged Add Instructions

opcode	op3	operation
TADDcc	100000	Tagged Add and modify icc
TADDccTV	100010	Tagged Add, modify icc and Trap on Overflow

Format (3):



Suggested Assembly Language Syntax	
taddcc	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
taddcctv	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description:

These instructions compute either"r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. An overflow condition exists if bit 1 or bit 0 of either operand is not zero, or if the addition generates an arithmetic overflow.

If a TADDccTV causes an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If a TADDccTV does not cause an overflow condition, all the integer condition codes are updated (in particular, the overflow bit (*v*) is set to 0) and the result of the addition is written into the *r register* specified by the *rd* field.

If a TADDcc causes an overflow condition, the overflow bit (*v*) of the PSR is set; if it does not cause an overflow, it is cleared. In either case, the remaining integer condition codes are also updated and the result of the addition is written into the *r register* specified by the *rd* field.

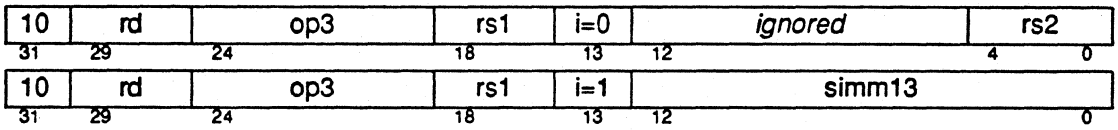
Traps:

tag_overflow (TADDccTV only)

B.12. Subtract Instructions

opcode	op3	operation
SUB	000100	Subtract
SUBcc	010100	Subtract and modify icc
SUBX	001100	Subtract with Carry
SUBXcc	011100	Subtract with Carry and modify icc

Format (3):



Suggested Assembly Language Syntax	
sub	reg_{rs1}, *reg_or_imm*, reg_{rd}
subcc	reg_{rs1}, *reg_or_imm*, reg_{rd}
subx	reg_{rs1}, *reg_or_imm*, reg_{rd}
subxcc	reg_{rs1}, *reg_or_imm*, reg_{rd}

Description:

These instructions compute either "r[rs1] - r[rs2]" if the *i* field is zero, or "r[rs1] - sign_ext(simm13)" if the *i* field is one, and place the result in the *r register* specified in the *rd* field.

SUBX and SUBXcc ("SUBtract eXtended") also subtract the PSR's carry (*c*) bit; that is, they compute "r[rs1] - r[rs2] - *c*" or "r[rs1] - sign_ext(simm13) - *c*" and place the result in the *r register* specified in the *rd* field.

SUBcc and SUBXcc modify **all** the integer condition codes.

B.12.1. Programming Note

A SUBcc with *rd* = 0 can be used for signed and unsigned integer compare.

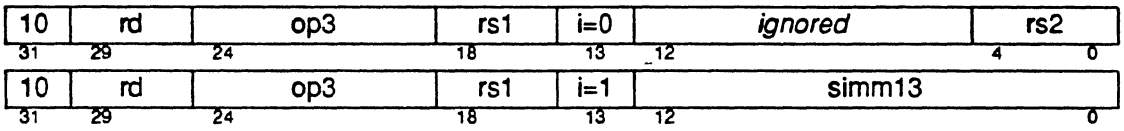
Traps:

(none)

B.13. Tagged Subtract Instructions

opcode	op3	operation
TSUBcc	100001	Tagged Subtract and modify icc
TSUBccTV	100011	Tagged Subtract, modify icc and Trap on Overflow

Format (3):



Suggested Assembly Language Syntax	
tsubcc	reg_{rs1}, *reg_or_imm*, reg_{rd}
tsubcctv	reg_{rs1}, *reg_or_imm*, reg_{rd}

Description:

These instructions compute either"r[rs1] - r[rs2]" if the *i* field is zero, or "r[rs1] - sign_ext(simm13)" if the *i* field is one. An overflow condition exists if bit 1 or bit 0 of either operand is not zero, or if the subtraction generates an arithmetic overflow.

If a TSUBccTV causes an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If a TSUBccTV does not cause an overflow condition, the integer condition codes are updated (in particular, the overflow bit (*v*) is set to 0) and the result of the subtraction is written into the *r register* specified by the *rd* field.

If a TSUBcc causes an overflow condition, the overflow bit (*v*) of the PSR is set; if it does not cause an overflow, it is cleared. In either case, the remaining integer condition codes are also updated and the result of the subtraction is written into the *r register* specified by the *rd* field.

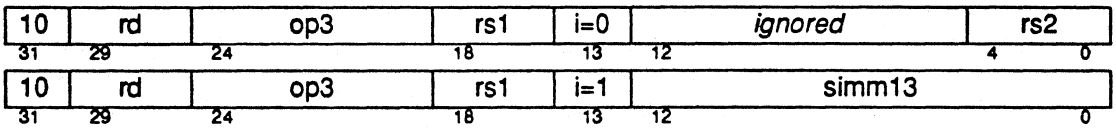
Traps:

tag_overflow (TSUBccTV only)

B.14. Multiply Step Instruction

opcode	op3	operation
MULScc	100100	Multiply Step and modify icc

Format (3):



Suggested Assembly Language Syntax	
mulscc	reg_{rs1}, *reg_or_imm*, reg_{rd}

Description:

The multiply step instruction can be used to generate the 64-bit product of two signed or unsigned words (See Appendix E). MULScc works as follows:

1. The value obtained by shifting "r[rs1]" (the incoming partial product) right by one bit and replacing its high-order bit by "N xor V" (the sign of the previous partial product) is computed.
2. If the least significant bit of the Y register (the multiplier) is set, the value from step (1) is added to the multiplicand. The multiplicand is "r[rs2]" if the *i* field is zero or is "sign_ext(simm13)" if the *i* field is one. If the LSB of the Y register is not set, then zero is added to the value from step (1).
3. The result from step (2) is written into "r[rd]" (the outgoing partial product). The PSR's integer condition codes are updated according to the addition performed in step (2).
4. The Y register (the multiplier) is shifted right by one bit and its high-order bit is replaced by the least significant bit of "r[rs1]" (the incoming partial product).

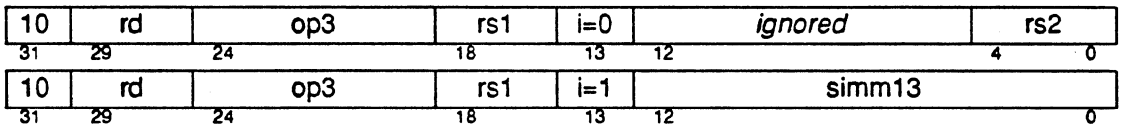
Traps:

(none)

B.15. Logical Instructions

opcode	op3	operation
AND	000001	And
ANDcc	010001	And and modify icc
ANDN	000101	And Not
ANDNcc	010101	And Not and modify icc
OR	000010	Inclusive Or
ORcc	010010	Inclusive Or and modify icc
ORN	000110	Inclusive Or Not
ORNcc	010110	Inclusive Or Not and modify icc
XOR	000011	Exclusive Or
XORcc	010011	Exclusive Or and modify icc
XNOR	000111	Exclusive Nor
XNORcc	010111	Exclusive Nor and modify icc

Format (3):



Suggested Assembly Language Syntax			
and	reg_{rs1},	*reg_or_imm*,	reg_{rd}
andcc	reg_{rs1},	*reg_or_imm*,	reg_{rd}
andn	reg_{rs1},	*reg_or_imm*,	reg_{rd}
andncc	reg_{rs1},	*reg_or_imm*,	reg_{rd}
or	reg_{rs1},	*reg_or_imm*,	reg_{rd}
orcc	reg_{rs1},	*reg_or_imm*,	reg_{rd}
orn	reg_{rs1},	*reg_or_imm*,	reg_{rd}
orncc	reg_{rs1},	*reg_or_imm*,	reg_{rd}
xor	reg_{rs1},	*reg_or_imm*,	reg_{rd}
xorcc	reg_{rs1},	*reg_or_imm*,	reg_{rd}
xnor	reg_{rs1},	*reg_or_imm*,	reg_{rd}
xnorcc	reg_{rs1},	*reg_or_imm*,	reg_{rd}

Description:

These instructions implement the bitwise logical operations. They compute either "r[rs1] **op** r[rs2]" if the *i* field is zero, or "r[rs1] **op** sign_ext(simm13)" if the *i* field is one (**op** = and, and not, or, or not, xor, xnor).

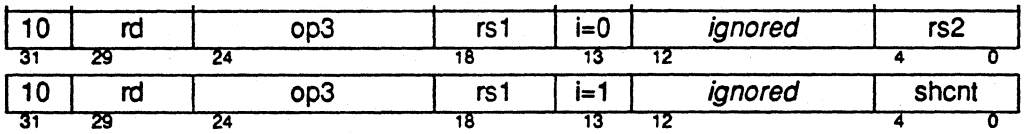
ANDcc, ANDNcc, ORcc, ORNcc, XORcc and XNORcc modify **all** the integer condition codes as described in the section *Registers*.

Traps: (none)

B.16. Shift Instructions

opcode	op3	operation
SLL	100101	Shift Left Logical
SRL	100110	Shift Right Logical
SRA	100111	Shift Right Arithmetic

Format (3):



Suggested Assembly Language Syntax	
sll	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
srl	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
sra	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description:

The shift count for these instructions is the least significant five bits of either "r[rs2]" if the *i* field is zero, or "simm13" if the *i* field is one. (The least significant five bits of "simm13" is called "shcnt" in the above format.)

SLL shifts the value of "r[rs1]" left by the number of bits implied by the shift count.

SRL and SRA shift the value of "r[rs1]" right by the number of bits implied by the shift count.

SLL and SRL replace vacated positions with zeroes, whereas SRA fills vacated positions with the most significant bit of "r[rs1]." No shift occurs when the shift count is zero.

All of these instructions place the shifted result in the *r register* specified in the *rd* field.

These instructions do **not** modify the condition codes.

B.16.1. Programming Note

"Arithmetic left shift by 1 (and calculate overflow)" can be implemented with an ADDcc instruction.

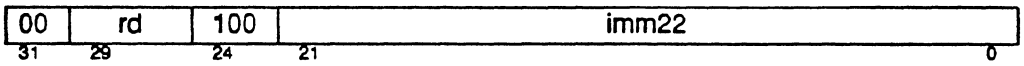
Traps:

(none)

B.17. SETHI Instruction

opcode	op	op2	operation
SETHI	00	100	Set High

Format (2):



Suggested Assembly Language Syntax	
sethi	*const22*, *reg*$_{rd}$
sethi	*%hi(value)*, *reg*$_{rd}$

Description:

SETHI zeroes the least significant 10 bits of "r[rd]" and replaces its high-order 22 bits with *imm22*.

The condition codes are not affected.

B.17.1. Programming Note

It is suggested that *sethi 0, %0* be used as the preferred NOP, since it will not cause an increase in execution time if it follows a load instruction.

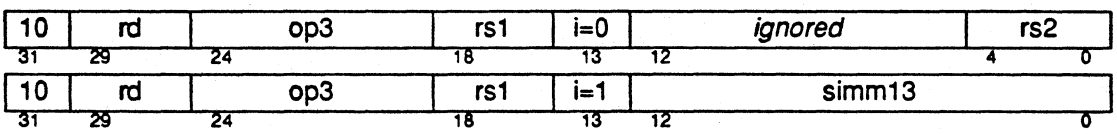
Traps:

(none)

B.18. SAVE and RESTORE Instructions

opcode	op3	operation
SAVE	111100	Save caller's window
RESTORE	111101	Restore caller's window

Format (3):



Suggested Assembly Language Syntax	
save	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
restore	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description:

The SAVE instruction subtracts one from the CWP (modulo the number of implemented windows) and compares this value, the "new_CWP," against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is set, "(WIM and 2^{new_CWP}) = 1," then a window_overflow trap is generated. If the WIM bit corresponding to the new_CWP is reset, then a window_overflow trap is not generated and new_CWP is written into CWP. This causes the **active** window to become the **previous** window, thereby saving the caller's window.

The RESTORE instruction adds one to the CWP (modulo the number of implemented windows) and compares this value, the "new_CWP," against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is set, "(WIM and 2^{new_CWP}) = 1," then a window_underflow trap is generated. If the WIM bit corresponding to the new_CWP is reset, then a window_underflow trap is not generated and new_CWP is written into CWP. This causes the **previous** window to become the **active** window, thereby restoring the caller's window.

Furthermore, if an overflow or underflow trap is **not** generated, SAVE and RESTORE behave like normal ADD instructions, except that the operands "r[rs1]" or "r[rs2]" are read from the **old** window (i.e., the window addressed by the original CWP) and the result is written into "r[rd]" of the **new** window (i.e., the window addressed by new_CWP).

Note that CWP arithmetic is performed modulo the number of implemented windows (NWINDOWS).

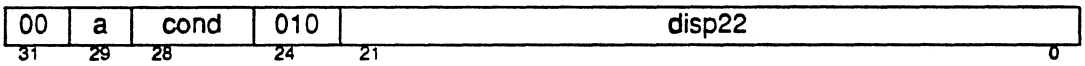
Traps:

window_overflow (SAVE only)
window_underflow (RESTORE only)

B.19. Branch on Integer Condition Instructions

opcode	cond	operation	Icc test
BA	1000	Branch Always	1
BN	0000	Branch Never	0
BNE	1001	Branch on Not Equal	not Z
BE	0001	Branch on Equal	Z
BG	1010	Branch on Greater	not (Z or (N xor V))
BLE	0010	Branch on Less or Equal	Z or (N xor V)
BGE	1011	Branch on Greater or Equal	not (N xor V)
BL	0011	Branch on Less	N xor V
BGU	1100	Branch on Greater Unsigned	not (C or Z)
BLEU	0100	Branch on Less or Equal Unsigned	(C or Z)
BCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	not C
BCS	0101	Branch on Carry Set (Less than, Unsigned)	C
BPOS	1110	Branch on Positive	not N
BNEG	0110	Branch on Negative	N
BVC	1111	Branch on Overflow Clear	not V
BVS	0111	Branch on Overflow Set	V

Format (2):



Suggested Assembly Language Syntax		
ba{,a}	*label*	
bn{,a}	*label*	
bne{,a}	*label*	*synonym: bnz*
be{,a}	*label*	*synonym: bz*
bg{,a}	*label*	
ble{,a}	*label*	
bge{,a}	*label*	
bl{,a}	*label*	
bgu{,a}	*label*	
bleu{,a}	*label*	
bcc{,a}	*label*	*synonym: bgeu*
bcs{,a}	*label*	*synonym: blu*
bpos{,a}	*label*	
bneg{,a}	*label*	
bvc{,a}	*label*	
bvs{,a}	*label*	

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

To set the "annul" bit for Bicc instructions, append an (optional) *",a" to the opcode. For example, use "bgu,a label". The preceding table indicates that the ",a" is optional by enclosing it in braces ({}).*

Description:

A Bicc instruction (except BA and BN) evaluates the integer condition codes (*icc*) according to the *cond* field. If the condition codes evaluate to true the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign_ext (disp22))." If the condition codes evaluate to false, the branch is not taken. If the branch is not taken and the *a* (annul) field is set, the delay instruction is not executed (annulled). If the branch is taken, the annul field is ignored. (Annulment, delay instructions, and delayed control transfers are described further in the section *Instructions.*)

BN (Branch Never) acts like a "NOP." except that, if the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

BA (Branch Always) causes a transfer of control, irrespective of the value of the condition code bits. If the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

Except for BA, all Bicc instructions with a=1 annul the delay instruction when the branch is not taken. However, BA with a=1 does the reverse: it annuls the delay instruction even though the branch is taken.

The delay instruction of a Bicc, other than a BA, should not be a delayed control-transfer instruction.

B.19.1. Programming Note

An untaken branch takes as much or more time than a taken branch. The additional time it takes is implementation-dependent.

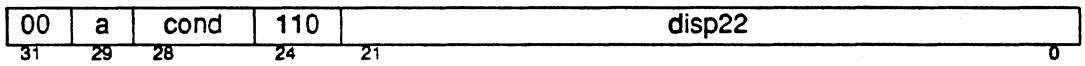
Traps:

(none)

B.20. Floating-point Branch on Condition Instructions

opcode	cond	operation	fcc test
FBA	1000	Branch Always	1
FBN	0000	Branch Never	0
FBU	0111	Branch on Unordered	U
FBG	0110	Branch on Greater	G
FBUG	0101	Branch on Unordered or Greater	G or U
FBL	0100	Branch on Less	L
FBUL	0011	Branch on Unordered or Less	L or U
FBLG	0010	Branch on Less or Greater	L or G
FBNE	0001	Branch on Not Equal	L or G or U
FBE	1001	Branch on Equal	E
FBUE	1010	Branch on Unordered or Equal	E or U
FBGE	1011	Branch on Greater or Equal	E or G
FBUGE	1100	Branch on Unordered or Greater or Equal	E or G or U
FBLE	1101	Branch on Less or Equal	E or L
FBULE	1110	Branch on Unordered or Less or Equal	E or L or U
FBO	1111	Branch on Ordered	E or L or G

Format (2):



Suggested Assembly Language Syntax		
fba{,*a*}	*label*	
fbn{,a}	*label*	
fbu{,a}	*label*	
fbg{,a}	*label*	
fbug{,a}	*label*	
fbl{,a}	*label*	
fbul{,a}	*label*	
fblg{,a}	*label*	
fbne{,a}	*label*	*synonym: fbnz*
fbe{,a}	*label*	*synonym: fbz*
fbue{,a}	*label*	
fbge{,a}	*label*	
fbuge{,a}	*label*	
fble{,a}	*label*	
fbule{,a}	*label*	
fbo{,a}	*label*	

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

To set the "annul" bit for FBfcc instructions, append an (optional) ",a" to the opcode. For example, use "*fbl,a label*". *The preceding table indicates that the ",a"* is optional by enclosing it in braces ({}).

Description:

An FBfcc instruction (except FBA and FBN) evaluates the floating-point condition codes (*fcc*) according to the *cond* field. If the condition codes evaluate to true the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign_ext (disp22))." If the condition codes evaluate to false, the branch is not taken. If the branch is not taken and the *a* (annul) field is set, the delay instruction is not executed (annulled). If the branch is taken, the annul field is ignored and the delay instruction is executed. (Annulment, delay instructions, and delayed control transfers are described further in the section *Instructions*.)

FBN (Branch Never) acts like a "NOP", except that if the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

FBA (Branch Always) causes a transfer of control, irrespective of the value of the condition code bits. If the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

An FBfcc instruction generates an fp_disabled trap (and does not branch on annul) if the PSR's EF bit is reset or if the FPU is not present.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

Except for FBA, all FBfcc instructions with a=1 annul the delay instruction when the branch is not taken. However, FBA with a=1 does the reverse: it annuls the delay instruction even though the branch is taken.

The instruction executed immediately before an FBfcc must not be a floating-point instruction.

B.20.1. Programming Note

An untaken branch takes as much or more time than a taken branch. The additional time it takes is implementation-dependent.

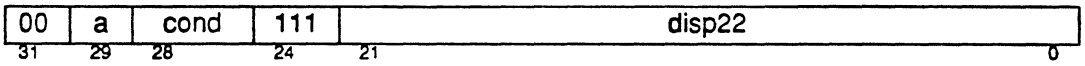
Traps:

fp_disabled
fp_exception

B.21. Coprocessor Branch on Condition Instructions

opcode	cond	bp_CP_cc[1:0] test
CBA	1000	Always
CBN	0000	Never
CB3	0111	3
CB2	0110	2
CB23	0101	2 or 3
CB1	0100	1
CB13	0011	1 or 3
CB12	0010	1 or 2
CB123	0001	1 or 2 or 3
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2

Format (2):



Suggested Assembly Language Syntax	
cba{,a}	*label*
cbn{,a}	*label*
cb3{,a}	*label*
cb2{,a}	*label*
cb23{,a}	*label*
cb1{,a}	*label*
cb13{,a}	*label*
cb12{,a}	*label*
cb123{,a}	*label*
cb0{,a}	*label*
cb03{,a}	*label*
cb02{,a}	*label*
cb023{,a}	*label*
cb01{,a}	*label*
cb013{,a}	*label*
cb012{,a}	*label*

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

To set the "annul" bit for CBccc instructions, append an (optional) ",a" to the opcode. For example, use "cb12,a label". The preceding table indicates that the ",a" is optional by enclosing it in braces ({}).

Description:

A CBccc instruction (except CBA and CBN) evaluates the coprocessor condition codes (supplied by the coprocessor on bp_CP_cc[1:0]) according to the *cond* field. If the condition codes evaluate to true the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign_ext (disp22))." If the condition codes evaluate to false, the branch is not taken and the instruction acts like a "NOP."

If the branch is not taken and the *a* (annul) field is set, the delay instruction is not executed (annulled). If the branch is taken, the annul field is ignored and the delay instruction is executed. (Annulment, delay instructions, and delayed control transfers are described further in the section *Instructions*.)

CBN (Branch Never) acts like a "NOP", except that if the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

CBA (Branch Always) causes a transfer of control, irrespective of the value of the condition code bits. If the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

A CBccc instruction generates a cp_disabled trap (and does not branch or annul) if the PSR's EC bit is reset or if no coprocessor is present.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

Except for CBA, all CBccc instructions with a=1 annul the delay instruction when the branch is not taken. However, CBA with a=1 does the reverse: it annuls the delay instruction even though the branch is taken.

A CBccc instruction must be immediately preceded by a non-coprocessor instruction.

B.21.1. Programming Note

An untaken branch takes as much or more time than a taken branch. The additional time it takes is implementation-dependent.

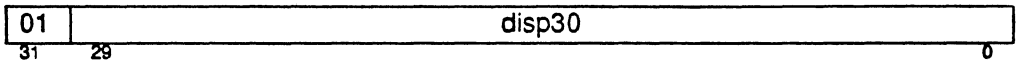
Traps:

cp_disabled
cp_exception

B.22. CALL Instruction

opcode	op	operation
CALL	01	Call

Format (1):



Suggested Assembly Language Syntax	
call	*label*

Description:

The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address "PC + (4 * disp30)". Since the word displacement (*disp30*) field is 30 bits wide, the target address can be arbitrarily distant. The CALL instruction also writes the value of PC, which contains the address of the CALL, into *out* register r[15].

The PC-relative displacement is formed by appending two low-order zeros to the instruction's 30-bit word displacement field.

B.22.1. Programming Note

A JMPL instruction with *rd* = 15 can be used as a register-indirect CALL.

B.22.2. Programming Note

The execution time of a CALL instruction may increase if the next instruction uses r[15] as a source operand. Whether this happens is implementation-dependent.

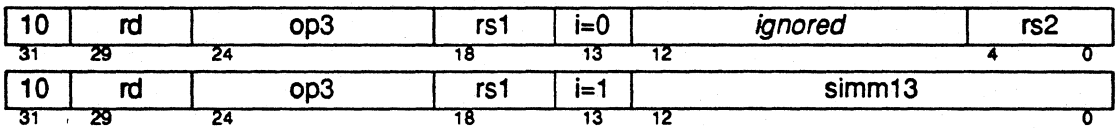
Traps:

(none)

B.23. Jump and Link Instruction

opcode	op3	operation
JMPL	111000	Jump and Link

Format (3):



Suggested Assembly Language Syntax
jmpl *address*, *reg*$_{rd}$

Description:

The JMPL instruction causes a register-indirect control transfer to an address specified by either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

The JMPL instruction writes the PC, which contains the address of the JMPL instruction, into the destination *r register* specified in the *rd* field.

If either of the low-order two bits of the jump address is nonzero, a mem_address_not_aligned trap occurs.

B.23.1. Programming Note

JMPL with *rd* = 0 can be used to return from a subroutine. The typical return address is "r[31]+8", if the subroutine was entered by a CALL instruction.

B.23.2. Programming Note

JMPL with *rd* = 15 can be used as a register-indirect CALL.

B.23.3. Programming Note

The execution time of a JMPL instruction may increase if the next instruction uses r[rd] as a source operand. Whether this happens is implementation-dependent.

Traps:

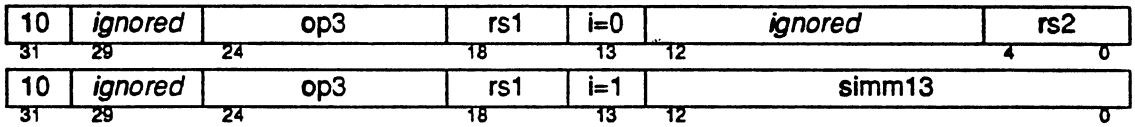
mem_address_not_aligned

B.24. Return from Trap Instruction

opcode	op3	operation
RETT†	111001	Return from Trap

† privileged instruction

Format (3):



Suggested Assembly Language Syntax	
rett	*address*

Description:

The RETT instruction adds one to the CWP (modulo the number of implemented windows) and compares this value, the "new_CWP," against the Window Invalid Mask (WIM) register. If the WIM bit indexed by the new_CWP is set, "(WIM and 2^{new_CWP}) = 1," then a window_underflow trap is generated. If the WIM bit indexed by the new_CWP is reset, then a window_underflow trap is not generated and new_CWP is written into CWP. This causes the *previous* window to become the *active* window, thereby restoring the window that existed at the time of the trap.

If a window_underflow trap is not generated, RETT causes a delayed control transfer to the target address. The target address is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. Furthermore, RETT restores the S field of the PSR from the PS field, and sets the ET field to one.

If traps are enabled (ET=1), an illegal_instruction trap occurs. If traps are disabled (ET=0) and the processor is not in supervisor mode (S=0), or if a window_underflow condition is detected, or if either of the low-order two bits of the target address is nonzero, a reset trap occurs. If a reset trap occurs, the *tt* field of the TBR encodes the trap condition: privileged_instruction, window_underflow, or mem_address_not_aligned.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

The instruction executed immediately before a RETT must be a JMPL instruction. (See discussion in the section "Instructions".)

B.24.1. Programming Note

To re-execute the trapped instruction when returning from a trap handler use the sequence:

```
jmpl%17, %0      ! old PC
rett%18          ! old nPC
```

To return to the instruction after the trapped instruction (e.g. when emulating an instruction) use the sequence:

```
jmpl%18, %0      ! old nPC
rett%18 + 4      ! old nPC + 4
```

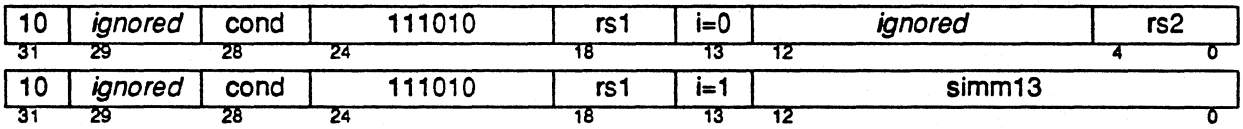
Traps:

illegal_instruction
reset (privileged_instruction)
reset (mem_address_not_aligned)
reset (window_underflow)

B.25. Trap on Integer Condition Instruction

opcode	cond	operation	Icc test
TA	1000	Trap Always	1
TN	0000	Trap Never	0
TNE	1001	Trap on Not Equal	not Z
TE	0001	Trap on Equal	Z
TG	1010	Trap on Greater	not (Z or (N xor V))
TLE	0010	Trap on Less or Equal	Z or (N xor V)
TGE	1011	Trap on Greater or Equal	not (N xor V)
TL	0011	Trap on Less	N xor V
TGU	1100	Trap on Greater Unsigned	not (C or Z)
TLEU	0100	Trap on Less or Equal Unsigned	(C or Z)
TCC	1101	Trap on Carry Clear (Greater than or Equal, Unsigned))	not C
TCS	0101	Trap on Carry Set (Less Than, Unsigned)	C
TPOS	1110	Trap on Positive	not N
TNEG	0110	Trap on Negative	N
TVC	1111	Trap on Overflow Clear	not V
TVS	0111	Trap on Overflow Set	V

Format (3):



Suggested Assembly Language Syntax		
`ta`	`address`	
`tn`	`address`	
`tne`	`address`	*synonym: tnz*
te	*address*	*synonym: tz*
tg	*address*	
tle	*address*	
tge	*address*	
tl	*address*	
tgu	*address*	
tleu	*address*	
tcc	*address*	*synonym: tgeu*
tcs	*address*	*synonym: tlu*
tpos	*address*	
tneg	*address*	
tvc	*address*	
tvs	*address*	

Description:

A Ticc instruction evaluates the integer condition codes (*icc*) according to the *cond* field. If the condition codes evaluate to true and there are no higher priority traps pending, then a trap_instruction trap is generated. If the condition codes evaluate to false, a trap_instruction trap does not occur.

If a trap_instruction trap is generated, the *tt* field of the Trap Base Register (TBR) is written with 128 plus the least significant seven bits of either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

See the section *Traps, Exceptions and Error Handling* for the complete definition of a trap.

Traps:

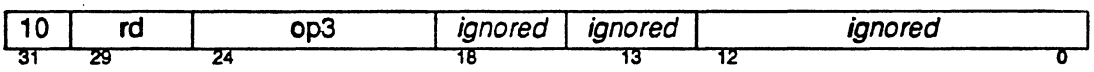
trap_instruction

B.26. Read State Register Instructions

opcode	op3	operation
RDY	101000	Read Y register
RDPSR†	101001	Read Processor State Register
RDWIM†	101010	Read Window Invalid Mask register
RDTBR†	101011	Read Trap Base Register

† privileged instruction

Format (3):



Suggested Assembly Language Syntax	
rd	*%y*, reg_{rd}
rd	*%psr*, reg_{rd}
rd	*%wim*, reg_{rd}
rd	*%tbr*, reg_{rd}

Description:

These instructions read the specified IU state registers into the *r* register specified in the *rd* field.

B.26.1. Programming Note

The execution time of any of these instructions may increase if the next instruction uses the register specified by the *rd* field of this instruction as a source operand. Whether it does or not is implementation-dependent.

Traps:

privileged_instruction (RDPSR, RDWIM and RDTBR only)

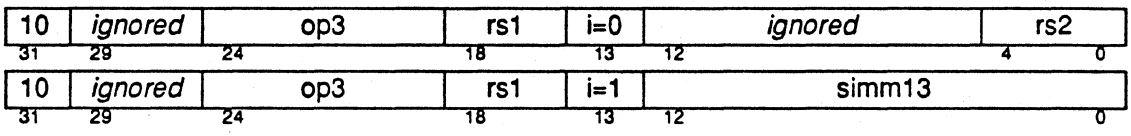
B.27. Write State Register Instructions

opcode	op3	operation
WRY	110000	Write Y register
WRPSR†	110001	Write Processor State Register
WRWIM†	110010	Write Window Invalid Mask register
WRTBR†	110011	Write Trap Base Register

† privileged instruction

Suggested Assembly Language Syntax	
wr	*reg*$_{rs1}$, *reg_or_imm*, *%y*
wr	*reg*$_{rs1}$, *reg_or_imm*, *%psr*
wr	*reg*$_{rs1}$, *reg_or_imm*, *%wim*
wr	*reg*$_{rs1}$, *reg_or_imm*, *%tbr*

Format (3):



Description:

These instructions write either "r[rs1] *xor* r[rs2]" if the *i* field is zero, or "r[rs1] *xor* sign_ext(simm13)" if the *i* field is one, to the writeable subfields of the specified IU state register.

WRPSR does not write the PSR and causes an illegal_instruction trap if the result would cause the CWP field of the PSR to point to an unimplemented window.

These instructions are *delayed-write* instructions:

1. If any of the three instructions after a WRPSR uses any field of the PSR that is changed by the WRPSR, the value of that field is unpredictable. (Note that any instruction which references a non-global register implicitly uses the CWP.)
2. If a WRPSR instruction is updating the PSR's PIL to a new value and is simultaneously setting ET to 1, this can result in an interrupt trap at a level equal to the old value of the PIL.

B.27.1. Programming Note

Two WRPSR instructions should be used when enabling traps and changing the value of the PIL. The first WRPSR should specify ET=0 with the new PIL value, and the second WRPSR should specify ET=1 and the new PIL value.

3. If any of the three instructions after a WRWIM is a SAVE, RESTORE or RETT, the occurrence of window_overflow and window_underflow traps is unpredictable.
4. If any of the three instructions that follow a WRY is a MULScc or RDY, the value of Y used is unpredictable.
5. If any of the three instructions that follow a WRTBR causes a trap, the trap base address (TBA) used may be either the old or the new value.

6. If any of the three instructions after a write state register instruction reads the modified state register, the value read is unpredictable.
7. If any of the three instructions after a write state register instruction is trapped, a subsequent read state register instruction in the trap handler will get the register's new value.

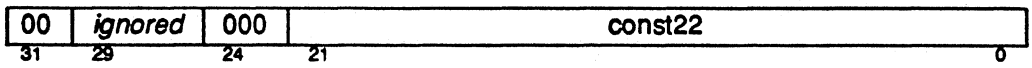
Traps:

privileged_instruction (WRPSR, WRWIM and WRTBR only)
illegal_instruction (WRPSR only)

B.28. Unimplemented Instruction

opcode	op	op2	operation
UNIMP	00	000	Unimplemented

Format (2):



Suggested Assembly Language Syntax	
unimp	*const22*

Description:

The UNIMP instruction causes an illegal_instruction trap. The const22 value is ignored.

B.28.1. Programming Note

This instruction can be used as part of the protocol for calling a function that is expected to return an aggregate value, such as a C-language structure. See *Appendix D* for an example.

a) An UNIMP instruction is placed after (not in) the delay slot after the CALL instruction in the calling function.

b) If the callee function is expecting to return a structure, it will find the size of the structure that the caller expects to be returned as the *const22* operand of the UNIMP instruction. The callee can check the opcode to make sure it is indeed UNIMP.

c) If the function is not going to return a structure, upon returning it attempts to execute the UNIMP instruction rather than skipping over it as it should. This causes the program to terminate. This behavior adds some run-time type checking to an interface that cannot be checked properly at compile time.

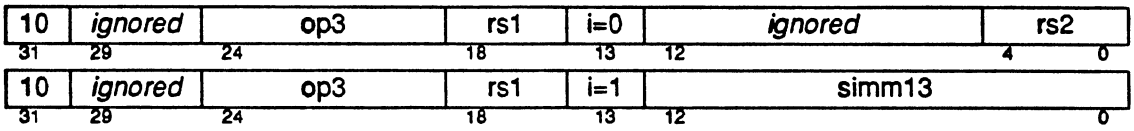
Traps:

illegal_instruction

B.29. Instruction Cache Flush Instruction

opcode	op3	operation
IFLUSH	111011	Instruction cache Flush

Format (3):



Suggested Assembly Language Syntax	
iflush	*address*

Description:

The IFLUSH instruction causes a word to be flushed from an instruction cache that may be internal to the processor. The address of the word to be flushed is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

B.29.1. Implementation Note:

If there is no instruction cache internal to the processor, IFLUSH acts as a "NOP." If there is an internal instruction cache, IFLUSH flushes the addressed word from the cache. If there is an external instruction cache, IFLUSH causes an illegal_instruction trap. The presence of an external instruction cache is determined by the *bp_I_cache_present* signal.

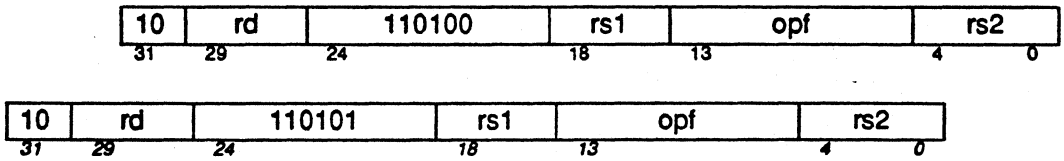
Traps:

illegal_instruction

B.30. Floating-point Operate (FPop) Instructions

opcode	op3	operation
FPop1	110100	Floating-point operate
FPop2	110101	Floating-point operate

Format (3):



The Floating-point Operate (FPop) instructions are encoded using two type 3 instruction formats called FPop1 and FPop2. The floating-point operations themselves are encoded by the *opf* field. (Note that the load/store floating-point instructions are not "FPop" instructions.)

All FPop instructions take all operands from and return all results to *f registers* and/or the FSR. They perform operations on ANSI/IEEE 754-1985 single, double, and extended formats (see the section *SPARC Architecture Overview*).

All multiple-precision floating-point instructions (including load/store floating-point) assume that operands are located in register *pairs* (for double precision) or *quadruples* (for extended precision). The following table indicates the alignment assumptions. Note that single-precision operands can be in any *f register*.

operand	f register address
double-e	0 mod 2
double-f	1 mod 2
extended-e	0 mod 4
extended-f	1 mod 4
extended-f-low	2 mod 4
extended-u	3 mod 4

According to this convention, the least significant bit of an *f register* address is ignored by double-precision FPops and the least significant two bits of an *f register* address are ignored by extended-precision FPops.

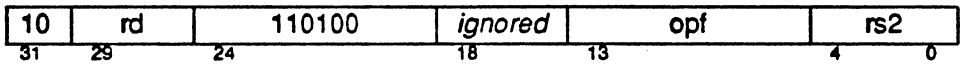
A program including floating-point computations generates the same results as if all instructions were executed sequentially (assuming it runs to completion). Note that floating-point loads and stores are not floating-point operate instructions.

Results are written (or traps are caused) in the order that FPops are encountered in the instruction stream. The section *Instructions* explains this in more detail. An FPop instruction causes an fp_disabled trap if the EF field of the PSR is 0 or if no FPU is present.

B.30.1. Convert Integer to Floating-point Instructions

opcode	opf	operation
FiTOs	011000100	Convert Integer to Single
FiTOd	011001000	Convert Integer to Double
FiTOx	011001100	Convert Integer to Extended

Format (3):



Suggested Assembly Language Syntax	
fitos	$freg_{rs2}$, $freg_{rd}$
fitod	$freg_{rs2}$, $freg_{rd}$
fitox	$freg_{rs2}$, $freg_{rd}$

Description:

These instructions convert the 32-bit integer argument in the *f register* specified by rs2 into a floating-point number in the destination format according to the ANSI/IEEE 754-1985 specification. They place the result in the destination *f register(s)* specified by rd.

For FiTOs and FiTOx with single-precision rounding, rounding is performed according to the rounding direction (RD) field of the FSR.

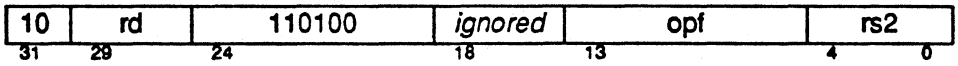
Traps:

fp_disabled
fp_exception (NX) (FiTOs and FiTOx when RP=single)

B.30.2. Convert Floating-point to Integer

opcode	opf	operation
FsTOi	011010001	Convert Single to Integer
FdTOi	011010010	Convert Double to Integer
FxTOi	011010011	Convert Extended to Integer

Format (3):



Suggested Assembly Language Syntax	
fstoi	$freg_{rs2}$, $freg_{rd}$
fdtoi	$freg_{rs2}$, $freg_{rd}$
fxtoi	$freg_{rs2}$, $freg_{rd}$

Description:

These instructions convert the floating-point source argument in the *f register* or *f registers* specified by rs2 to a 32-bit integer (in the *f register* specified by the *rd* field) according to the ANSI/IEEE 754-1985 specification.

The floating-point argument is rounded toward zero and the *rd* field of the FSR is ignored.

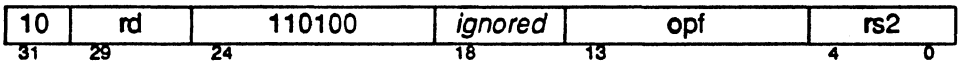
Traps:

fp_disabled
fp_exception (NV, NX)

B.30.3. Convert Between Floating-point Formats Instructions

opcode	opf	operation
FsTOd	011001001	Convert Single to Double
FsTOx	011001101	Convert Single to Extended
FdTOs	011000110	Convert Double to Single
FdTOx	011001110	Convert Double to Extended
FxTOs	011000111	Convert Extended to Single
FxTOd	011001011	Convert Extended to Double

Format (3):



Suggested Assembly Language Syntax	
fstod	$freg_{rs2}$, $freg_{rd}$
fstox	$freg_{rs2}$, $freg_{rd}$
fdtox	$freg_{rs2}$, $freg_{rd}$
fdtox	$freg_{rs2}$, $freg_{rd}$
fxtod	$freg_{rs2}$, $freg_{rd}$
fxtos	$freg_{rs2}$, $freg_{rd}$

Description:

These instructions convert the floating-point source argument in the *f register* or *f registers* specified by rs2 to a floating-point number in the destination format according to the ANSI/IEEE 754-1985 specification. They place the result in the *f register* or *f registers* specified by rd.

Rounding is performed according to the rounding direction (RD) field of the FSR. In the case of FdTOx, the outcome is also a function of the rounding precision (RP) field.

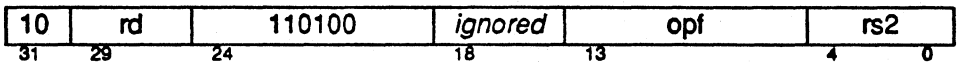
Traps:

fp_disabled
fp_exception (OF, UF, NV, NX)

B.30.4. Floating-point Move Instructions

opcode	opf	operation
FMOVs	000000001	Move
FNEGs	000000101	Negate
FABSs	000001001	Absolute Value

Format (3):



Suggested Assembly Language Syntax	
fmovs	$freg_{rs2}$, $freg_{rd}$
fnegs	$freg_{rs2}$, $freg_{rd}$
fabss	$freg_{rs2}$, $freg_{rd}$

Description:

FMOVs moves a word from f[rs2] to f[rd]. Multiple FMOVs's are required to transfer a multiple-precision number between *f registers*.

FNEGs complements the sign bit, and FABs clears it.

These instructions do not round.

B.30.5. Programming Note

FNEGs or FABSs instructions can also operate on the high-order words (the word that contains the sign bit) of *double* and *extended* operands. Thus an FNEGs instruction and an FMOVs instruction would be used to negate a *double* and put the results in a different pair of *f registers*.

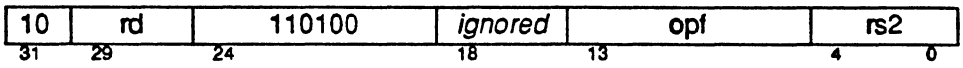
Traps:

fp_disabled

B.31. Floating-point Square Root Instructions

opcode	opf	operation
FSQRTs	000101001	Square Root Single
FSQRTd	000101010	Square Root Double
FSQRTx	000101011	Square Root Extended

Format (3):



Suggested Assembly Language Syntax	
fsqrts	$freg_{rs2}$, $freg_{rd}$
fsqrtd	$freg_{rs2}$, $freg_{rd}$
fsqrtx	$freg_{rs2}$, $freg_{rd}$

Description:

These instructions generate the square root of the floating-point source argument in the *f register* or *f registers* specified by rs2 according to the ANSI/IEEE 754-1985 specification. They place the result in the destination *f register* or *f registers* specified by the *rd* field.

Rounding is performed according to the rounding direction (RD) field of the FSR. In the case of FSQRTx, the outcome is also a function of the rounding precision (RP) field.

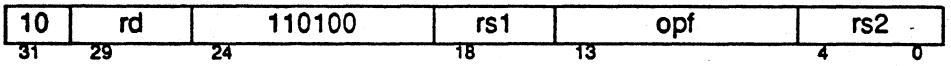
Traps:

fp_disabled
fp_exception (NV, NX)

B.31.1. Floating-point Add and Subtract Instructions

opcode	opf	operation
FADDs	001000001	Add Single
FADDd	001000010	Add Double
FADDx	001000011	Add Extended
FSUBs	001000101	Subtract Single
FSUBd	001000110	Subtract Double
FSUBx	001000111	Subtract Extended

Format (3):



Suggested Assembly Language Syntax			
fadds	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
faddd	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
faddx	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
fsubs	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
fsubd	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
fsubx	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$

Description:

These instructions add or subtract their operands according to the ANSI/IEEE 754-1985 specification, and place the result in the *f register* or *f registers* specified in the *rd* field. The subtract instructions subtract the floating-point value specified by *rs2* from the one specified by *rs1*.

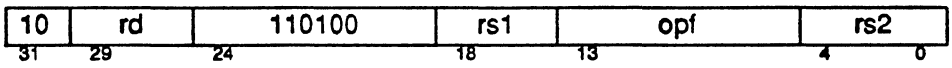
Traps:

fp_disabled
fp_exception (OF, UF, NX)

B.31.2. Floating-point Multiply and Divide Instructions

opcode	opf	operation
FMULs	001001001	Multiply Single
FMULd	001001010	Multiply Double
FMULx	001001011	Multiply Extended
FDIVs	001001101	Divide Single
FDIVd	001001110	Divide Double
FDIVx	001001111	Divide Extended

Format (3):



Suggested Assembly Language Syntax			
fmuls	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
fmuld	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
fmulx	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
fdivs	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
fdivd	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$
fdivx	$freg_{rs1}$,	$freg_{rs2}$,	$freg_{rd}$

Description:

These instructions multiply or divide their operands according to the ANSI/IEEE 754-1985 specification, and place the result in the *f register* or *f registers* specified in the *rd* field. The divide instructions divide the floating-point value specified by *rs1* by the one specified by *rs2*.

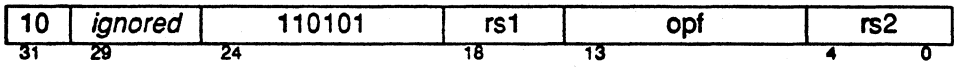
Traps:

fp_disabled
fp_exception (OF, UF, DZ (FDIV only), NV, NX)

B.31.3. Floating-point Compare Instructions

opcode	opf	operation
FCMPs	001010001	Compare Single
FCMPd	001010010	Compare Double
FCMPx	001010011	Compare Extended
FCMPEs	001010101	Compare Single and Exception if Unordered
FCMPEd	001010110	Compare Double and Exception if Unordered
FCMPEx	001010111	Compare Extended and Exception if Unordered

Format (3):



Suggested Assembly Language Syntax	
fcmps	*freg*$_{rs1}$, *freg*$_{rs2}$
fcmpd	*freg*$_{rs1}$, *freg*$_{rs2}$
fcmpx	*freg*$_{rs1}$, *freg*$_{rs2}$
fcmpes	*freg*$_{rs1}$, *freg*$_{rs2}$
fcmped	*freg*$_{rs1}$, *freg*$_{rs2}$
fcmpex	*freg*$_{rs1}$, *freg*$_{rs2}$

Description:

These instructions compare their operands according to the ANSI/IEEE 754-1985 specification. The floating-point condition codes in the FSR are set as follows:

NOTE:
This table is a duplicate of Table 3-5 in the section "Registers".

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

In this table, *fs1* refers to the value specified by the *rs1* field and *fs2* refers to the value specified by the *rs2* field of the compare instruction.

The "Compare and Cause Exception if Unordered" instructions (FCMPE) cause an invalid exception (NV) if either of the operands is a signaling or quiet NaN. FCMP also causes an invalid exception if either operand is a signaling NaN.

☆ ☆ ☆ **NOTE** ☆ ☆ ☆

A non-floating point instruction must be executed between an FCMP and a subsequent FBfcc.

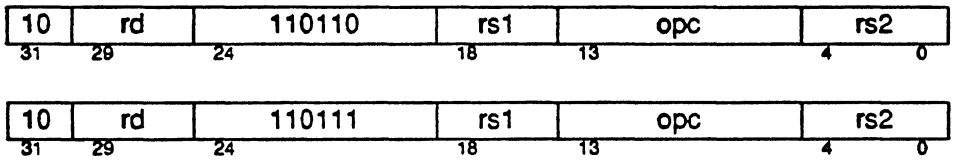
Traps:

fp_disabled
fp_exception (NV)

B.32. Coprocessor Operate Instructions

opcode	op3	operation
CPop1	110110	Coprocessor Operate
CPop2	110111	Coprocessor Operate

Format (3):



☆ ☆ ☆ **NOTE** ☆ ☆ ☆

The assembly language syntax for these instructions is unspecified.

The Coprocessor Operate (CPop) instructions are encoded via two type 3 instruction formats called CPop1 and CPop2. The coprocessor operations themselves are encoded by the *opc* field and are coprocessor-dependent. (Note that the load/store coprocessor instructions are not "CPop" instructions.)

All CPop instructions take all operands from and return all results to coprocessor registers. The data types supported by the coprocessor are coprocessor-dependent. Operand alignment is coprocessor-dependent.

A CPop instruction causes a cp_disabled trap if the EC field of the PSR is 0 or if no coprocessor is present.

Whether a CPop generates a cp_exception trap is coprocessor-dependent.

APPENDIX C: ISP DESCRIPTIONS

C.1. Introduction

This appendix provides a description of the SPARC architecture using the Instruction-Set Processor (ISP) description language. It includes register definitions, instruction fields, processor states, instruction dispatch, traps, and instruction descriptions.

The instruction interpreter defines the ordering of events. Except for a few cases (which are documented), the interpreter together with the instruction and register definitions provide a supplemental description of the processor.

Note that the use of a particular variable in the notation does not necessarily imply that its related signal is present in an implementation, or visible to the programmer.

The instruction description language is a modified version of Bell and Newell's ISP instruction description language, which was created to accurately describe computer instruction sets. While the semantics are somewhat intuitive, the following guidelines provide important details:

- The only data type is the *bit vector.* Variables are defined as bit vectors of particular widths, declared as **variable<n:m>**. Variable subfields can be defined, also with the **<n:m>** notation. The value of a vector is a number in a base indicated by its subscript. The default base is decimal. Arrays of vectors are declared as **array[n:m]**.
- The notation ← indicates variable assignment, and **:=** indicates a macro definition.
- When a bit vector is assigned to another of greater length, the operand is right-justified in the destination vector and the high-order positions are zero-filled. The macro **zero_extend** is sometimes used to make this clear. Conversely, the macro **sign_extend** causes the high-order positions of the result to be filled with the highest-order (sign) bit of its operand.
- The semicolon ';' separates statements. Parentheses '()' group statements and expressions that could otherwise be interpreted ambiguously.

- All statements are generally executed "simultaneously." However, if the term **next** appears, it indicates that the statement or statements which follow the **next** are executed after those that appear before the **next**. Thus, all statements between **next** phrases are executed concurrently. More precisely, this means that all expressions on the right hand sides of assignments located between **next**'s are evaluated first, after which the variables on the left hand sides are updated. (This convention emulates synchronous, clocked hardware.)

  For example, if A=0 and B=0, execution of the following two statements,

  ```
  A ← B+1;
  B ← A+1;
  ```

  results in A=1 and B=1. However,

  ```
  A ← B+1;
  next;
  B ← A+1;
  ```

  results in A=1 and B=2.

- The symbol [] designates concatenation of vectors. A comma ',' on the left side of an assignment separates quantities that are concatenated for the purpose of assignment. For example, if the 2-bit vector T2 equals 3, and X, Y, and Z are 1-bit vectors, then:

  ```
  X, Y, Z  ←  0[]T2
  ```

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

- The operators '+' and '–' perform two's complement arithmetic.

- The phrase **fork**, used only in the instruction interpreter for the FPop instructions, indicates that the associated routine may be executed concurrently with *all* other subsequent statements. There is no notation for rejoining: after the forked routine executes its last statement, it terminates.

- The major difference between the notation used here and the 1971 version of ISP is that the notation here uses the more common:

  ```
  if cond then S1 else S2
  ```

  whereas Bell and Newell used the following:

  ```
  (cond → S1, ¬ cond → S2)
  ```

- The macros memory_read and memory_write, are implementation-dependent. These routines define the interface without referring to implementation-specific signals:

  ```
  load_data ← memory_read(addr_space, address)

  memory_write(addr_space, address, byte_mask,
       store_data)
  ```

  Memory_read returns the word in memory specified by both the address and the address space identifier.

  Memory_write writes all or part of the word store_data into the word specified by the given address. If there is an exception, memory_write does not change the state of the external system or the MMU. Byte_mask is a 4-bit value that indicates which of the four bytes in store_data are to be written into the addressed word.

C.2. Register Definitions

```
PSR<31:0>;                          {Processor State Register}
    impl                    := PSR<31:28>;
    ver                     := PSR<27:24>;
    icc                     := PSR<23:20>;
      N                       := PSR<23>;
      Z                       := PSR<22>;
      V                       := PSR<21>;
      C                       := PSR<20>;
    reserved                := PSR<19:14>;
    EC                      := PSR<13>;
    EF                      := PSR<12>;
    PIL                     := PSR<11:8>;
    S                       := PSR<7>;
    PS                      := PSR<6>;
    ET                      := PSR<5>;
    CWP                     := PSR<4:0>;

TBR<31:0>;                          {Trap Base Register}
    TBA                     := TBR<31:12>;
    tt                      := TBR<11:4>;
    zero                    := TBR<3:0>;

FSR<31:0>;                          {Floating-Point State Register}
    RD                      := FSR<31:30>;
    RP                      := FSR<29:28>;
    TEM                     := FSR<27:23>;
      NVM                     := FSR<27>;
      OFM                     := FSR<26>;
      UFM                     := FSR<25>;
      DZM                     := FSR<24>;
      NXM                     := FSR<23>;
      AU                      := FSR<22>;

    reserved                := FSR<21:17>;
    ftt                     := FSR<16:14>;
    qne                     := FSR<13>;
    reserved                := FSR<12>;
    fcc                     := FSR<11:10>;
    aexc                    := FSR<9:5>;
      nva   := FSR<9>;
      ofa   := FSR<8>;
      ufa   := FSR<7>;
      dza   := FSR<6>;
      nxa   := FSR<5>;
    cexc                    := FSR<4:0>;
      nvc   := FSR<4>;
      ofc   := FSR<3>;
      ufc   := FSR<2>;
      dzc   := FSR<1>;
      nxc   := FSR<0>;

CSR<31:0>;                          {CP State Register}
WIM<31:0>;                          {Window Invalid Mask Register}
```

```
Y<31:0>;                                    {Y Register}
PC<31:0>;                                   {Program Counter}
nPC<31:0>;                                  {Next Program Counter}
FQ<63:0>;                                   {Floating-Point Queue}
CQ<63:0>;                                   {Coprocessor Queue}
G[1:7]<31:0>;                               {Global Registers}
R[0:(16*NWINDOWS)-1]<31:0>;                 {Windowed Registers}
f[0:31]<31:0>;                              {Floating-Point Registers}
```

```
r[n] := if  (n = 0)
     then  0
     else if  (1 ≤ n ≤ 7)
          then  G[n]                          {globals}
          else  R[(n-8) + (CWP*16)] ;  {windowed registers}
```

C.3. System Interface Definitions

```
bp_IRL<3:0>;
bp_reset_in;
pb_error;
pb_retain_bus;
bp_FPU_present;
bp_CP_present;
bp_I_cache_present;
bp_CP_exception;
bp_CP_cc <1:0>;
bp_memory_exception;
```

C.4. Instruction Fields

The numbers in braces are the widths of the fields in bits.

```
instruction<31:0> ;
     op       {2} := instruction<31:30>;
     op2      {3} := instruction<24:22>;
     op3      {6} := instruction<24:19>;
     opf      {9} := instruction<13:5>;
     opc      {9} := instruction<13:5>;
     asi      {8} := instruction<12:5>;
     i        {1} := instruction<13>;
     rd       {5} := instruction<29:25>;
     a        {1} := instruction<29>;
     cond     {4} := instruction<28:25>;
     rs1      {5} := instruction<18:14>;
     rs2      {5} := instruction<4:0>;
     simm13  {13} := instruction<12:0>;
     shcnt    {5} := instruction<4:0>;
     disp30  {30} := instruction<29:0>;
     disp22  {22} := instruction<21:0>;
```

C.5. Processor States and Instruction Fetch

The IU can be in one of three states: execute_mode, reset_mode, or error_mode.

The FPU can be in one of five states: reset_mode, error_mode, fpu_execute_mode, fpu_exception_pending_mode, or fpu_exception_mode. The FPU's reset_mode and error_mode correspond to the IU's reset and error modes. The remaining FPU states are described in Section C.6.

The processor (that is the IU and FPU) is in reset_mode when bp_reset_in is asserted. The processor remains in reset_mode until bp_reset_in is deasserted, at which point the IU enters execute_mode and the FPU enters fpu_execute_mode.

When bp_reset_in is deasserted, the first instruction address is 0, with ASI=9 (supervisor instruction).

The processor enters error_mode from any state except reset_mode if a synchronous trap is generated while traps are disabled. (See the section *Traps, Exceptions, and Error Handling*).
5.) The processor remains in error_mode until bp_reset_in is asserted, at which time it enters reset_mode.

C.5.1. Implementation Note

The external system should assert *bp_reset_in* whenever *pb_error* is detected.

The following ISP code defines the three IU states. In execute_mode, the IU fetches and dispatches instructions.

```
while (reset_mode) (
    if (bp_reset_in = 0) then (
          reset_mode ← 0;
          execute_mode ← 1;
          trap ← 1;
          reset ← 1
    )
);

addr_space := S=0 then 8 else 9;

while (execute_mode) (
        check_interrupts;              { see Section C.8}
        next;

    { the following code emulates the delayed nature of the
      write state register instructions.}

    PSR ← PSR'; PSR' ← PSR''; PSR'' ← PSR'''; PSR''' ← PSR'''';
    TBR ← TBR'; TBR' ← TBR''; TBR'' ← TBR'''; TBR''' ← TBR'''';
    WIM ← WIM'; WIM' ← WIM''; WIM'' ← WIM'''; WIM''' ← WIM'''';
    Y ← Y'; Y' ← Y''; Y'' ← Y'''; Y''' ← Y'''';
    next;

        if (trap = 1) then
                execute_trap;          { see Section C.8}
        next;

        instruction ← memory_read(addr_space, PC);
        next;

        if (bp_memory_exception = 1) then (
                trap ← 1;
                instruction_access_exception ← 1
        ) else (
                if (annul = 0) then (
                        dispatch_instruction  { see Section C.5 }
                ) else (
                        annul ← 0;
                        PC ← nPC;
                        nPC ← nPC + 4
                )
        )
);


while (error_mode)  (
        if (bp_reset_in = 1) then
           error mode ← 0
           reset mode ← 1
           pb_error ← 0

        )
```

```
);
```

C.6. Instruction Dispatch

The "dispatch_instruction" macro determines if the fetched instruction is an FPop or CPop. If it is an FPop, it is executed by the "execute_FPU_instruction" macro (Section C.6) as soon as the FPU can accept another instruction. If the fetched instruction is a CPop, it is executed by the "execute_CP_instruction" macro (Section C.7) as soon as the CP can accept another instruction.

If the instruction is neither an FPop or a CPop, it is executed by the "execute_IU_instruction" macro, which includes all the macro definitions in Section C.9 (except for FPop and CPop).

Unused bit patterns in the *op*, *op2*, *op3*, *opf*, and *i* fields of instructions cause illegal_instruction traps. Other fields that are defined to be *unused* are ignored and do not cause traps.

The macro 'floating-point_instr' returns a 1 if the instruction is a floating-point instruction. Similarly, the macro 'coprocessor_instr' returns a 1 if the instruction is a coprocessor instruction.

```
unimplemented_IU_instr := (
     if ( ( (op=00_2) and (op2=000_2) )     [UNIMP instruction}
               or
          ( ((op=11_2) or (op=10_2)) and (op3=unassigned) )
               or
          ( (i = 1) and
            (LDSBA or LDSHA  or LDUBA  or LDUHA  or LDA or
          LDDA  or STDA   or LDSTUBA or SWAPA
             STBA  or STHA   or STA
             )
        ) then 1 else 0
);

floating_point_instr := (
     if (LDF   or LDDF  or LDFSR or
         STF   or STDF  or STFSR or STDFQ or
         FPop1 or FPop2 or FBfcc) then 1 else 0
);

coprocessor_instr: = (
     if (LDC   or LDDC  or LDCSR or
         STC   or STDC  or STCSR or STDCQ or CPop1 or CPop2 or CBccc) then 1 else 0
);

dispatch_instruction := (
     if (unimpl_IU_instr = 1) then (
          trap ← 1;
          illegal_instruction ← 1
     );
     if (floating-point_instr = 1) then (
          if (EF = 0) then (
               trap ← 1;
               fp_disabled ← 1
          ) else (
               if ( fpu_exception_pending_mode = 1 ) then (
                    fpu_exception_pending_mode ← 0;
                    fpu_exception_mode ← 1;
                    trap ← 1
               );
               while ( (fp_not_ready = 1) and (trap = 0) )
                    check_interrupts;
          )
     );
     if (coprocessor_instr = 1) then (
          if (EC = 0) then (
               trap ← 1;
               cp_disabled ← 1
          ) else (
               check_CP_exception;
               next;
               while ( (cp_not_ready = 1) and (trap = 0) ) (
                    check_interrupts;
          )
     );
```

```
     next;
     if (trap = 0) then
          if (FPop1 or FPop2) then  fork execute_FPU_instruction
          else if (CPop1 or CPop2) then  fork execute_CP_instruction
          else    execute_IU_instruction
     );
)

execute_IU_instruction := (
     { do routine for specific instruction, defined below }
     next;
     if (trap = 0 and
         not (CALL or RETT or JMPL or Bicc or FBfcc or CBccc or Ticc) ) then (
          PC ← nPC;
          nPC← nPC + 4
     )
);

execute_FPU_instruction := (
     if (FPU_exception_mode)_ then (
          ftt <- sequence_error;
          FPU_exception_mode <- 0;              {see following discussion}
          FPU_exception_pending_mode <- 1
     ) else (
          enqueue_FQ(instruction, PC)
          { execute description defined below }
     )
);
```

C.7. Floating-Point Instruction Execution

The FPU can execute floating-point operate (FPop) instructions concurrently with other FPops and with non-floating-point instructions. To do this, it maintains a Floating-point Queue (FQ) of FPop instructions pending completion, and can force the IU to wait until resource and data dependencies have been resolved.

The architecture ensures that a program containing FPops generates the same numerical results as if there were no concurrency.

After the FPU begins to execute an FPop, the IU continues to fetch and execute instructions until one of five "hold" conditions occurs. Any one of these causes the IU to stop fetching instructions until the condition is no longer true:

1) If, for a load floating-point register instruction, the destination *f register* is the source or destination register of an executing FPop, the IU waits until the executing FPop no longer requires the register.

2) If, for a store floating-point register instruction, the source *f register* is the destination register of an executing FPop, the IU waits until executing FPop no longer require the register.

3) A load or store floating-point state register instruction (LDFSR, STFSR) causes the IU to wait until all executing and pending FPops have completed.

4) A branch on floating-point condition (FBfcc) instruction causes the IU to wait until any executing or pending floating-point compare instructions (FCMP, FCMPE) have finished.

5) When the IU encounters an FPop, it stops fetching instructions until the FPop has been accepted by the FPU.

C.7.1. Floating-Point Queue (FQ)

The floating-point queue (FQ) has at least one entry for each of the FPU's arithmetic units that can execute in parallel with other arithmetic units. The depth of the queue is implementation-dependent.

Each entry in the queue (for the purposes of the definition in this appendix) contains 1) the FPop instruction itself, 2) the PC from which the FPop was fetched, 3) an indication of the arithmetic unit executing it, 4) a completion status bit that indicates whether the operation finished properly, and 5) a temporary result, including any exceptions or condition codes generated by the instruction. Parts (1) and (2) of the front entry are visible to the programmer using the STDFQ instruction; the other parts and the other entries are invisible to the programmer.

(Note that load floating-point, store floating-point, and FBfcc instructions are never entered in the queue.)

For the purposes of the definition in this appendix, when an arithmetic unit finishes, it deposits its computed result, any exceptions or conditions it may have generated, and a completion status bit, into the reserved location in the queue. As FPops complete, each entry moves toward the front of the queue (if it is not already there).

The FPU can stop executing an FPop in one of four ways: 1) completed without exception (normal), 2) IEEE_exception, 3) unfinished_FPop, or 4) unimplemented_FPop. The following paragraphs describe each:

Normal Completion

If the FPop represented by the front entry in the queue caused no unmasked exceptions, the FPU 1) writes the result into the *f register(s)* specified by the *rd* field of the instruction (if any), 2) updates the FSR's *cexc* and *fcc* fields, 3) removes the entry from the queue, and 4) advances the queue.

IEEE_Exception

If the FPop pointed to by the front entry in the queue caused an IEEE_exception trap, the FPU updates the FSR's *cexc* and *ftt* fields to identify the exception, and does **not** write the result into the *f register(s)* specified by the *rd* field of the instruction, **nor** does it remove the entry from the queue. However, if an IEEE_exception does not result in a fp_exception trap, all results are written, including the destination *f register*, *cexc*, *aexc*, and *fcc*.

Unimplemented_FPop or Unfinished_FPop

If the FPop pointed to by the front entry in the queue is not implemented, or if the arithmetic unit was unable to complete it according to the ANSI/IEEE 754-1985 specification (for example, a multiply unit may not be able to postnormalize a denormalized result or handle a NaN), the FPU updates the *ftt* field of the FSR to identify the exception, and does **not** write the result into the *f register(s)* specified by the *rd* field of the instruction, **nor** does it remove the entry from the queue. The front entry in the queue identifies the FPop that generated the floating-point exception trap.

C.7.2. FQ_Front_Done

The implementation-dependent macro 'FQ_front_done' returns a 1 if an arithmetic unit has finished processing the FPop at the front of the FQ. The implementation-dependent macro 'stop_FPU' stops all current processing of FQ entries.

C.7.3. FPU States

The FPU can be in any of three modes: FPU_execute_mode, FPU_exception_pending_mode, or FPU_exception_mode. In FPU_execute_mode, it executes floating-point instructions.

The FPU enters the FPU_exception_pending_mode state when an FPop instruction causes an IEEE_exception, unfinished_FPop exception, unimplemented_FPop exception, or a sequence_error. The FPU remains in FPU_exception_pending_mode until the IU fetches another floating-point instruction, at which time a fp_exception trap is caused and the FPU enters the FPU_exception_mode state.

In FPU_exception_mode, the FPU executes only store floating point instructions. If an FPop or a load floating point instruction is fetched while the unit is in FPU_exception_mode, the *ftt* field of the FSR will be updated to indicate "sequence_error", and the FPU will enter FPU_exception_pending_mode. The instruction that caused the sequence_error is not entered into the FQ.

The FPU returns to FPU_execute_mode after the FQ has been emptied via STDFQ instructions, that is, *qne* is 0.

```
while (FPU_execute_mode) (
     if (FQ_front_done = 1) then (
          if (fp_unimplemented = 1) then (        {not implemented}
               fp_exception ← 1; ftt ← unimplemented_FPop;
          );
          if (FQ_c = 0) then (                    {not finished}
               fp_exception ← 1; ftt ← unfinished_FPop;
          ) else (                          {executed and finished}
               cexc ← texc;
               next;
               if ( cexc and TEM ≠ 0) then (   {floating-point trap}
                    fp_exception ← 1; ftt ← IEEE_Exception;
               ) else (                          {no floating-point trap}
                    aexc ← aexc or cexc;
                    if (FQ_single_result = 1) then
                         f[rd] ← result;
                    if (FQ_double_result = 1) then
                         f[rdE], f[rdO] ← result;
                    if (FQ_extended_result = 1) then
                         f[rdEE], f[rdEO], f[rdOE] ← result;
                    if (FQ_compare = 1) then
                         fcc ← tfcc;
                    dequeue_FQ;
               )
          )
          next;
          if (fp_exception = 1) then (
               FPU_execute_mode ← 0;
               FPU_exception_pending_mode ← 1
          )
     )
)
```

C.8. Coprocessor Instruction Execution

The CP can execute coprocessor operate (CPop) instructions concurrently with integer instructions and other CPops. Although the instruction set includes a "store CP double queue" instruction, the existence of the queue and the type of concurrency available in the coprocessor is dependent on the coprocessor itself.

The FPU leaves FPU_exception_mode and enters FPU_execute_mode after the FQ has been emptied (via execution of STDFQ instructions.)

```
execute_CP_instruction  :=  (  {not specified}  ) ;
```

C.9. Traps

```
execute_trap := (
     select_trap;
     ET ← 0;                    {ignore asynchronous traps}
     PS ← S;
     annul ← 0;
     CWP ← (CWP - 1) mod NWINDOWS;{point to next window}
     r[17] ← PC;                    {preserve program counters}
     r[18] ← nPC;
     next;
     S ← 1;                     {set supervisor mode}
     if (reset_trap = 0) then (
          PC ← TBR;
          nPC ← TBR + 4
     ) else (
          reset_trap ← 0;
          PC ← 0;
          nPC ← 4
     )
);

select_trap := (
     if (ET = 0 or reset_trap = 1) then
          error_mode ← 1
     else if (instruction_access_exception = 1) then
          tt ← 00000001₂
     else if (illegal_instruction = 1) then
          tt ← 00000010₂
     else if (privileged_instruction = 1) then
          tt ← 00000011₂
     else if (fp_disabled = 1) then
          tt ← 00000100₂
     else if (cp_disabled = 1) then
          tt ← 00100100₂
     else if (window_overflow = 1) then
          tt ← 00000101₂
     else if (window_underflow = 1) then
          tt ← 00000110₂
     else if (mem_address_not_aligned = 1) then
          tt ← 00000111₂
     else if (fp_exception = 1) then
          tt ← 00001000₂;
     else if (cp_exception = 1) then
          tt ← 00101000₂;
     else if (data_access_exception = 1) then
          tt ← 00001001₂
     else if (tag_overflow = 1) then
          tt ← 00001010₂
     else if (trap_instruction = 1) then
          tt ← 1₂□ticc_trap_type
     else if (interrupt_level > 0) then
          tt ← 0001₂□interrupt_level
     next;
     trap ← 0;      {since the tt field has been set, reset the trap signal}
     reset_trap ← 0;
```

```
        instruction_access_exception ← 0;
        illegal_instruction ← 0;
        privileged_instruction ← 0;
        fp_disabled ← 0;
        cp_disabled ← 0;
        window_overflow ← 0;
        window_underflow ← 0;
        mem_address_not_aligned ← 0;
        fp_exception ← 0;
        cp_exception ← 0;
        data_access_exception ← 0;
        tag_overflow ← 0;
        trap_instruction ← 0;
        interrupt_level ← 0
);

check_interrupts := (
        if (bp_reset_in = 1) then (
                reset_mode ← 1
        ) else if (ET = 1 and (bp_IRL = 15 or bp_IRL > PIL)) then (
                trap ← 1;
                interrupt_level ← bp_IRL
        );
);
```

C.10. Instruction Definitions

This section contains the ISP definitions of the SPARC architecture instructions. These complement the instruction descriptions in *Appendix B, Instruction Descriptions.*

C.10.1. Load Instructions

```
if ( (LDF or LDDF or LDFSR) then (
     if (EF = 0 or bp_FPU_present = 0)  then (
          trap ← 1; fp_disabled ← 1
     ) else if (FPU_exception_mode = 1) then (
          ftt ← sequence_error;
          FPU_exception_mode ← 0 ;
          FPU_exception_pending_mode ← 1 ;
) ;
if ( (LDC or LDDC or LDCSR) and (EC = 0 or bp_CP_present = 0) ) then (
     trap ← 1; cp_disabled ← 1 ) ;
next;
if (trap = 0) then (
     if (LDD or LD or LDSH or LDUH or LDSB or LDUB
         or LDDF or LDF or LDFSR or LDDC or LDC or LDCSR) then (
          address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
          addr_space ← (if (S = 0) then 10 else 11)
     ) else if (LDDA or LDA or LDSHA or LDUHA or LDSBA or LDUBA) then (
          if (S = 0) then (
               trap ← 1;  privileged_instruction ← 1
          )
          address ← r[rs1] + r[rs2];
          addr_space ← asi
     );
);
next;
if (trap = 0) then (
     if ( ((LDD or LDDA or LDDF or LDDC) and address<2:0> ≠ 0) or
          ((LD or LDA or LDF or LDFSR or LDC or LDCSR) and address<1:0> ≠ 0) or
          ((LDSH or LDSHA or LDUH or LDUHA) and address<0> ≠ 0) ) then (
          trap ← 1;  mem_addr_not_aligned ← 1
     )
);
next;
if (trap = 0) then (
     data ← memory_read(addr_space, address);
     MAE ← bp_memory_exception;
     next;
     if (MAE = 1) then (
          trap ← 1;  data_access_exception ← 1
     ) else (
          if (LDSB or LDSBA or LDUB or LDUBA) then (
               if      (address<1:0> = 0)  byte ← data<31:24>
               else if (address<1:0> = 1)  byte ← data<23:16>
               else if (address<1:0> = 2)  byte ← data<15:8>
               else if (address<1:0> = 3)  byte ← data<7:0>;
               next;
               if (LDSB or LDSBA) then
                    word0 ← sign_extend_byte(byte)
               else
                    word0 ← zero_extend_byte(byte)
          ) else if (LDSH or LDSHA or LDUH or LDUHA) then (
               if      (address<1:0> = 0)  halfword ← data<31:16>
               else if (address<1:0> = 2)  halfword ← data<15:0>;
               next;
```

```
                    if (LDSH or LDSHA) then
                         word0 ← sign_extend_halfword(halfword)
                    else
                         word0 ← zero_extend_halfword(halfword)
               ) else
                    word0 ← data
          )
     );
     next;
     if (trap = 0) then (
          if ( rd ≠ 0 and (LD or LDA or LDSH or or LDSHA or LDUHA or LDUH or LDSB or LDSBA or I
               r[rd] ← word0
          else if ( ((rd and 11110₂) ≠ 0) and (LDD or LDDA) ) then
               r[rd and 11110₂] ← word0
          else if (LDF) then
               f[rd] ← word0
          else if (LDFSR) then (
               wait_for_FAUs_to_complete;  {implementation-defined}
               FSR ← word0 )
          else if (LDC) then
               c[rd] ← word0
          else if (LDCSR) then
               CSR ← word0
     );
     next;
     if (trap = 0 and (LDD or LDDA or LDDF or LDDC)) then (
          word1 ← memory_read(addr_space, address + 4);
          MAE ← bp_memory_exception;
          next;
          if (MAE = 1) then (
               trap ← 1;  data_access_exception ← 1 )
          else if (LDD or LDDA) then
               r[rd or 1] ← word1
          else if (LDDF) then
               f[rd or 1] ← word1
          else if (LDDC) then
               c[rd or 1] ← word1

     );
```

C.10.2. Store Instructions

```
if ((STF or STDF or STFSR or STDFQ) and (EF = 0 or bp_FPU_present = 0) ) then (
    trap ← 1; fp_disabled ← 1 ) ;
if ((STC or STDC or STCSR or STDCQ) and (EC = 0 or bp_CP_present = 0) ) then (
    trap ← 1; cp_disabled ← 1 ) ;
if (trap = 0) then (
    if (STD or ST or STH or STB or STF or STDF or STFSR or STDFQ or STCSR or STC or STDC o
        address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
        addr_space ← (if S=0 then 10 else 11)
    ) else if (STDA or STA or STHA or STBA) then (
        if (S = 0) then (
            trap ← 1; privileged_instruction ← 1
        ) else (
            address ← r[rs1] + r[rs2];
            addr_space ← asi;
        )
    );
);
next;
if (trap = 0) then (
    if (STD or STDA or STDF or STDFQ or STDC or STDCQ) then (
        if (address<2:0> ≠ 0) then
            trap ← 1; mem_addr_not_aligned ← 1 )
    else if (ST or STA or STF or STFSR or STC or STCSR) then (
        if (address<1:0> ≠ 0) then
            trap ← 1; mem_addr_not_aligned ← 1 )
    else if (STH or STHA) then (
        if (address<0> ≠ 0) then (
            trap ← 1; mem_addr_not_aligned ← 1 )
    );
);
next;
if (trap = 0) then (
    if (STDF) then (
        byte_mask ← 1111₂; data0 ← f[rd and 1110₂] )
    else if (STDFQ) then (
        byte_mask ← 1111₂; data0 ← FQ.ADDR )
    else if (STDC) then (
        byte_mask ← 1111₂; data0 ← c[rd and 1110₂] )
    else if (STDCQ) then (
        byte_mask ← 1111₂; data0 ← CQ.ADDR )
    else if (STD or STDA) then (
        byte_mask ← 1111₂; data0 ← r[rd and 1110₂] )
    else if (ST or STA) then )
        byte_mask = 1111₂; data0 = r[rd])
    else if (STH or STHA) then (
        if (address<1:0> = 0) then (
            byte_mask ← 1100₂; data0 ← shift_left_logical(r[rd], 16) )
        else if (address<1:0> = 2) then (
            byte_mask ← 0011₂; data0 ← r[rd] ) )
    else if (STB or STBA) then (
        if (address<1:0> = 0) then (
            byte_mask ← 1000₂; data0 ← shift_left_logical(r[rd], 24) )
        else if (address<1:0> = 1) then (
            byte_mask ← 0100₂; data0 ← shift_left_logical(r[rd], 16) )
```

```
        else if (address<1:0> = 2) then (
            byte_mask ← 0010₂; data0 ← shift_left_logical(r[rd], 8) )
        else if (address<1:0> = 3) then (
            byte_mask ← 0001₂; data0 ← r[rd] )
    );
);
next;
if (trap = 0) then (
    memory_write(addr_space, address, byte_mask, data0);
    MAE ← bp_memory_exception
    next;
    if (MAE = 1) then (
        trap ← 1; data_access_exception ← 1
    )
);
next;
if (trap = 0) then (
    if (STD or STDA) then data1  ← r[rd or 1]
    else if (STDF) then data1  ← f[rd or 1]
        else if (STDFQ) then (
                data1 <- FQ.INSTR;
                dequeue_FQ;
        next;
        if (qne = 0) then (
            FPU_exception_mode ← 0 ;
            FPU_execute_mode  ← 1
        )
        )
    else if (STDC) then data1  ← c[rd or 1]
    else if (STDCQ) then data1  ← CQ.INSTR
    next;
    memory_write(addr_space, address + 4, 1111₂, data1);
    MAE ← bp_memory_exception;
    next;
    if (MAE = 1) then (
        trap ← 1; data_access_exception ← 1
    )
);
```

C.10.3. Atomic Load-Store Unsigned Byte Instructions

```
if (LDSTUB) then (
     address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
     addr_space ← (if (S = 0) then 10 else 11)
) else if (LDSTUBA) then (
     if (S = 0) then (
          trap ← 1; privileged_instruction ← 1
     )
     address ← r[rs1] + r[rs2];
     addr_space ← asi
);
next;
if (trap = 0) then (
     pb_retain_bus ← 1;
     next;
     data ← memory_read(addr_space, address);
     MAE ← bp_memory_exception;
     next;
     if (MAE = 1) then (
          trap ← 1;  data_access_exception ← 1
     ) else (
          if      (address<1:0> = 0)  word ← zero_extend_byte(data<31:24>)
          else if (address<1:0> = 1)  word ← zero_extend_byte(data<23:16>)
          else if (address<1:0> = 2)  word ← zero_extend_byte(data<15:8>)
          else if (address<1:0> = 3)  word ← zero_extend_byte(data<7:0>);
          next;
          if (rd ≠ 0) then r[rd] ← word
     )
);
next;
if (trap = 0) then (
     if        (address<1:0> = 0) then ( byte_mask ← 1000₂)
      else if (address<1:0> = 1) then ( byte_mask ← 0100₂)
      else if (address<1:0> = 2) then ( byte_mask ← 0010₂)
      else if (address<1:0> = 3) then ( byte_mask ← 0001₂)
     ;
     next;
     memory_write(addr_space, address, byte_mask, FFFFFFFF₁₆);
     MAE ← bp_memory_exception;
     next;
     pb_retain_bus ← 0;
     if (MAE = 1) then (
          trap ← 1; data_access_exception ← 1
     )
);
```

C.10.4. Swap r Register with Memory Instructions

```
if (SWAP) then (
   address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
   addr_space ← (if (S = 0) then 10 else 11)
) else if (SWAPA) then (
   if (S = 0) then (
      trap ← 1; privileged_instruction ← 1
   )
   address ← r[rs1] + r[rs2];
   addr_space ← asi
);
next;
if (trap = 0) then (
   temp ← r[rd];
   pb_retain_bus ← 1;
   next;
   word ← memory_read(addr_space, address);
   MAE ← bp_memory_exception;
   next;
   if (MAE = 1) then (
      trap ← 1;  data_access_exception ← 1
   ) else (
      if (rd ≠ 0) then r[rd] ← word
   )
);
next;
if (trap = 0) then (
   memory_write(addr_space, address, 1111₂, temp);
   MAE ← bp_memory_exception;
   next;
   pb_retain_bus ← 0;
   if (MAE = 1) then (
      trap ← 1; data_access_exception ← 1
   )
);
```

C.10.5. Add Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

if (ADD or ADDcc) then
     result ← r[rs1] + operand2;
else if (ADDX or ADDXcc) then
     result ← r[rs1] + operand2 + C;
next;
if (rd ≠ 0) then
     r[rd] ← result;
if (ADDcc or ADDXcc) then (
     N ← result<31>;
     Z ← if result=0 then 1 else 0;
     V ← (r[rs1]<31> and operand2<31> and not result<31>) or
          (not r[rs1]<31> and not operand2<31> and result<31>);
     C ← (r[rs1]<31> and operand2<31>) or
          (not result<31> and (r[rs1]<31> or operand2<31>))
);
```

C.10.6. Tagged Add Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

result ← r[rs1] + operand2;
next;
temp_v ← (r[rs1]<31> and operand2<31> and not result<31>) or
     (not r[rs1]<31> and not operand2<31> and result<31>) or
     (r[rs1]<1:0> ≠ 0 or operand2<1:0> ≠ 0);
next;
if (TADDccTV and temp_v = 1) then (
     trap ← 1; tag_overflow ← 1
) else (
     N ← result<31>;
     Z ← if result=0 then 1 else 0;
     V ← temp_v;
     C ← (r[rs1]<31> and operand2<31>) or
          (not result<31> and (r[rs1]<31> or operand2<31>));
     if (rd ≠ 0) then
          r[rd] ← result;
);
```

C.10.7. Subtract Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

if (SUB or SUBcc) then
     result ← r[rs1] - operand2;
else if (SUBX or SUBXcc) then
     result ← r[rs1] - operand2 - C;
next;
if (rd ≠ 0) then
     r[rd] ← result;
if (SUBcc or SUBXcc) then (
     N ← result<31>;
     Z ← if result=0 then 1 else 0;
     V ← (r[rs1]<31> and not operand2<31> and not result<31>) or
          (not r[rs1]<31> and operand2<31> and result<31>);
     C ← (not r[rs1]<31> and operand2<31>) or
          (result<31> and (not r[rs1]<31> or operand2<31>))
);
```

C.10.8. Tagged Subtract Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

result ← r[rs1] - operand2;
next;
temp_v ← (r[rs1]<31> and not operand2<31> and not result<31>) or
     (not r[rs1]<31> and operand2<31> and result<31>) or
     (r[rs1]<1:0> ≠ 0 or operand2<1:0> ≠ 0);
next;
if (TSUBccTV and temp_v = 1) then (
     trap ← 1; tag_overflow ← 1
) else (
     N ← result<31>;
     Z ← if result=0 then 1 else 0;
     V ← temp_v;
     C ← (not r[rs1]<31> and operand2<31>) or
          (result<31> and (not r[rs1]<31> or operand2<31>));
     if (rd ≠ 0) then
          r[rd] ← result;
);
```

C.10.9. Multiply Step Instruction

```
operand1 := (N xor V)||r[rs1]<31:1>;
operand2 :=  (
     if (Y<0> = 0) then 0
     else if (i = 0) then r[rs2] else sign_extend(simm13)
);

result ← operand1 + operand2;
Y ← r[rs1]<0>||Y<31:1>;
next;
if (rd ≠ 0) then
     r[rd] ← result;
N ← result<31>;
Z ← if result=0 then 1 else 0;
V ← (operand1<31> and operand2<31> and not result<31>) or
     (not operand1<31> and not operand2<31> and result<31>);
C ← (operand1<31> and operand2<31>) or
     (not result<31> and (operand1<31> or operand2<31>))
```

C.10.10. Logical Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

if (AND or ANDcc) then result ← r[rs1] and operand2
else if (ANDN or ANDNcc) then result ← r[rs1] and not operand2
else if (OR or ORcc) then result ← r[rs1] or operand2
else if (ORN or ORNcc) then result ← r[rs1] or not operand2
else if (XOR or XORcc) then result ← r[rs1] xor operand2
else if (XNOR or XNORcc) then result ← r[rs1] xor not operand2;
next;
if (rd ≠ 0) then r[rd] ← result;
if (ANDcc or ANDNcc or ORcc or ORNcc or XORcc or XNORcc) then (
     N ← result<31>;
     Z ← if result=0 then 1 else 0;
     V ← 0;
     C ← 0
);
```

C.10.11. Shift Instructions

```
shift_count := if i=0 then r[rs2]<4:0> else shcnt;

if (SLL and rd ≠ 0) then
    r[rd] ← shift_left_logical(r[rs1], shift_count)
else if (SRL and rd ≠ 0) then
    r[rd] ← shift_right_logical(r[rs1], shift_count)
else if (SRA and rd ≠ 0) then
    r[rd] ← shift_right_arithmetic(r[rs1], shift_count)
```

C.10.12. SETHI Instruction

```
if (rd ≠ 0) then (
    r[rd]<31:10> ← imm22;
    r[rd]<9:0> ← 0
)
```

C.10.13. SAVE and RESTORE Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

if (SAVE) then (
    new_cwp ← (CWP - 1) mod NWINDOWS;
    next;
    if ((WIM and 2^new_cwp) ≠ 0) then (
        trap ← 1; window_overflow ← 1
    ) else (
        result ← r[rs1] + operand2;  {operands from old window}
        CWP ← new_cwp
    )
) else if (RESTORE) then (
    new_cwp ← (CWP + 1) mod NWINDOWS;
    next;
    if ((WIM and 2^new_cwp) ≠ 0) then (
        trap ← 1; window_underflow ← 1
    ) else (
        result ← r[rs1] + operand2;  {operands from old window}
        CWP ← new_cwp
    )
);
next;
if (trap = 0 and rd ≠ 0) then
    r[rd] ← result                {destination in new window}
```

C.10.14. Branch on Integer Condition Instructions

```
eval_icc := (
    if (BNE and (Z = 0)) then 1 else 0;
    if (BE and (Z = 1)) then 1 else 0;
    if (BG and ((Z or (N xor V)) = 0)) then 1 else 0;
    if (BLE and ((Z or (N xor V)) = 1)) then 1 else 0;
    if (BGE and ((N xor V) = 0)) then 1 else 0;
    if (BL and ((N xor V) = 1)) then 1 else 0;
    if (BGU and (C = 0 and Z = 0)) then 1 else 0;
    if (BLEU and (C = 1 or Z = 1)) then 1 else 0;
    if (BCC and (C = 0)) then 1 else 0;
    if (BCS and (C = 1)) then 1 else 0;
    if (BPOS and (N = 0)) then 1 else 0;
    if (BNEG and (N = 1)) then 1 else 0;
    if (BVC and (V = 0)) then 1 else 0;
    if (BVS and (V = 1)) then 1 else 0;
    if (BA) then 1;
    if (BN) then 0
);

PC ← nPC;
if (eval_icc) = 1 then (
    nPC ← PC + sign_extend(disp22□00₂);
    if (BA and a = 1) then
        annul ← 1
) else (
    nPC ← nPC + 4;
    if (a = 1) then
        annul ← 1
)
```

C.10.15. Floating-Point Branch on Condition Instructions

```
E := if fcc=0 then 1 else 0;
L := if fcc=1 then 1 else 0;
G := if fcc=2 then 1 else 0;
U := if fcc=3 then 1 else 0;

eval_fcc := (
    if (FBU and U)) then 1 else 0;
    if (FBG and G) then 1 else 0;
    if (FBUG and (G or U) then 1 else 0;
    if (FBL and L) then 1 else 0;
    if (FBUL and (L or U)) then 1 else 0;
    if (FBLG and (L or G)) then 1 else 0;
    if (FBNE and (L or G or U)) then 1 else 0;
    if (FBE and E)) then 1 else 0;
    if (FBUE and (E or U)) then 1 else 0;
    if (FBGE and (E or G)) then 1 else 0;
    if (FBUGE and (E or G or U)) then 1 else 0;
    if (FBLE and (E or L)) then 1 else 0;
    if (FBULE and (E or L or U)) then 1 else 0;
    if (FBO and (E or L or G)) then 1 else 0;
    if (FBA) then 1;
    if (FBN) then 0
);

PC ← nPC;
if (eval_fcc = 1) then (
    nPC ← PC + sign_extend(disp22[] 00₂);
    if (FBA and (a = 1)) then
        annul ← 1
) else (
    nPC ← nPC + 4;
    if (a = 1) then
        annul ← 1
)
```

C.10.16. Coprocessor Branch on Condition Instructions

```
C0 := if bp_CP_cc<1:0>=0 then 1 else 0;
C1 := if bp_CP_cc<1:0>=1 then 1 else 0;
C2 := if bp_CP_cc<1:0>=2 then 1 else 0;
C3 := if bp_CP_cc<1:0>=3 then 1 else 0;

eval_bp_CP_cc := (
     if (CB3 and C3)) then 1 else 0;
     if (CB2 and C2) then 1 else 0;
     if (CB23 and (C2 or C3) then 1 else 0;
     if (CB1 and C1) then 1 else 0;
     if (CB13 and (C1 or C3)) then 1 else 0;
     if (CB12 and (C1 or C2)) then 1 else 0;
     if (CB123 and (C1 or C2 or C3)) then 1 else 0;
     if (CB0 and C0)) then 1 else 0;
     if (CB03 and (C0 or C3)) then 1 else 0;
     if (CB02 and (C0 or C2)) then 1 else 0;
     if (CB023 and (C0 or C2 or C3)) then 1 else 0;
     if (CB01 and (C0 or C1)) then 1 else 0;
     if (CB013 and (C0 or C1 or C3)) then 1 else 0;
     if (CB012 and (C0 or C1 or C2)) then 1 else 0;
     if (CBA) then 1;
     if (CBN) then 0
);

PC ← nPC;
if (eval_bp_CP_cc = 1) then (
     nPC ← PC + sign_extend(disp22□00₂);
     if (CBA and (a = 1)) then
          annul ← 1
) else (
     nPC ← nPC + 4;
     if (a = 1) then
          annul ← 1
)
```

C.10.17. CALL Instruction

```
r[15] ← PC;
PC ← nPC;
nPC ← PC + disp30□00₂
```

C.10.18. Jump and Link Instruction

```
jump_address ← r[rs1] + (if i=0 then r[rs2] else sign_ext(simm13));
next;
if (jump_address<1:0> ≠ 0) then (
     trap ← 1;
     mem_address_not_aligned ← 1
) else (
     if (rd ≠0) then r[rd] ← PC;
     PC ← nPC;
     nPC ← jump_address
)
```

C.10.19. Return from Trap Instruction

```
new_cwp ← (CWP + 1) mod NWINDOWS;
address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
next;
if (ET) then (
        trap ← 1;
        illegal_instruction ← 1
) else if (S = 0) then (
        trap ← 1;
        privileged_instruction ← 1
) else if ((WIM and 2^new_cwp) ≠ 0) then (
        trap ← 1;
        window_underflow ← 1
) else if (address<1:0> ≠ 0) then (
        trap ← 1;
        mem_address_not_aligned ← 1
) else (
        ET ← 1;
        PC ← nPC;
        nPC ← address;
        CWP ← new_cwp;
        S ← pS
)
```

C.10.20. Trap on Integer Condition Instructions

```
trap_eval_icc := (
   if (TNE and (Z = 0)) then 1 else 0;
   if (TE and (Z = 1)) then 1 else 0;
   if (TG and ((Z or (N xor V)) = 0)) then 1 else 0;
   if (TLE and ((Z or (N xor V)) = 1)) then 1 else 0;
   if (TGE and ((N xor V) = 0)) then 1 else 0;
   if (TL and ((N xor V) = 1)) then 1 else 0;
   if (TGU and (C = 0 and Z = 0)) then 1 else 0;
   if (TLEU and (C = 1 or Z = 1)) then 1 else 0;
   if (TCC and (C = 0)) then 1 else 0;
   if (TCS and (C = 1)) then 1 else 0;
   if (TPOS and (N = 0)) then 1 else 0;
   if (TNEG and (N = 1)) then 1 else 0;
   if (TVC and (V = 0)) then 1 else 0;
   if (TVS and (V = 1)) then 1 else 0;
   if (TA) then 1;
   if (TN) then 0
);

trap_number := r[rs1] + (if i=0 then r[rs2] else sign_ext(simm13));

if (Ticc) then (
   if (trap_eval_icc = 1) then (
      trap ← 1;
      trap_instruction ← 1;
      ticc_trap_type ← trap_number <6:0>
   ) else (
      PC ← nPC;
      nPC ← nPC + 4
   )
);
```

C.10.21. Read State Register Instructions

```
if ((RDPSR or RDWIM or RDTBR) and S = 0) then (
     trap ← 1;
     privileged_instruction ← 1
) else if (rd ≠ 0) then (
     if (RDY) then
          r[rd] ← Y
     else if (RDPSR) then
          r[rd] ← PSR
     else if (RDWIM) then
          r[rd] ← WIM
     else if (RDTBR) then
          r[rd] ← TBR;
);
```

C.10.22. Write State Register Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);
result := r[rs1] xor operand2;

if (WRY) then
     Y' ← result
else if (WRPSR) then (
     if (result<4:0> ≥ NWINDOWS) then (
          trap ← 1;
          illegal_instruction ← 1
     ) else if (S = 0) then (
          trap ← 1;
          privileged_instruction ← 1
     ) else
          PSR' ← result
) else if (WRWIM) then (
     if (S = 0) then (
          trap ← 1;
          privileged_instruction ← 1
     ) else
          WIM' ← result
) else if (WRTBR) then
     if (S = 0) then (
          trap ← 1;
          privileged_instruction ← 1
     ) else
          TBR' ← result
);
```

C.10.23. Unimplemented Instruction

```
trap ← 1;
illegal_instruction ← 1
```

C.10.24. Instruction Cache Flush Instruction

```
address := r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));

if (IU_cache_present) then
    flush_IU_cache_word(address) {implementation-dependent}
else if (bp_I_cache_present) then (
    trap ← 1;
    illegal_instruction ← 1
)
```

C.11. Floating-Point Operate Instructions

The multiple precision FPops use the following notation to indicate *f register* alignment:

```
double precision
     rs1E := rs1<4:1>□0₂; rs1O := rs1<4:1>□1₂;
     rs2E := rs2<4:1>□0₂; rs2O := rs2<4:1>□1₂;
     rdE := rd<4:1>□0₂; rdO := rd<4:1>□1₂

extended precision
     rs1EE := rs1<4:2>□00₂; rs1EO := rs1<4:2>□01₂; rs1OE := rs1<4:2>□10₂;
     rs2EE := rs2<4:2>□00₂; rs2EO := rs2<4:2>□01₂; rs2OE := rs2<4:2>□10₂;
     rdEE := rd<4:2>□00₂; rdEO := rd<4:2>□01₂; rdOE := rd<4:2>□10₂
```

Most of the floating-point routines defined below (or not defined since they are implementation-dependent) return: (1) a single, double, or extended *result*; (2) a 5-bit exception vector (*texc*) similar to the *cexc* field of the FSR, or a 2-bit condition code vector (*tfcc*) identical to the *fcc* field of the FSR; and (3) a completion status bit (*c*) which indicates whether the arithmetic unit was able to complete the operation.

C.11.1. Convert Integer to Floating-Point Instructions

```
if (FiTOs) then
        result, texc, c ← cvt_integer_to_single(f[rs2])
else if (FiTOd) then
        result, texc, c ← cvt_integer_to_double(f[rs2])
else if (FiTOx) then
        result, texc, c ← cvt_integer_to_extended(f[rs2])
```

C.11.2. Convert Floating-Point to Integer

```
if (FsTOi) then
        result, texc, c ← cvt_single_to_integer(f[rs2])
else if (FdTOi) then
        result, texc, c ← cvt_double_to_integer(f[rs2E]□f[rs2O])
else if (FxTOi) then
        result, texc, c ← cvt_extended_to_integer(f[rs2EE]□f[rs2EO]□f[rs2OE]);
```

C.11.3. Convert Between Floating-Point Formats Instructions

```
if (FsTOd) then
        result, texc, c ← cvt_single_to_double(f[rs2])
else if (FsTOx) then
        result, texc, c ← cvt_single_to_extended(f[rs2])
else if (FdTOs) then
        result, texc, c ← cvt_double_to_single(f[rs2E]||f[rs2O])
else if (FdTOx) then
        result, texc, c ← cvt_double_to_extended(f[rs2E]||f[rs2O])
else if (FxTOs) then
        result, texc, c ← cvt_extended_to_single(f[rs2E]||f[rs2O]||f[rs2OE])
else if (FxTOd) then
        result, texc, c ← cvt_extended_to_double(f[rs2EE]||f[rs2EO]||f[rs2OE])
```

C.11.4. Floating-Point Move Instructions

```
if (FMOVs) then
        result ← f[rs2]
else if (FNEGs) then
        result ← f[rs2] xor 80000000₁₆
else if (FABSs) then
        result ← f[rs2] and 7FFFFFFF₁₆;
texc ← 0;
C ← 1
```

C.11.5. Floating-Point Square Root Instructions

```
if (FSQRTs) then
        result, texc, c ← sqrt_single(f[rs2])
else if (FSQRTd) then
        result, texc, c ← sqrt_double(f[rs2E]||f[rs2O])
else if (FSQRTx) then
        result, texc, c ← sqrt_extended(f[rs2EE]||f[rs2EO]||f[rs2OE])
```

C.11.6. Floating-Point Add and Subtract Instructions

```
if (FADDs) then
        result, texc, c ← add_single(f[rs1], f[rs2])
else if (FSUBs) then
        result, texc, c ← sub_single(f[rs1], f[rs2])
else if (FADDd) then
        result, texc, c ← add_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FSUBd) then
        result, texc, c ← sub_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FADDx) then
        result, texc, c ← add_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
                f[rs2EE]□f[rs2EO]□f[rs2OE])
else if (FSUBx) then
        result, texc, c ← sub_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
                f[rs2EE]□f[rs2EO]□f[rs2OE])
```

C.11.7. Floating-Point Multiply and Divide Instructions

```
if (FMULs) then
        result, texc, c ← mul_single(f[rs1], f[rs2])
else if (FDIVs) then
        result, texc, c ← div_single(f[rs1], f[rs2])
else if (FMULd) then
        result, texc, c ← mul_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FDIVd) then
        result, texc, c ← div_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FMULx) then
        result, texc, c ← mul_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
                f[rs2EE]□f[rs2EO]□f[rs2OE])
else if (FDIVx) then
        result, texc, c ← div_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
                f[rs2EE]□f[rs2EO]□f[rs2OE])
```

C.11.8. Floating-Point Compare Instructions

```
if (FCMPs) then
        tfcc, texc, c ← compare_single(f[rs1], f[rs2])
else if (FCMPd) then
        tfcc, texc, c ← compare_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FCMPx) then
        tfcc, texc, c ← compare_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
                f[rs2EE]□f[rs2EO]□f[rs2OE])
else if (FCMPEs) then
        tfcc, texc, c ← compare_e_single(f[rs1], f[rs2]);
else if (FCMPEd) then
        tfcc, texc, c ← compare_e_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FCMPEx) then
        tfcc, texc, c ← compare_e_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
                f[rs2EE]□f[rs2EO]□f[rs2OE])
```

APPENDIX D: SOFTWARE CONSIDERATIONS

D.1. Introduction

This appendix describes how software can use the SPARC architecture effectively. It describes assumptions that compilers may make about the resources available, and how compilers can use them. It does not discuss how the operating system may use the architecture.

How to use registers is typically a very important resource allocation problem for compilers. The SPARC architecture provides windowed registers (*in*, *out*, *local*), global registers, and floating-point registers.

D.1.1. *In* and *Out* Registers

The *in* and *out* registers are used primarily for passing parameters to subroutines and receiving results from them, and for keeping track of the memory stack. When a routine is called, the caller's *out*s become the callee's *in*s.

One of the caller's *out* registers is used as the stack pointer, SP. It points to an area in which the system can store *r16* through *r31* when the register file overflows. **It is essential that this register have the correct value when the corresponding underflow trap occurs so that the register window can be reloaded.** It is also important that this register be kept up to date with register window changes, and that the overhead for doing calls be kept as small as possible. Since SP is in one of the caller's *out* registers, it can be used by the callee as its FP, and the callee can use the SAVE instruction to set its own SP from its FP.

Up to six parameters† may be passed by placing them in the *out* registers; additional parameters are passed in the memory stack. When the callee is entered, the parameters passed in registers are now in its corresponding *in*s. One of the other two *in*/*out* registers is used as the caller's old SP, which is also the current routine's frame pointer, FP (see below). The other is used to pass the subroutine's return address. With the exception of SP, *out* registers may be used as temporaries between subroutine calls.

If a routine is passed more than six parameters, the remainder are passed on the memory stack. If, on the other hand, it is passed fewer than six parameters, it may use the other parameter registers as if they were *local*s. If a register parameter has its address taken, it must be stored on the memory stack, and used from there for the lifetime of the pointer (or for the extent of the procedure, if the compiler cannot figure this out). A function returns its value by writing it into its *in*s (which are the caller's *out*s).

D.1.2. Local Registers

The *local*s are used for automatic variables and most temporaries. The compiler may also copy parameters out of the memory stack into the *local*s and use them from there. If an automatic variable has its address taken, it must be stored in the memory stack for the lifetime of the

† Six is more than adequate, since the overwhelming majority of procedures in system code — at least 97% measured statically, according to the studies cited by Weicker (Weicker, R.P., Dhrystone: A Synthetic Systems Programming Benchmark, *CACM* 27:10, October 1984) — take fewer than six parameters. The average number of parameters, measured statically or dynamically, is no greater than 2.1 in any of these studies.

pointer (or for the extent of the procedure, if the compiler cannot figure this out).

D.1.3. Global Registers

Unlike the *ins*, *locals*, and *outs*, the *globals* are not part of any register window, but are a single set of registers with global scope, like the registers of a more traditional architecture. This means that if they are used on a per-procedure basis, they must be saved and restored.

The *global* registers can be used for temporaries and for global variables or pointers, either visible to the user or maintained as part of the program's execution environment. For instance, one could by convention address all global scalars by offsets from register *r7*. This would allow 2^{13} bytes of global scalars, and would enable access to these variables faster than if they were only accessible via absolute addresses. This is because absolute addresses longer than 13 bits require a SETHI instruction.

D.1.4. Floating-Point Registers

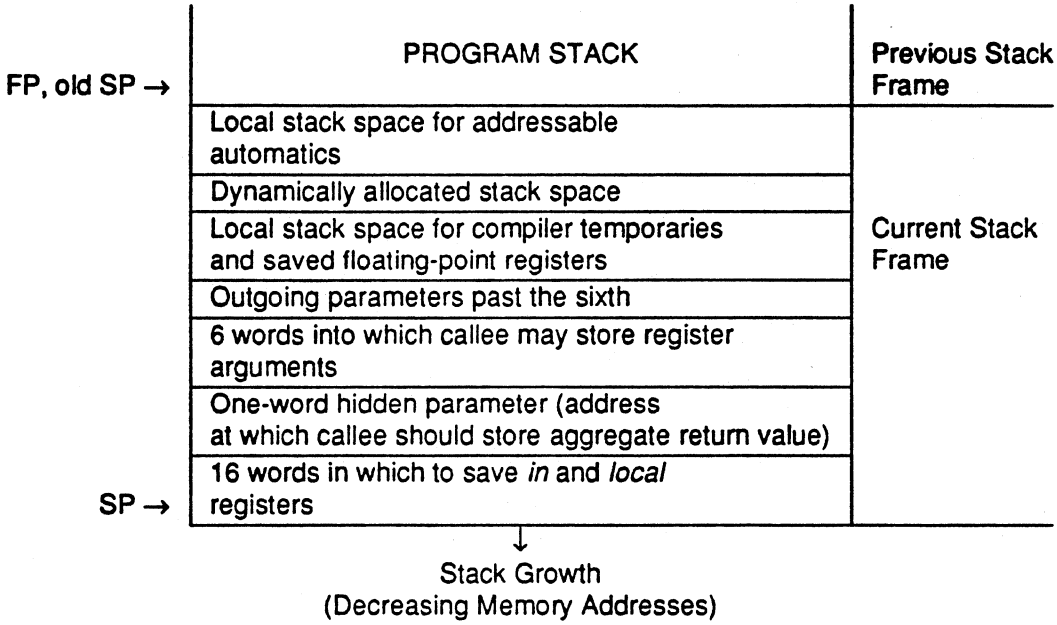
There are thirty-two 32-bit floating-point registers. They are accessed differently from the other registers and cannot be moved to or from anything but memory. Like the global registers, they must be managed by software. Compilers probably will **not** pass parameters in them, but will use them for user variables and compiler temporaries. Across a procedure call, either the caller saves the live floating-point registers, or the callee saves the ones it uses and subsequently restores them.

	r31	(i7)	return address
	r30	(FP)	frame pointer
	r29	(i5)	incoming param reg 5
in	r28	(i4)	incoming param reg 4
	r27	(i3)	incoming param reg 3
	r26	(i2)	incoming param reg 2
	r25	(i1)	incoming param reg 1
	r24	(i0)	incoming param reg 0
	r23	(l7)	local 7
	r22	(l6)	local 6
	r21	(l5)	local 5
local	r20	(l4)	local 4
	r19	(l3)	local 3
	r18	(l2)	local 2
	r17	(l1)	local 1
	r16	(l0)	local 0
	r15	(o7)	temp
	r14	(SP)	stack pointer
	r13	(o5)	outgoing param reg 5
out	r12	(o4)	outgoing param reg 4
	r11	(o3)	outgoing param reg 3
	r10	(o2)	outgoing param reg 2
	r9	(o1)	outgoing param reg 1
	r8	(o0)	outgoing param reg 0
	r7	(g7)	global 7
	r6	(g6)	global 6
	r5	(g5)	global 5
global	r4	(g4)	global 4
	r3	(g3)	global 3
	r2	(g2)	global 2
	r1	(g1)	global 1
	r0	(g0)	0
	f31		floating-point value
floating point	:		:
	f0		floating-point value

D.2. The Memory Stack

Parameters beyond the sixth are passed on the stack. Parameters which must be addressable are stored in the stack. Space is reserved on the stack for passing a one-word hidden parameter. This is used when the caller is expecting to be returned a C language *struct* by value; it gives the address of stack space allocated by the caller for that purpose (see Section D.4). Space is reserved on the stack for keeping the procedure's *in* and *local* registers, should the register stack overflow. Automatic variables which must be addressable are kept there, as are some compiler-generated temporaries. These include automatic arrays and automatic records. Space is reserved on the stack for saving floating-point registers across calls. Space on the stack may be

dynamically allocated using the *alloca* function from the C library. Automatic variables on the stack are addressed relative to FP, while temporaries and outgoing parameters are addressed relative to SP. When a procedure is active, its stack frame appears as in Figure D-2.



D.3. Example Code

In the following example we assume the following pseudo-instructions are provided by the assembler:

pseudo-instruction	equivalent instruction
ret	*jmp %i7 + 8*
retl	*jmp %o7 + 8*
mov reg_or_imm, reg	or %g0, *reg_or_imm, reg*

The following code fragment shows a simple procedure call with a value returned, and the procedure itself:

```
! CALLER:
!     int i;                              /* in register %l7 */
!     i = sum3( 1, 2, 3 );
      ...
      mov  1, %o0
      mov  2, %o1
      call sum3
      mov  3, %o2                  ! last parameter in delay slot
      mov  %o0, %l7
      ...

! CALLEE:
!     int sum3( a, b, c )
!     int a, b, c;                    /* received %i0, %i1 and %i2 */
!     {
!          return a+b+c;
!     }

sum3:
      save %sp, -(16*4), %sp        ! setup new sp
      add  %i0, %i1, %l7            ! compute sum in local
      add  %l7, %i2, %l7
      ret
      restore   %l7, 0, %o0             ! move result into output reg, restore
```

Since "sum3" does not call any subroutines (i.e. it is a "leaf" routine) it can be recoded as:

```
sum3:
      add     %o0, %o1, %o3           ! use %o3 as a local
      retl                            ! can't use ret; use retl
      add     %o2, %o3, %o0
```

D.4. Functions Returning Aggregate Values

Some programming languages, including C, some dialects of Pascal, and Modula-2, allow the user to define a function returning an aggregate value, such as a C **struct** or a Pascal **record**. Since such a value may not fit into the registers, another value returning protocol must be defined to return the result in memory. Reentrancy and efficiency considerations require that the memory used to hold such a return value be allocated by the function's caller. The address of this memory area is passed as the one-word hidden parameter mentioned in the section *The Memory Stack* in this appendix. Because of the lack of type safety in the C language, a function should not assume that its caller is expecting an aggregate return value and has provided a valid memory address. Thus some additional handshaking is required.

When a procedure expecting an aggregate function value is compiled, an UNIMP instruction is placed after the delay-slot instruction following the call to the function in question. The immediate field in this UNIMP instruction is the low-order twelve bits of the size in bytes of the aggregate value expected. When an aggregate-returning function is about to return its value in the memory allocated by its caller, it first tests for the presence of this UNIMP instruction in its caller's instruction stream. If it is found, then the hidden parameter is assumed to be valid, and the function returns control to the location following the unimplemented instruction. Otherwise, the hidden parameter is assumed *not* to be valid, and no value can be returned. Conversely, if a scalar-returning function is called when an aggregate value is expected, the function returns as usual, executing the UNIMP instruction and causing a trap.

APPENDIX E: EXAMPLE INTEGER MULTIPLICATION AND DIVISION ROUTINES

E.1. Introduction

This appendix contains routines a SPARC architecture system might use to perform integer multiplication and division.

In these examples, it is assumed that the assembler provides the following pseudo-instructions:

Pseudo Instruction	Equivalent Instruction
nop	*sethi 0,%g0*
jmp	*jmpl address, %g0*
ret	*jmp %i7 +8*
retl	*jmp %o7 +8*
mov reg_or_imm, neg	*or %g0, reg_or_imm, reg*
tst reg	*subcc reg,%g0, %g0*
neg reg	*sub %g0, reg, reg*
cmp reg, reg_or_imm	*subcc reg, reg_or_imm, %g0*
inc reg	*add reg, 1, reg*
incc reg	*addcc reg, 1, reg*
dec reg	*sub reg, 1, reg*
deccc reg	*subcc reg, 1, reg*

It is also assumed that the assembler recognizes "*/*...**/*'-style comments, and "*!*' as the beginning of a comment which extends to the end of the current line.

E.2. Signed Multiplication

```
/*
 * Procedure to perform a 32-bit by 32-bit multiply.
 * Pass the multiplicand in %i0, and the multiplier in %i1.
 * The least significant 32 bits of the result are returned in %i0,
 * and the most significant in %i1.
 *
 * This code has an optimization built-in for short (less than 13-bit)
 * multiplies.  Short multiplies require 26 or 27 instruction cycles, and
 * long ones require 47 to 51 instruction cycles.  For two positive numbers
 * (the most common case) a long multiply takes 47 instruction cycles.
 *
 * This code indicates that overflow has occurred by leaving the Z condition
 * code clear. The following call sequence would be used if you wish to
 * deal with overflow:
 *
 *        call      .mul
 *        nop                           ! (or set up last parameter here)
 *        bnz       overflow_code       ! (or tnz to overflow handler)
 *
 * Note that this is a Leaf routine; i.e. it calls no other routines and does
 * all of its work in the Out registers.  Thus, the usual SAVE and RESTORE
 * instructions are not needed.
 */

    .global .mul
.mul:
    mov       %o0, %y          ! multiplier to Y register
    andncc    %o0, 0xfff, %g0  ! mask out lower 12 bits
    be        mul_shortway     ! can do it the short way
    andcc     %g0, %g0, %o4    ! zero the partial product and clear N and V conditions
    !
    ! long multiply
    !
    mulscc    %o4, %o1, %o4    ! first iteration of 33
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
    mulscc    %o4, %o1, %o4
```

```
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4   ! 32nd iteration
        mulscc  %o4, %g0, %o4   ! last iteration only shifts
        !
        ! If %o0 (multiplier) was negative, the result is:
        !    (%o0 * %o1) + %o1 * (2**32)
        ! We fix that here.
        !
        tst     %o0
        rd      %y, %o0
        bge     1f
        tst     %o0             ! for when we check for overflow

        sub     %o4, %o1, %o4   ! bit 33 and up of the product are in
                                ! %o4, so we don't have to shift %o1
        !
        ! We haven't overflowed if:
        !    low-order bits are positive and high-order bits are 0
        !    low-order bits are negative and high-order bits are -1
        !
        ! If you are not interested in detecting overflow,
        ! replace the following few instructions with:
        !
        !       1:  retl
        !           mov     %o4, %o1
        !
1:
        bge     2f              ! if low-order bits were positive.
        addcc   %o4, %g0, %o1   ! return most sig. bits of prod and set
                                ! Z appropriately (for positive product)
        retl                    ! leaf-routine return
        subcc   %o4, -1, %g0    ! set Z if high order bits are -1 (for negative product)
2:
        retl                    ! leaf-routine return
        nop
        !
        ! short multiply
        !
mul_shortway:
        mulscc  %o4, %o1, %o4   ! first iteration of 13
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
```

```
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4
        mulscc  %o4, %o1, %o4   ! 12th iteration
        mulscc  %o4, %g0, %o4   ! last iteration only shifts

        rd      %y, %o5
        sll     %o4, 12, %o0    ! left shift middle bits by 12 bits
        srl     %o5, 20, %o5    ! right shift low bits by 20 bits
        !
        ! We haven't overflowed if:
        !    low-order bits are positive and high-order bits are 0
        !    low-order bits are negative and high-order bits are -1
        !
        ! if you are not interested in detecting overflow,
        ! replace the following code with:
        !
        !       or      %o5, %o4, %o0
        !       retl
        !       mov     %o4, %o1
        !
        orcc    %o5, %o0, %o0   ! merge for true product
        bge     3f              ! if low-order bits were positive.
        sra     %o4, 20, %o1    ! right shift high bits by 20 bits
                                ! and put into  %o1
        retl                    ! leaf-routine return
        subcc   %o1, -1, %g0    ! set Z if high order bits are -1 (for
                                ! negative product)
3:
        retl                    ! leaf-routine return
        addcc   %o1, %g0, %g0   ! set Z if high order bits are 0
```

E.3. Unsigned Multiplication

```
/*
 * Procedure to perform a 32 by 32 unsigned multiply.
 * Pass the multiplier in %o0, and the multiplicand in %o1.
 * The least significant 32 bits of the result will be returned in %o0,
 * and the most significant in %o1.
 *
 * This code has an optimization built-in for short (less than 13 bit)
 * multiplies. Short multiplies require 25 instruction cycles, and long ones
 * require 46 or 48 instruction cycles.
 *
 * This code indicates that overflow has occured, by leaving the Z condition
 * code clear. The following call sequence would be used if you wish to
 * deal with overflow:
 *
 *      call    .umul
 *      nop                     ! (or set up last parameter here)
 *      bnz     overflow_code   ! (or tnz to overflow handler)
 *
 * Note that this is a Leaf routine; i.e. it calls no other routines and does
 * all of its work in the Out registers.  Thus, the usual SAVE and RESTORE
 * instructions are not needed.
 */
    .global .umul
.umul:
    or      %o0, %o1, %o4   ! logical or of multiplier and multiplcand
    mov     %o0, %y         ! multiplier to Y register
    andncc  %o4, 0xfff, %o5 ! mask out lower 12 bits
    be      mul_shortway ! can do it the short way
    andcc       %g0, %g0, %o4   ! zero the partial product and clear N and V conditions
    !
    ! long multiply
    !
    mulscc  %o4, %o1, %o4   ! first iteration of 33
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
```

```
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4    ! 32nd iteration
    mulscc  %o4, %g0, %o4    ! last iteration only shifts
    /*
     * Normally, with the shifty-add approach, if both numbers are positive,
     * you get the correct result.  With 32-bit twos-complement numbers,
     * -x can be represented as ((2 - (x/(2**32)) mod 2) * 2**32.  To avoid
     * a lot of 2**32's, we can just move the radix point up to be just
     * to the left of the sign bit.  So:
     *
     *   x *  y = (xy) mod 2
     *  -x *  y = (2 - x) mod 2 * y = (2y - xy) mod 2
     *   x * -y = x * (2 - y) mod 2 = (2x - xy) mod 2
     *  -x * -y = (2 - x) * (2 - y) = (4 - 2x - 2y + xy) mod 2
     *
     * For signed multiplies, we subtract (2**32) * x from the partial
     * product to fix this problem for negative multipliers (see multiply.s)
     * Because of the way the shift into the partial product is calculated
     * (N xor V), this term is automatically removed for the multiplicand,
     * so we don't have to adjust.
     *
     * But for unsigned multiplies, the high order bit wasn't a sign bit,
     * and the correction is wrong.  So for unsigned multiplies where the
     * high order bit is one, we end up with xy - (2**32) * y.  To fix it
     * we add y * (2**32).
     */
    tst     %o1
    bge     1f
    nop
    add     %o4, %o0, %o4
1:
    rd      %y, %o0         ! return least sig. bits of prod
    retl                    ! leaf-routine return
    addcc       %o4, %g0, %o1    ! delay slot; return high bits and set
                            ! zero bit appropriately
    !
    ! short multiply
    !
mul_shortway:
    mulscc  %o4, %o1, %o4    ! first iteration of 13
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
    mulscc  %o4, %o1, %o4
```

```
mulscc  %o4, %o1, %o4
mulscc  %o4, %o1, %o4
mulscc  %o4, %o1, %o4
mulscc  %o4, %o1, %o4
mulscc  %o4, %o1, %o4    ! 12th iteration
mulscc  %o4, %g0, %o4    ! last iteration only shifts

rd      %y, %o5
sll     %o4, 12, %o4 ! left shift partial product by 12 bits
srl     %o5, 20, %o5 ! right shift product by 20 bits
or      %o5, %o4, %o0    ! merge for true product
!
! The delay instruction (addcc) moves zero into %o1,
! sets the zero condition code, and clears the other conditions.
! This is the equivalent result to a long umultiply which doesn't overflow.
!
retl                ! leaf-routine return
addcc       %g0, %g0, %o1
```

E.4. Division

Integer division implemented in software or microcode is usually done by a method such as the non-restoring algorithm, which provides one digit of quotient per step. A W-by-W digit division, of radix-B digits, is most easily achieved using 2*W-digit arithmetic.

E.4.1. Program 1

A binary-radix, 16-digit version of this method is illustrated by the C language function in Program 1, which performs an unsigned division, producing the quotient in Q and the remainder in R.

```
#include <stdio.h>
#include <assert.h>

#define W 16    /* maximum number of bits in the dividend & divisor */

unsigned short
divide( dividend, divisor )
    unsigned  short dividend, divisor;
{
    long R;     /* partial remainder -- need 2*W bits */
    unsigned short Q; /* partial quotient */
    int iter;

    R = dividend;
    Q = 0;
    for ( iter = W; iter >= 0; iter -= 1 ){
     assert( Q*divisor+R == dividend );
     if (R >= 0){
         R -= divisor <<iter;
         Q  += 1<<iter;
     } else {
         R += divisor <<iter;
         Q  -= 1<<iter;
     }
    }
    if ( R < 0 ){
     R += divisor;
     Q -= 1;
    }
    return Q;
}
```

E.4.2. Program 2

In the simple form shown above, this method has two drawbacks:

- It requires a 2*W-digit accumulator
- It always requires W steps.

Both these problems may be overcome by estimating the quotient before the actual division is carried out. This can cut the time required for a division from O(W) to $O(log_B(\text{quotient}))$. Program 2 illustrates how this estimate may be used to reduce the number of divide steps required and the size of the accumulator.

```
#include <stdio.h>
#include <assert.h>

#define W 32 /* maximum number of bits in a divisor or dividend */

#define Big_value (unsigned)(1<<(W-2)) /* 2 ^ (W-1) */

int
estimate_log_quotient( dividend, divisor )
    unsigned dividend, divisor;
{
    unsigned log_quotient;

    for (log_quotient = 0; log_quotient < W; log_quotient += 1 )
    if ( ( divisor <<log_quotient) > Big_value )
        break;
    else if ( (divisor <<log_quotient) >= dividend )
        break;

    return log_quotient;
}

unsigned
divide( dividend, divisor )
    unsigned dividend, divisor;
{
    int  R; /* remainder */
    unsigned Q; /* quotient  */

    int iter;
    R = dividend;
    Q = 0;
    for ( iter = estimate_log_quotient( dividend, divisor); iter >= 0; iter -= 1 ){
    assert( Q*divisor+R == dividend );
    if (R >= 0){
        R -= divisor <<iter;
        Q  += 1<<iter;
    } else {
        R += divisor <<iter;
        Q  -= 1<<iter;
    }
    }
    if ( R < 0 ){
    R += divisor;
    Q -= 1;
    }
    return Q;
}
```

E.4.3. Program 3

Another way of reducing the number of division steps required is to choose a larger base, B'. This is only feasible if the cost of the radix-B' inner loop does not exceed the cost of the radix-B inner loop by more than $\log_B$ (B'). When $B' = B^N$ for some integer N, a radix-B' inner loop can easily be constructed from the radix-B inner loop by arranging an N-high, B-ary decision tree. Programs 3 and 4 illustrate how this can be done. Program 3 uses N-level recursion to show the principle, but the overhead of recursion in this example far outweighs the loop overhead saved by reducing the number of steps required. Program 4 shows how run-time recursion can be eliminated if N is fixed at two.

```
#include <stdio.h>
#include <assert.h>

#define W 32 /* bits in a word */

int B,          /* number base of division (must be a power of 2) */
    N;          /* log2(B) */
#define WB  (W/N)    /* base B digits in a word */
#define Big_value (unsigned)(B<<(WB-2)) /* B ^ (WB-1) */

int Q,     /* partial quotient */
    R,     /* partial remainder */
    V;     /* multiple of the divisor */
int
estimate_log_quotient( dividend, divisor )
    unsigned dividend, divisor;
{
    unsigned log_quotient;

    for (log_quotient = 0; log_quotient < WB; log_quotient += 1 )
     if ( ( divisor <<log_quotient*N) > Big_value )
         break;
     else if ( (divisor <<log_quotient*N) >= dividend )
         break;

    return log_quotient;
}


int
compute_digit( level, quotient_digit)
     int level, quotient_digit;
{
     if (R >= 0){
          R -= V << level;
          quotient_digit += 1<<level;
     } else {
          R += V << level;
          quotient_digit -= 1<<level;
     }
     if (level > 0)
          return compute_digit( level-1, quotient_digit );
     else
          return quotient_digit;
}


unsigned
divide( dividend, divisor )
    unsigned dividend, divisor;
{

    int iter;
```

```
    B = (1<<(N));
    R = dividend;
    Q = 0;
    for ( iter = estimate_log_quotient( dividend, divisor); iter >= 0; iter -= 1 ){
     assert( Q*divisor+R == dividend );
     V = divisor << (iter*N);
     Q += compute_digit( N-1, 0) << (iter*N);
    }
    if ( R < 0 ){
     R += divisor;
     Q -= 1;
    }
    return Q;
}
```

E.4.4. Program 4

```
#include <stdio.h>
#include <assert.h>

#define W 32    /* bits in a word */

#define B 4     /* number base of division (must be a power of 2) */
#define N 2     /* log2(B) */
#define WB  (W/N)    /* base B digits in a word */
#define Big_value (unsigned)(B<<(WB-2)) /* B ^ WB-1 */

int
estimate_log_quotient( dividend, divisor )
    unsigned dividend, divisor;
{
    unsigned log_quotient;

    for (log_quotient = 0; log_quotient < WB; log_quotient += 1 )
     if ( ( divisor <<log_quotient*N) > Big_value )
         break;
     else if ( (divisor <<log_quotient*N) >= dividend )
         break;

    return log_quotient;
}


int
unsigned
divide( dividend, divisor )
    unsigned dividend, divisor;
{

int Q,     /* partial quotient */
    R,     /* partial remainder */
    V;     /* multiple of the divisor */
    int iter;

    R = dividend;
    Q = 0;
    for ( iter = estimate_log_quotient( dividend, divisor); iter >= 0; iter -= 1 ){
     assert( Q*divisor+R == dividend );
     V = divisor << (iter*N);
     /* N-deep, B-wide decision tree */
     if ( R >= 0 ){
          R -= V<<1;
          if ( R >= 0 ){
               R -= V;
               Q += 3 <<(N*iter);
          } else {
               R += V;
               Q += 1 <<(N*iter);
          }
     } else {
          R += V<<1;
```

```
            if ( R >= 0 ){
                R -= V;
                Q -= 1 <<(N*iter);
            } else {
                R += V;
                Q -= 3 <<(N*iter);
            }
        }
    }
    if ( R < 0 ){
     R += divisor;
     Q -= 1;
    }
    return Q;
}
```

E.4.5. Program 5

At the risk of losing even more clarity, we can optimize away several of the bookkeeping operations, as shown in Program 5.

```
#include <stdio.h>
#include <assert.h>

#define W 32    /* bits in a word */

#define B 4     /* number base of division (must be a power of 2) */
#define N 2     /* log2(B) */
#define WB  (W/N)    /* base B digits in a word */
#define Big_value (unsigned)(B<<(WB-2)) /* B ^ WB-1 */


int
unsigned
divide( dividend, divisor )
    unsigned dividend, divisor;
{

    int Q, /* partial quotient */
        R, /* partial remainder */
        V; /* multiple of the divisor */
    int iter;

    R = dividend;
    Q = 0;
    V = divisor;
    for ( iter = 0; V <= Big_value && V <= dividend; iter += 1 )
     V <<= N;

    for ( V <<= (N-1); iter >= 0; iter -= 1 ){
     Q <<= N;
     assert( Q*(1<<(iter*N))*divisor+R == dividend );
     /* N-deep, B-wide decision tree */
     if ( R >= 0 ){
          R -= V;
          V >>= 1;
          if ( R >= 0 ){
               R -= V;
               V >>= 1;
               Q += 3 ;
          } else {
               R += V;
               V >>= 1;
               Q += 1 ;
          }
     } else {
          R += V;
          V >>= 1;
          if ( R >= 0 ){
               R -= V;
               V >>= 1;
               Q -= 1;
          } else {
               R += V;
               V >>= 1;
```

```
                Q -= 3;
            }
        }
    }
    if ( R < 0 ){
     R += divisor;
     Q -= 1;
    }
    return Q;
}
```

E.4.6. Program 6

Program 6 is, essentially, the method we recommend for SPARC. The depth of the decision tree — two in the preceding examples — is controlled by the constant N, and is currently set to three, based on empirical evidence. The decision tree is not explicitly coded, but defined by the recursive *m4* macro DEVELOP_QUOTIENT_BITS. Other differences include:

- Handling of signed and unsigned operands
- More care is taken to avoid overflow for very large quotients or divisors
- Special tests are made for division by zero and zero quotient
- The routine is conditionally compiled for either division or remaindering.

```
/*
 * Divison/Remainder
 *
 * Input is:
 *   dividend -- the thing being divided
 *   divisor  -- how many ways to divide it
 * Important parameters:
 *   N -- how many bits per iteration we try to get
 *        as our current guess: define(N, 3)
 *   WORDSIZE -- how many bits altogether we're talking about:
 *        obviously: define(WORDSIZE, 32)
 * A derived constant:
 *   TOPBITS -- how many bits are in the top "decade" of a number:
 *        define(TOPBITS, eval( WORDSIZE - N*((WORDSIZE-1)/N) ) )
 * Important variables are:
 *   Q -- the partial quotient under development -- initally 0
 *   R -- the remainder so far -- initially == the dividend
 *   ITER -- number of iterations of the main division loop which will
 *        be required. Equal to CEIL( lg2(quotient)/N )
 *        Note that this is log_base_(2^N) of the quotient.
 *   V -- the current comparand -- initially divisor*2^(ITER*N-1)
 * Cost:
 *   current estimate for non-large dividend is
 *        CEIL( lg2(quotient) / N ) x ( 10 + 7N/2 ) + C
 *   a large dividend is one greater than 2^(31-TOPBITS) and takes a
 *   different path, as the upper bits of the quotient must be developed
 *   one bit at a time.
 *   This uses the m4 and cpp macro preprocessors.
 */

#include "sw_trap.h"

define(dividend, `%i0')
define(divisor, `%i1')
define(Q, `%i2')
define(R, `%i3')
define(ITER,    `%l0')
define(V, `%l1')
define(SIGN,    `%l2')
define(T, `%l3')     ! working variable
define(SC, `%l4')
/*
 * This is the recursive definition of how we develop quotient digits.
 * It takes three important parameters:
 *   $1 -- the current depth, 1<=$1<=N
 *   $2 -- the current accumulation of quotient bits
 *   N  -- max depth
 * We add a new bit to $2 and either recurse or insert the bits in the quotient.
 * Dynamic input:
 *   R -- current remainder
 *   Q -- current quotient
 *   V -- current comparand
 *   cc -- set on current value of R
 * Dynamic output:
```

```
 *   R', Q', V', cc'
 */

define(DEVELOP_QUOTIENT_BITS,
`    !depth $1, accumulated bits $2
     bl   L.$1.eval(2^N+$2)
     srl  V,1,V
     ! remainder is positive
     subcc R,V,R
     ifelse( $1, N,
     `    b    9f
          add  Q, ($2*2+1), Q
     ',`  DEVELOP_QUOTIENT_BITS( incr($1), `eval(2*$2+1)')
     ')
L.$1.eval(2^N+$2):              ! remainder is negative
     addcc R,V,R
     ifelse( $1, N,
     `    b    9f
          add  Q, ($2*2-1), Q
     ',`  DEVELOP_QUOTIENT_BITS( incr($1), `eval(2*$2-1)')
     ')
     ifelse( $1, 1, `9:')
')
ifelse( ANSWER, `quotient', `
     .global   .div, .udiv
.udiv:    ! UNSIGNED DIVIDE
     save %sp,-64,%sp
     b    divide
     mov  0,SIGN                ! result always positive

.div:! SIGNED DIVIDE
     save %sp,-64,%sp
     orcc divisor,dividend,%g0 ! are either dividend or divisor negative
     bge  divide                ! if not, skip this junk
     xor  divisor,dividend,SIGN ! record sign of result in sign of SIGN
     tst  divisor
     bge  2f
     tst  dividend
     !    divisor < 0
     bge  divide
     neg  divisor
     2:
     !    dividend < 0
     neg  dividend
     !    FALL THROUGH
',`
     .global   .rem, .urem
.urem:    ! UNSIGNED REMAINDER
     save %sp,-64,%sp           ! do this for debugging
     b    divide
     mov  0,SIGN                ! result always positive

.rem:! SIGNED REMAINDER
     save %sp,-64,%sp           ! do this for debugging
```

```
        orcc divisor,dividend,%g0 ! are either dividend or divisor negative
        bge  divide               ! if not, skip this junk
        mov  dividend,SIGN        ! record sign of result in sign of SIGN
        tst  divisor
        bge  2f
        tst  dividend
        !    divisor < 0
        bge  divide
        neg  divisor
        2:
        !    dividend < 0
        neg  dividend
        !    FALL THROUGH
')


divide:
!       compute size of quotient, scale comparand
        orcc divisor,%g0,V   ! movcc   divisor,V
        te   ST_DIV0 ! if divisor = 0
        mov  dividend,R
        mov  0,Q
        sethi %hi(1<<(WORDSIZE-TOPBITS-1)),T
        cmp  R,T
        blu  not_really_big
        mov  0,ITER
        !
        ! Here, the dividend is >= 2^(31-N) or so.  We must be careful here, as
        ! our usual N-at-a-shot divide step will cause overflow and havoc.  The
        ! total number of bits in the result here is N*ITER+SC, where SC <= N.
        ! Compute ITER, in an unorthodox manner: know we need to Shift V into
        !       the top decade: so don't even bother to compare to R.
        1:
                cmp  V,T
                bgeu 3f
                mov  1,SC
                sll  V,N,V
                b    1b
                inc  ITER
        !
        ! Now compute SC
        !
        2:      addcc V,V,V
                bcc  not_too_big ! bcc    not_too_big
                add  SC,1,SC
                        !
                        ! We're here if the divisor overflowed when Shifting.
                        ! This means that R has the high-order bit set.
                        ! Restore V and subtract from R.
                        sll  T,TOPBITS,T ! high order bit
                        srl  V,1,V ! rest of V
                        add  V,T,V
                        b    do_single_div
                        dec  SC
```

```
not_too_big:
3:    cmp  V,R
      blu  2b
      nop
      be   do_single_div
      nop
! V > R: went too far: back up 1 step
!     srl  V,1,V
!     dec  SC
! do single-bit divide steps
!
! We have to be careful here. We know that R >= V, so we can do the
! first divide step without thinking.  BUT, the others are conditional,
! and are only done if R >= 0.  Because both R and V may have the high-
! order bit set in the first step, just falling into the regular
! division loop will mess up the first time around.
! So we unroll slightly...
do_single_div:
      deccc SC
      bl   end_regular_divide
      nop
      sub  R,V,R
      mov  1,Q
      b    end_single_divlcop
      nop
single_divloop:
      sll  Q,1,Q
      bl   1f
      srl  V,1,V
      ! R >= 0
      sub  R,V,R
      b    2f
      inc  Q
1:    ! R < 0
      add  R,V,R
      dec  Q
2:
end_single_divloop:
      deccc SC
      bge  single_divloop
      tst  R
      b    end_regular_divide
      nop


not_really_big:
1:
      sll  V,N,V
      cmp  V,R
      bleu 1b
      inccc ITER
      be   got_result
      dec  ITER
do_regular_divide:
```

```
        ! Do the main division iteration
        tst  R
        ! Fall through into divide loop
divloop:
        sll  Q,N,Q
        DEVELOP_QUOTIENT_BITS( 1, 0 )
end_regular_divide:
        deccc ITER
        bge  divloop
        tst  R
        bge  got_result
        nop
        ! non-restoring fixup here
ifelse( ANSWER, 'quotient',
`       dec  Q
',`     add  R,divisor,R
')

got_result:
        tst  SIGN
        bge  1f
        restore
        ! answer < 0
        retl            ! leaf-routine return
ifelse( ANSWER, 'quotient',
`       neg  %o2,%o0    ! quotient  <- -Q
',`     neg  %o3,%o0    ! remainder <- -R
')
1:      retl            ! leaf-routine return
ifelse( ANSWER, 'quotient',
`       mov  %o2,%o0    ! quotient  <-  Q
',`     mov  %o3,%o0    ! remainder <-  R
')
```

APPENDIX F: OPCODES AND CONDITION CODES

F.1. Introduction

This appendix lists the opcodes and condition codes.

op	Instruction
01	CALL

op2	Instruction
000	UNIMP
001	*unimplemented*
010	Bicc
011	*unimplemented*
100	SETHI
101	*unimplemented*
110	FBfcc
111	CBccc

op3	Instruction
000000	ADD
000001	AND
000010	OR
000011	XOR
000100	SUB
000101	ANDN
000110	ORN
000111	XNOR
001000	ADDX
001001	*unimplemented*
001010	*unimplemented*
001011	*unimplemented*
001100	SUBX
001101	*unimplemented*
001110	*unimplemented*
001111	*unimplemented*
010000	ADDcc
010001	ANDcc
010010	ORcc
010011	XORcc
010100	SUBcc
010101	ANDNcc
010110	ORNcc
010111	XNORcc
011000	ADDXcc
011001	*unimplemented*
011010	*unimplemented*
011011	*unimplemented*
011100	SUBXcc
011101	*unimplemented*
011110	*unimplemented*
011111	*unimplemented*

op3	Instruction
100000	TADDcc
100001	TSUBcc
100010	TADDccTV
100011	TSUBccTV
100100	MULScc
100101	SLL
100110	SRL
100111	SRA
101000	RDY
101001	RDPSR
101010	RDWIM
101011	RDTBR
101100	*unimplemented*
101101	*unimplemented*
101110	*unimplemented*
101111	*unimplemented*
110000	WRY
110001	WRPSR
110010	WRWIM
110011	WRTBR
110100	FPop1
110101	FPop2
110110	CPop1
110111	CPop2
111000	JMPL
111001	RETT
111010	Ticc
111011	IFLUSH
111100	SAVE
111101	RESTORE
111110	*unimplemented*
111111	*unimplemented*

op3	Instruction
000000	LD
000001	LDUB
000010	LDUH
000011	LDD
000100	ST
000101	STB
000110	STH
000111	STD
001000	*unimplemented*
001001	LDSB
001010	LDSH
001011	*unimplemented*
001100	*unimplemented*
001101	LDSTUB
001110	*unimplemented*
001111	SWAP
010000	LDA
010001	LDUBA
010010	LDUHA
010011	LDDA
010100	STA
010101	STBA
010110	STHA
010111	STDA
011000	*unimplemented*
011001	LDSBA
011010	LDSHA
011011	*unimplemented*
011100	*unimplemented*
011101	LDSTUBA
011110	*unimplemented*
011111	SWAPA

op3	Instruction
100000	LDF
100001	LDFSR
100010	*unimplemented*
100011	LDDF
100100	STF
100101	STFSR
100110	STDFQ
100111	STDF
101000 — 101111	*unimplemented*
110000	LDC
110001	LDCSR
110010	*unimplemented*
110011	LDDC
110100	STC
110101	STCSR
110110	STDCQ
110111	STDC
111000 — 111111	*unimplemented*

opf	Instruction
000000001	FMOVs
000000101	FNEGs
000001001	FABSs
000101001	FSQRTs
000101010	FSQRTd
000101011	FSQRTx
001000001	FADDs
001000010	FADDd
001000011	FADDx
001000101	FSUBs
001000110	FSUBd
001000111	FSUBx
001001001	FMULs
001001010	FMULd
001001011	FMULx
001001101	FDIVs
001001110	FDIVd
001001111	FDIVx
011000100	FiTOs
011000110	FdTOs
011000111	FxTOs
011001000	FiTOd
011001001	FsTOd
011001011	FxTOd
011001100	FiTOx
011001101	FsTOx
011001110	FdTOx
011010001	FsTOi
011010010	FdTOi
011010011	FxTOi

opf	instruction
001010001	FCMPs
001010010	FCMPd
001010011	FCMPx
001010101	FCMPEs
001010110	FCMPEd
001010111	FCMPEx

cond	test
0000	never
0001	equal
0010	less than or equal
0011	less than
0100	less than or equal, unsigned
0101	carry set (less than, unsigned)
0110	negative
0111	overflow set
1000	always
1001	not equal
1010	greater than
1011	greater than or equal
1100	greater than, unsigned
1101	carry clear (greater than or equal, unsigned)
1110	positive
1111	overflow clear

cond	test
0000	never
0001	not equal
0010	less than or greater than
0011	unordered or less than
0100	less than
0101	unordered or greater than
0110	greater than
0111	unordered
1000	always
1001	equal
1010	unordered or equal
1011	greater than or equal
1100	unordered or greater than or equal
1101	less than or equal
1110	unodered or less than or equal
1111	ordered

opcode	cond	bp_CP_cc[1:0] test
CBN	0000	Never
CB123	0001	1 or 2 or 3
CB12	0010	1 or 2
CB13	0011	1 or 3
CB1	0100	1
CB23	0101	2 or 3
CB2	0110	2
CB3	0111	3
CBA	1000	Always
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2