B

Instruction Definitions

This Appendix includes a description of each SPARC instruction. More detailed algorithmic definitions appear in Appendix C, “ISP Descriptions.”

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 list 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. Note that in these descriptions, the symbol □ designates concatenation of bit vectors. A comma ‘**,**’ on the left side of an assignment separates quantities that are concatenated for the purpose of assignment. For example, if X, Y, and Z are 1-bit vectors, and the 2-bit vector T equals 11_2, then:

```
(X, Y, Z)  ←  0□T
```

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

(5) A list of the traps that can occur as a consequence of attempting to execute the instruction(s). Traps due to an instruction_access_error, instruction_access_exception, or r_register_access_error, and interrupt requests are not listed since they can occur on any instruction. Also, any instruction may generate an illegal_instruction trap if it is not implemented in hardware.

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

The following table summarizes the instruction set; the instruction definitions follow the table.

Table B-1 *Instruction Set*

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 deferred-trap Queue
STC	Store Coprocessor
STDC	Store Double Coprocessor
STCSR	Store Coprocessor State Register
STDCQ†	Store Double Coprocessor deferred-trap Queue
LDSTUB (LDSTUBA†)	Atomic Load-Store Unsigned Byte (in Alternate space)
SWAP (SWAPA†)	Swap r Register with Memory (in Alternate space)
SETHI	Set High 22 bits of *r* Register
NOP	No Operation
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
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)

Table B-1 *Instruction Set— Continued*

Opcode	*Name*
MULScc	Multiply Step (and modify icc)
UMUL (UMULcc) SMUL (SMULcc)	Unsigned Integer Multiply (and modify icc) Signed Integer Multiply (and modify icc)
UDIV (UDIVcc) SDIV (SDIVcc)	Unsigned Integer Divide (and modify icc) Signed Integer Divide (and modify icc)
SAVE RESTORE	Save caller's window Restore caller's window
Bicc FBfcc CBccc	Branch on integer condition codes Branch on floating-point condition codes Branch on coprocessor condition codes
CALL JMPL RETT†	Call and Link Jump and Link Return from Trap
Ticc	Trap on integer condition codes
RDASR‡ RDY RDPSR† RDWIM† RDTBR†	Read Ancillary State Register Read Y Register Read Processor State Register Read Window Invalid Mask Register Read Trap Base Register
WRASR‡ WRY WRPSR† WRWIM† WRTBR†	Write Ancillary State Register Write Y Register Write Processor State Register Write Window Invalid Mask Register Write Trap Base Register
STBAR	Store Barrier
UNIMP	Unimplemented
FLUSH	Flush Instruction Memory
FPop	Floating-point Operate: FiTO(s,d,q), F(s,d,q)TOi, FsTOd, FsTOq, FdTOs, FdTOq, FqTOs, FqTOd, FMOVs, FNEGs, FABSs, FSQRT(s,d,q), FADD(s,d,q), FSUB(s,d,q), FMUL(s,d,q), FDIV(s,d,q), FsMULd, FdMULq, FCMP(s,d,q), FCMPE(s,d,q)
CPop	Coprocessor Operate: implementation-dependent

† privileged instruction

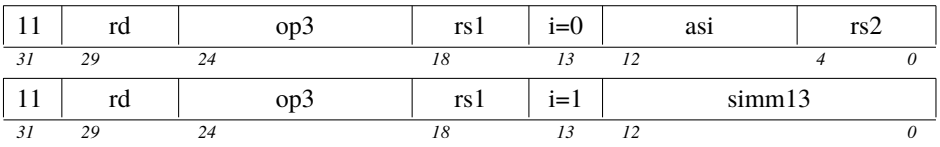
‡ privileged instruction if the referenced ASR register is privileged

B.1. Load Integer Instructions

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

† privileged instruction

Format (3):



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

Description:

The load integer instructions copy a byte, a halfword, or a word from memory into r[*rd*]. A fetched byte or halfword is right-justified in destination register r[*rd*]; it is either sign-extended or zero-filled on the left, depending on whether or not the opcode specifies a signed or unsigned operation, respectively.

The load doubleword integer instructions (LDD, LDDA) move a doubleword from memory into an *r* register pair. The more significant word at the effective memory address is moved into the even *r* register. The less significant word (at the effective memory address + 4) is moved into the following odd *r* register. (Note that a load doubleword with *rd* = 0 modifies only r[1].) The least significant bit of the *rd* field is unused and should be set to zero by software. An attempt to execute a load doubleword instruction that refers to a mis-aligned (odd) destination register number may cause an illegal_instruction trap.

The effective address for a load instruction is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one. Instructions that load from an alternate address space contain the address space identifier to be used for the load in the *asi* field, and must contain zero in the *i* field or an illegal_instruction trap will occur. Load instructions that do not load from an alternate address space access either a user data space or system data space, according to the S bit of the PSR.

A successful load (notably, load doubleword) instruction operates atomically.

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.

See Appendix L, "Implementation Characteristics," for information on the timing of the integer load instructions.

Implementation Note — During execution of a load doubleword instruction, if an exception is generated during the memory cycle in which the second word is being loaded, the destination register(s) may be modified before the trap is taken. See Chapter 7, "Traps."

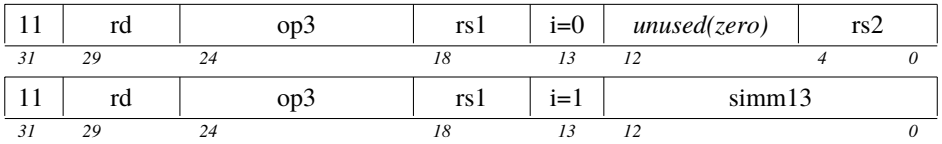
Traps:

illegal_instruction (load alternate with *i* = 1; LDD, LDDA with odd *rd*)
privileged_instruction (load alternate space only)
mem_address_not_aligned (excluding LDSB, LDSBA, LDUB, LDUBA)
data_access_exception
data_access_error

B.2. 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 f[*rd*].

The load doubleword 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 following odd *f* register. The least significant bit of the *rd* field is unused and should always be set to zero by software. If this bit is non-zero, it is recommended that LDDF cause an fp_exception trap with FSR.*ftt* = invalid_fp_register.

The load floating-point state register instruction (LDFSR) waits for all FPop instructions that have not finished execution to complete, and then loads a word from memory into the FSR. If any of the three instructions that follow (in time) a LDFSR is an FBfcc, the value of the *fcc* field of the FSR which is seen by the FBfcc is undefined.

The effective address for the load instruction is "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; LDDF traps if the address is not doubleword-aligned. If the EF field of the PSR is 0, or if no FPU is present, a load floating-point instruction causes an fp_disabled trap.

Implementation Note — If a load floating-point instruction traps with a data access exception, the destination *f* register(s) either remain unchanged or are set to an implementation-dependent predetermined constant value. See Chapter 7, "Traps," and Appendix L, "Implementation Characteristics."

Traps:

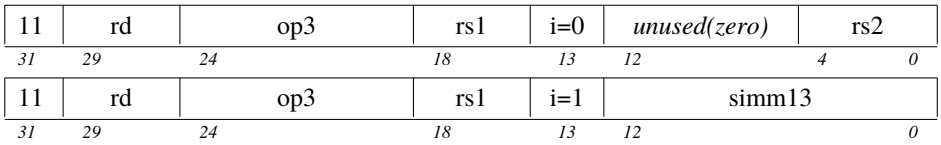
fp_disabled
fp_exception (sequence_error, invalid_fp_register(LDDF))
data_access_exception

data_access_error
mem_address_not_aligned

B.3. 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 "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; LDDC traps if the address is not doubleword-aligned. If the EC field of the PSR is 0, or if no coprocessor is present, a load coprocessor instruction causes a cp_disabled trap.

Implementation Note

An implementation might cause a data_access_exception trap due to a "non-resumable machine-check" error during an "effective address + 4" memory access, even though the corresponding "effective address" access did not cause an error. Thus, the *even* destination CP register may be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.) See Chapter 7, "Traps."

Traps:

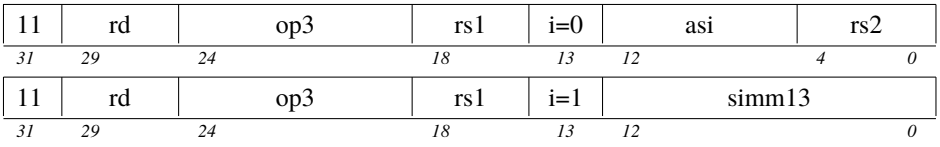
cp_disabled
cp_exception
mem_address_not_aligned
data_access_exception
data_access_error

B.4. Store Integer Instructions

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

† privileged instruction

Format (3):



Suggested Assembly Language Syntax		
`stb`	*reg*$_{rd}$, [*address*]	*(synonyms:* `stub`, `stsb`*)*
`sth`	*reg*$_{rd}$, [*address*]	*(synonyms:* `stuh`, `stsh`*)*
`st`	*reg*$_{rd}$, [*address*]	
`std`	*reg*$_{rd}$, [*address*]	
`stba`	*reg*$_{rd}$, [*regaddr*] *asi*	*(synonyms:* `stuba`, `stsba`*)*
`stha`	*reg*$_{rd}$, [*regaddr*] *asi*	*(synonyms:* `stuha`, `stsha`*)*
`sta`	*reg*$_{rd}$, [*regaddr*] *asi*	
`stda`	*reg*$_{rd}$, [*regaddr*] *asi*	

Description:

The store integer instructions copy the word, the less significant halfword, or the least significant byte from r[*rd*] into memory.

The store doubleword integer instructions (STD, STDA) copy a doubleword from an *r* register pair into memory. The more significant word (in the even-numbered *r* register) is written into memory at the effective address, and the less significant word (in the following odd-numbered *r* register) is written into memory at the "effective address + 4". The least significant bit of the *rd* field of a store doubleword instruction is unused and should always be set to zero by software. An attempt to execute a store doubleword instruction that refers to a mis-aligned (odd) *rd* may cause an illegal_instruction trap.

The effective address for a store instruction is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one. Instructions that store to an alternate address space contain the address space identifier to be

used for the store in the *asi* field, and must contain zero in the *i* field or an illegal_instruction trap will occur. Store instructions that do not store to an alternate address space access either a user data space or system data space, according to the S bit of the PSR.

A successful store (notably, store doubleword) instruction operates atomically.

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

See Chapter 6, "Memory Model," for the definition of how stores by different processors are ordered relative to one another in a multiprocessor environment.

Implementation Note

An implementation might cause a data_access_exception trap due to a "non-resumable machine-check" error during an "effective address + 4" memory access, even though the corresponding "effective address" access did not cause an error. Thus, memory data at the effective memory address may be changed in this case. Note that this cannot happen across a page boundary because of the doubleword-alignment restriction. See Chapter 7, "Traps."

Traps:

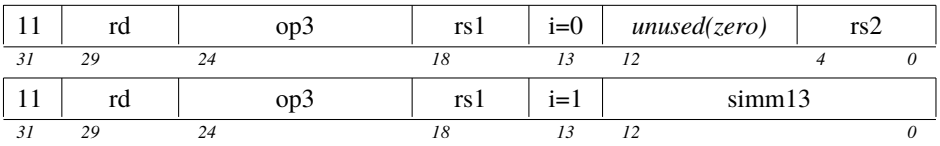
illegal_instruction (store alternate with *i* = 1; STD, STDA with odd *rd*)
privileged_instruction (store alternate only)
mem_address_not_aligned (excluding STB and STBA)
data_access_exception
data_access_error
data_store_error

B.5. 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 deferred-trap 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) copies f[*rd*] into memory.

The store double floating-point instruction (STDF) copies a doubleword from an *f* register pair into memory. The more significant word (in the even-numbered *f* register) is written into memory at the effective address, and the less significant word (in the odd-numbered *f* register) is written into memory at "effective address + 4". The least significant bit of the *rd* field is unused and should always be set to zero by software. If this bit is non-zero, it is recommended that STDF cause an fp_exception trap with FSR.*ftt* = invalid_fp_register.

The store floating-point deferred-trap queue instruction (STDFQ) stores the front doubleword of the Floating-point Queue (FQ) into memory. An attempt to execute STDFQ on an implementation without a floating-point queue causes an fp_exception trap with FSR.*ftt* set to 4 (sequence_error). On an implementation with a floating-point queue, an attempt to execute STDFQ when the FQ is empty (FSR.*qne* = 0) should cause an fp_exception trap with FSR.*ftt* set to 4 (sequence_error). Any additional semantics of this instruction are implementation-dependent. See Appendix L, "Implementation Characteristics," for information on the formats of the deferred-trap queues.

The store floating-point state register instruction (STFSR) waits for any concurrently executing FPop instructions that have not completed to complete, and then writes the FSR into memory. STFSR may zero FSR.*ftt* after

writing the FSR to memory.

The effective address for a store instruction is "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. If the EF field of the PSR is 0, or if the FPU is not present, a store floating-point instruction causes an fp_disabled trap.

See Chapter 6, "Memory Model," for the definition of how stores by different processors are ordered relative to one another in a multiprocessor environment.

Implementation Note — An implementation might cause a data_access_exception trap due to a "non-resumable machine-check" error during an "effective address + 4" memory access, even though the corresponding "effective address" access did not cause an error. Thus, memory data at the effective memory address may be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.) See Appendix L, "Implementation Characteristics."

Traps:

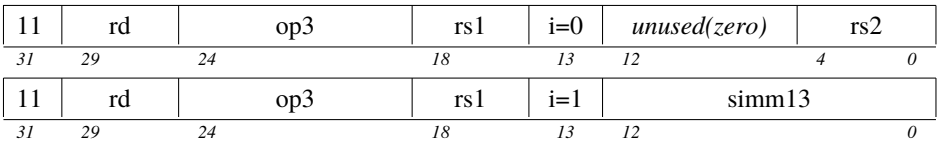
fp_disabled
fp_exception (sequence_error(STDFQ),
invalid_fp_register(STDF, STDFQ))
privileged_instruction (STDFQ only)
mem_address_not_aligned
data_access_exception
data_access_error
data_store_error

B.6. 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) copies the contents of a coprocessor register into memory.

The store double coprocessor instruction (STDC) copies the contents of a coprocessor register pair into memory.

The store coprocessor state register instruction (STCSR) copies the contents of the coprocessor state register into memory. The store doubleword coprocessor queue instruction (STDCQ) moves the front entry of the coprocessor queue into memory. On an implementation without a coprocessor queue, STDCQ may cause a cp_exception trap. The semantics of these instructions depend on the implementation of the attached coprocessor, if any.

The effective address for a store instruction is "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. 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.

See Chapter 6, "Memory Model," for the definition of how stores by different processors are ordered relative to one another in a multiprocessor environment.

Traps:

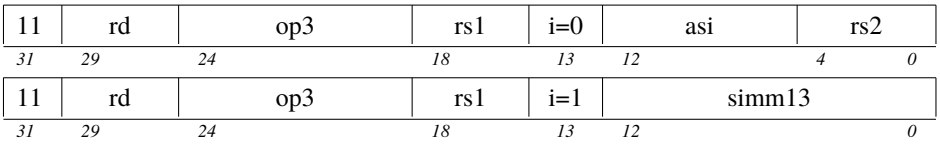
cp_disabled
cp_exception
privileged_instruction (STDCQ only)
mem_address_not_aligned
data_access_exception
illegal_instruction (STDCQ only; implementation-dependent)
data_access_error
data_store_error

B.7. 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 copy a byte from memory into r[*rd*], then rewrite the addressed byte in memory to all ones. The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more processors executing atomic load-store unsigned byte, SWAP, or SWAPA instructions addressing the same byte or word simultaneously are guaranteed to execute them in an undefined, but serial order.

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

See Chapter 6, "Memory Model," for the definition of how stores by different processors are ordered relative to one another in a multiprocessor environment.

Implementation Note — An implementation might cause a data_access_exception trap due to a "non-resumable machine-check" error during the store memory access, even though there was no error during the corresponding load access. In this case, the destination register may be changed. See Chapter 7, "Traps."

Traps:

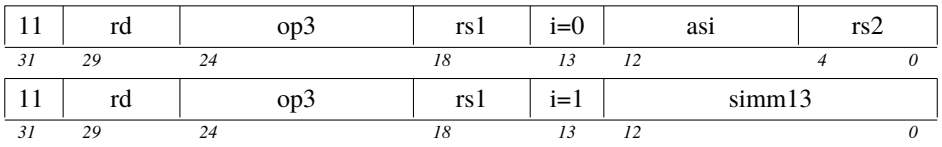
illegal_instruction (LDSTUBA with *i* = 1 only)
privileged_instruction (LDSTUBA only)
data_access_exception
data_access_error
data_store_error

B.8. SWAP Register with Memory Instruction

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

† privileged instruction

Format (3):



Suggested Assembly Language Syntax	
`swap`	[*address*] , reg_{rd}
`swapa`	[*regaddr*] *asi* , reg_{rd}

Description:

The SWAP and SWAPA instructions exchange r[*rd*] with the contents of the word at the addressed memory location. The operation is performed atomically, that is, without allowing intervening interrupts or deferred traps. In a multiprocessor system, two or more processors executing SWAP, SWAPA, or atomic load-store unsigned byte instructions addressing the same word or byte simultaneously are guaranteed to execute them in an undefined, but serial order.

The effective address of a SWAP instruction is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one. SWAPA must contain zero in the *i* field, or an illegal_instruction trap will occur. The address space identifier used for the memory accesses is taken from the *asi* field. For SWAP, the address space is either a user or a system data space, 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.

See Chapter 6, "Memory Model," for the definition of how stores by different processors are ordered relative to one another in a multiprocessor environment.

Programming Note — See Appendix G, "SPARC ABI Software Considerations," regarding use of SWAP instructions in SPARC ABI software.

Implementation Note — An implementation might cause a data_access_exception trap due to a "non-resumable machine-check" error during the store memory access, but not during the load access. In this case, the destination register can be changed. See Chapter 7, "Traps."

Implementation Note — See Appendix L, "Implementation Characteristics," for information on the presence of hardware support for these instructions in the various SPARC implementations.

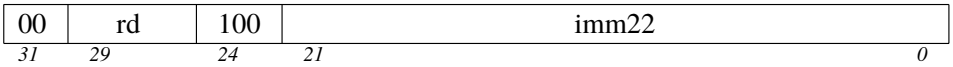
Traps:

illegal instruction (when $i = 1$, SWAPA only)
privileged_instruction (SWAPA only)
mem_address_not_aligned
data_access_exception
data_access_error
data_store_error

B.9. SETHI Instruction

opcode	*op*	*op2*	*operation*
SETHI	00	100	Set High-Order 22 bits

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 the value from its *imm22* field.

SETHI does not affect the condition codes.

A SETHI instruction with *rd* = 0 and *imm22* = 0 is defined to be a NOP instruction. See the NOP instruction page in Section B.10.

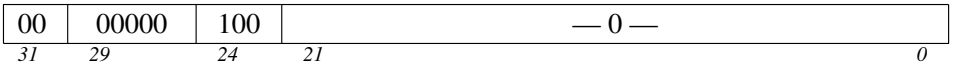
Traps:

(none)

B.10. NOP Instruction

opcode	*op*	*op2*	*operation*
NOP	00	100	No Operation

Format (2):



Suggested Assembly Language Syntax
`nop`

Description:

The NOP instruction changes no program-visible state (except the PC and nPC).

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

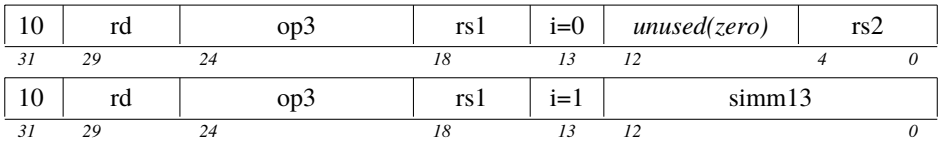
Traps:

(none)

B.11. 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 "r[*rs1*] **operation** r[*rs2*]" if the *i* field is zero, or "r[*rs1*] **operation** sign_ext(*simm13*)" if the *i* field is one, and write the result into r[*rd*].

ANDcc, ANDNcc, ORcc, ORNcc, XORcc, and XNORcc modify the integer condition codes (*icc*).

ANDN, ANDNcc, ORN, and ORNcc logically negate their second operand before applying the main (AND or OR) operation.

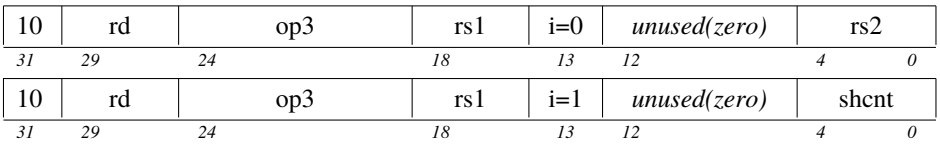
Programming Note: XNOR and XNORcc logically implement XOR-Not and XOR-Not-cc, respectively.

Traps: (none)

B.12. 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 r[*rs2*] if the *i* field is zero, or the value in *shcnt* if the *i* field is one.

When *i* is 0, the most significant 27 bits of the value in r[*rs2*] are ignored. When *i* is 1, bits 5 through 12 of the shift instruction are reserved and should be supplied as zero by software.

SLL shifts the value of r[*rs1*] left by the number of bits given 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 write the shifted result into r[*rd*].

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

Programming Note — "Arithmetic left shift by 1 (and calculate overflow)" can be effected with an ADDcc instruction.

Implementation Note — *shcnt* in shift instructions corresponds to the least significant five bits of *simm13* in other Format 3 instructions. However, bits 12 through 5 in shift instructions must be zero.

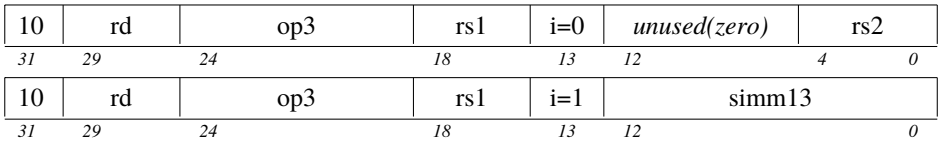
Traps:

(none)

B.13. 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 "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one, and write the sum into r[*rd*].

ADDX and ADDXcc ("ADD eXtended") also add the PSR's carry (*c*) bit; that is, they compute "r[*rs1*] + r[*rs2*] + *c*" or "r[*rs1*] + sign_ext(*simm13*) + *c*" and write the sum into r[*rd*].

ADDcc and ADDXcc modify the integer condition codes (*icc*). Overflow occurs on addition if both operands have the same sign and the sign of the sum is different.

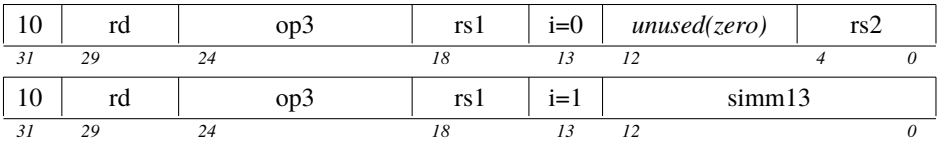
Traps:

(none)

B.14. 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 a sum that is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one.

TADDcc modifies the integer condition codes (*icc*), and TADDccTV does so also if it does not trap.

A tag_overflow occurs if bit 1 or bit 0 of either operand is nonzero, or if the addition generates an arithmetic overflow (both operands have the same sign and the sign of the sum is different).

If a TADDccTV causes a tag_overflow, a tag_overflow trap is generated and r[*rd*] and the condition codes remain unchanged. If a TADDccTV does not cause a tag_overflow, the integer condition codes are updated (in particular, the overflow bit (*v*) is set to 0) and the sum is written into r[*rd*].

If a TADDcc causes a tag_overflow, the overflow bit (*v*) of the PSR is set; if it does not cause a tag_overflow, the overflow bit is cleared. In either case, the remaining integer condition codes are also updated and the sum is written into r[*rd*].

See Appendix D, "Software Considerations," for a suggested tagging scheme.

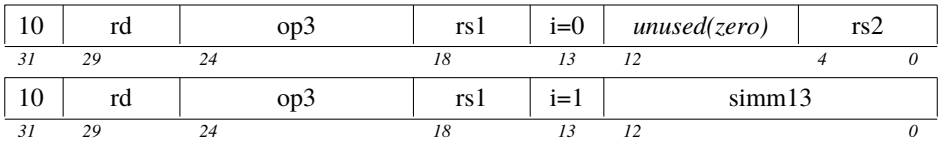
Traps:

tag_overflow (TADDccTV only)

B.15. 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 "r[*rs1*] – r[*rs2*]" if the *i* field is zero, or "r[*rs1*] – sign_ext(*simm13*)" if the *i* field is one, and write the difference into r[*rd*].

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 write the difference into r[*rd*].

SUBcc and SUBXcc modify the integer condition codes (*icc*). Overflow occurs on subtraction if the operands have different signs and the sign of the difference differs from the sign of r[*rs1*].

Programming Note — A SUBcc with *rd* = 0 can be used to effect a signed or unsigned integer comparison. See the `cmp` synthetic instruction in Appendix A.

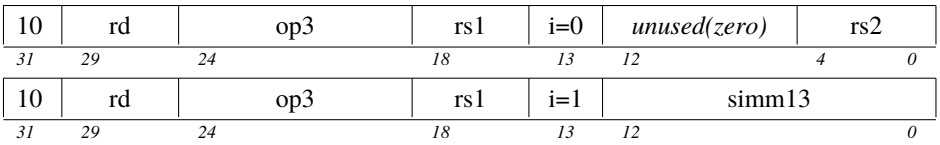
Traps:

(none)

B.16. 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 "r[*rs1*] – r[*rs2*]" if the *i* field is zero, or "r[*rs1*] – sign_ext(*simm13*)" if the *i* field is one.

TSUBcc modifies the integer condition codes (*icc*) and TSUBccTV does so also if it does not trap.

A tag_overflow occurs if bit 1 or bit 0 of either operand is nonzero, or if the subtraction generates an arithmetic overflow (the operands have different signs and the sign of the difference differs from the sign of r[*rs1*]).

If a TSUBccTV causes a tag_overflow, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If a TSUBccTV does not cause a tag_overflow condition, the integer condition codes are updated (in particular, the overflow bit (*v*) is set to 0) and the difference is written into r[*rd*].

If a TSUBcc causes a tag_overflow, the overflow bit (*v*) of the PSR is set; if it does not cause a tag_overflow, the overflow bit is cleared. In either case, the remaining integer condition codes are also updated and the difference is written into r[*rd*].

See Appendix D, "Software Considerations." for a suggested tagging scheme.

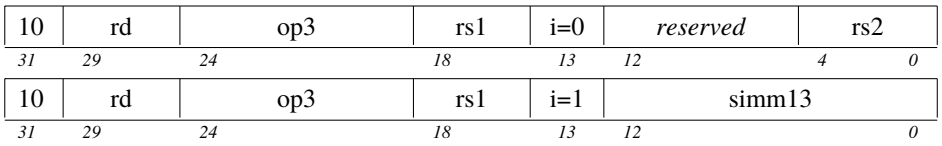
Traps:

tag_overflow (TSUBccTV only)

B.17. 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:

MULScc treats r[*rs1*] and the Y register as a single 64-bit, right-shiftable doubleword register. The least significant bit of r[*rs1*] is treated as if it were adjacent to the most significant bit of the Y register. The MULScc instruction conditionally adds, based on the least significant bit of Y.

Multiplication assumes that the Y register initially contains the multiplier, r[*rs1*] contains the most significant bits of the product, and r[*rs2*] contains the multiplicand. Upon completion of the multiplication, the Y register contains the least significant bits of the product.

Note that a standard MULScc instruction has *rs1* = *rd*. See Appendix E, "Example Integer Multiplication and Division Routines," for a 32 32 → 64 signed multiplication example program based on MULScc.

MULScc operates as follows:

(1) The multiplier is established as r[*rs2*] if the *i* field is zero, or sign_ext(*simm13*) if the *i* field is one.

(2) A 32-bit value is computed by shifting r[*rs1*] right by one bit with "N **xor** V" from the PSR replacing the high-order bit. (This is the proper sign for the previous partial product.)

(3) If the least significant bit of the Y register = 1, the shifted value from step (2) is added to the multiplier.
If the LSB of the Y register = 0, then 0 is added to the shifted value from step (2).

(4) The sum from step (3) is written into r[*rd*].

(5) The integer condition codes, *icc*, are updated according to the addition performed in step (3).

(6) The Y register is shifted right by one bit, with the LSB of the unshifted r[*rs1*] replacing the MSB of Y.

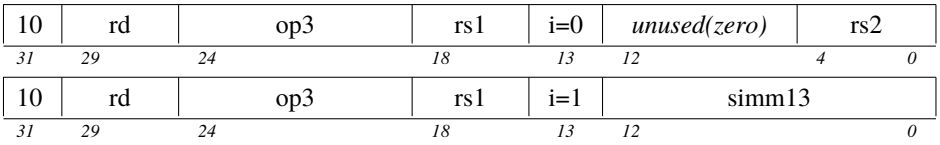
Traps:

(none)

B.18. Multiply Instructions

opcode	*op3*	*operation*
UMUL	001010	Unsigned Integer Multiply
SMUL	001011	Signed Integer Multiply
UMULcc	011010	Unsigned Integer Multiply and modify icc
SMULcc	011011	Signed Integer Multiply and modify icc

Format (3):



Suggested Assembly Language Syntax	
`umul`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`smul`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`umulcc`	reg_{rs1} , *reg_or_imm* , reg_{rd}
`smulcc`	reg_{rs1} , *reg_or_imm* , reg_{rd}

Description:

The multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They compute "r[*rs1*] r[*rs2*]" if the *i* field is zero, or "r[*rs1*] sign_ext(*simm13*)" if the *i* field is one. They write the 32 most significant bits of the product into the Y register and the 32 least significant bits into r[*rd*].

An unsigned multiply (UMUL, UMULcc) assumes unsigned integer word operands and computes an unsigned integer doubleword product. A signed multiply (SMUL, SMULcc) assumes signed integer word operands and computes a signed integer doubleword product.

UMUL and SMUL do not affect the condition code bits. UMULcc and SMULcc write the integer condition code bits, *icc*, as follows. Note that negative (N) and zero (Z) are set according to the **less** significant word of the product.

icc bit	UMULcc	SMULcc
N	Set if product[31] = 1	Set if product[31] = 1
Z	Set if product[31:0] = 0	Set if product[31:0] = 0
V	Zero †	Zero †
C	Zero †	Zero †

† Specification of this condition code may change in a future revision to the architecture. Software should not test this condition code.

Programming Note — 32-bit overflow after UMUL/UMULcc is indicated by Y != 0.
32-bit overflow after SMUL/SMULcc is indicated by Y != (r[*rd*] >> 31).

Programming Note — See Appendix G, “SPARC ABI Software Considerations,” regarding use of multiply instructions in SPARC ABI software.

Implementation Note — An implementation may assume that the smaller operand will typically be in r[*rs2*] or *simm13*.

Implementation Note — See Appendix L, “Implementation Characteristics,” for information on whether these instructions are executed in hardware or software in the various SPARC implementations.

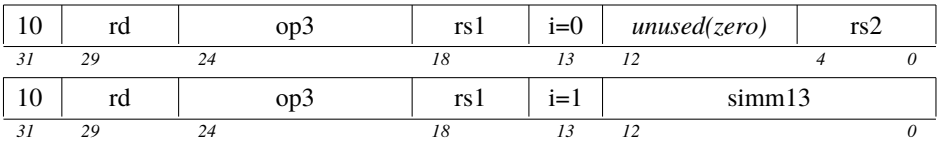
Traps:

(none)

B.19. Divide Instructions

opcode	*op3*	*operation*
UDIV	001110	Unsigned Integer Divide
SDIV	001111	Signed Integer Divide
UDIVcc	011110	Unsigned Integer Divide and modify icc
SDIVcc	011111	Signed Integer Divide and modify icc

Format (3):



Suggested Assembly Language Syntax	
`udiv`	*reg*$_{rs1}$, *reg_or_imm* , *reg*$_{rd}$
`sdiv`	*reg*$_{rs1}$, *reg_or_imm* , *reg*$_{rd}$
`udivcc`	*reg*$_{rs1}$, *reg_or_imm* , *reg*$_{rd}$
`sdivcc`	*reg*$_{rs1}$, *reg_or_imm* , *reg*$_{rd}$

Description:

The divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. If the *i* field is zero, they compute "(Y r[*rs1*]) r[*rs2*]". Otherwise (the *i* field is one), the divide instructions compute "(Y r[*rs1*]) sign_ext(*simm13*)". In either case, the 32 bits of the integer quotient are written into r[*rd*]. The remainder (if generated) is discarded.

An unsigned divide (UDIV, UDIVcc) assumes an unsigned integer doubleword dividend (Y r[*rs1*]) and an unsigned integer word divisor (r[*rs2*]) and computes an unsigned integer word quotient (r[*rd*]). A signed divide (SDIV, SDIVcc) assumes a signed integer doubleword dividend (Y r[*rs1*]) and a signed integer word divisor (r[*rs2*] or sign_ext(*simm13*)) and computes a signed integer word quotient (r[*rd*]).

Signed division rounds an inexact quotient toward zero if there is a nonzero remainder; for example, –3 2 equals –1 with a remainder of –1 (not –2 with a remainder of 1). An implementation may choose to strictly adhere to this rounding, in which case overflow for a negative result must be detected using method [A] below. Or, it may choose to make an exception for rounding with the maximum negative quotient, in which case overflow for a negative result must be detected using method [B].

The result of a divide instruction can overflow the 32-bit destination register r[rd] under certain conditions. When overflow occurs (whether or not the instruction sets the condition codes in *icc*), the largest appropriate integer is returned as the quotient in r[rd]. The conditions under which overflow occurs and the value returned in r[rd] under those conditions are specified in

the following table.

Divide Overflow Detection and Value Returned		
Instruction	Condition under which overflow occurs ("result" refers to quotient + remainder)	Value returned in r[rd]
UDIV, UDIVcc	result > ($2^{32}-1$ with a remainder of divisor−1)	$2^{32}-1$ (0xffffffff)
SDIV, SDIVcc (positive result)	result > ($2^{31}-1$ with a remainder of |divisor|−1)	$2^{31}-1$ (0x7fffffff)
SDIV, SDIVcc (negative result)	**either** † **[A]** result < (-2^{31} with a remainder of −(|divisor|−1)) **or** † **[B]** result < (-2^{31} with a remainder of 0)	-2^{31} (0x80000000)

† which of these two overflow-detection conditions is used is implementation-dependent, but must be consistent within an implementation.

UDIV and SDIV do not affect condition code bits. UDIVcc and SDIVcc write the integer condition code bits as follows. Note that negative(N) and zero(Z) are set according to the value of the quotient (after it has been set to reflect overflow, if any), and that UDIVcc and SDIVcc set overflow(V) differently.

icc bit	UDIVcc	SDIVcc
N	Set if quotient[31] = 1	Set if quotient[31] = 1
Z	Set if quotient[31:0] = 0	Set if quotient[31:0] = 0
V	Set if overflow (*per above table*)	Set if overflow (*per above table*)
C	Zero	Zero

For future compatibility, software should assume that the contents of the Y register are **not** preserved by the divide instructions.

Programming Note — See Appendix G, "SPARC ABI Software Considerations," regarding use of divide instructions in SPARC ABI software.

Implementation Note — The integer division instructions may generate a remainder. If they do, it is recommended that the remainder be stored in the Y register.

Implementation Note — See Appendix L, "Implementation Characteristics," for information on whether these instructions are executed in hardware or software, and the condition which triggers signed overflow for negative quotients in the various SPARC implementations.

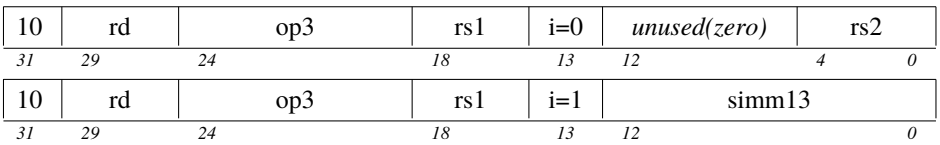
Traps:

division_by_zero

B.20. 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 NWINDOWS) and compares this value (new_CWP) against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is 1, that is, (WIM and 2^{new_CWP}) = 1, then a window_overflow trap is generated. If the WIM bit corresponding to the new_CWP is 0, then no window_overflow trap is generated and new_CWP is written into CWP. This causes the current window to become the CWP−1 window, thereby saving the caller's window.

The RESTORE instruction adds one to the CWP (modulo NWINDOWS) and compares this value (new_CWP) against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is 1, that is, (WIM and 2^{new_CWP}) = 1, then a window_underflow trap is generated. If the WIM bit corresponding to the new_CWP = 0, then no window_underflow trap is generated and new_CWP is written into CWP. This causes the CWP+1 window to become the current window, thereby restoring the caller's window.

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

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

Programming Note — The SAVE instruction can be useed to atomically allocate a new window in the register file and a new software stack frame in main memory. See Appendix D, "Software Considerations," for details.

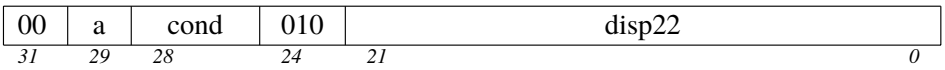
Programming Note — Typically, if a SAVE (RESTORE) instruction traps, the overflow (underflow) trap handler returns to the trapped instruction to reexecute it. So, although the ADD operation is not performed the first time (when the instruction traps), it is performed the second time.

Traps:
window_overflow (SAVE only)
window_underflow (RESTORE only)

B.21. Branch on Integer Condition Codes 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 "`,a`" to the opcode mnemonic. For example, use "`bgu,a label`". The preceding table indicates that the "`,a`" is optional by enclosing it in braces ({}).

Description:

Unconditional Branches (BA, BN)

If its annul field is 0, a BN (Branch Never) instruction acts like a "NOP". If its annul field is 1, the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

BA (Branch Always) causes a PC-relative, delayed control transfer to the address "PC + (4 sign_ext(*disp22*))", regardless of the values of the integer condition code bits. If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

Icc-Conditional Branches

Conditional Bicc instructions (all except BA and BN) evaluate the integer condition codes (*icc*), according to the *cond* field of the instruction. Such evaluation produces either a "true" or "false" result. If "true", the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 sign_ext(*disp22*))". If "false", the branch is not taken.

If a conditional branch is taken, the delay instruction is always executed regardless of the value of the annul field. If a conditional branch is not taken and the *a* (annul) field is 1, the delay instruction is annulled (not executed). (Note that the annul bit has a **different** effect on conditional branches than it does on unconditional branches.)

Annulment, delay instructions, and delayed control transfers are described further in Chapter 5, "Instructions." In particular, note that a Bicc should not be placed in the delay slot of a conditional branch instruction.

See Appendix L, "Implementation Characteristics," for information on the timing of the Bicc instructions.

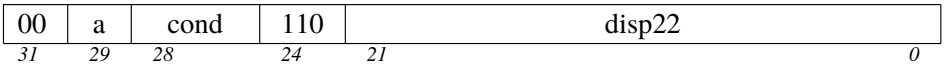
Traps:

(none)

B.22. Branch on Floating-point Condition Codes 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 "`,a`" to the opcode mnemonic. For example, use "`fbl,a label`". The preceding table indicates that the "`,a`" is optional by enclosing it in braces ({}).

Description:

Unconditional Branches (FBA, FBN)

If its annul field is 0, a FBN (Branch Never) instruction acts like a "NOP". If its annul field is 1, the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

FBA (Branch Always) causes a PC-relative, delayed control transfer to the address "PC + (4 sign_ext(*disp22*))", regardless of the value of the floating-point condition code bits. If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

Fcc-Conditional Branches

Conditional FBfcc instructions (all except FBA and FBN) evaluate the floating-point condition codes (*fcc*), according to the *cond* field of the instruction. Such evaluation produces either a "true" or "false" result. If "true", the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 sign_ext(*disp22*))". If "false", the branch is not taken.

If a conditional branch is taken, the delay instruction is always executed regardless of the value of the annul field. If a conditional branch is not taken and the *a* (annul) field is 1, the delay instruction is annulled (not executed). (Note that the annul bit has a **different** effect on conditional branches than it does on unconditional branches.)

Annulment, delay instructions, and delayed control transfers are described further in Chapter 5, "Instructions." In particular, note that an FBfcc should not be placed in the delay slot of a conditional branch instruction.

If the PSR's EF bit is 0, or if an FPU is not present, an FBfcc instruction does not branch, does not annul the following instruction, and generates an fp_disabled trap.

If the instruction executed immediately before an FBfcc is an FPop2 instruction, the result of the FBfcc is undefined. Therefore, at least one non-FPop2 instruction should be executed between an FPop2 and a subsequent FBfcc.

If any of the three instructions that follow (in time) an LDFSR is an FBfcc, the value of the *fcc* field of the FSR that is seen by the FBfcc is undefined.

See Appendix L, "Implementation Characteristics," for information on the timing of the FBfcc instructions.

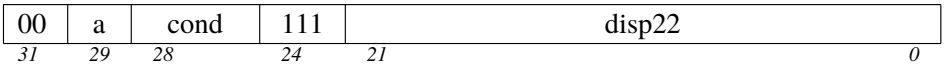
Traps:

fp_disabled
fp_exception

B.23. Branch on Coprocessor Condition Codes 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 "`,a`" to the opcode mnemonic. For example, use "`cb12,a label`". The preceding table indicates that the "`,a`" is optional by enclosing it in braces ({}).

Description:

Unconditional Branches (CBA, CBN)

If its annul field is 0, a CBN (Branch Never) instruction acts like "NOP". If its annul field is 1, the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

CBA (Branch Always) causes a PC-relative, delayed control transfer to the address "PC + (4 sign_ext(*disp22*))", regardless of the value of the condition code bits. If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

Ccc-Conditional Branches

Conditional CBccc instructions (all except CBA and CBN) evaluate the coprocessor condition codes (*ccc*), according to the *cond* field of the instruction. Such evaluation produces either a "true" or "false" result. If "true", the branch is taken; that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 sign_ext(*disp22*))". If "false", the branch is not taken.

If a conditional branch is taken, the delay instruction is always executed regardless of the value of the annul field. If a conditional branch is not taken and the *a* (annul) field is 1, the delay instruction is annulled (not executed). (Note that the annul bit has a **different** effect on conditional branches than it does on unconditional branches.)

Annulment, delay instructions, and delayed control transfers are described further in Chapter 5, "Instructions." In particular, note that a CBccc should not be placed in the delay slot of a conditional branch instruction.

If the PSR's EC bit is 0, or if a coprocessor is not present, a CBccc instruction does not branch, does not annul the following instruction, and generates a cp_disabled trap.

See Appendix L, "Implementation Characteristics," for information on the timing of the CBccc instructions.

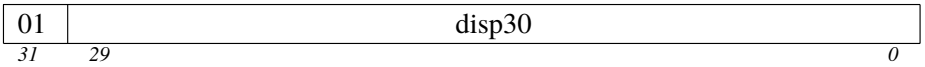
Traps:

cp_disabled
cp_exception

B.24. Call and Link Instruction

opcode	*op*	*operation*
CALL	01	Call and Link

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

The CALL instruction also writes the value of PC, which contains the address of the CALL, into r[15] (*out* register 7).

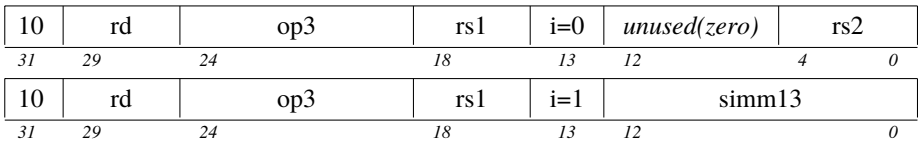
Traps:

(none)

B.25. 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 delayed control transfer to the address given by "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 copies the PC, which contains the address of the JMPL instruction, into register r[*rd*].

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

Programming Note — A JMPL instruction with *rd* = 15 functions as a register-indirect call using the standard link register. JMPL with *rd* = 0 can be used to return from a subroutine. The typical return address is "r[31]+8", if a non-leaf (uses SAVE instruction) subroutine is entered by a CALL instruction, or "r[15]+8" if a leaf (doesn't use SAVE instruction) subroutine is entered by a CALL instruction.

Implementation Note — When a RETT instruction appears in the delay slot of a JMPL, the target of the JMPL must be fetched from the address space implied by the **new** (i.e. post-RETT) value of the PSR's S bit. In particular, this applies to a return from trap to a user address space.

Traps:

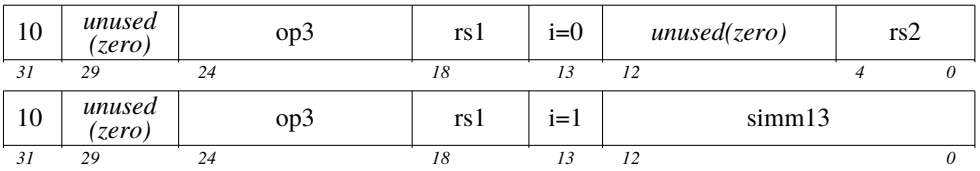
mem_address_not_aligned

B.26. Return from Trap Instruction

opcode	*op3*	*operation*
RETT†	111001	Return from Trap

† privileged instruction

Format (3):



Suggested Assembly Language Syntax
`rett` *address*

Description:

RETT is used to return from a trap handler. Under some circumstances, RETT may itself cause a trap. If a RETT instruction does not cause a trap, it (1) adds 1 to the CWP (modulo NWINDOWS), (2) causes a delayed control transfer to the target address, (3) restores the S field of the PSR from the PS field, and (4) sets the ET field of the PSR to 1. The target address is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one.

One of several traps may occur when an RETT is executed. These are described in priority order (highest priority first):

- If traps are enabled (ET=1) and the processor is in user mode (S=0), a privileged_instruction trap occurs.
- If traps are enabled (ET=1) and the processor is in supervisor mode (S=1), an illegal_instruction trap occurs.
- If traps are disabled (ET=0), and (a) the processor is in user mode (S=0), or (b) a window_underflow condition is detected (WIM **and** 2^{new_CWP}) = 1, or (c) either of the low-order two bits of the target address is nonzero, then the processor indicates a trap condition of (a) privileged_instruction, (b) window_underflow, or (c) mem_address_not_aligned (respectively) in the *tt* field of the TBR register, and enters the error_mode state.

The instruction executed immediately before an RETT must be a JMPL instruction. (If not, one or more instruction accesses following the RETT may be to an incorrect address space.)

Programming Note — To reexecute the trapped instruction when returning from a trap handler use the sequence:

```
jmpl   %r17,%r0   ! old PC
rett   %r18       ! old nPC
```

To return to the instruction after the trapped instruction (for example, after emulating an instruction) use the sequence:

```
jmpl   %r18,%r0   ! old nPC
```

```
        rett   %r18+4     ! old nPC + 4
```

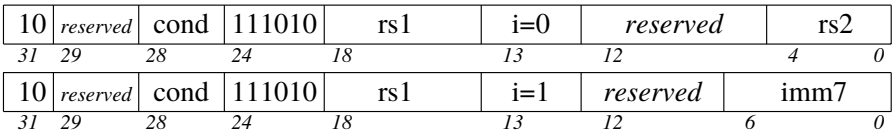
Traps:

illegal_instruction
privileged_instruction
privileged_instruction (may cause processor to enter error_mode)
mem_address_not_aligned (may cause processor to enter error_mode)
window_underflow (may cause processor to enter error_mode)

B.27. Trap on Integer Condition Codes 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`	*software_trap#*	
`tn`	*software_trap#*	
`tne`	*software_trap#*	*(synonym:* `tnz`*)*
`te`	*software_trap#*	*(synonym:* `tz`*)*
`tg`	*software_trap#*	
`tle`	*software_trap#*	
`tge`	*software_trap#*	
`tl`	*software_trap#*	
`tgu`	*software_trap#*	
`tleu`	*software_trap#*	
`tcc`	*software_trap#*	*(synonym:* `tgeu`*)*
`tcs`	*software_trap#*	*(synonym:* `tlu`*)*
`tpos`	*software_trap#*	
`tneg`	*software_trap#*	
`tvc`	*software_trap#*	
`tvs`	*software_trap#*	

Description:

A Ticc instruction evaluates the integer condition codes (*icc*) according to the *cond* field of the instruction, producing either a "true" or "false" result. If "true" and no higher priority exceptions or interrupt requests are pending, then a trap_instruction trap is generated. If "false", a trap_instruction trap does not occur and the instruction behaves like a NOP.

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 "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or 128 plus the least significant seven bits of "r[*rs1*] + sign_ext(*software_trap#*)" if the *i* field is one.

After a taken Ticc, the processor enters supervisor mode, disables traps, decrements the CWP (modulo NWINDOWS), and saves PC and nPC into r[17] and r[18] (*local* registers 1 and 2) of the new window. See Chapter 7, "Traps."

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

Traps:

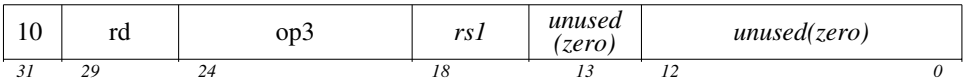
trap_instruction

B.28. Read State Register Instructions

opcode	*op3*	*rs1*	*operation*
RDY	101000	0	Read Y Register
RDASR‡	101000	1 – 15	Read Ancillary State Register (*reserved*)
RDASR‡	101000	16 – 31	(*implementation-dependent*)
RDPSR†	101001	reserved	Read Processor State Register
RDWIM†	101010	reserved	Read Window Invalid Mask Register
RDTBR†	101011	reserved	Read Trap Base Register

† privileged instruction
‡ privileged instruction if source register is privileged

Format (3):



Suggested Assembly Language Syntax
`rd %y,` reg_{rd}
`rd` asr_reg_{rs1} `,` reg_{rd}
`rd %psr,` reg_{rd}
`rd %wim,` reg_{rd}
`rd %tbr,` reg_{rd}

Description:

These instructions read the specified IU state register into r[*rd*].

Note that RDY is distinguished from RDASR only by the *rs1* field. The *rs1* field must be zero and *op3* = 0x28 to read the Y register.

If *rs1* ≠ 0 and *op3* = 0x28, then an implementation-dependent ancillary state register is read. Values of *rs1* in the range 1...14 are reserved for future versions of the architecture; values 16...31 are available for implementations to use. An RDASR instruction with *rs1* = 15 and *rd* = 0 is defined to be an STBAR instruction (see Section B.30 for its description). RDASR with *rs1* = 15 and *rd* ≠ 0 is reserved for future versions of the architecture.

An *rs1* value of 1...14 in an RDASR instruction produces undefined results, but does not cause an illegal_instruction trap.

For an RDASR instruction with *rs1* in the range 16...31, the following are implementation-dependent: the interpretation of bits 13:0 and 29:25 in the instruction, whether the instruction is privileged or not, and whether the instruction causes an illegal_instruction trap or not.

Implementation Note — Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers. See Appendix L, "Implementation Characteristics," for information on implemented ancillary state registers.

Traps:

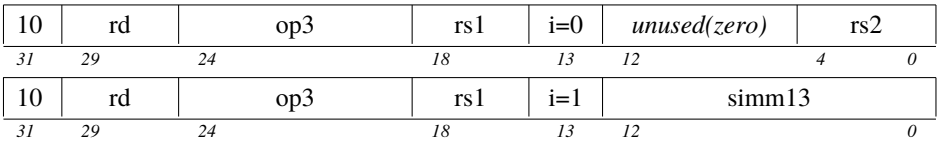
privileged_instruction (except RDY)
illegal_instruction (RDASR only; implementation-dependent)

B.29. Write State Register Instructions

opcode	*op3*	*rd*	*operation*
WRY	110000	0	Write Y Register
WRASR‡	110000	1 – 15	Write Ancillary State Register (*reserved*)
WRASR‡	110000	16 – 31	(*implementation-dependent*)
WRPSR†	110001	reserved	Write Processor State Register
WRWIM†	110010	reserved	Write Window Invalid Mask Register
WRTBR†	110011	reserved	Write Trap Base Register

† privileged instruction
‡ privileged instruction if destination register is privileged

Format (3):



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

Description:

WRY, WRPSR, WRWIM, and WRTBR write "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 writable fields of the specified IU state register. (Note the exclusive-or operation.)

Note that WRY is distinguished from WRASR only by the *rd* field. The *rd* field must be zero and *op3* = 0x30 to write the Y register.

WRASR writes a value to the ancillary state register (ASR) indicated by *rd*. The operation performed to generate the value written may be *rd*-dependent or implementation-dependent (see below). A WRASR instruction is indicated by $rd \neq 0$ and *op3* = 0x30.

WRASR instructions with *rd* in the range 1...15 are reserved for future versions of the architecture; executing a WRASR instruction with *rd* in that range produces undefined results.

WRASR instructions with *rd* in the range 16...31 are available for implementation-dependent uses. For a WRASR instruction with *rd* in the range 16...31, the following are implementation-dependent: the interpretation of bits 18:0 in the instruction, the operation(s) performed (for example,

xor) to generate the value written to the ASR, whether the instruction is privileged or not, and whether the instruction causes an illegal_instruction trap or not. In some existing implementations, WRASR instructions may write the Y register (see Appendix L, "Implementation Characteristics"). WRASR in new implementations must not write the Y register.

If the result of a WRPSR instruction would cause the CWP field of the PSR to point to an unimplemented window, it causes an illegal_instruction trap and does not write the PSR.

The write state register instructions are **delayed-write** instructions. That is, they may take until completion of the third instruction following the write instruction to consummate their write operation. The number of delay instructions (0 to 3) is implementation-dependent.

WRPSR appears to write the ET and PIL fields immediately with respect to interrupts.

The following paragraphs define the relationship between the writing of a field of a state register and that field's being simultaneously or subsequently accessed:

1. If any of the three instructions after a write state register instruction **writes** any field of the same state register, the subsequent contents of that field are undefined. The exception to this is that another instance of the **same** write state register instruction (e.g. a WRPSR following within three instructions of another WRPSR) will write the field as intended.

   Programming Note
   Many instructions implicitly write the CWP or *icc* fields of the PSR. For example, SAVE, RESTORE, traps, and RETT write CWP, and many instructions write *icc*.

2. If any of the three instructions after a write state register instruction **reads** any field that was **changed** by the original write state register instruction, the contents of that field read by that instruction are undefined.

   Programming Note
   Many instructions implicitly read CWP or *icc*. For example, CALL implicitly reads CWP; instructions that reference an integer non-global (windowed) register implicitly read CWP; SAVE, RESTORE, RETT, and traps (including Ticc) read CWP, and Bicc and Ticc read the *icc* field.

   Programming Note
   SAVE, RESTORE and RETT implicitly read WIM. If any of them executes within three instructions after a WRWIM which changes the contents of the WIM, the occurrence of window_overflow and window_underflow traps is unpredictable.

   Programming Note
   MULScc, RDY, SDIV, SDIVcc, UDIV, and UDIVcc implicitly read the Y register. If any of these instructions execute within three instructions after a WRY which changed the contents of the Y register, its results are undefined.

3. In some implementations, if a WRPSR instruction updates the PSR's PIL field to a new value and simultaneously sets ET to 1, an interrupt trap at a level equal to the old value of the PIL may result.

Programming Note

A pair of 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.

Programming Note

If traps are enabled (ET=1), care must be taken if software is to disable them (ET=0). Since the "RDPSR, WRPSR" sequence is interruptible — allowing the PSR to be changed between the two instructions — this sequence is not a reliable mechanism to disable traps. Two alternatives are:

1) Generate a Ticc trap, the handler for which disables traps. The trap handler should verify that it was indeed "called" from supervisor mode (by examining the PS bit of the PSR) before returning from the trap to the supervisor.
2) Use the "RDPSR, WRPSR" sequence, but write all the interrupt and 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.

4. If any of the three instructions that follow a WRPSR causes a trap, the values of the S and CWP fields read from the PSR while taking the trap may be either the old or the new values.
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 any write state register instruction causes a trap, a subsequent read state register instruction in the trap handler will get the state register's new value.

Implementation Note — Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trap-control registers. See Appendix L, "Implementation Characteristics," for information on implemented ancillary state registers.

Implementation Note — Two possible ways to cause WRPSR to appear to write ET and PIL immediately with respect to interrupts are:

- Write ET and PIL immediately (propagating forward through the pipeline as needed)
- Disable interrupts during the subsequent three instructions

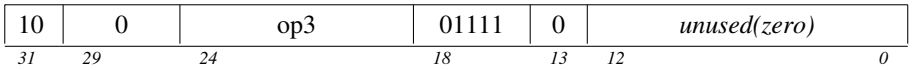
Traps:

privileged_instruction (except WRY)
illegal_instruction (WRPSR, if CWP ≥ NWINDOWS)
illegal_instruction (WRASR; implementation-dependent)

B.30. STBAR Instruction

opcode	*op3*	*operation*
STBAR	101000	Store Barrier

Format (3):



Suggested Assembly Language Syntax
`stbar`

Description:

The store barrier instruction (STBAR) forces **all** store and atomic load-store operations issued by the processor prior to the STBAR to complete before **any** store or atomic load-store operations issued by the processor subsequent to the STBAR are executed by memory.

STBAR executes as a no-op on a machine that implements only the Strong Consistency memory model or the Total Store Ordering (TSO) memory model, and on a machine that implements the Partial Store Ordering (PSO) memory model but is running with the PSO mode disabled.

Note that the encoding of STBAR is identical to that of the RDASR instruction, except that *rs1* = 15 and *rd* = 0.

Implementation Note

For correctness, it is sufficient for the processor to stop issuing new store and atomic load-store operations when an STBAR is encountered and resume after all stores have completed and are observed in memory by all processors. More efficient implementations may take advantage of the fact that the processor is allowed to issue store and load-store operations after the STBAR, as long as these operations are guaranteed not to be executed by memory before all the earlier stores and atomic load-stores have been executed by memory.

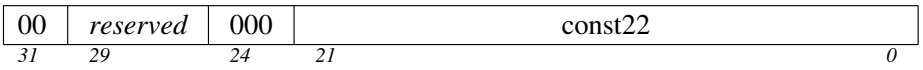
Traps:

(none)

B.31. 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 by the hardware; specifically, its values are **not** reserved by the architecture for any future use.

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 `struct` or `union` or Pascal `record`. See Appendix D, "Software Considerations," for an example.

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

b) If the called 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 called function 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.

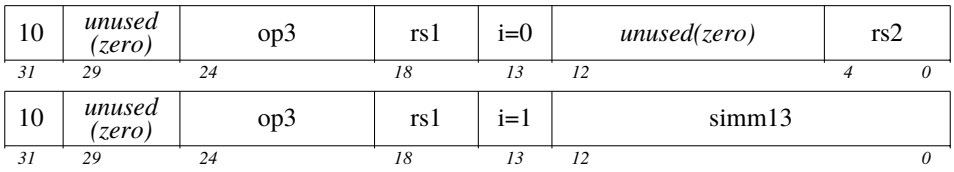
Traps:

illegal_instruction

B.32. Flush Instruction Memory

opcode	*op3*	*operation*
FLUSH	111011	Flush Instruction Memory

Format (3):



Suggested Assembly Language Syntax
`flush` *address*

Description:

The FLUSH instruction ensures that subsequent instruction fetches to the target of the FLUSH by the processor executing the FLUSH appear to execute after any loads, stores, and atomic load-stores issued by that processor prior to the FLUSH.

In a multiprocessor system, FLUSH also ensures that stores and atomic load-stores to the target of the FLUSH, issued prior to the FLUSH by the processor executing the FLUSH, become visible to the instruction fetches of all other processors some time after the execution of the FLUSH.

When a processor executes a sequence of store or atomic load-stores interspersed with appropriate FLUSH and STBAR instructions, (the latter needed only for the PSO memory model), the changes appear to the instruction fetches of all processors to occur in the order in which they were made. See Chapter 6, “Memory Model,” and Appendix K, “Formal Specification of the Memory Model” for a definition of what constitutes appropriate FLUSH and STBAR instructions in such a sequence.

FLUSH operates on the doubleword containing the addressed location.

A FLUSH is needed only between a store and a subsequent **instruction access** to the modified location. The memory model guarantees that **data** loads observe the results of the most recent store even if there is no intervening FLUSH. See Chapter 6, “Memory Model.”

The effective virtual address operand for the FLUSH instruction is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one. The least significant two address bits of the result are unused and should be supplied as zero by software. Bit 2 of the address is ignored.

By the time five instructions subsequent to a FLUSH have executed, any internal copy of the addressed location in the issuing processor will contain

the same value as the one which would be seen if read from memory. For example, the processor pipeline or instruction buffers might contain an internal copy of the addressed location. See IBuf in Chapter 6, "Memory Model." FLUSH does **not** necessarily affect such internal copies in other processors attached to the memory system.

Programming Notes

(1) FLUSH is typically used in self-modifying code.

(2) Although FLUSH provides support for self-modifying code, the use of self-modifying code is not encouraged. FLUSH may be a time-consuming operation on some implementations.

Implementation Notes

(1) FLUSH may operate on more than just the doubleword implied by the effective address. In particular, it may flush one or more containing cache lines or blocks.

(2) In a uniprocessor system with a combined I and D cache (or no cache) and a total pipeline (store buffer plus IBuf) depth of no more than five instructions, FLUSH may not need to perform any operation.

In a uniprocessor system with split I and D caches, FLUSH ensures that if both caches contain a copy of the contents of the addressed location, those cached copies eventually become consistent.

In a multiprocessor system with caches, FLUSH ensures that all cached copies of the contents of the addressed location are consistent.

Cache consistency may be implemented by any combination of invalidation, write-back of cached data, or other implementation-dependent consistency mechanisms.

(3) If FLUSH is not implemented in hardware as described above, FLUSH causes an unimplemented_FLUSH (or illegal_instruction) trap, and the function of FLUSH is performed by system software. Whether FLUSH traps or not is implementation-dependent. If it does trap, it causes an unimplemented_FLUSH or illegal_instruction trap. On implementations where unimplemented_FLUSH supports faster software emulation of FLUSH than does illegal_instruction, use of unimplemented_FLUSH is preferred. An implementation may select the trapping behavior of the FLUSH instruction based on a pin or an implementation-dependent bit in a control register.

(4) In a given implementation, FLUSH may need to flush the processor's IBuf and/or pipeline to fulfill the requirement that the IBuf and pipeline will be consistent with the cache within five instructions.

(5) The number of instructions which must execute after a FLUSH before its effect is complete is implementation-dependent, but is at most 5.

(6) See Appendix L, "Implementation Characteristics," for implementation-specific information about the FLUSH instruction.

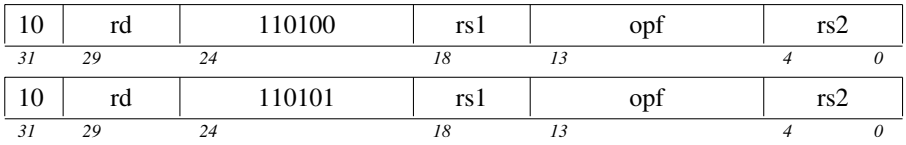
Traps:

unimplemented_FLUSH (*implementation-dependent*) illegal_instruction (*implementation-dependent*)

B.33. Floating-point Operate (FPop) Instructions

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

Format (3):



Description:

The Floating-point Operate (FPop) instructions are encoded using two type 3 formats: FPop1 and FPop2. The particular floating-point operation is indicated by *opf* field. Note that the load/store floating-point instructions are not FPop instructions.

FPop1 instructions do not affect the floating-point condition codes. FPop2 instructions may affect the floating-point condition codes.

The FPop instructions support operations between integer words and single-, double-, and quad-precision floating-point operands in *f* register(s).

All FPop instructions operate according to ANSI/IEEE Std. 754-1985 on single, double, and quad formats. See Chapter 3, "Data Formats," for definitions of the floating-point data types.

The least significant bit of an *f* register address is unused by double-precision FPops, and the least significant 2 bits of an *f* register address are unused by quad-precision FPop instructions. The unused register address bits are reserved and, for future compatibility, should be supplied as zeros by software. If these bits are non-zero in an FPop with a double- or quad-precision operand, it is recommended that the FPop cause an fp_exception trap with FSR.*ftt* = invalid_fp_register.

If an FPop2 (for example, FCMP, FCMPE) instruction sets the floating-point condition codes, then at least one non-FPop2 (non-floating-point-operate2) instruction must be executed between the FPop2 and a subsequent FBfcc instruction. Otherwise, the result of the FBfcc is undefined.

An FPop instruction causes an fp_disabled trap if either the EF field of the PSR is 0 or no FPU is present.

Floating-point exceptions may cause either precise or deferred traps. See Chapter 7, "Traps."

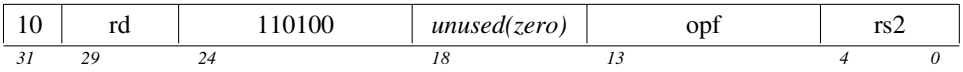
Programming Note — See Appendix G, "SPARC ABI Software Considerations," regarding use of FSQRT, FsMULd, and quad-precision floating-point instructions in SPARC ABI software.

Implementation Note — See Appendix L, "Implementation Characteristics," for information on whether FsMULd, FdMULq, and the quad-precision instructions are executed in hardware or software in the various SPARC implementations.

Convert Integer to Floating point Instructions

opcode	*opf*	*operation*
FiTOs	011000100	Convert Integer to Single
FiTOd	011001000	Convert Integer to Double
FiTOq	011001100	Convert Integer to Quad

Format (3):



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

Description:

These instructions convert the 32-bit integer word operand in f[*rs2*] into a floating-point number in the destination format. They write the result into the *f* register(s) specified by *rd*.

FiTOs rounds according to the RD field of the FSR.

Programming Note — See Appendix G, “SPARC ABI Software Considerations,” regarding use of the FiTOq instruction in SPARC ABI software.

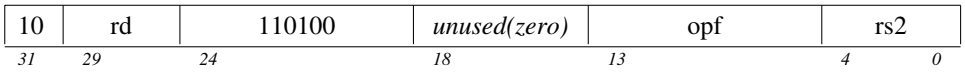
Traps:

fp_disabled
fp_exception (NX (FiTOs only), invalid_fp_register(FiTOd, FiTOq))

Convert Floating point to Integer Instructions

opcode	*opf*	*operation*
FsTOi	011010001	Convert Single to Integer
FdTOi	011010010	Convert Double to Integer
FqTOi	011010011	Convert Quad to Integer

Format (3):



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

Description:

These instructions convert the floating-point operand in the *f* register(s) specified by *rs2* into a 32-bit integer word in f[*rd*].

The result is always rounded toward zero (the RD field of the FSR register is ignored).

Programming Note — See Appendix G, "SPARC ABI Software Considerations," regarding use of the FqTOi instruction in SPARC ABI software.

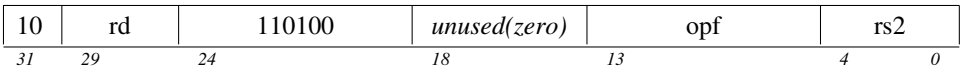
Traps:

fp_disabled
fp_exception (NV, NX, invalid_fp_register(FdTOi, FqTOi))

Convert Between Floating-point Formats Instructions

opcode	*opf*	*operation*
FsTOd	011001001	Convert Single to Double
FsTOq	011001101	Convert Single to Quad
FdTOs	011000110	Convert Double to Single
FdTOq	011001110	Convert Double to Quad
FqTOs	011000111	Convert Quad to Single
FqTOd	011001011	Convert Quad to Double

Format (3):



Suggested Assembly Language Syntax	
`fstod`	$freg_{rs2}$, $freg_{rd}$
`fstoq`	$freg_{rs2}$, $freg_{rd}$
`fdtos`	$freg_{rs2}$, $freg_{rd}$
`fdtoq`	$freg_{rs2}$, $freg_{rd}$
`fqtos`	$freg_{rs2}$, $freg_{rd}$
`fqtod`	$freg_{rs2}$, $freg_{rd}$

Description:

These instructions convert the floating-point operand in the *f* register(s) specified by *rs2* to a floating-point number in the destination format. They write the result into the *f* register(s) specified by *rd*.

Rounding is performed according to the RD field of the FSR.

FqTOd, FqTOs, and FdTOs (the "narrowing" conversion instructions) can raise OF, UF, and NX exceptions. FdTOq, FsTOq, and FsTOd (the "widening" conversion instructions) cannot.

Any of these six instructions can trigger an NV exception if the source operand is a signaling NaN.

Programming Note — See Appendix G, "SPARC ABI Software Considerations," regarding use of the FsTOq, FdTOq, FqTOs, and FqTOd instructions in SPARC ABI software.

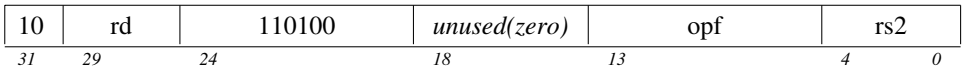
Traps:

fp_disabled
fp_exception (OF, UF, NV, NX, invalid_fp_register)

Floating-point Move Instructions

opcode	*opf*	*operation*
FMOVs	000000001	Move
FNEGs	000000101	Negate
FABSs	000001009	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 copies the contents of f[*rs2*] to f[*rd*].

FNEGs copies the contents of f[*rs2*] to f[*rd*] with the sign bit complemented.

FABSs copies the contents of f[*rs2*] to f[*rd*] with the sign bit cleared.

These instructions do not round.

Programming Note — To transfer a multiple-precision value between *f* registers, one FMOVs instruction is required per word to be transferred.

Programming Note — If the source and destination registers ($freg_{rs2}$ and $freg_{rd}$) are the same, a single FNEGs (FABSs) instruction performs negation (absolute-value) for any operand precision, including double and quad.

If the source and destination registers are different, a double-precision negation (absolute value) is performed by an FNEGs (FABSs) and an FMOVs instruction. Similarly, a quad-precision negation (absolute value) requires an FNEGs (FABSs) and three FMOVs instructions.

See Section 3, “Data Formats,” for the formats of the floating-point data types.

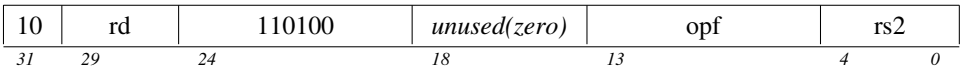
Traps:

fp_disabled

Floating-point Square Root Instructions

opcode	*opf*	*operation*
FSQRTs	000101001	Square Root Single
FSQRTd	000101010	Square Root Double
FSQRTq	000101011	Square Root Quad

Format (3):



Suggested Assembly Language Syntax	
`fsqrts`	*freg*$_{rs2}$, *freg*$_{rd}$
`fsqrtd`	*freg*$_{rs2}$, *freg*$_{rd}$
`fsqrtq`	*freg*$_{rs2}$, *freg*$_{rd}$

Description:

These instructions generate the square root of the floating-point operand in the *f* register(s) specified by the *rs2* field. They place the result in the destination *f* register(s) specified by the *rd* field.

Rounding is performed according to the *rd* field of the FSR.

Programming Note — See Appendix G, "SPARC ABI Software Considerations," regarding use of FSQRT instructions in SPARC ABI software.

Implementation Note — See Appendix L, "Implementation Characteristics," for information on whether the FSQRT instructions are executed in hardware or in software in the various SPARC implementations.

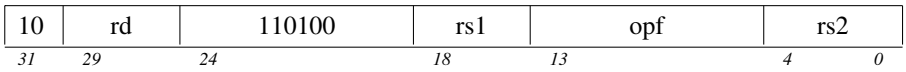
Traps:

fp_disabled
fp_exception (NV, NX, invalid_fp_register(FSQRTd, FSQRTq))

Floating-point Add and Subtract Instructions

opcode	*opf*	*operation*
FADDs	001000001	Add Single
FADDd	001000010	Add Double
FADDq	001000011	Add Quad
FSUBs	001000101	Subtract Single
FSUBd	001000110	Subtract Double
FSUBq	001000111	Subtract Quad

Format (3):



Suggested Assembly Language Syntax	
`fadds`	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
`faddd`	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
`faddq`	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
`fsubs`	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
`fsubd`	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
`fsubq`	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description:

The floating-point add instructions add the *f* register(s) specified by the *rs1* field and the *f* register(s) specified by the *rs2* field, and write the sum into the *f* register(s) specified by the *rd* field.

The floating-point subtract instructions subtract the *f* register(s) specified by the *rs2* field from the *f* register(s) specified by the *rs1* field, and write the difference into the *f* register(s) specified by the *rd* field.

Programming Note — See Appendix G, “SPARC ABI Software Considerations,” regarding use of the FADDq and FSUBq instructions in SPARC ABI software.

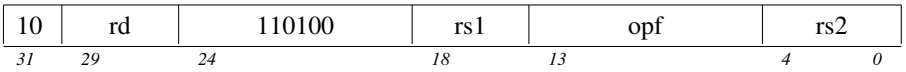
Traps:

fp_disabled
fp_exception (OF, UF, NX, NV ($\infty - \infty$),
invalid_fp_register(all except FADDs and FSUBs))

Floating-point Multiply and Divide Instructions

opcode	*opf*	*operation*
FMULs	001001001	Multiply Single
FMULd	001001010	Multiply Double
FMULq	001001011	Multiply Quad
FsMULd	001101001	Multiply Single to Double
FdMULq	001101110	Multiply Double to Quad
FDIVs	001001101	Divide Single
FDIVd	001001110	Divide Double
FDIVq	001001111	Divide Quad

Format (3):



Suggested Assembly Language Syntax	
`fmuls`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fmuld`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fmulq`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fsmuld`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fdmulq`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fdivs`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fdivd`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
`fdivq`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description:

The floating-point multiply instructions multiply the *f* register(s) specified by the *rs1* field by the *f* register(s) specified by the *rs2* field, and write the product into the *f* register(s) specified by the *rd* field.

The FsMULd instruction provides the exact double-precision product of two single-precision operands, without underflow, overflow, or rounding error. Similarly, FdMULq provides the exact quad-precision product of two double-precision operands.

The floating-point divide instructions divide the *f* register(s) specified by the *rs1* field by the *f* register(s) specified by the *rs2* field, and write the quotient into the *f* register(s) specified by the *rd* field.

Programming Note — See Appendix G, "SPARC ABI Software Considerations," regarding use of the FMULq, FDIVq, FsMULd, and FdMULq instructions in SPARC ABI software.

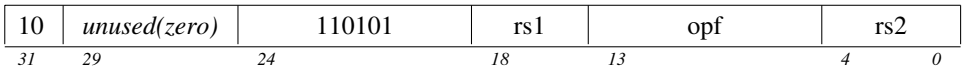
Traps:

fp_disabled
fp_exception (OF, UF, DZ (FDIV only), NV, NX,
invalid_fp_register(all except FMULs and FDIVs))

Floating-point Compare Instructions

opcode	*opf*	*operation*
FCMPs	001010001	Compare Single
FCMPd	001010010	Compare Double
FCMPq	001010011	Compare Quad
FCMPEs	001010101	Compare Single and Exception if Unordered
FCMPEd	001010110	Compare Double and Exception if Unordered
FCMPEq	001010111	Compare Quad and Exception if Unordered

Format (3):



Suggested Assembly Language Syntax	
`fcmps`	$freg_{rs1}$, $freg_{rs2}$
`fcmpd`	$freg_{rs1}$, $freg_{rs2}$
`fcmpq`	$freg_{rs1}$, $freg_{rs2}$
`fcmpes`	$freg_{rs1}$, $freg_{rs2}$
`fcmped`	$freg_{rs1}$, $freg_{rs2}$
`fcmpeq`	$freg_{rs1}$, $freg_{rs2}$

Description:

These instructions compare the *f* register(s) specified by the *rs1* field with the *f* register(s) specified by the *rs2* field, and set the floating-point condition codes according to the following table:

Table B-2 *Floating-point Condition Codes (fcc)*

fcc	Relation
0	$freg_{rs1}$ = $freg_{rs2}$
1	$freg_{rs1}$ < $freg_{rs2}$
2	$freg_{rs1}$ > $freg_{rs2}$
3	$freg_{rs1}$? $freg_{rs2}$ (unordered)

The “compare and cause exception if unordered” (FCMPEs, FCMPEd, and FCMPEq) instructions cause an invalid (NV) exception if either operand is a signaling NaN or a quiet NaN. FCMP causes an invalid (NV) exception if either operand is a signaling NaN.

A non-FPop2 (non-floating-point-operate2) instruction must be executed between an FPop2 (FCMP or FCMPE) instruction and a subsequent FBfcc instruction. Otherwise, the result of the FBfcc is unpredictable.

Programming Note See Appendix G, “SPARC ABI Software Considerations,” regarding use of the FCMPq and FCMPEq instructions in SPARC ABI software.

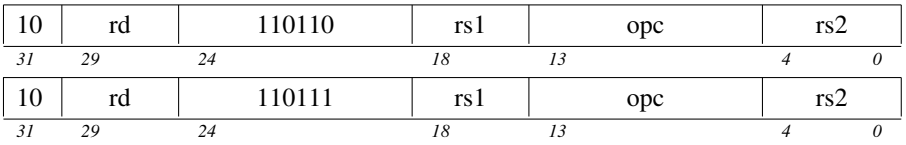
Traps:

fp_disabled
fp_exception (NV, invalid_fp_register(all except FCMPs and FCMPEs))

B.34. Coprocessor Operate Instructions

opcode	*op3*	*operation*
CPop1	110110	Coprocessor Operate
CPop2	110111	Coprocessor Operate

Format (3):



Suggested Assembly Language Syntax	
`cpop1`	*opc* , $creg_{rs1}$, $creg_{rs2}$, $creg_{rd}$
`cpop2`	*opc* , $creg_{rs1}$, $creg_{rs2}$, $creg_{rd}$

Note — The above is a suggested "generic" assembly language syntax for these instructions, which may be used in an implementation-independent SPARC assembler. It is expected that assemblers supporting specific coprocessor implementations will (also) support syntaxes with more mnemonic instruction names and fewer operands.

Description:

The Coprocessor Operate (CPop) instructions are encoded via two type 3 formats: CPop1 and CPop2. Interpretation of the *rd*, *rs1*, *opc*, and *rs2* fields is coprocessor-dependent. Note that the load/store coprocessor instructions are not "CPop" instructions.

CPop1 instructions do not affect the coprocessor condition codes. CPop2 instructions may affect the coprocessor condition codes.

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

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

The conditions under which execution of a CPop instruction causes a cp_exception trap are coprocessor-dependent.

Implementation Note — Typically, the particular coprocessor operation is indicated by the *opc* field.

Traps:

cp_disabled
cp_exception (*coprocessor-dependent*)