6 Instructions

Instructions are accessed by the processor from memory and are executed, annulled, or trapped. Instructions are encoded in four major formats and partitioned into eleven general categories.

6.1 Instruction Execution

The instruction at the memory location specified by the program counter is fetched and then executed. Instruction execution may change program-visible processor and/or memory state. As a side-effect of its execution, new values are assigned to the program counter (PC) and the next program counter (nPC).

An instruction may generate an exception if it encounters some condition that makes it impossible to complete normal execution. Such an exception may in turn generate a precise trap. Other events may also cause traps: an exception caused by a previous instruction (a deferred trap), an interrupt or asynchronous error (a disrupting trap), or a reset request (a reset trap). If a trap occurs, control is vectored into a trap table. See Chapter 7, "Traps," for a detailed description of exception and trap processing.

If a trap does not occur and the instruction is not a control transfer, the next program counter (nPC) is copied into the PC and the nPC is incremented by 4 (ignoring overflow, if any). If the instruction is a control-transfer instruction, the next program counter (nPC) is copied into the PC and the target address is written to nPC. Thus, the two program counters provide for a delayed-branch execution model.

For each instruction access and each normal data access, the IU appends an 8-bit address space identifier, or ASI, to the 64-bit memory address. Load/store alternate instructions (see 6.3.1.3, "Address Space Identifiers (ASIs),") can provide an arbitrary ASI with their data addresses, or use the ASI value currently contained in the ASI register.

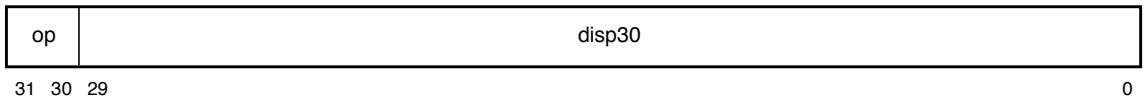
Implementation Note:

The time required to execute an instruction is implementation-dependent, as is the degree of execution concurrency. In the absence of traps, an implementation should cause the same program-visible register and memory state changes as if a program had executed according to the sequential model implied in this document. See Chapter 7, "Traps," for a definition of architectural compliance in the presence of traps.

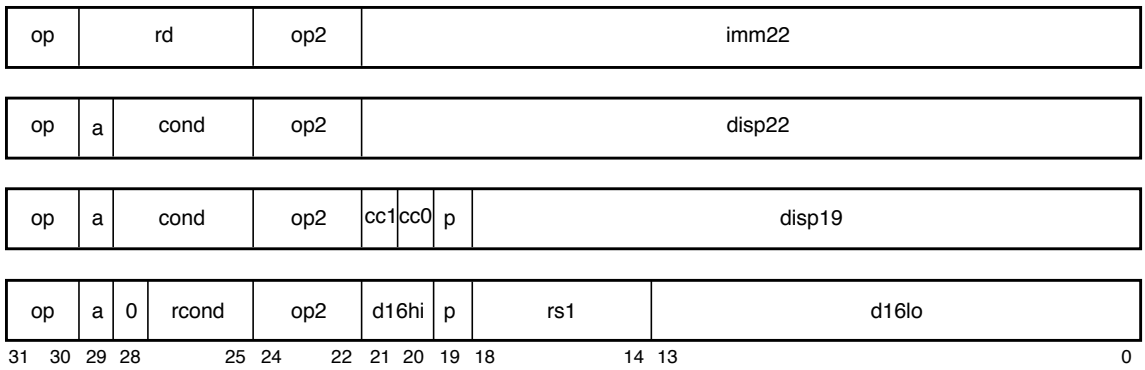
6.2 Instruction Formats

Instructions are encoded in four major 32-bit formats and several minor formats, as shown in figures 33 and 34.

Format 1 (*op* = 1): CALL



Format 2 (*op* = 0): SETHI & Branches (Bicc, BPcc, BPr, FBfcc, FBPfcc)



Format 3 (*op* = 2 or 3): Arithmetic, Logical, MOVr, MEMBAR, Load, and Store

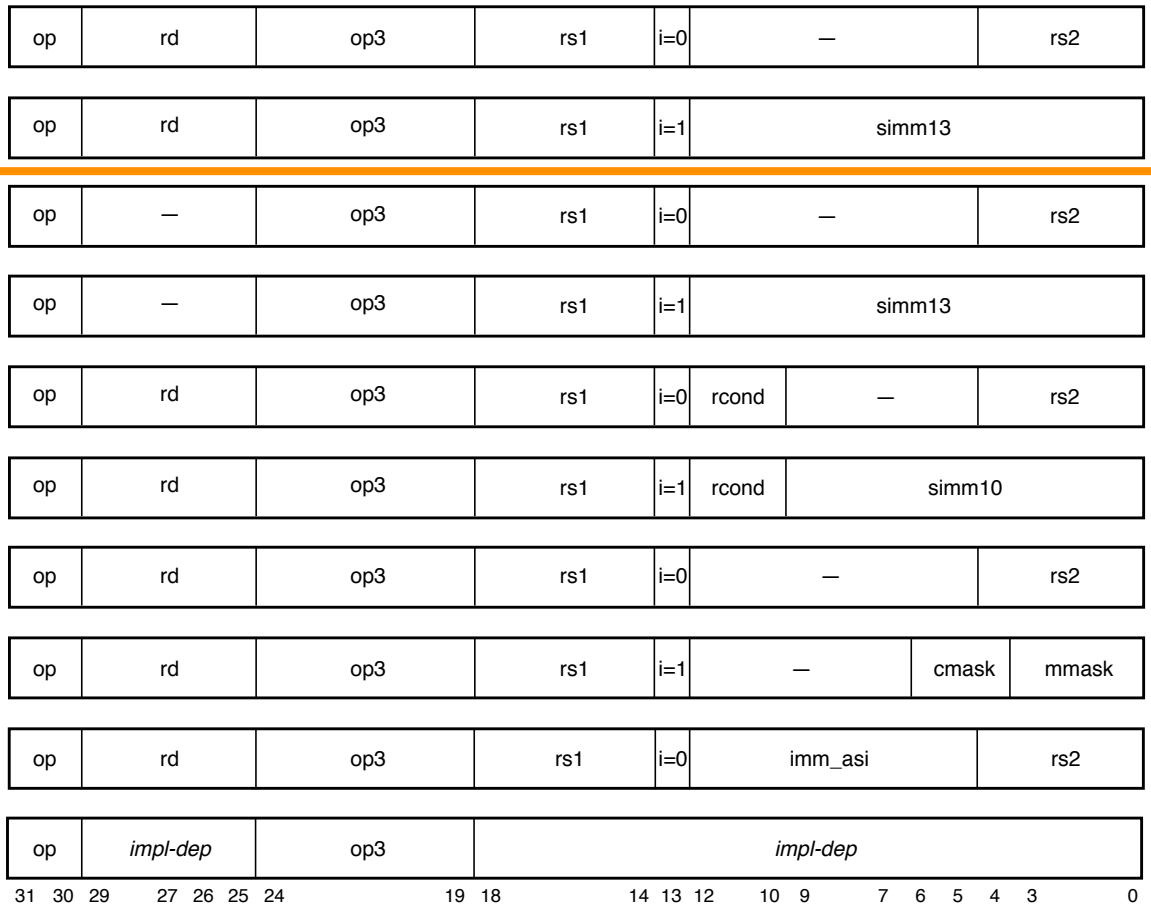
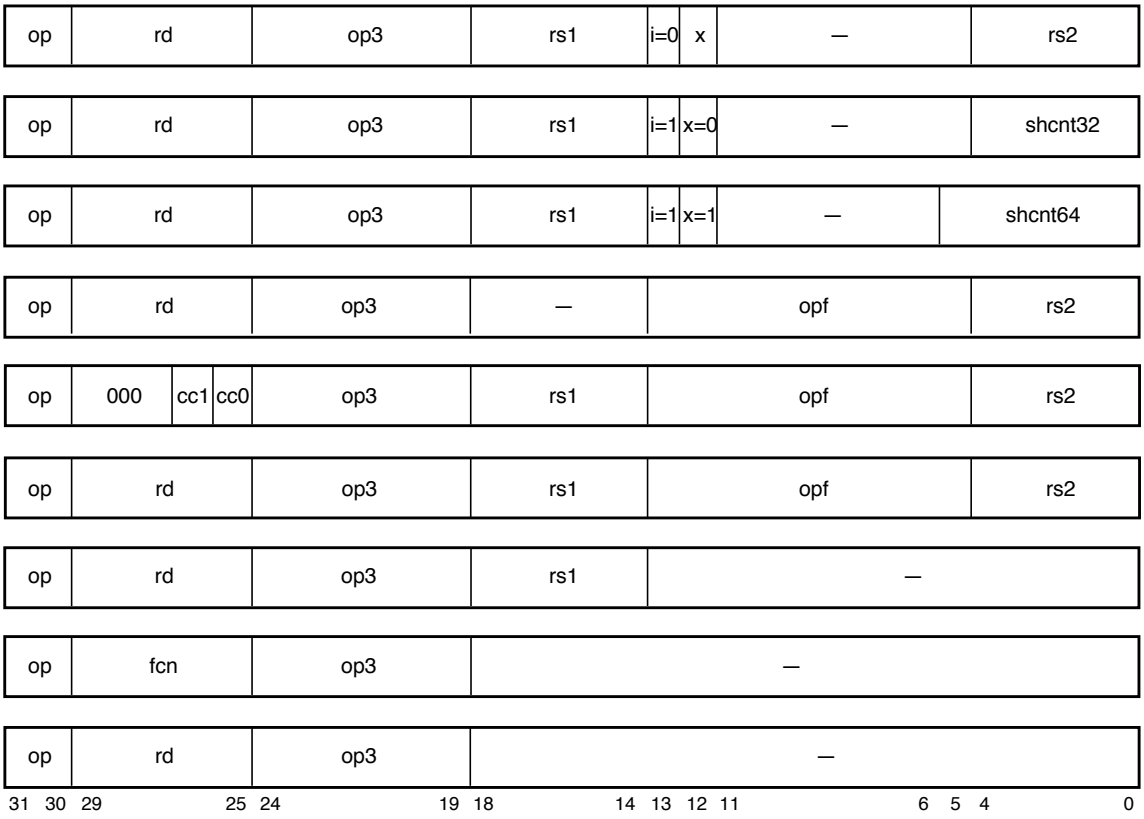
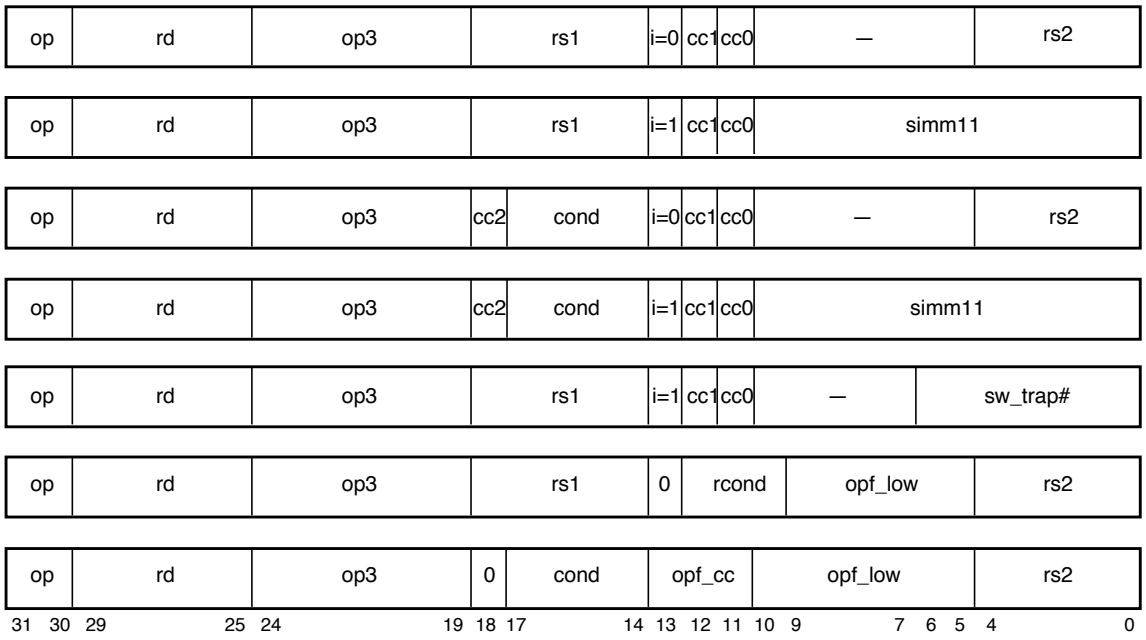


Figure 33—Summary of Instruction Formats: Formats 1, 2, and 3

Format 3 (op = 2 or 3): *Continued*



Format 4 (op = 2): MOVcc, FMOVr, FMOVcc, and Tcc



E Opcode Maps

E.1 Overview

This appendix contains the SPARC-V9 instruction opcode maps.

Opcodes marked with a dash '—' are reserved; an attempt to execute a reserved opcode shall cause a trap, unless it is an implementation-specific extension to the instruction set. See 6.3.11, "Reserved Opcodes and Instruction Fields," for more information.

In this appendix and in Appendix A, "Instruction Definitions," certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in table 21 on page 133. For deprecated opcodes, see the appropriate instruction pages in Appendix A, "Instruction Definitions," for preferred substitute instructions.

E.2 Tables

Table 30—*op*[1:0]

op [1:0]			
0	**1**	**2**	**3**
Branches & SETHI *See table 31*	CALL	Arithmetic & Misc. *See table 32*	Loads/Stores *See table 33*

Table 31—*op2*[2:0] (*op* = 0)

op2 [2:0]							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ILLTRAP	BPcc *See table 36*	Bicc^{D} *See table 36*	BPr *See table 37*	SETHI NOP†	FBPfcc *See table 36*	FBfcc^{D} *See table 36*	—

†*rd* = 0, *imm22* = 0

Table 32—*op3*[5:0] (*op* = 2)

		op3 [5:4]			
		0	**1**	**2**	**3**
op3 [3:0]	**0**	ADD	ADDcc	TADDcc	WRY[D] (*rd* = 0) — (*rd* = 1) WRCCR (*rd* = 2) WRASI (*rd* = 3) WRASR[PASR] (*see* A.63) WRFPRS (*rd* = 6) SIR (*rd* = 15, *rs1* = 0, *i* = 1)
	1	AND	ANDcc	TSUBcc	SAVED[P] (*fcn* = 0), RESTORED[P] (*fcn* = 1)
	2	OR	ORcc	TADDccTV[D]	WRPR[P]
	3	XOR	XORcc	TSUBccTV[D]	—
	4	SUB	SUBcc	MULScc[D]	FPop1 *See table 34*
	5	ANDN	ANDNcc	SLL (*x* = 0), SLLX (*x* = 1)	FPop2 *See table 35*
	6	ORN	ORNcc	SRL (*x* = 0), SRLX (*x* = 1)	IMPDEP1
	7	XNOR	XNORcc	SRA (*x* = 0), SRAX (*x* = 1)	IMPDEP2
	8	ADDC	ADDCcc	RDY[D] (*rs1* = 0) — (*rs1* = 1) RDCCR (*rs1* = 2) RDASI (*rs1* = 3) RDTICK[PNPT] (*rs1* = 4) RDPC (*rs1* = 5) RDFPRS (*rs1* = 6) RDASR[PASR] (*see* A.44) MEMBAR (*rs1* = 15, *rd* = 0, *i* = 1) STBAR[D] (*rs1* = 15, *rd* = 0, *i* = 0)	JMPL
	9	MULX	—	—	RETURN
	A	UMUL[D]	UMULcc[D]	RDPR[P]	Tcc *See table 36*
	B	SMUL[D]	SMULcc[D]	FLUSHW	FLUSH
	C	SUBC	SUBCcc	MOVcc	SAVE
	D	UDIVX	—	SDIVX	RESTORE
	E	UDIV[D]	UDIVcc[D]	POPC (*rs1* = 0) — (*rs1* > 0)	DONE[P] (*fcn* = 0) RETRY[P] (*fcn* = 1)
	F	SDIV[D]	SDIVcc[D]	MOVr *See table 37*	—

Table 33—*op3*[5:0] (*op* = 3)

		op3 [5:4]			
		0	**1**	**2**	**3**
op3 [3:0]	**0**	LDUW	LDUWA$^{P_{ASI}}$	LDF	LDFA$^{P_{ASI}}$
	1	LDUB	LDUBA$^{P_{ASI}}$	LDFSR^{D}, LDXFSR	—
	2	LDUH	LDUHA$^{P_{ASI}}$	LDQF	LDQFA$^{P_{ASI}}$
	3	LDD^{D}	LDDA$^{D, P_{ASI}}$	LDDF	LDDFA$^{P_{ASI}}$
	4	STW	STWA$^{P_{ASI}}$	STF	STFA$^{P_{ASI}}$
	5	STB	STBA$^{P_{ASI}}$	STFSR^{D}, STXFSR	—
	6	STH	STHA$^{P_{ASI}}$	STQF	STQFA$^{P_{ASI}}$
	7	STD^{D}	STDA$^{P_{ASI}}$	STDF	STDFA$^{P_{ASI}}$
	8	LDSW	LDSWA$^{P_{ASI}}$	—	—
	9	LDSB	LDSBA$^{P_{ASI}}$	—	—
	A	LDSH	LDSHA$^{P_{ASI}}$	—	—
	B	LDX	LDXA$^{P_{ASI}}$	—	—
	C	—	—	—	CASA$^{P_{ASI}}$
	D	LDSTUB	LDSTUBA$^{P_{ASI}}$	PREFETCH	PREFETCHA$^{P_{ASI}}$
	E	STX	STXA$^{P_{ASI}}$	—	CASXA$^{P_{ASI}}$
	F	SWAP^{D}	SWAPA$^{D, P_{ASI}}$	—	—

Table 34—*opf*[8:0] (*op* = 2,*op3* = 34_{16} = FPop1)

	opf[3:0]															
opf [8:4]	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
00	—	FMOVs	FMOVd	FMOVq	—	FNEGs	FNEGd	FNEGq	—	FABSs	FABSd	FABSq	—	—	—	—
01	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
02	—	—	—	—	—	—	—	—	—	FSQRTs	FSQRTd	FSQRTq	—	—	—	—
03	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
04	—	FADDs	FADDd	FADDq	—	FSUBs	FSUBd	FSUBq	—	FMULs	FMULd	FMULq	—	FDIVs	FDIVd	FDIVq
05	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
06	—	—	—	—	—	—	—	—	—	FsMULd	—	—	—	—	FdMULq	—
07	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
08	—	FsTOx	FdTOx	FqTOx	FxTOs	—	—	—	FxTOd	—	—	—	FxTOq	—	—	—
09	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0A	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0B	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0C	—	—	—	—	FiTOs	—	FdTOs	FqTOs	FiTOd	FsTOd	—	FqTOd	FiTOq	FsTOq	FdTOq	—
0D	—	FsTOi	FdTOi	FqTOi	—	—	—	—	—	—	—	—	—	—	—	—
0E..1F	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 35—*opf*[8:0] (*op* = 2, *op3* = 35_{16} = FPop2)

	opf[3:0]								
opf [8:4]	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8..F**
00	—	FMOVs (fcc0)	FMOVd (fcc0)	FMOVq (fcc0)	—	†	†	†	—
01	—	—	—	—	—	—	—	—	—
02	—	—	—	—	—	FMOVRsZ	FMOVRdZ	FMOVRqZ	—
03	—	—	—	—	—	—	—	—	—
04	—	FMOVs (fcc1)	FMOVd (fcc1)	FMOVq (fcc1)	—	FMOVRsLEZ	FMOVRdLEZ	FMOVRqLEZ	—
05	—	FCMPs	FCMPd	FCMPq	—	FCMPEs	FCMPEd	FCMPEq	—
06	—	—	—	—	—	FMOVRsLZ	FMOVRdLZ	FMOVRqLZ	—
07	—	—	—	—	—	—	—	—	—
08	—	FMOVs (fcc2)	FMOVd (fcc2)	FMOVq (fcc2)	—	†	†	†	—
09	—	—	—	—	—	—	—	—	—
0A	—	—	—	—	—	FMOVRsNZ	FMOVRdNZ	FMOVRqNZ	—
0B	—	—	—	—	—	—	—	—	—
0C	—	FMOVs (fcc3)	FMOVd (fcc3)	FMOVq (fcc3)	—	FMOVRsGZ	FMOVRdGZ	FMOVRqGZ	—
0D	—	—	—	—	—	—	—	—	—
0E	—	—	—	—	—	FMOVRsGEZ	FMOVRdGEZ	FMOVRqGEZ	—
0F	—	—	—	—	—	—	—	—	—
10	—	FMOVs (icc)	FMOVd (icc)	FMOVq (icc)	—	—	—	—	—
11..17	—	—	—	—	—	—	—	—	—
18	—	FMOVs (xcc)	FMOVd (xcc)	FMOVq (xcc)	—	—	—	—	—
19..1F	—	—	—	—	—	—	—	—	—

† Undefined variation of FMOVR

Table 36—*cond*[3:0]

		BPcc	Bicc[D]	FBPfcc	FBfcc[D]	Tcc
		op = 0 op2 = 1	op = 0 op2 = 2	op = 0 op2 = 5	op = 0 op2 = 6	op = 2 op3 = $3A_{16}$
cond [3:0]	0	BPN	BN[D]	FBPN	FBN[D]	TN
	1	BPE	BE[D]	FBPNE	FBNE[D]	TE
	2	BPLE	BLE[D]	FBPLG	FBLG[D]	TLE
	3	BPL	BL[D]	FBPUL	FBUL[D]	TL
	4	BPLEU	BLEU[D]	FBPL	FBL[D]	TLEU
	5	BPCS	BCS[D]	FBPUG	FBUG[D]	TCS
	6	BPNEG	BNEG[D]	FBPG	FBG[D]	TNEG
	7	BPVS	BVS[D]	FBPU	FBU[D]	TVS
	8	BPA	BA[D]	FBPA	FBA[D]	TA
	9	BPNE	BNE[D]	FBPE	FBE[D]	TNE
	A	BPG	BG[D]	FBPUE	FBUE[D]	TG
	B	BPGE	BGE[D]	FBPGE	FBGE[D]	TGE
	C	BPGU	BGU[D]	FBPUGE	FBUGE[D]	TGU
	D	BPCC	BCC[D]	FBPLE	FBLE[D]	TCC
	E	BPPOS	BPOS[D]	FBPULE	FBULE[D]	TPOS
	F	BPVC	BVC[D]	FBPO	FBO[D]	TVC

Table 37—Encoding of *rcond*[2:0] Instruction Field

		BPr	MOVr	FMOVr
		op = 0 op2 = 3	op = 2 op3 = $2F_{16}$	op = 2 op3 = 35_{16}
rcond [2:0]	0	—	—	—
	1	BRZ	MOVRZ	FMOVZ
	2	BRLEZ	MOVRLEZ	FMOVLEZ
	3	BRLZ	MOVRLZ	FMOVLZ
	4	—	—	—
	5	BRNZ	MOVRNZ	FMOVNZ
	6	BRGZ	MOVRGZ	FMOVGZ
	7	BRGEZ	MOVRGEZ	FMOVGEZ

Table 38—*cc*/*opf_cc* Fields (MOVcc and FMOVcc)

opf_cc			Condition code selected
cc2	cc1	cc0	
0	0	0	fcc0
0	0	1	fcc1
0	1	0	fcc2
0	1	1	fcc3
1	0	0	icc
1	0	1	—
1	1	0	xcc
1	1	1	—

Table 39—*cc* Fields (FBPfcc, FCMP and FCMPE)

cc1	cc0	Condition code selected
0	0	fcc0
0	1	fcc1
1	0	fcc2
1	1	fcc3

Table 40—*cc* Fields (BPcc and Tcc)

cc1	cc0	Condition code selected
0	0	icc
0	1	—
1	0	xcc
1	1	—

A.27 Load Integer

The LDD instruction is deprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC-V9 software. It is recommended that the LDX instruction be used in its place.

Opcode	op3	Operation
LDSB	00 1001	Load Signed Byte
LDSH	00 1010	Load Signed Halfword
LDSW	00 1000	Load Signed Word
LDUB	00 0001	Load Unsigned Byte
LDUH	00 0010	Load Unsigned Halfword
LDUW	00 0000	Load Unsigned Word
LDX	00 1011	Load Extended Word
LDD[D]	00 0011	Load Doubleword

Format (3):

Suggested Assembly Language Syntax		
`ldsb`	[*address*], reg_{rd}	
`ldsh`	[*address*], reg_{rd}	
`ldsw`	[*address*], reg_{rd}	
`ldub`	[*address*], reg_{rd}	
`lduh`	[*address*], reg_{rd}	
`lduw`	[*address*], reg_{rd}	(*synonym*: `ld`)
`ldx`	[*address*], reg_{rd}	
`ldd`	[*address*], reg_{rd}	

Description:

The load integer instructions copy a byte, a halfword, a word, an extended word, or a doubleword from memory. All except LDD copy the fetched value into *r*[*rd*]. A fetched byte, halfword, or word is right-justified in the destination register *r*[*rd*]; it is either sign-extended or zero-filled on the left, depending on whether the opcode specifies a signed or unsigned operation, respectively.

The load doubleword integer instructions (LDD) copy a doubleword from memory into an *r*-register pair. The word at the effective memory address is copied into the even *r* register. The word at the effective memory address + 4 is copied into the following odd-numbered *r*

register. The upper 32 bits of both the even-numbered and odd-numbered *r* registers are zero-filled. Note that a load doubleword with *rd* = 0 modifies only *r*[1]. The least significant bit of the *rd* field in an LDD instruction is unused and should be set to zero by software. An attempt to execute a load doubleword instruction that refers to a misaligned (odd-numbered) destination register causes an *illegal_instruction* exception.

IMPL. DEP. #107(1): It is implementation-dependent whether LDD is implemented in hardware. If not, an attempt to execute it will cause an *unimplemented_ldd* exception.

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

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

LDUH and LDSH cause a *mem_address_not_aligned* exception if the address is not halfword-aligned. LDUW and LDSW cause a *mem_address_not_aligned* exception if the effective address is not word-aligned. LDX and LDD cause a *mem_address_not_aligned* exception if the address is not doubleword-aligned.

Programming Note:

LDD is provided for compatibility with SPARC-V8. It may execute slowly on SPARC-V9 machines because of data path and register-access difficulties. In some systems it may trap to emulation code. It is suggested that programmers and compilers avoid using these instructions.

If LDD is emulated in software, an LDX instruction should be used for the memory access in order to preserve atomicity.

Compatibility Note:

The SPARC-V8 LD instruction has been renamed LDUW in SPARC-V9. The LDSW instruction is new in SPARC-V9.

Exceptions:

async_data_error
unimplemented_LDD (LDD only (impl. dep. #107))
illegal_instruction (LDD with odd *rd*)
mem_address_not_aligned (all except LDSB, LDUB)
data_access_exception
data_access_protection
data_access_MMU_miss
data_access_error

A.54 Store Integer

The STD instruction isdeprecated; it is provided only for compatibility with previous versions of the architecture. It should not be used in new SPARC-V9 software. It is recommended that the STX instruction be used in its place.

Opcode	op3	Operation
STB	00 0101	Store Byte
STH	00 0110	Store Halfword
STW	00 0100	Store Word
STX	00 1110	Store Extended Word
STD[D]	00 0111	Store Doubleword

Format (3):

Suggested Assembly Language Syntax		
`stb`	*reg*$_{rd}$`,` [*address*]	(*synonyms*: `stub`, `stsb`)
`sth`	*reg*$_{rd}$`,` [*address*]	(*synonyms*: `stuh`, `stsh`)
`stw`	*reg*$_{rd}$`,` [*address*]	(*synonyms*: `st`, `stuw`, `stsw`)
`stx`	*reg*$_{rd}$`,` [*address*]	
`std`	*reg*$_{rd}$`,` [*address*]	

Description:

The store integer instructions (except store doubleword) copy the whole extended (64-bit) integer, the less-significant word, the least significant halfword, or the least significant byte of *r*[*rd*] into memory.

The store doubleword integer instruction (STD) copies two words from an *r* register pair into memory. The least significant 32 bits of the even-numbered *r* register are written into memory at the effective address, and the least significant 32 bits of the following odd-numbered *r* register are 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 misaligned (odd-numbered) *rd* causes an *illegal_instruction* exception.

IMPL. DEP. #108(1): IT is implementation-dependent whether STD is implemented in hardware. if not, an attempt to execute it will cause an *unimplemented_STD* exception.

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

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

STH causes a *mem_address_not_aligned* exception if the effective address is not halfword-aligned. STW causes a *mem_address_not_aligned* exception if the effective address is not word-aligned. STX and STD causes a *mem_address_not_aligned* exception if the effective address is not doubleword-aligned.

Programming Note:

STD is provided for compatibility with SPARC-V8. It may execute slowly on SPARC-V9 machines because of data path and register-access difficulties. In some SPARC-V9 systems it may cause a trap to emulation code; therefore, STD should be avoided.

If STD is emulated in software, STX should be used in order to preserve atomicity.

Compatibility Note:

The SPARC-V8 ST instruction has been renamed STW in SPARC-V9.

Exceptions:

async_data_error
unimplemented_STD (STD only) (impl. dep. #108)
illegal_instruction (STD with odd *rd*)
mem_address_not_aligned (all except STB)
data_access_exception
data_access_error
data_access_protection
data_access_MMU_miss