304-425

Assembler Synt

1.1. Introduction

Sun Microsystems' Sun-4 Assembler takes assembly language programs, as specified in this document, and produces relocatable object files for processi by the Sun-4 link editor. The assembly language described in this document corresponds with the SPARC instruction set defined in the *SPARC™ Architec Manual*, Version 8, is intended for use on Sun-4s and SPARCStations.

1.2. Other References

You should also become familiar with the manual pages *as*(1), *ld*(1), *cpp*(1), *a.out*(5), and the *SPARC Architecture Manual.*

1.3. A Short Example

The following example illustrates how a short assembly language program m look.

```
/*
 * a simple program to copy a string
 * showing correct syntax, delay slots, and use of annul bit.
 * pseudo-operations:     .seg, .global, .asciz, .skip
 * synthetic instructions:   set, ret, retl, mov, inc, deccc, nop
 * numeric label:         1
 * symbolic substitution: WINDOWSIZE
*/

#include <sun4/asm_linkage.h>

        .seg    "text"
        .global _main
_main:
        save    %sp, -WINDOWSIZE, %sp
        set     str, %o                     ! source string
        set     out, %o1                    ! destination location
        call    _bcopy
        mov     24, %o2                     ! delay slot, length to copy

        ret
        restore %o0, 0, %o0                 ! return value from main


        .global _bcopy
```

```
1:
        inc     %o0                             ! inc from address
        stb     %o4, [%o1]                      ! write to address
        inc     %o1                             ! in the delay slot: inc to address

_bcopy:
        deccc   %o2                             ! dec count, set condition codes
        bge,a   1b                              ! loop until done
        ldub    [%o0], %o4                      ! delay slot, read from address
        retl                                    ! leaf routine return
        nop                                     ! delay slot

        .seg    "data"
str:
        .asciz  "this is a sample string"


        .seg    "bss"
out:
        .skip   30                              ! reserve 30 bytes
```

1.4. Syntax Notation

In the descriptions of assembly language syntax in this chapter, brackets "[ ]" enclose optional items, and the star "*" indicates items to be repeated zero or more times. Braces "{ }" enclose alternate item choices, which are separated from each other by vertical bars "|". Wherever blanks are allowed, arbitrary numbers of blanks and horizontal tabs may be used.

The syntax of assembly language lines is:

```
[statement  [ ; statement]*]  [!comment]
[!comment]
```

1.5. Statement Syntax

The syntax of an assembly language *statement* is:

```
[label:]  [instruction]
```

In the above syntax, *label* is a symbol name (described below), *instruction* is an encoded pseudo-op, synthetic instruction, or instruction, and *comment* is any text up to the line end.

1.6. Lexical Features

This section describes lexical features of the assembler's syntax.

Case Distinction

Upper and lower case are distinct everywhere, *except* in the names of special symbols (see below), where there is no case distinction.

Comments

A comment is preceded by an exclamation mark; the "`!`" and all following c acters up to the end of the line are ignored. C-style comments with "`/*...*` are also permitted, and may span multiple lines.

Numbers

Decimal, hexadecimal, and octal numeric constants are recognized, and are w ten as in the C language. For floating-point pseudo operations, floating-point constants are written with `0r` or `0R` (for `REAL`) followed by a string acceptat to `atof`(3): an optional sign followed by a nonempty string of digits with optional decimal point and optional exponent, or followed by a special name, shown below.

The special names `0rnan` and `0rinf` represent the special floating-point va Not-A-Number and INFinity, respectively. Negative Not-A-Number and Ne tive INFinity are specified as `0r-nan` and `0r-inf`, respectively.

NOTE *Notice that the names of these floating-point constants begin with a zero, not letter " O "*

Strings

Strings may be quoted with either double-quote (") or single-quote (') mark When used in an expression, the numeric value of a string is the numeric valu the ASCII representation of its first character.

The suggested style is to use single quote marks for the ASCII value of a sing character, and double quote marks for quoted-string operands, such as used b pseudo-ops. Here is some assembly code in the suggested style:

```
add    %g1,'a'-'A',%g1    ! g1 + ('a' - 'A') --> g1
.seg   "data"
.ascii "a string"
.byte  'M'
```

The following escape codes are recognized in strings; they are derived from C

`\b`	backspace
`\f`	formfeed
`\n`	newline (linefeed)
`\r`	carriage return
`\t`	horizontal tab
`\`*nnn*	octal value *nnn*

Symbol Names

The syntax for a symbol name is:

{ *letter* | _ | $ | . } { *letter* | _ | $ | . | *digit* }*

Upper-case and lower-case letters are distinct, and the underscore, dollar sign and period are treated as alphabetic characters.

Symbol names that begin with `L` are assumed to be compiler-generated local symbols, and, to simplify debugging somewhat, are best avoided in hand-coded assembly language routines.

The symbol "`.`" is predefined, and always refers to the address of the beginning of the current assembly language statement.

NOTE *By convention, system run-time routine names start with "." and names from C assembly language and `f77` begin with a "_".*

Labels

A label is either a symbol or a single decimal digit *n* (`0...9`). Note that a label immediately followed by a colon.

Numeric labels may be defined repeatedly in an assembly, whereas normal symbolic labels may be defined only once.

A numeric label *n* is referenced after its definition (backward reference) as *n*`b`, and before its definition (forward reference) as *n*`f`.

Special Symbols

Special symbol names begin with `%` so as not to conflict with user symbols, and include:

Table 1-1 *Special Symbols*

Symbol Object	*Name*	*Comment*
general-purpose registers	`%r0 ... %r31`	
general-purpose global registers	`%g0 ... %g7`	*(same as `%r0 ... %r7`)*
general-purpose "out" registers	`%o0 ... %o7`	*(same as `%r8 ... %r15`)*
general-purpose "local" registers	`%l0 ... %l7`	*(same as `%r16 ... %r23`)*
general-purpose "in" registers	`%i0 ... %i7`	*(same as `%r24 ... %r31`)*
stack-pointer register	`%sp`	*(`%sp` ≡ `%o6` ≡ `%14`)*
frame-pointer register	`%fp`	*(`%fp` ≡ `%i6` ≡ `%30`)*
floating-point registers	`%f0 ... %f31`	
floating-point status register	`%fsr`	
front of floating-point queue	`%fq`	
coprocessor registers	`%c0 ... %c31`	
coprocessor status register	`%csr`	
coprocessor queue	`%cq`	
program status register	`%psr`	
trap vector base address register	`%tbr`	
window invalid mask	`%wim`	
Y register	`%y`	
unary operators	`%lo`	*(extracts least significant 10 bits)*
	`%hi`	*(extracts most significant 22 bits)*

There is no case distinction in special symbols; therefore using something like `%PSR` is equivalent to `%psr`. Use of all lower-case is the suggested style. The lack of case distinction allows for the use of non-recursive preprocessor

substitutions, such as

```
#define psr %PSR
```

The special symbols `%hi` and `%lo` are true unary operators which can be us any expression, and like other unary operators have higher precedence than binary operations. For example:

```
%hi a+b  ≡  (%hi a)+b
%lo a+b  ≡  (%lo a)+b
```

It is a good idea to enclose operands of `%hi` or `%lo` in parentheses to avoid ambiguity. For example:

```
%hi(a) + b
```

Operators and Expressions

The following operators are recognized in constant expressions:

Binary	*Operators*	*Unary*	*Operators*
+	Integer Addition	+	(no effect)
-	Integer Subtraction	-	2's Complement
*	Integer Multiplication	~	1's Complement
/	Integer Division	%lo	(see above)
%	Modulo	%hi	(see above)
^	Exclusive OR		
<<	Left Shift		
>>	Right Shift		
&	Bitwise AND		
|	Bitwise OR		

Note that the modulo operator `%` must not be immediately followed by a lett digit, to avoid confusion with register names or with `%hi` or `%lo`. The mod operator is typically followed by a space or left parenthesis.

Although the above operators have the same precedence as in the C languag parenthesization of expressions is recommended to avoid ambiguity.

1.7. `as` Error Messages

Messages generated by the assembler are generally self explanatory and give sufficient information to allow one to correct a problem. Certain conditions cause the assembler to issue warnings associated with delay slots following trol Transfer Instructions (CTIs):

- `set` instructions in delay slots
- labels in delay slots
- segments that end in control/transfer instructions

These are not necessarily incorrect, but point to places where a problem could exist. If you have intentionally written code this way, you can inform the assembler that you know what you are doing by inserting a pseudo-op in a manner similar to a C programmer's using casts.

The `.empty` pseudo-operation in a delay slot tells the assembler that the delay slot can be empty or contain whatever follows, because you have verified that either the code is correct or the content of the delay slot doesn't matter. Avoid using `.empty` in assembly-language programs just as you would avoid using casts in C programs. The `.empty` pseudo-operation is used only in programs written in assembly language; Sun's compilers don't generate it.

Instruction-Set Mapp

The tables in this chapter describe the relationship between hardware instru of the SPARC architecture, as defined in *SPARC Processor Architecture*, a instruction set used by Sun Microsystems' SPARC Assembler.

2.1. Table Notation

The following table describes the notation used in the tables in the rest of t chapter to describe the instruction set of the assembler.

Table 2-1 *Notation*

Symbol	*Describes*	*Comment*

reg	`%r0 ... %r31`
	`%g0 ... %g7`
	`%o0 ... %o7`
	`%l0 ... %l7`
	`%i0 ... %i7`
		
		(same as `%r0...%r7`*)*
		(same as `%r8...%r15`*)*
		(same as `%r16...%r23`*)*
		(same as `%r24...%r31`*)*

freg	`%f0 ... %f31`	
creg	`%c0 ... %c31`	
value		*(an expression involving at most one relocatable symbol)*
const13	*value*	*(a signed constant which fits in 13 bits)*
const22	*value*	*(a constant which fits in 22 bits)*
asi	*value*	*(alternate address space identifier; an unsigned 8-bit value)*
reg_{rd}		*Destination register.*
reg_{rs1}, reg_{rs2}		*Source register 1, source register 2.*
regaddr	reg_{rs1} $reg_{rs1} + reg_{rs2}$	*Address formed with register contents only.*
address	$reg_{rs1} + reg_{rs2}$ $reg_{rs1} + const13$ $reg_{rs1} - const13$ $const13 + reg_{rs1}$ *const13*	*Address formed from register contents, immediate constant, or both.*

Table 2-1 Notation— Continued

Symbol	*Describes*	*Comment*
reg_or_imm	reg_{rs2} *const13*	*Value from either a single register, or an immediate constant.*

2.2. Integer Instructions

The following table outlines the correspondence between SPARC hardware integer instructions and SPARC assembly language instructions. The following notations are suffixed repeatedly to assembler mnemonics (and in upper case for SPARC architecture instruction names):

sr — status register.

a — instructions dealing with alternate space.

b — byte instructions.

h — halfword instructions.

d — doubleword instructions.

f — referencing floating-point registers.

c — referencing coprocessor registers.

rd — as a subscript, refers to a destination register in the argument list of an instruction.

rs — as a subscript, refers to a source register in the argument list of an instruction.

NOTE *The syntax of individual instructions is designed so that a destination operand (if any), which may be either a register or a reference to a memory location, is always the* **last** *operand in a statement.*

In the table below, curly brackets ({ }) mark optional arguments. Square brackets ([ ]) mark indirection: the *contents* of the addressed memory location are being read from or written to.

NOTE *All* `Bicc` *and* `Bfcc` *instructions, described in the following table, may indicate that the* **annul** *bit is to be set by appending* "`,a`" *to the opcode; e.g.* "`bgeu,a` *label*".

Table 2-2 *SPARC to Assembly Language Mapping*

SPARC	*Mnemonic*	*Argument List*	*Name*	*Comments*
ADD	`add`	*reg_{rs1}, reg_or_imm, reg_{rd}*	*Add*	
ADDcc	`addcc`	*reg_{rs1}, reg_or_imm, reg_{rd}*	*Add and modify icc*	
ADDX	`addx`	*reg_{rs1}, reg_or_imm, reg_{rd}*	*Add with carry*	
ADDXcc	`addxcc`	*reg_{rs1}, reg_or_imm, reg_{rd}*		
AND	`and`	*reg_{rs1}, reg_or_imm, reg_{rd}*	*And*	
ANDcc	`andcc`	*reg_{rs1}, reg_or_imm, reg_{rd}*		
ANDN	`andn`	*reg_{rs1}, reg_or_imm, reg_{rd}*		
ANDNcc	`andncc`	*reg_{rs1}, reg_or_imm, reg_{rd}*		
Bicc	`bn{,a}`	*label*	*Branch on integer condition codes*	*(branch never)*
	`bne{,a}`	*label*		*(synonym:* `bnz`*)*
	`be{,a}`	*label*		*(synonym:* `bz`*)*
	`bg{,a}`	*label*		
Bicc	`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*		
	`ba{,a}`	*label*		*(synonym:* `b`*)*
CALL	`call`	*label{,n}*	*(n = # of out registers used as arguments)*	
CBccc	`cbn{,a}`	*label*	*Branch on coprocessor condition codes*	*(branch never)*
	`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*		
	`cba{,a}`	*label*		

Table 2-2 *SPARC to Assembly Language Mapping— Continued*

SPARC	*Mnemonic*	*Argument List*	*Name*	*Comments*
FBfcc	`fbn{,a}`	*label*	*Branch on floating-point condition codes*	*(branch never)*
	`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*		
	`fba{,a}`	*label*		
FLUSH	`flush`	*address*	*Instruction cache flush*	
JMPL	`jmpl`	*address*, reg_{rd}	*Jump and link*	
LDSB	`ldsb`	[*address*], reg_{rd}	*Load signed byte*	
LDSH	`ldsh`	[*address*], reg_{rd}	*Load signed halfword*	
LDSTUB	`ldstub`	[*address*], reg_{rd}	*Load-store unsigned byte*	
LDUB	`ldub`	[*address*], reg_{rd}	*Load unsigned byte*	
LDUH	`lduh`	[*address*], reg_{rd}	*Load unsigned halfword*	
LD	`ld`	[*address*], reg_{rd}	*Load word*	
LDD	`ldd`	[*address*], reg_{rd}	*Load double word*	(reg_{rd} *must be even)*
LDF	`ld`	[*address*], $freg_{rd}$	*Load floating-point register*	
LDFSR	`ld`	[*address*], `%fsr`		
LDDF	`ldd`	[*address*], $freg_{rd}$	*Load double floating-point*	
LDC	`ld`	[*address*], $creg_{rd}$	*Load coprocessor*	
LDCSR	`ld`	[*address*], `%csr`		
LDDC	`ldd`	[*address*], $creg_{rd}$	*Load double coprocessor*	
LDSBA	`ldsba`	[*regaddr*] *asi*, reg_{rd}	*Load signed byte from alternate space*	
LDSHA	`ldsha`	[*regaddr*] *asi*, reg_{rd}		
LDUBA	`lduba`	[*regaddr*] *asi*, reg_{rd}		
LDUHA	`lduha`	[*regaddr*] *asi*, reg_{rd}		
LDA	`lda`	[*regaddr*] *asi*, reg_{rd}		
LDDA	`ldda`	[*regaddr*] *asi*, reg_{rd}		(reg_{rd} *must be even)*
LDSTUBA	`ldstuba`	[*regaddr*] *asi*, reg_{rd}		

Table 2-2 *SPARC to Assembly Language Mapping— Continued*

SPARC	*Mnemonic*	*Argument List*	*Name*	*Comments*
MULScc	`mulscc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*Multiply step (and modify icc)*	
NOP	`nop`		*no operation*	
OR	`or`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*Inclusive or*	
ORcc	`orcc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$		
ORN	`orn`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$		
ORNcc	`orncc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$		
RDASR	`rd`	`%asrn`$_{rs1}$, *reg*$_{rd}$		
RDY	`rd`	`%y`, *reg*$_{rd}$		*(see synthetic instruction*
RDPSR	`rd`	`%psr`, *reg*$_{rd}$		*(see synthetic instruction*
RDWIM	`rd`	`%wim`, *reg*$_{rd}$		*(see synthetic instruction*
RDTBR	`rd`	`%tbr`, *reg*$_{rd}$		*(see synthetic instruction*
RESTORE	`restore`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$		*(see synthetic instruction*
RETT	`rett`	*address*	*Return from trap*	
SAVE	`save`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$		*(see synthetic instruction*
SDIV	`sdiv`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*signed divide*	
SDIVcc	`sdiv`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*signed divide and modify icc*	
SMUL	`smul`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*signed multiply*	
SMULcc	`smulcc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*signed multiply and modify icc*	
SETHI	`sethi`	*const22*, *reg*$_{rd}$	*Set high 22 bits of r register*	
	`sethi`	`%hi(`*value*`)`, *reg*$_{rd}$		*(see synthetic instructior*
SLL	`sll`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*Shift left logical*	
SRL	`srl`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*Shift right logical*	
SRA	`sra`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*Shift right arithmetic*	
STB	`stb`	*regaddr*, [*address*]	*Store byte.*	*(synonyms:* `stub, st`
STH	`sth`	*regaddr*, [*address*]		*(synonyms:* `stuh, stsh`
ST	`st`	*reg*$_{rd}$, [*address*]		
STD	`std`	*reg*$_{rd}$, [*address*]		*(reg*$_{rd}$ *must be even)*
STF	`st`	*freg*$_{rd}$, [*address*]		
STDF	`std`	*freg*$_{rd}$, [*address*]		
STFSR	`st`	`%fsr`, [*address*]	*Store floating-point status register*	
STDFQ	`std`	`%fq`, [*address*]	*Store double floating-point queue*	
STC	`st`	*creg*$_{rd}$, [*address*]	*Store coprocessor*	

Table 2-2 *SPARC to Assembly Language Mapping— Continued*

SPARC	*Mnemonic*	*Argument List*	*Name*	*Comments*

STDC
STCSR
STDCQ
	`std`
	`st`
	`std`
		creg$_{rd}$, [*address*]
		`%csr`, [*address*]
		`%cq`, [*address*]
			Store double coprocessor queue	

STBA
 	
STHA
STA
STDA
	`stba`
	
	`stha`
	`sta`
	`stda`
		regaddr, [*regaddr*] *asi*
		
		regaddr, [*regaddr*] *asi*
		reg$_{rd}$, [*regaddr*] *asi*
		reg$_{rd}$, [*regaddr*] *asi*
			Store byte into alternate space	*(synonyms:* `stuba`, `stsba`*)*
				
				(synonyms: `stuha`, `stsha`
				
				(reg$_{rd}$ *must be even)*

SUB
SUBcc
SUBX
SUBXcc
	`sub`
	`subcc`
	`subx`
	`subxcc`
		reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
		reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
		reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
		reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
			Subtract
			Subtract and modify icc
			Subtract with carry	

SWAP SWAPA	`swap` `swapa`	[*address*], *reg*$_{rd}$ [*regaddr*] *asi*, *reg*$_{rd}$	*Swap memory word with register*	

Ticc	`tn`
	
	`tne`
	`te`
	`tg`
	`tle`
	`tge`
	`tl`
	`tgu`
	`tleu`
	`tlu`
	`tgeu`
	`tpos`
	`tneg`
	`tvc`
	`tvs`
	`ta`
		address
		
		address
		address
		address
		address
		address
		address
		address
		address
		address
		address
		address
		address
		address
		address
		address
			Trap on integer condition code. (See note.)	*(trap never)*
				
				(synonym: `tnz`*)*
				(synonym: `tz`*)*
				
				(synonym: `tcc`*)*
				(synonym: `tcs`*)*
				
				(synonym: `t`*)*

TADDcc
TSUBcc
TADDccTV
 	
TSUBccTV
	`taddcc`
	`tsubcc`
	`taddcctv`
	
	`tsubcctv`
		reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
		reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
		reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
		
		reg$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$
			Tagged add and modify icc
			
			Tagged add and modify icc and trap on overflow	

UDIV UDIVcc	`udiv` `udivcc`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$ *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*unsigned divide* *unsigned divide and modify icc*	
UMUL	`umul`	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	*unsigned multiply*	

Table 2-2 *SPARC to Assembly Language Mapping— Continued*

SPARC	*Mnemonic*	*Argument List*	*Name*	*Comments*
UMULcc	`umulcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}	*unsigned multiply and modify icc*	
UNIMP	`unimp`	*const22*	*Unimplemented instruction*	
WRASR	`wr`	*reg_or_imm*, `%asrn`$_{rs1}$		
WRY	`wr`	reg_{rs1}, *reg_or_imm*, `%y`		*(see synthetic instructions*
WRPSR	`wr`	reg_{rs1}, *reg_or_imm*, `%psr`		*(see synthetic instructions*
WRWIM	`wr`	reg_{rs1}, *reg_or_imm*, `%wim`		*(see synthetic instructions*
WRTBR	`wr`	reg_{rs1}, *reg_or_imm*, `%tbr`		*(see synthetic instructions*
XNOR	`xnor`	reg_{rs1}, *reg_or_imm*, reg_{rd}	*Exclusive nor*	
XNORcc	`xnorcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}		
XOR	`xor`	reg_{rs1}, *reg_or_imm*, reg_{rd}	*Exclusive or*	
XORcc	`xorcc`	reg_{rs1}, *reg_or_imm*, reg_{rd}		

NOTE *Trap numbers 16-31 are available for use by the user, and will not be usurpe Sun. Currently-defined trap numbers are those defined in* `/usr/include/sun4/trap.h`, *as follows:*

```
0x00  ST_SYSCALL
0x01  ST_BREAKPOINT
0x02  ST_DIV0
0x03  ST_FLUSH_WINDOWS
0x04  ST_CLEAN_WINDOWS
0x05  ST_RANGE_CHECK
0x06  ST_FIX_ALIGN
0x07  ST_INT_OVERFLOW
```

2.3. Floating-Point Instructions

In the table below, the types of numbers being manipulated by an instructior denoted by the following lowercase letters:

i — integer

s — single

d — double

q — quad

In some cases where more than numeric type is involved, each instruction ir group is described. Otherwise, only the first member of a group is describec

Table 2-3 *Floating-point Instructions*

SPARC	*Mnemonic*	*Argument List*	*Description*
FiTOs	fitos	$freg_{rs2}$, $freg_{rd}$	*Convert integer to single.*
FiTOd	fitod	$freg_{rs2}$, $freg_{rd}$	*Convert integer to double.*
FiTOq	fitoq	$freg_{rs2}$, $freg_{rd}$	*Convert integer to quad.*
FsTOi	fstoi	$freg_{rs2}$, $freg_{rd}$	*Convert single to integer.*
FdTOi	fdtoi	$freg_{rs2}$, $freg_{rd}$	*Convert double to integer.*
FqTOi	fqtoi	$freg_{rs2}$, $freg_{rd}$	*Convert quad to integer.*
FsTOd	fstod	$freg_{rs2}$, $freg_{rd}$	*Convert single to double.*
FsTOq	fstoq	$freg_{rs2}$, $freg_{rd}$	*Convert single to quad.*
FdTOs	fdtos	$freg_{rs2}$, $freg_{rd}$	*Convert double to single.*
FdTOq	fdtoq	$freg_{rs2}$, $freg_{rd}$	*Convert double to quad.*
FqTOd	fqtod	$freg_{rs2}$, $freg_{rd}$	*Convert quad to double.*
FqTOs	fqtos	$freg_{rs2}$, $freg_{rd}$	*Convert quad to single.*
FMOVs	fmovs	$freg_{rs2}$, $freg_{rd}$	*Move*
FNEGs	fnegs	$freg_{rs2}$, $freg_{rd}$	*Negate*
FABSs	fabss	$freg_{rs2}$, $freg_{rd}$	*Absolute value*
FSQRTs	fsqrts	$freg_{rs2}$, $freg_{rd}$	*Square root*
FSQRTd	fsqrtd	$freg_{rs2}$, $freg_{rd}$	
FSQRTq	fsqrtq	$freg_{rs2}$, $freg_{rd}$	
FADDs	fadds	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	*Add*
FADDd	faddd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FADDq	faddq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FSUBs	fsubs	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	*Subtract*
FSUBd	fsubd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FSUBq	fsubx	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FMULs	fmuls	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	*Multiply*
FMULd	fmuld	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FMULq	fmulq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FdMULq	fmulq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	Multiply double to quad.
FsMULd	fsmuld	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	Multiply single to double.
FDIVs	fdivs	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	*Divide*
FDIVd	fdivd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FDIVq	fdivq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FCMPs	fcmps	$freg_{rs1}$, $freg_{rs2}$	*Compare*
FCMPd	fcmpd	$freg_{rs1}$, $freg_{rs2}$	
FCMPq	fcmpq	$freg_{rs1}$, $freg_{rs2}$	

Table 2-3 *Floating-point Instructions— Continued*

SPARC	*Mnemonic*	*Argument List*	*Description*
FCMPEs	fcmpes	$freg_{rs1}$, $freg_{rs2}$	*Compare, Generate exception if unordered.*
FCMPEd	fcmped	$freg_{rs1}$, $freg_{rs2}$	
FCMPEq	fcmpeq	$freg_{rs1}$, $freg_{rs2}$	

2.4. Coprocessor Instructions

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

If the EC field of the PSR is 0, or if no coprocessor is present, a cpop*n* inst tion causes a cp_disabled trap.

The conditions causing a cp_exception trap are coprocessor-dependent.

NOTE *A non-* cpop*n (non-coprocessor-operate) instruction must be executed betv a* cpop2 *instruction and a subsequent* cbccc *instruction.*

Table 2-4 *Coprocessor Instructions*

SPARC	*Mnemonic*	*Argument List*	*Name*	*Comments*
CPop1	cpop1	*opd*, reg_{rs1}, reg_{rs2}, reg_{rd}	*Coprocessor operation*	
CPop2	cpop2	*opd*, reg_{rs1}, reg_{rs2}, reg_{rd}	*Coprocessor operation*	*(may modify cc*

2.5. Synthetic Instructions

This section describes the mapping of synthetic instructions to hardware ins tions.

Table 2-5 *Synthetic Instruction to Hardware Instruction Mapping*

Synthetic Instruction		*Hardware Equivalent(s)*		*Comment*
cmp	reg_{rs1}, *reg_or_imm*	subcc	.reg_{rs1}, *reg_or_imm*, %g0	*(compare)*
jmp	*address*	jmpl	*address*, %g0	
call	*reg_or_imm*	jmpl	*reg_or_imm*, %o7	
tst	reg_{rs1}	orcc	reg_{rs1}, %g0, %g0	*(test)*
ret		jmpl	%i7+8, %g0	*(return from subroutine)*
retl		jmpl	%o7+8, %g0	*(return from leaf subrouti*
restore		restore	%g0, %g0, %g0	*(trivial* restore*)*
save		save	%g0, %g0, %g0	*(trivial* save*)* *Warning: trivial* save *should only be used in ke code!*
set	*value*, reg_{rd}	or	%g0, *value*, reg_{rd}	*(if -4096 ≤ value ≤ 4095)*

Table 2-5 *Synthetic Instruction to Hardware Instruction Mapping— Continued*

Synthetic Instruction		*Hardware Equivalent(s)*		*Comment*
`set`	*value*, reg_{rd}	`sethi`	`%hi(`*value*`)`, reg_{rd}	*(if ((value&0x1ff) == 0))*
`set`	*value*, reg_{rd}	`sethi` `or`	`%hi(`*value*`)`, reg_{rd}; reg_{rd}, `%lo(`*value*`)`, reg_{rd}	*(otherwise)* *Warning: do not use* `set` *in an instruction's delay slot.*
`not`	reg_{rs1}, reg_{rd}	`xnor`	reg_{rs1}, `%g0`, reg_{rd}	*(one's complement)*
`not`	reg_{rd}	`xnor`	reg_{rd}, `%g0`, reg_{rd}	*(one's complement)*
`neg`	reg_{rs2}, reg_{rd}	`sub`	`%g0`, reg_{rs2}, reg_{rd}	*(two's complement)*
`neg`	reg_{rd}	`sub`	`%g0`, reg_{rd}, reg_{rd}	*(two's complement)*
`inc`	reg_{rd}	`add`	reg_{rd}, `1`, reg_{rd}	*(increment by 1)*
`inc`	*const13*, reg_{rd}	`add`	reg_{rd}, *const13*, reg_{rd}	*(increment by const13)*
`inccc`	reg_{rd}	`addcc`	reg_{rd}, `1`, reg_{rd}	*(increment by 1 and set icc)*
`inccc`	*const13*, reg_{rd}	`addcc`	reg_{rd}, *const13*, reg_{rd}	*(increment by const13 and set icc)*
`dec`	reg_{rd}	`sub`	reg_{rd}, `1`, reg_{rd}	*(decrement by 1)*
`dec`	*const13*, reg_{rd}	`sub`	reg_{rd}, *const13*, reg_{rd}	*(decrement by const13)*
`deccc`	reg_{rd}	`subcc`	reg_{rd}, `1`, reg_{rd}	*(decrement by 1 and set icc)*
`deccc`	*const13*, reg_{rd}	`subcc`	reg_{rd}, *const13*, reg_{rd}	*(decrement by const13 and set icc)*
`btst`	*reg_or_imm*, reg_{rs1}	`andcc`	reg_{rs1}, *reg_or_imm*, `%g0`	*(bit test)*
`bset`	*reg_or_imm*, reg_{rd}	`or`	reg_{rd}, *reg_or_imm*, reg_{rd}	*(bit set)*
`bclr`	*reg_or_imm*, reg_{rd}	`andn`	reg_{rd}, *reg_or_imm*, reg_{rd}	*(bit clear)*
`btog`	*reg_or_imm*, reg_{rd}	`xor`	reg_{rd}, *reg_or_imm*, reg_{rd}	*(bit toggle)*
`clr`	reg_{rd}	`or`	`%g0`, `%g0`, reg_{rd}	*(clear(zero) register)*
`clrb`	[*address*]	`stb`	`%g0`, [*address*]	*(clear byte)*
`clrh`	[*address*]	`sth`	`%g0`, [*address*]	*(clear halfword)*
`clr`	[*address*]	`st`	`%g0`, [*address*]	*(clear word)*
`mov`	*reg_or_imm*, reg_{rd}	`or`	`%g0`, *reg_or_imm*, reg_{rd}	
`mov`	`%y`, reg_{rs1}	`rd`	`%y`, reg_{rs1}	
`mov`	`%psr`, reg_{rs1}	`rd`	`%psr`, reg_{rs1}	
`mov`	`%wim`, reg_{rs1}	`rd`	`%wim`, reg_{rs1}	
`mov`	`%tbr`, reg_{rs1}	`rd`	`%tbr`, reg_{rs1}	
`mov`	*reg_or_imm*, `%y`	`wr`	`%g0`, *reg_or_imm*, `%y`	
`mov`	*reg_or_imm*, `%psr`	`wr`	`%g0`, *reg_or_imm*, `%psr`	
`mov`	*reg_or_imm*, `%wim`	`wr`	`%g0`, *reg_or_imm*, `%wim`	
`mov`	*reg_or_imm*, `%tbr`	`wr`	`%g0`, *reg_or_imm*, `%tbr`	

2.6. Leaf Procedures

Leaf procedures are the outermost routines on the tree of a program, as a tree leaf is at the end of a stem on the branch of a tree.

Some leaf procedures can be made to operate *without* their own register win or stack frame, using their caller's instead. Such a leaf procedure is called a **optimized leaf procedure**. This can be done when the candidate procedure meets all of the following conditions:

- it contains no `CALL`s or `JMPL`s to other procedures
- it contains no references to `%sp`, except in its `SAVE` instruction
- it contains no references to `%fp`
- it refers to, or can be made to refer to, no more than 8 of the 32 integer r ters, inclusive of `%o7`, the "return address".

If a procedure conforms to all of the above conditions, it can be made to ope using its caller's stack frame and registers an optimization that saves both ti and space. When optimized, the procedure may only safely use registers wh its caller already assumes to be volatile across a procedure call: `%o0` ... `%o5` `%o7`, and `%g1`. This may be expanded to registers `%g1` ... `%g7` if SPARC A compliance isn't required.

Leaf routines are most useful when they prevent expensive window overflow/underflow situations, saving many tens of cycles each.