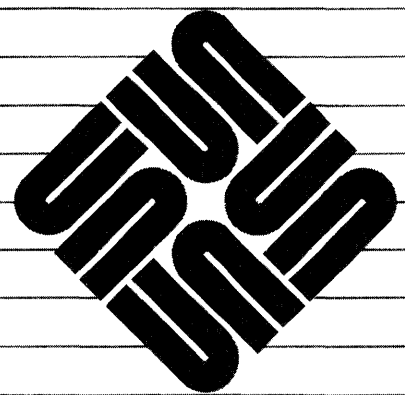
Sun-4 Assembly Language Reference Manual



Part Number: 800-3806-10
Revision A of 27 March, 1990

SPARC™ is a trademark of Sun Microsystems, Inc.

Sun Workstation® is a trademark of Sun Microsystems, Incorporated.

Copyright © 1990 Sun Microsystems, Inc. – Printed in U.S.A.

All rights reserved. No part of this work covered by copyright hereon may be reproduced in any form or by any means – graphic, electronic, or mechanical – including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

Restricted rights legend: use, duplication, or disclosure by the U.S. government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and in similar clauses in the FAR and NASA FAR Supplement.

The Sun Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

This product is protected by one or more of the following U.S. patents: 4,777,485 4,688,190 4,527,232 4,745,407 4,679,014 4,435,792 4,719,569 4,550,368 in addition to foreign patents and applications pending.

Contents

Chapter 1 Assembler Syntax **1**

1.1. Introduction 1

1.2. Other References 1

1.3. A Short Example 1

1.4. Syntax Notation 2

1.5. Statement Syntax 2

1.6. Lexical Features 2

Case Distinction 3

Comments 3

Numbers 3

Strings 3

Symbol Names 3

Labels 4

Special Symbols 4

Operators and Expressions 5

1.7. `as` Error Messages 5

Chapter 2 Instruction-Set Mapping **7**

2.1. Table Notation 7

2.2. Integer Instructions 8

2.3. Floating-Point Instructions 13

2.4. Coprocessor Instructions 15

2.5. Synthetic Instructions 15

2.6. Leaf Procedures 17

Appendix A Pseudo-Operations **19**

Appendix B The Sun-4 Assembler **23**

B.1. `as` Options 23

Index **25**

Tables

Table 1-1 Special Symbols 4

Table 2-1 Notation 7

Table 2-2 SPARC to Assembly Language Mapping 9

Table 2-3 Floating-point Instructions 14

Table 2-4 Coprocessor Instructions 15

Table 2-5 Synthetic Instruction to Hardware Instruction Mapping 15

Table A-1 List of Pseudo-Operations 19

1

Assembler Syntax

1.1. Introduction

Sun Microsystems' Sun-4 Assembler takes assembly language programs, as specified in this document, and produces relocatable object files for processing by the Sun-4 link editor. The assembly language described in this document corresponds with the SPARC instruction set defined in the *SPARC™ Architecture 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 might 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 characters 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 written 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 acceptable 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, as shown below.

The special names `0rnan` and `0rinf` represent the special floating-point values Not-A-Number and INFinity, respectively. Negative Not-A-Number and Negative 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 the letter "O"*

Strings

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

The suggested style is to use single quote marks for the ASCII value of a single character, and double quote marks for quoted-string operands, such as used by 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 is 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 used in 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 letter or digit, to avoid confusion with register names or with `%hi` or `%lo`. The modulo operator is typically followed by a space or left parenthesis.

Although the above operators have the same precedence as in the C language, 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 will cause the assembler to issue warnings associated with delay slots following Control 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.

2

Instruction-Set Mapping

The tables in this chapter describe the relationship between hardware instructions of the SPARC architecture, as defined in *SPARC Processor Architecture*, and the 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 the 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 ORcc ORN ORNcc	or orcc orn orncc	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}	*Inclusive or*	
RDASR RDY RDPSR RDWIM RDTBR	rd rd rd rd rd	%asrn_{rs1}, reg_{rd} %y, reg_{rd} %psr, reg_{rd} %wim, reg_{rd} %tbr, reg_{rd}		 *(see synthetic instructions)* *(see synthetic instructions)* *(see synthetic instructions)* *(see synthetic instructions)*
RESTORE	restore	reg_{rs1}, reg_or_imm, reg_{rd}		*(see synthetic instructions)*
RETT	rett	*address*	*Return from trap*	
SAVE	save	reg_{rs1}, reg_or_imm, reg_{rd}		*(see synthetic instructions)*
SDIV SDIVcc	sdiv sdiv	reg_{rs1}, reg_or_imm, reg_{rd} reg_{rs1}, reg_or_imm, reg_{rd}	*signed divide* *signed divide and modify icc*	
SMUL SMULcc	smul smulcc	reg_{rs1}, reg_or_imm, reg_{rd} reg_{rs1}, reg_or_imm, reg_{rd}	*signed multiply* *signed multiply and modify icc*	
SETHI	sethi sethi	*const22*, reg_{rd} %hi(*value*), reg_{rd}	*Set high 22 bits of r register*	 *(see synthetic instructions)*
SLL SRL SRA	sll srl sra	reg_{rs1}, reg_or_imm, reg_{rd} reg_{rs1}, reg_or_imm, reg_{rd} reg_{rs1}, reg_or_imm, reg_{rd}	*Shift left logical* *Shift right logical* *Shift right arithmetic*	
STB STH ST STD STF STDF STFSR STDFQ STC	stb sth st std st std st std st	*regaddr*, [*address*] *regaddr*, [*address*] reg_{rd}, [*address*] reg_{rd}, [*address*] $freg_{rd}$, [*address*] $freg_{rd}$, [*address*] %fsr, [*address*] %fq, [*address*] $creg_{rd}$, [*address*]	*Store byte.* *Store floating-point status register* *Store double floating-point queue* *Store coprocessor*	*(synonyms:* stub, stsb*)* *(synonyms:* stuh, stsh*)* *(reg_{rd} must be even)*

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

SPARC	Mnemonic	Argument List	Name	Comments
STDC	std	$creg_{rd}$, [*address*]		
STCSR	st	%csr, [*address*]		
STDCQ	std	%cq, [*address*]	*Store double coprocessor queue*	
STBA	stba	*regaddr*, [*regaddr*] *asi*	*Store byte into alternate space*	*(synonyms:* stuba, stsba*)*
STHA	stha	*regaddr*, [*regaddr*] *asi*		*(synonyms:* stuha, stsha*)*
STA	sta	reg_{rd}, [*regaddr*] *asi*		
STDA	stda	reg_{rd}, [*regaddr*] *asi*		*(reg_{rd} must be even)*
SUB	sub	reg_{rs1}, *reg_or_imm*, reg_{rd}	*Subtract*	
SUBcc	subcc	reg_{rs1}, *reg_or_imm*, reg_{rd}	*Subtract and modify icc*	
SUBX	subx	reg_{rs1}, *reg_or_imm*, reg_{rd}	*Subtract with carry*	
SUBXcc	subxcc	reg_{rs1}, *reg_or_imm*, reg_{rd}		
SWAP	swap	[*address*], reg_{rd}	*Swap memory word with register*	
SWAPA	swapa	[*regaddr*] *asi*, reg_{rd}		
Ticc	tn	*address*	*Trap on integer condition code. (See note.)*	*(trap never)*
	tne	*address*		*(synonym:* tnz*)*
	te	*address*		*(synonym:* tz*)*
	tg	*address*		
	tle	*address*		
	tge	*address*		
	tl	*address*		
	tgu	*address*		
	tleu	*address*		
	tlu	*address*		*(synonym:* tcc*)*
	tgeu	*address*		*(synonym:* tcs*)*
	tpos	*address*		
	tneg	*address*		
	tvc	*address*		
	tvs	*address*		
	ta	*address*		*(synonym:* t*)*
TADDcc	taddcc	reg_{rs1}, *reg_or_imm*, reg_{rd}	*Tagged add and modify icc*	
TSUBcc	tsubcc	reg_{rs1}, *reg_or_imm*, reg_{rd}		
TADDccTV	taddcctv	reg_{rs1}, *reg_or_imm*, reg_{rd}	*Tagged add and modify icc and trap on overflow*	
TSUBccTV	tsubcctv	reg_{rs1}, *reg_or_imm*, reg_{rd}		
UDIV	udiv	reg_{rs1}, *reg_or_imm*, reg_{rd}	*unsigned divide*	
UDIVcc	udivcc	reg_{rs1}, *reg_or_imm*, reg_{rd}	*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 usurped by 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 instruction are 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 in a group is described. Otherwise, only the first member of a group is described.

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 coprocessor 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* instruction 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 between 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 ccc's)*

2.5. Synthetic Instructions

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

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 retl		jmpl jmpl	%i7+8, %g0 %o7+8, %g0	*(return from subroutine)* *(return from leaf subroutine)*
restore save		restore save	%g0, %g0, %g0 %g0, %g0, %g0	*(trivial* restore*)* *(trivial* save*)* *Warning: trivial* save *should only be used in kernel 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's 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 window or stack frame, using their caller's instead. Such a leaf procedure is called an **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 registers, inclusive of `%o7`, the "return address".

If a procedure conforms to all of the above conditions, it can be made to operate using its caller's stack frame and registers an optimization that saves both time and space. When optimized, the procedure may only safely use registers which 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 ABI compliance isn't required.

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

A

Pseudo-Operations

The following pseudo-operations are supported by the Sun-4 assembler:

Table A-1 *List of Pseudo-Operations*

Mnemonic	***Argument(s)***	***Description***
`.alias`		Turns off preceding `.noalias.` (Compiler-generated only.)
`.noalias`	*%reg1*, *%reg2*	*%reg1* and *%reg2* will not alias each other (point to the same destination) until a `.alias` is issued. (Compiler-generated only.)
`.ascii`	"*string*" [,"*string*"]*	Generates the given sequence(s) of ASCII characters.
`.asciz`	"*string*" [,"*string*"]*	Generates the given sequence(s) of ASCII characters, with each string followed by a null byte.
`.optim`	"*string*"	Any optimization that can also be given as a flag in the command line, such as `-O[`*n*`]` with *n* = {0,1,2,3}. (Compiler-generated only.)
`.seg`	"*string*"	Changes the current segment to the one named, and sets the location counter to the location of the next available byte in that segment. The default segment at the beginning of assembly is `text`. Currently, only segments `text`, `data`, `data1`, and `bss` are supported.
`.skip`	*n*	Increments the location counter by *n*, which allocates *n* bytes of empty space in the current segment.
`.align`	*boundary*	Aligns the location counter on a 0-mod-*boundary* boundary; *boundary* may be 1 (which has no effect), 2, 4, or 8.
`.byte`	*8bitval* [,*8bitval*]*	Generates (a sequence of) initialized bytes in the current segment.
`.half`	*16bitval* [,*16bitval*]*	Generates (a sequence of) initialized halfwords in the current segment. The location counter must already be aligned on a halfword boundary (use `.align 2`).

Table A-1 *List of Pseudo-Operations— Continued*

Mnemonic	*Argument(s)*	*Description*
`.word`	*32bitval* [,*32bitval*] *	Generates (a sequence of) initialized words in the current segment. The location counter must already be aligned on a word boundary (use `.align 4`).
`.single`	0r*floatval* [,0r*floatval*] *	Generates (a sequence of) initialized single-precision floating-point values in the current segment. The location counter must already be aligned on a word boundary (use `.align 4`).
`.double`	0r*floatval* [,0r*floatval*] *	Generates (a sequence of) initialized double-precision floating-point values in the current segment. The location counter must already be aligned on a doubleword boundary (use `.align 8`).
`.quad`	0r*floatval* [,0r*floatval*] *	Generates (a sequence of) initialized quad-precision floating-point values in the current segment (`.quad` currently generates quad-precision values with only *double-precision* significance). The location counter must already be aligned on a doubleword boundary (use `.align 8`).
`.global`	*symbol_name* [,*symbol_name*] *	Marks the (list of) user symbols as "global". Note that when a symbol is both declared to be global and defined (that is, used as a label, used as the left operand of an = pseudo-op, or used as the first operand of a `.reserve` pseudo-op) in the same module, the `.global` must appear *before* the definition.
`.common`	*symbol_name*, *size* [,"*segment*"]	Declares the name and size (in bytes) of a FORTRAN-style COMMON area. If "*segment*" is "bss" or not specified, then the common area will appear in either the `bss` or the `data` segment, depending on how *symbol_name* is defined elsewhere. These are the only choices currently supported.
`.reserve`	*symbol_name*, *size* [,"*segment*"[,*boundary*]]	Defines symbol *symbol_name*, and reserves *size* bytes of space for it in segment *segment* (optionally aligned on a *boundary*-byte address boundary). This is equivalent to: `.seg` "*segment*" [`.align` *boundary*] *symbol_name*: `.skip` *size* `.seg` "< *previous segment*>" If "*segment*" is not specified, space is reserved in the current segment.
`.empty`		Used in the delay slot of a Control Transfer Instruction (CTI), this suppresses assembler complaints about the next instruction's presence in a delay slot. Some instructions should not be in the delay slot of a CTI. See the *SPARC Architecture Manual* for details.

Table A-1 *List of Pseudo-Operations— Continued*

Mnemonic	*Argument(s)*	*Description*
`.proc`	*n*	Signals the beginning of a "procedure" (unit of optimization) to the peephole optimizer in the Sun-4 assembler; *n* specifies which registers will contain useful information upon return from the procedure, as follows: 0 no return value 6 return value in `%f0` 7 return value in `%f0` and `%f1` *(other)* return value in `%i0` (caller's `%o0`) The pseudo-operation `.proc` may be produced by code generators for higher-level languages. *See note below.*
`.stabs`	"*string*", *const4*, 0, *const16*, *const32*	Inserts a symbol table entry consisting of "*string*", followed by a 4-bit constant *const4*, a literal zero, a 16-bit constant *const16*, and a 32-bit constant *const32*. Used by Sun compilers only to pass information through the object file to symbolic debuggers.
`.stabn`	*const4*, 0, *const16*, *const32*	Inserts a symbol table entry consisting of a 4-bit numeric entry *const4*, followed by a literal zero, a 16-bit constant *const16*, and a 32-bit constant *const32*. Used by Sun compilers only to pass line-number information through the object file to symbolic debuggers.
`.stabd`	*const4*, 0, *const16*	Inserts a symbol-table entry consisting of a 4-bit numeric entry *const4*, followed by a literal zero and a 16-bit constant *const16*. Used by Sun compilers only to pass location-counter information through the object file to symbolic debuggers.
=	*symbol_name = constant_expression*	Assigns the value of *constant_expression* to *symbol_name*.

NOTE *Since peephole optimization is not performed on hand-written assembly-language code, there is no need for* `.proc` *statements in such code.*

B

The Sun-4 Assembler

You invoke `as` as follows:

```
as [options] [inputfile] ...
```

`as` translates the assembly language source files, *inputfile* into an executable object file, *objfile*. The Sun-4 assembler recognizes the filename argument '-' as the standard input.

All undefined symbols in the assembly are treated as global.

The Sun-4 assembler supports macros, `#include` files, and symbolic substitution through use of the C preprocessor `cpp`. The assembler invokes the preprocessor before assembly begins if it has been specified from the command line as an option (see `-P` below).

B.1. `as` Options

`-L` Save defined labels beginning with an L, which are normally discarded to save space in the resultant symbol table. The compilers generate many such temporary labels.

`-R` Make the initialized data segment read-only by concatenating it to the text segment.

`-o` *objfile*
The next argument is taken as the name of the object file to be produced. If the `-o` flag isn't used, the object file is named `a.out`.

`-P` Run `cpp`, the C preprocessor, on the files being assembled. The preprocessor is run separately on each input file, not on their concatenation. The preprocessor output is passed to the assembler.

`-k` Generate position-independent code as required by

```
cc -pic/-PIC
```

WARNING ***Don't apply the `-k` flag to hand-coded assembler programs unless they are written to be position-independent.***

pseudo-operations, *continued*
- .optim, 19
- .proc, 20
- .quad, 20
- .reserve, 20
- .seg, 19
- .single, 20
- .skip, 19
- .stabd, 21
- .stabn, 21
- .stabs, 21
- .word, 19

R

register routines
- RESTORE, 17
- SAVE, 17

registers
- GLOBAL, 17
- OUT, 17

RESTORE, 17

S

SAVE, 17

segments
- bss, 19
- data, 19
- data1, 19
- text, 19

special symbols
- %cq, 4
- %csr, 4
- %fp, 4
- %fq, 4
- %fsr, 4
- %hi, 4
- %lo, 4
- %psr, 4
- %sp, 4
- %tbr, 4
- %wim, 4
- %y, 4

ST_BREAKPOINT, 13
ST_CLEAN_WINDOWS, 13
ST_DIV0, 13
ST_FIX_ALIGN, 13
ST_FLUSH_WINDOWS, 13
ST_INT_OVERFLOW, 13
ST_RANGE_CHECK, 13
statement syntax, 2
syntax, 1
- assembler, 1
- notation, 2
- statement, 2

synthetic instructions, 15 *thru* 16
- hardware equivalents, 15

T

text, 19
traps
- ST_BREAKPOINT, 13
- ST_CLEAN_WINDOWS, 13

traps, *continued*
- ST_DIV0, 13
- ST_RANGE_CHECK, 13
- ST_SYSCALL, 13
- ST_WINDOWS, 13