



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

<b>Chapter 1 Assembler Syntax .....</b>	<b>1</b>
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
<b>Chapter 2 Instruction-Set Mapping .....</b>	<b>7</b>
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

<b>Appendix A</b> Pseudo-Operations .....	<b>19</b>
<b>Appendix B</b> The Sun-4 Assembler .....	<b>23</b>
B.1. <code>as</code> Options .....	23
<b>Index</b> .....	<b>25</b>

---

## 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



---

## 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:

<code>\b</code>	backspace
<code>\f</code>	formfeed
<code>\n</code>	newline (linefeed)
<code>\r</code>	carriage return
<code>\t</code>	horizontal tab
<code>\nnn</code>	octal value <i>nnn</i>

**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 `£77` 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  $nb$ , and before its definition (forward reference) as  $nf$ .

## Special Symbols

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

Table 1-1 *Special Symbols*

<i>Symbol Object</i>	<i>Name</i>	<i>Comment</i>
general-purpose registers	<code>%r0 ... %r31</code>	
general-purpose global registers	<code>%g0 ... %g7</code>	<i>(same as %r0 ... %r7)</i>
general-purpose "out" registers	<code>%o0 ... %o7</code>	<i>(same as %r8 ... %r15)</i>
general-purpose "local" registers	<code>%l0 ... %l7</code>	<i>(same as %r16 ... %r23)</i>
general-purpose "in" registers	<code>%i0 ... %i7</code>	<i>(same as %r24 ... %r31)</i>
stack-pointer register	<code>%sp</code>	<i>(%sp ≡ %o6 ≡ %l4)</i>
frame-pointer register	<code>%fp</code>	<i>(%fp ≡ %i6 ≡ %o30)</i>
floating-point registers	<code>%f0 ... %f31</code>	
floating-point status register	<code>%fsr</code>	
front of floating-point queue	<code>%fq</code>	
coprocessor registers	<code>%c0 ... %c31</code>	
coprocessor status register	<code>%csr</code>	
coprocessor queue	<code>%cq</code>	
program status register	<code>%psr</code>	
trap vector base address register	<code>%tbr</code>	
window invalid mask	<code>%wim</code>	
Y register	<code>%y</code>	
unary operators	<code>%lo</code> <code>%hi</code>	<i>(extracts least significant 10 bits)</i> <i>(extracts most significant 22 bits)</i>

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:

<i>Binary</i>	<i>Operators</i>	<i>Unary</i>	<i>Operators</i>
+	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.

## 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

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

Table 2-1 Notation—Continued

<i>Symbol</i>	<i>Describes</i>	<i>Comment</i>
<i>reg_or_imm</i>	<i>reg</i> <sub><i>rs2</i></sub> <i>const13</i>	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	jmp1	address, reg <sub>rd</sub>	Jump and link		
LDSB	ldsb	[address], reg <sub>rd</sub>	Load signed byte	(reg <sub>rd</sub> must be even)	
LDSH	ldsh	[address], reg <sub>rd</sub>	Load signed halfword		
LDSTUB	ldstub	[address], reg <sub>rd</sub>	Load-store unsigned byte		
LDUB	ldub	[address], reg <sub>rd</sub>	Load unsigned byte		
LDUH	lduh	[address], reg <sub>rd</sub>	Load unsigned halfword		
LD	ld	[address], reg <sub>rd</sub>	Load word		
LDD	ldd	[address], reg <sub>rd</sub>	Load double word		
LDF	ld	[address], freg <sub>rd</sub>	Load floating-point register		
LDFSR	ld	[address], %fsr			
LDDF	ldd	[address], freg <sub>rd</sub>	Load double floating-point		
LDC	ld	[address], creg <sub>rd</sub>	Load coprocessor		
LDCSR	ld	[address], %csr			
LDDC	ldd	[address], creg <sub>rd</sub>	Load double coprocessor		
LDSBA	ldsba	[regaddr] asi, reg <sub>rd</sub>	Load signed byte from alternate space		(reg <sub>rd</sub> must be even)
LDSHA	ldsha	[regaddr] asi, reg <sub>rd</sub>			
LDUBA	lduba	[regaddr] asi, reg <sub>rd</sub>			
LDUHA	lduha	[regaddr] asi, reg <sub>rd</sub>			
LDA	lda	[regaddr] asi, reg <sub>rd</sub>			
LDDA	ldda	[regaddr] asi, reg <sub>rd</sub>			
LDSTUBA	ldstuba	[regaddr] asi, reg <sub>rd</sub>			



Table 2-2 SPARC to Assembly Language Mapping—Continued

SPARC	Mnemonic	Argument List	Name	Comments
MULScc	mulbcc	$reg_{rs1}, reg\_or\_imm, reg_{rd}$	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	$\%asr_n, reg_{rd}$ $\%y, reg_{rd}$ $\%psr, reg_{rd}$ $\%wim, reg_{rd}$ $\%tbr, reg_{rd}$		(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	stb sth st std st std st	$regaddr, [address]$ $regaddr, [address]$ $reg_{rd}, [address]$ $reg_{rd}, [address]$ $freg_{rd}, [address]$ $freg_{rd}, [address]$ $\%fsr, [address]$	Store byte.	(synonyms: stub, stsb) (synonyms: stuh, stsh)  ( $reg_{rd}$ must be even)
STDFQ	std	$\%fq, [address]$	Store floating-point status register 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	Trap on integer condition code. (See note.)	(trap never)  (synonym: tnz) (synonym: tz)          (synonym: tcc) (synonym: tcs)          (synonym: t)
TADDcc TSUBcc TADDccTV	taddcc tsubcc taddcctv	$reg_{rs1}$ , $reg\_or\_imm$ , $reg_{rd}$ $reg_{rs1}$ , $reg\_or\_imm$ , $reg_{rd}$ $reg_{rs1}$ , $reg\_or\_imm$ , $reg_{rd}$	Tagged add and modify icc Tagged add and modify icc and trap on overflow	
TSUBccTV	tsubcctv	$reg_{rs1}$ , $reg\_or\_imm$ , $reg_{rd}$		
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	$reg\_or\_imm, \%asr_{rs1}$		(see synthetic instructions) (see synthetic instructions) (see synthetic instructions) (see synthetic instructions)
WRY	wr	$reg_{rs1}, reg\_or\_imm, \%y$		
WRPSR	wr	$reg_{rs1}, reg\_or\_imm, \%psr$		
WRWIM	wr	$reg_{rs1}, reg\_or\_imm, \%wim$		
WRTBR	wr	$reg_{rs1}, reg\_or\_imm, \%tbr$		
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

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

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 `cpopn` 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 `cpopn` instruction causes a `cp_disabled` trap.

The conditions causing a `cp_exception` trap are coprocessor-dependent.

**NOTE** A non-`cpopn` (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	(may modify ccc's)
CPop2	cpop2	$opd, reg_{rs1}, reg_{rs2}, reg_{rd}$	Coprocessor operation	

## 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
<code>cmp</code> $reg_{rs1}, reg\_or\_imm$	<code>subcc</code> $.reg_{rs1}, reg\_or\_imm, \%g0$	(compare)
<code>jmp</code> $address$	<code>jmp1</code> $address, \%g0$	
<code>call</code> $reg\_or\_imm$	<code>jmp1</code> $reg\_or\_imm, \%o7$	
<code>tst</code> $reg_{rs1}$	<code>orcc</code> $reg_{rs1}, \%g0, \%g0$	(test)
<code>ret</code>	<code>jmp1</code> $\%i7+8, \%g0$	(return from subroutine)
<code>retl</code>	<code>jmp1</code> $\%o7+8, \%g0$	(return from leaf subroutine)
<code>restore</code>	<code>restore</code> $\%g0, \%g0, \%g0$	(trivial restore)
<code>save</code>	<code>save</code> $\%g0, \%g0, \%g0$	(trivial save) Warning: trivial save should only be used in kernel code!
<code>set</code> $value, reg_{rd}$	<code>or</code> $\%g0, value, reg_{rd}$	(if $-4096 \leq value \leq 4095$ )

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

Synthetic Instruction		Hardware Equivalent(s)		Comment
set	value, reg <sub>rd</sub>	sethi	%hi (value), reg <sub>rd</sub>	(if ((value&0x1ff) == 0))
set	value, reg <sub>rd</sub>	sethi or	%hi (value), reg <sub>rd</sub> ; reg <sub>rd</sub> , %lo (value), reg <sub>rd</sub>	(otherwise)  Warning: do not use set in an instruction's delay slot.
not	reg <sub>rs1</sub> , reg <sub>rd</sub>	xnor	reg <sub>rs1</sub> , %g0, reg <sub>rd</sub>	(one's complement)
not	reg <sub>rd</sub>	xnor	reg <sub>rd</sub> , %g0, reg <sub>rd</sub>	(one's complement)
neg	reg <sub>rs2</sub> , reg <sub>rd</sub>	sub	%g0, reg <sub>rs2</sub> , reg <sub>rd</sub>	(two's complement)
neg	reg <sub>rd</sub>	sub	%g0, reg <sub>rd</sub> , reg <sub>rd</sub>	(two's complement)
inc	reg <sub>rd</sub>	add	reg <sub>rd</sub> , 1, reg <sub>rd</sub>	(increment by 1)
inc	const13, reg <sub>rd</sub>	add	reg <sub>rd</sub> , const13, reg <sub>rd</sub>	(increment by const13)
inccc	reg <sub>rd</sub>	addcc	reg <sub>rd</sub> , 1, reg <sub>rd</sub>	(increment by 1 and set icc)
inccc	const13, reg <sub>rd</sub>	addcc	reg <sub>rd</sub> , const13, reg <sub>rd</sub>	(increment by const13 and set icc)
dec	reg <sub>rd</sub>	sub	reg <sub>rd</sub> , 1, reg <sub>rd</sub>	(decrement by 1)
dec	const13, reg <sub>rd</sub>	sub	reg <sub>rd</sub> , const13, reg <sub>rd</sub>	(decrement by const13)
deccc	reg <sub>rd</sub>	subcc	reg <sub>rd</sub> , 1, reg <sub>rd</sub>	(decrement by 1 and set icc)
deccc	const13, reg <sub>rd</sub>	subcc	reg <sub>rd</sub> , const13, reg <sub>rd</sub>	(decrement by const13 and set icc)
btst	reg_or_imm, reg <sub>rs1</sub>	andcc	reg <sub>rs1</sub> , reg_or_imm, %g0	(bit test)
bset	reg_or_imm, reg <sub>rd</sub>	or	reg <sub>rd</sub> , reg_or_imm, reg <sub>rd</sub>	(bit set)
bclr	reg_or_imm, reg <sub>rd</sub>	andn	reg <sub>rd</sub> , reg_or_imm, reg <sub>rd</sub>	(bit clear)
btog	reg_or_imm, reg <sub>rd</sub>	xor	reg <sub>rd</sub> , reg_or_imm, reg <sub>rd</sub>	(bit toggle)
clr	reg <sub>rd</sub>	or	%g0, %g0, reg <sub>rd</sub>	(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 <sub>rd</sub>	or	%g0, reg_or_imm, reg <sub>rd</sub>	
mov	%y, reg <sub>rs1</sub>	rd	%y, reg <sub>rs1</sub>	
mov	%psr, reg <sub>rs1</sub>	rd	%psr, reg <sub>rs1</sub>	
mov	%wim, reg <sub>rs1</sub>	rd	%wim, reg <sub>rs1</sub>	
mov	%tbr, reg <sub>rs1</sub>	rd	%tbr, reg <sub>rs1</sub>	
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 CALLs or JMLs 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.





## Pseudo-Operations

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

Table A-1 *List of Pseudo-Operations*

<i>Mnemonic</i>	<i>Argument(s)</i>	<i>Description</i>
.alias		Turns off preceding .noalias. (Compiler-generated only.)
.noalias	<i>%reg1, %reg2</i>	<i>%reg1</i> and <i>%reg2</i> 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	<i>n</i>	Increments the location counter by <i>n</i> , which allocates <i>n</i> bytes of empty space in the current segment.
.align	<i>boundary</i>	Aligns the location counter on a 0-mod- <i>boundary</i> boundary; <i>boundary</i> may be 1 (which has no effect), 2, 4, or 8.
.byte	<i>8bitval</i> [, <i>8bitval</i> ] *	Generates (a sequence of) initialized bytes in the current segment.
.half	<i>16bitval</i> [, <i>16bitval</i> ] *	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

<i>Mnemonic</i>	<i>Argument(s)</i>	<i>Description</i>
.word	<i>32bitval</i> [, <i>32bitval</i> ]*	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	<i>0rfloatval</i> [, <i>0rfloatval</i> ]*	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	<i>0rfloatval</i> [, <i>0rfloatval</i> ]*	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	<i>0rfloatval</i> [, <i>0rfloatval</i> ]*	Generates (a sequence of) initialized quad-precision floating-point values in the current segment (.quad currently generates quad-precision values with only <i>double-precision</i> significance). The location counter must already be aligned on a doubleword boundary (use .align 8).
.global	<i>symbol_name</i> [, <i>symbol_name</i> ]*	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 <i>before</i> the definition.
.common	<i>symbol_name, size</i> [, " <i>segment</i> "]	Declares the name and size (in bytes) of a FORTRAN-style COMMON area. If " <i>segment</i> " is "bss" or not specified, then the common area will appear in either the bss or the data segment, depending on how <i>symbol_name</i> is defined elsewhere. These are the only choices currently supported.
.reserve	<i>symbol_name, size</i> [, " <i>segment</i> " [, <i>boundary</i> ]]	Defines symbol <i>symbol_name</i> , and reserves <i>size</i> bytes of space for it in segment <i>segment</i> (optionally aligned on a <i>boundary</i> -byte address boundary). This is equivalent to: <pre>         .seg    "<i>segment</i>"             [.align <i>boundary</i>]         <i>symbol_name</i>: .skip    <i>size</i>             .seg    "&lt; <i>previous segment</i>&gt;" </pre> If " <i>segment</i> " 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 <i>SPARC Architecture Manual</i> for details.

Table A-1 List of Pseudo-Operations—Continued

<i>Mnemonic</i>	<i>Argument(s)</i>	<i>Description</i>
<code>.proc</code>	<i>n</i>	Signals the beginning of a “procedure” (unit of optimization) to the peephole optimizer in the Sun-4 assembler; <i>n</i> 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 ( <i>other</i> ) return value in %i0 (caller’s %o0) The pseudo-operation <code>.proc</code> may be produced by code generators for higher-level languages. <i>See note below.</i>
<code>.stabs</code>	<code>"string", const4, 0, const16, const32</code>	Inserts a symbol table entry consisting of <code>"string"</code> , followed by a 4-bit constant <code>const4</code> , a literal zero, a 16-bit constant <code>const16</code> , and a 32-bit constant <code>const32</code> . Used by Sun compilers only to pass information through the object file to symbolic debuggers.
<code>.stabn</code>	<code>const4, 0, const16, const32</code>	Inserts a symbol table entry consisting of a 4-bit numeric entry <code>const4</code> , followed by a literal zero, a 16-bit constant <code>const16</code> , and a 32-bit constant <code>const32</code> . Used by Sun compilers only to pass line-number information through the object file to symbolic debuggers.
<code>.stabd</code>	<code>const4, 0, const16</code>	Inserts a symbol-table entry consisting of a 4-bit numeric entry <code>const4</code> , followed by a literal zero and a 16-bit constant <code>const16</code> . Used by Sun compilers only to pass location-counter information through the object file to symbolic debuggers.
<code>=</code>	<code>symbol_name = constant_expression</code>	Assigns the value of <code>constant_expression</code> to <code>symbol_name</code> .

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



---

## 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.*

- I*path*
- D*name*
- D*name=def*
- U*name*

When `-P` is in effect, these options are passed to the `cpp` preprocessor, without interpretation by `as`. Otherwise, they are ignored.

- O[*n*]

Enable peephole optimization corresponding to optimization level *n* (1 if *n* is not specified) of the Sun high-level language compilers. This option can be used safely only when assembling code produced by a Sun compiler.

- S[*C*]

Produce a disassembly of the emitted code to the standard output. This is most useful in combination with the `-O` option, to review optimized code. Adding the character *C* to the option prevents comment lines from appearing in the output.

---

# Index

## *Special Characters*

atof(3), 3

## A

as, 23

assembler

- and `cpp`, 23
- case distinction, 3
- comments, 3
- coprocessor instructions, 15
- delay slots, 5
- error messages, 5
- expressions, 5
- hardware instructions, 8
- `#include` files, 23
- instruction-set, 7
- invoking, 23
- labels, 4
- leaf procedure, 17
- lexical features, 2
- numbers, 3
- operators, 5
- optimization, 21
- options, 23
- pseudo-operations, 19
- register routines, 17
- special symbols, 4
- strings, 3
- symbol names, 3
- syntax, 1
- synthetic instructions, 15

## B

bss, 19

## C

COMMON, 19  
coprocessor instructions, 15  
`cp_disabled` trap, 15  
`cp_exception` trap, 15

## D

data, 19  
`data1`, 19  
delay slots, 5

## E

error messages, 5  
expressions, 5

## F

`£77`, 4

floating-point constant

- Or-inf, 3
- Or-nan, 3
- Orinf, 3
- Ornan, 3

floating-point instructions, 13 *thru* 15

## H

hardware instructions  
floating-point, 13  
integer, 8

## I

instruction-set mapping, 7

instructions

- coprocessor operate (CPop), 15
- integer instructions, 8 *thru* 13

## L

leaf procedure, 17

## O

operators, 5  
optimization, 17, 21  
options

- assembler, 23 *thru* 24

## P

pseudo-operations, 19

- =, 21
- .alias, 19
- .align, 19
- .ascii, 19
- .asciz, 19
- .byte, 19
- .common, 20
- .double, 20
- .empty, 6, 20
- .global, 20
- .half, 19
- .noalias, 19

pseudo-operations, *continued*

- .optim, 19
- .proc, 20
- .quad, 20
- .reserve, 20
- .seg, 19
- .single, 20
- .skip, 19
- .stabd, 21
- .stabs, 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\_DIVO, 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\_DIVO, 13
- ST\_RANGE\_CHECK, 13
- ST\_SYSCALL, 13
- ST\_WINDOWS, 13