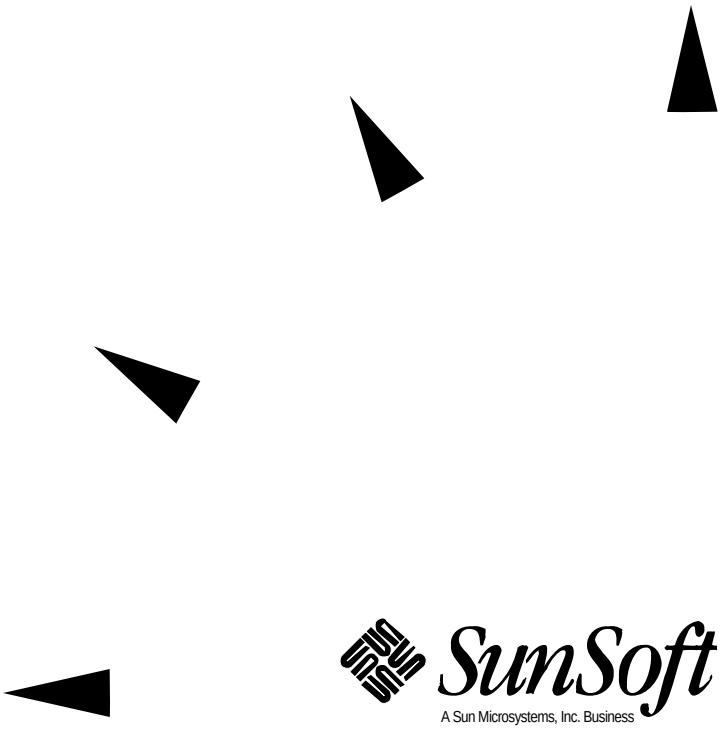
SPARC Assembly Language Reference Manual



2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Adobe PostScript

Contents

1. SPARC Assembler for SunOS 5.*x* **1**

Introduction 1

Operating Environment 1

SPARC Assembler for SunOS 4.1 Versus SunOS 5.*x* 2

New Labeling Format 2

New Object File Format 2

Pseudo-Operations 2

New Command Line Options 4

Other References 5

2. Assembler Syntax **7**

Syntax Notation 7

Assembler File Syntax 7

Lines Syntax 8

Statement Syntax 8

Lexical Features 8

Case Distinction 8
Comments 8
Labels 8
Numbers 9
Strings 9
Symbol Names 10
Special Symbols 11
Operators and Expressions 13
Assembler Error Messages 14
3. Extensible and Linking Format 15
ELF Header 16
Sections 17
Section Header 18
Predefined User Sections 22
Predefined Non-User Sections 24
Locations 25
Addresses 25
Relocation Tables 25
Symbol Tables 25
String Tables 27
Assembler Directives 28
Section Control Directives 28
Symbol Attribute Directives 28
Assignment Directive 29

Data Generating Directives 29
4. Converting Files to the New Format 31
Introduction 31
Conversion Instructions 31
Examples. 32
5. Instruction-Set Mapping 33
Table Notation 33
Integer Instructions 36
Floating-Point Instruction. 47
Coprocessor Instructions 49
Synthetic Instructions 50
A. Pseudo-Operations 55
Alphabetized Listing with Descriptions 55
B. Examples of Pseudo-Operations 65
C. Using the Assembler Command Line 69
Assembler Command Line 69
Assembler Command Line Options 70
Disassembling Object Code 72
D. An Example Language Program 73
Index 79

Tables

Table 2-1	Escape Codes Recognized in Strings	10
Table 2-2	Special Symbol Names	11
Table 2-3	Operators Recognized in Constant Expressions	13
Table 3-1	Reserved Object File Types	17
Table 3-2	Section Attribute Flags	19
Table 3-3	Section Types	20
Table 3-4	Predefined User Sections	22
Table 3-5	Predefined Non-User Sections	24
Table 3-6	Symbol Types	26
Table 3-7	Symbol Bindings	27
Table 5-1	Notations Used to Describe Instruction Sets	34
Table 5-2	Commonly Suffixed Notations	36
Table 5-3	SPARC to Assembly Language Mapping	37
Table 5-4	Floating-Point Instructions	47
Table 5-5	Coprocessor-Operate Instructions	49
Table 5-6	Synthetic Instruction to Hardware Instruction Mapping	50

SPARC Assembler for SunOS 5.x 1

Introduction

This chapter discusses the features of the SunOS 5.*x*™ SPARC® assembler. This document is distributed by SunSoft, Inc. as part of SunSoft's developer documentation set with every SunOS operating system release.

This document is also distributed with the on-line documentation set for the convenience of SPARCworks™ and SPARCompiler™ 3.0.1 users who have products that run on the SunOS 5.*x* operating system. It is included as part of the SPARCworks/SPARCompiler 3.0.1 Common Tools and Related Material AnswerBook, which is the on-line information retrieval system.

Operating Environment

The SunOS SPARC assembler runs under SunOS 5.*x* operating system or the Solaris™ 2.x operating environment. SunOS 5.*x* refers to SunOS 5.2 and later releases. Solaris 2.*x* refers to Solaris 2.2 and later releases.

The current release is SunOS 5.3.2 or Solaris 2.3.2. In the context of the SPARCworks and SPARCompiler products from SunSoft, the release refers to the current SPARCworks and SPARCompiler 3.0.1 products that run on SunOS 5.2 or Solaris 2.2 and later releases respectively.

1

SPARC Assembler for SunOS 4.1 Versus SunOS 5.x

This section describes the differences between the SunOS 4.1 SPARC assembler and the SunOS 5.*x* SPARC assembler.

New Labeling Format

- Symbol names beginning with a dot (.) are assumed to be local symbols.
- Names beginning with an underscore (_) are reserved by ANSI C.

New Object File Format

The type of object files created by the SPARC assembler are now ELF (*Extensible and Linking Format*) files. These relocatable object files hold code and data suitable for linking with other object files to create an executable file or a shared object file, and are the assembler normal output.

Pseudo-Operations

See Appendix A, "Pseudo-Operations," for a detailed description of the pseudo-operations (pseudo-ops) listed in this section.

New Pseudo-Ops

`.file`

Specifies the name of the source file associated with the object file.

`.local`

Declares each symbol in the list to be local.

`.nonvolatile`

Defines the end of a block of instructions which should not be modified at optimization time. This is the companion to **.volatile** pseudo-op.

`.popsection`

Makes the previous section the new current section.

`.pushsection`

Makes the named section the current section.

`.section`

Specifies information about the object file, including program and control information.

`.size`

Declares the symbol size in bytes.

`.type`

Declares the type of symbol.

`.uahalf`

Generates a (sequence of) 16-bit value(s).

`.uaword`

Generates a (sequence of) 32-bit value(s).

`.version`

Identifies the minimum assembler version necessary to assemble the input file.

`.volatile`

Defines the beginning of a block of instructions which should not be modified at optimization time. This is the companion to **.nonvolatile** pseudo-op.

`.weak`

Declares each symbol in the list to be defined as a “weak” global symbol.

Changed Pseudo-Ops

`.common`

Currently, only `".bss"` (uninitialized data segments) is supported for the section name. (`".data"` is not currently supported.)

`.global`

Does not need to occur before a definition, or tentative definition, of the specified symbol.

`.seg`

The SunOS 4.1 SPARC assembler directive `".seg"`:
`.seg "test", .seg "data" .seg "data1", .seg "bss"`,
would be interpreted as the following in the SunOS 5.*x* SPARC assembler:
`.section ".text, .section ".data",`
`.section ".data1", .section ".bss"`.

Note – This pseudo-op is being maintained for compatibility with existing SunOS 4.1 SPARC assembly language programs only. The suggested usage for the SunOS 5.*x* SPARC assembler is `.section`.

New Command Line Options

See Appendix C, "Using the Assembler Command Line," for a detailed description of the new command line options listed in this section.

`-K PIC`

Generates position-independent code. This option has the same functionality as the `-k` option under the SunOS 4.1 SPARC assembler.

`-m`

Runs `m4` macro preprocessing on input.

`-Q`

Produces information about the object file.

`-q`

Causes the assembler to perform a quick assembly by disabling context-dependent error checking.

`-b`

Generates SPARCworks SourceBrowser information.

`-T`

This is a migration option so that SunOS 4.1 assembly files can be assembled to run on a SunOS 5.*x* system.

`-V`

Writes the assembler version information on the standard error output.

Other References

You should also become familiar with the following:

- Manual pages: `as`(1), `ld`(1), `cpp`(1, `elf`(3f), `dis`(1), `a.out`(1)
- SPARC Architecture Manual
- ELF-related sections of the *Programming Utilities Guide* manual
- SPARC Applications Binary Interface (ABI)

1

Assembler Syntax 2

The SunOS 5.*x* SPARC assembler takes assembly language programs, as specified in this document, and produces relocatable object files for processing by the SunOS 5.*x* SPARC link editor. The assembly language described in this document corresponds to the SPARC instruction set defined in the *SPARC Architecture Manual* and is intended for use on machines that use the SPARC architecture.

Syntax Notation

In the descriptions of assembly language syntax in this chapter:

- Brackets ([ ]) enclose optional items.
- Asterisks (*) indicate 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. Newline characters are not allowed in place of blanks.

Assembler File Syntax

The syntax of assembly language *files* is:

```
[line]*
```

2

Lines Syntax

The syntax of assembly language *lines* is:

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

Statement Syntax

The syntax of an assembly language *statement* is:

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

where:

label is a symbol name.

instruction is an encoded pseudo-op, synthetic instruction, or instruction.

Lexical Features

This section describes the lexical features of the assembler syntax.

Case Distinction

Uppercase and lowercase letters are distinct everywhere *except* in the names of special symbols. Special symbol names have no case distinction.

Comments

A comment is preceded by an exclamation mark character (!); the exclamation mark character and all following characters up to the end of the line are ignored. C language-style comments ("`/*…*/`") are also permitted and may span multiple lines.

Labels

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

Numeric labels may be defined repeatedly in an assembly file; 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`.

Numbers

Decimal, hexadecimal, and octal numeric constants are recognized and are written as in the C language. However, integer suffixes (such as `L`) are not recognized.

For floating-point pseudo-operations, floating-point constants are written with `0r` or `0R` (where `r` or `R` means `REAL`) followed by a string acceptable to `atof(3)`; that is, an optional sign followed by a non-empty string of digits with optional decimal point and optional exponent.

The special names `0rnan` and `0rinf` represent the special floating-point values *Not-A-Number* (NaN) and *INFinity. Negative Not-A-Number* and *Negative INFinity* are specified as `0r-nan` and `0r-inf`.

Note – The names of these floating-point constants begin with the digit zero, *not* the letter “O.”

Strings

A *string* is a sequence of characters quoted with either double-quote mark (") or single-quote mark (’) characters. The sequence must not include a *newline* character. 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 mark* characters for the ASCII value of a single character, and *double quote mark* characters for quoted-string operands such as used by pseudo-ops. An example of assembly code in the suggested style is:

```
add %g1,’a’-’A’,%g1 ! g1 + (’a’ - ’A’) --> g1
```

The escape codes described in Table 2-1, derived from ANSI C, are recognized in strings.

Table 2-1 Escape Codes Recognized in Strings

Escape Code	Description
`\a`	Alert
`\b`	Backspace
`\f`	Form feed
`\n`	Newline (line feed)
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\`*`nnn`*	Octal value *nnn*
`\x`*`nn...`*	Hexadecimal value *nn...*

Symbol Names

The syntax for a symbol *name* is:

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

In the above syntax:

- Uppercase and lowercase letters are distinct; the underscore (_), dollar sign ($), and dot (.) are treated as alphabetic characters.
- Symbol names that begin with a dot (.) are assumed to be local symbols. To simplify debugging, avoid using this type of symbol name in hand-coded assembly language routines.
- The symbol dot (.) is predefined and always refers to the address of the beginning of the current assembly language statement.

- External variable names beginning with the underscore character are reserved by the ANSI C Standard. Do *not* begin these names with the underscore; otherwise, the program will not conform to ANSI C and unpredictable behavior may result.

Special Symbols

Special symbol names begin with a *percentage sign* (%) to avoid conflict with user symbols. Table 2-2 lists these special symbol names.

Table 2-2 Special Symbol Names

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 = %r14`)
Frame-pointer register	`%fp`	(`%fp = %i6 = %r30`)
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`	

Table 2-2 Special Symbol Names (Continued)

Symbol Object	Name	Comment
Unary operators	`%lo`	Extracts least significant 10 bits
	`%hi`	Extracts most significant 22 bits
	`%r_disp32`	Used only in Sun compiler-generated code.
	`%r_plt32`	Used only in Sun compiler-generated code.
Ancillary state registers	`%asr1` … `%asr31`	

There is no case distinction in special symbols; for example,

```
%PSR
```

is equivalent to

```
%psr
```

The suggested style is to use lowercase letters.

The lack of case distinction allows for the use of non-recursive preprocessor substitutions, for example:

```
#define psr %PSR
```

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

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

To avoid ambiguity, enclose operands of the `%hi` or `%lo` operators in parentheses. For example:

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

Operators and Expressions

The operators described in Table 2-3 are recognized in constant expressions.

Table 2-3 Operators 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	Extract least significant 10 bits
%	Modulo	%hi	Extract most significant 22 bits
^	Exclusive OR	%r_disp32	Used in Sun compiler-generated code only to instruct the assembler to generate specific relocation information for the given expression.
<<	Left shift	%r_plt32	Used in Sun compiler-generated code only to instruct the assembler to generate specific relocation information for the given expression.
>>	Right shift		
&	Bitwise AND		
|	Bitwise OR		

Since these operators have the same precedence as in the C language, put expressions in parentheses to avoid ambiguity.

To avoid confusion with register names or with the %hi, %lo, %r_disp32, or %r_plt32 operators, the modulo operator % must *not* be immediately followed by a letter or digit. The modulo operator is typically followed by a space or left parenthesis character.

2

Assembler Error Messages

Messages generated by the assembler are generally self-explanatory and give sufficient information to allow correction of a problem.

Certain conditions will cause the assembler to issue warnings associated with delay slots following Control Transfer Instructions (CTI). These warnings are:

- Set synthetic instructions in delay slots
- Labels in delay slots
- Segments that end in control transfer instructions

These warnings point to places where a problem could exist. If you have intentionally written code this way, you can insert an `.empty` pseudo-operation immediately after the control transfer instruction.

The `.empty` pseudo-operation in a delay slot tells the assembler that the delay slot can be empty or can contain whatever follows because you have verified that either the code is correct or the content of the delay slot does not matter.

Extensible and Linking Format 3

The type of object files created by the SPARC assembler version for SunOS 5.*x* are now *Extensible and Linking Format* (ELF) files. These relocatable ELF files hold code and data suitable for linking with other object files to create an executable or a shared object file, and are the assembler normal output. The assembler may also write information to standard output (for example, under the `-S` option) and to standard error (for example, under the `-V` option). The SPARC assembler creates a default output file when standard input or multiple files are used.

The ELF object file format consists of:

- Header
- Sections
- Locations
- Addresses
- Relocation tables
- Symbol tables
- String tables

For more information, see Chapter 4, “Object Files,” in the *System V Application Binary Interface (SPARC™ Processor Supplement)* manual.

3

ELF Header

The *ELF header* is always located at the beginning of the ELF file. It describes the ELF file organization and contains the actual sizes of the object file control structures. The initial bytes of an ELF header specify how the file is to be interpreted.

The ELF header contains the following information:

ehsize – ELF header size in bytes.

entry – Virtual address at which the process is to start. A value of 0 indicates no associated entry point.

flag – Processor-specific flags associated with the file.

ident – Marks the file as an object file and provides machine-independent data to decode and interpret the file contents.

machine – Specifies the required architecture for an individual file. A value of 2 specifies SPARC.

phentsize – Size in bytes of entries in the program header table. All entries are the same size.

phnum – Number of entries in program header table. A value of 0 indicates the file has no program header table.

phoff – Program header table file offset in bytes. The value of 0 indicates no program header.

shentsize – Size in bytes of the section header. A section header is one entry in the section header table; all entries are the same size.

shnum – Number of entries in section header table. A value of 0 indicates the file has no section header table.

shoff – Section header table file offset in bytes. The value of 0 indicates no section header.

shstrndx – Section header table index of the entry associated with the section name string table. A value of `SHN_UNDEF` indicates the file does not have a section name string table.

type – Identifies the object file type. Table 3-1 describes the reserved object file types.

version – Identifies the object file version.

Table 3-1 Reserved Object File Types

Type	Value	Description
`none`	`0`	No file type
`rel`	`1`	Relocatable file
`exec`	`2`	Executable file
`dyn`	`3`	Shared object file
`core`	`4`	Core file
`loproc`	`0xff00`	Processor-specific
`hiproc`	`0xffff`	Processor-specific

Sections

A section is the smallest unit of an object that can be relocated. The following sections are commonly present in an ELF file:

- Section header
- Executable text
- Read-only data
- Read-write data
- Read-write uninitialized data (*section header* only)

Sections do not need to be specified in any particular order. The *current section* is the section to which code is generated.

These sections contain all other information in an object file and satisfy several conditions.

1. Every section must have one section header describing the section. However, a section header does not need to be followed by a section.
2. Each section occupies one contiguous sequence of bytes within a file. The section may be empty (that is, of zero-length).
3. A byte in a file can reside in only one section. Sections in a file cannot overlap.
4. An object file may have inactive space. The contents of the data in the inactive space are unspecified.

Sections can be added for multiple text or data segments, shared data, user-defined sections, or information in the object file for debugging.

Note – Not all of the sections need to be present.

Section Header

The *section header* allows you to locate all of the file sections. An entry in a section header table contains information characterizing the data in a section.

The section header contains the following information:

addr – Address at which the first byte resides if the section appears in the memory image of a process; the default value is 0.

addralign – Aligns the address if a section has an address alignment constraint; for example, if a section contains a double-word, the entire section must be ensured double-word alignment. Only 0 and positive integral powers of 2 are currently allowed. A value of 0 or 1 indicates no address alignment constraints.

entsize – Size in bytes for entries in fixed-size tables such as the symbol table.

*flag*s – One-bit descriptions of section attributes. Table 3-2 describes the section attribute flags.

Table 3-2 Section Attribute Flags

Flag	Default Value	Description
SHF_WRITE	0x1	Contains data that is writable during process execution.
SHF_ALLOC	0x2	Occupies memory during process execution. This attribute is *off* if a control section does not reside in the memory image of the object file.
SHF_EXECINSTR	0x4	Contains executable machine instructions.
SHF_MASKPROC	0xf0000000	Reserved for processor-specific semantics.

info – Extra information. The interpretation of this information depends on the section type, as described in Table 3-3.

link – Section header table index link. The interpretation of this information depends on the section type, as described in Table 3-3.

name – Specifies the section name. An index into the section header string table section specifies the location of a null-terminated string.

offset – Specifies the byte offset from the beginning of the file to the first byte in the section.

Note – If the section type is SHT_NOBITS, *offset* specifies the conceptual placement of the file.

---

size – Specifies the size of the section in bytes.

Note – If the section type is SHT_NOBITS, *size* may be non-zero; however, the section still occupies no space in the file.

---

type – Categorizes the section contents and semantics. Table 3-3 describes the section types.

Table 3-3 Section Types

Name	Value	Description	Interpretation by info	Interpretation by link
`null`	0	Marks section header as inactive.		
`progbits`	1	Contains information defined explicitly by the program.		
`symtab`	2	Contains a symbol table for link editing. This table may also be used for dynamic linking; however, it may contain many unnecessary symbols. Note: *Only one section of this type is allowed in a file*	One greater than the symbol table index of the last local symbol.	The section header index of the associated string table.
`strtab`	3	Contains a string table. A file may have multiple string table sections.		
`rela`	4	Contains relocation entries with explicit addends. A file may have multiple relocation sections.	The section header index of the section to which the relocation applies.	The section header index of the associated symbol table.
`hash`	5	Contains a symbol rehash table. Note: *Only one section of this type is allowed in a file*	0	The section header index of the symbol table to which the hash table applies.
`dynamic`	6	Contains dynamic linking information. Note: *Only one section of this type is allowed in a file*	0	The section header index of the string table used by entries in the section.
`note`	7	Contains information that marks the file.		

Table 3-3 Section Types (Continued)

Name	Value	Description	Interpretation by info	Interpretation by link
nobits	8	Contains information defined explicitly by the program; however, a section of this type does not occupy any space in the file.		
rel	9	Contains relocation entries without explicit addends. A file may have multiple relocation sections.	The section header index of the section to which the relocation applies.	The section header index of the associated symbol table.
shlib	10	Reserved.		
dynsym	11	Contains a symbol table with a minimal set of symbols for dynamic linking. Note: *Only one section of this type is allowed in a file*	One greater than the symbol table index of the last local symbol.	The section header index of the associated string table.
loproc hiproc	0x70000000 0x7*fffffff*	Lower and upper bound of range reserved for processor-specific semantics.		
louser hiuser	0x80000000 0x*ffffffff*	Lower and upper bound of range reserved for application programs. Note: *Section types in this range may be used by an application without conflicting with system-defined section types.*		

Note – Some section header table indexes are reserved and the object file will not contain sections for these special indexes.

Predefined User Sections

A section that can be manipulated by the section control directives is known as a *user section*. You can use the section control directives to change the user section in which code or data is generated. Table 3-4 lists the predefined user sections that can be named in the section control directives.

Table 3-4 Predefined User Sections

Section Name	Description
`".bss"`	Section contains uninitialized read-write data.
`".comment"`	Comment section.
`".data" & ".data1"`	Section contains initialized read-write data.
`".debug"`	Section contains debugging information.
`".fini"`	Section contains runtime finalization instructions.
`".init"`	Section contains runtime initialization instructions.
`".rodata" & ".rodata1"`	Section contains read-only data.
`".text"`	Section contains executable text.
`".line"`	Section contains line # info for symbolic debugging.
`".note"`	Section contains note information.

Creating an `.init` *Section in an Object File*

The `.init` sections contain codes that are to be executed before the the main program is executed. To create an `.init` section in an object file, use the assembler pseudo-ops shown in Code Example 3-1.

```
    .section ".init"
    .align   4
    <instructions>
```

Code Example 3-1 Creating an `.init` Section

At link time, the `.init` sections in a sequence of `.o` files are concatenated into an `.init` section in the linker output file. The code in the `.init` section are executed before the main program is executed.

Note – The codes are executed inside a stack frame of 96 bytes. Do not reference or store to locations that are greater than `%sp+96` in the `.init` section.

Creating a `.fini` *Section in an Object File*

`.fini` sections contain codes that are to be executed after the the main program is executed. To create an `.fini` section in an object file, use the assembler pseudo-ops shown in Code Example 3-2.

```
    .section ".fini"
    .align   4
    <instructions>
```

Code Example 3-2 Creating an `.fini` Section

At link time, the `.fini` sections in a sequence of `.o` files are concatenated into a `.fini` section in the linker output file. The codes in the `.fini` section are executed after the main program is executed.

Note – The codes are executed inside a stack frame of 96 bytes. Do not reference or store to locations that are greater than `%sp+96` in the `.fini` section.

Predefined Non-User Sections

Table 3-5 lists sections that are predefined but cannot not be named in the section control directives because they are not under user control.

Table 3-5 Predefined Non-User Sections

Section Name	Description
".dynamic"	Section contains dynamic linking information.
".dynstr	Section contains strings needed for dynamic linking.
".dynsym"	Section contains the dynamic linking symbol table.
".got"	Section contains the global offset table.
".hash"	Section contains a symbol hash table.
".interp	Section contains the path name of a program interpreter.
".plt"	Section contains the procedure linking table.
"relname & .relaname"	Section containing relocation information. *name* is the section to which the relocations apply. e.g. ".rel.text", ".rela.text".
".shstrtab"	String table for the section header table names.
".strtab"	Section contains the string table.
".symtab"	Section contains a symbol table.

Locations

A *location* is a specific position within a section. Each location is identified by a section and a byte offset from the beginning of the section. The *current location* is the location within the current section where code is generated.

A *location counter* tracks the current offset within each section where code or data is being generated. When a section control directive (for example, `.section` pseudo-op) is processed, the location information from the location counter associated with the new section is assigned to and stored with the name and value of the current location.

The current location is updated at the end of processing each statement, but can be updated during processing of data-generating assembler directives (for example, the `.word` pseudo-op).

Note – Each section has one location counter; if more than one section is present, only one location can be current at any time.

Addresses

Locations represent *addresses in memory* if a section is allocatable; that is, its contents are to be placed in memory at program runtime. Symbolic references to these locations must be changed to addresses by the SPARC link editor.

Relocation Tables

The assembler produces a companion *relocation table* for each relocatable section. The table contains a list of relocations (that is, adjustments to data in the section) to be performed by the link editor.

Symbol Tables

A *symbol table* contains information to locate and relocate symbolic definitions and references. The SPARC assembler creates a symbol table section for the object file. It makes an entry in the symbol table for each symbol that is defined or referenced in the input file and is needed during linking. The symbol table is then used by the SPARC link editor during relocation. The section header contains the symbol table index for the first non-local symbol.

A symbol table contains the following information:

name – Index into the object file symbol string table. A value of zero indicates the symbol table entry has no name; otherwise, the value represents the string table index that gives the symbol name.

value – Value of the associated symbol. This value is dependent on the context; for example, it may be an address, or it may be an absolute value.

size – Size of symbol. A value of 0 indicates that the symbol has either no size or an unknown size.

info – Specifies the symbol type and binding attributes. Table 3-6 and Table 3-7 describes these values.

other – Undefined meaning. Current value is 0.

shndx – Contains the section header table index to another relevant section, if specified. As a section moves during relocation, references to the symbol will continue to point to the same location because the value of the symbol will change as well.

Table 3-6 Symbol Types

Value	Type	Description
0	`notype`	Type not specified.
1	`object`	*Symbol* is associated with a data object; for example, a variable or an array.
2	`func`	*Symbol* is associated with a function or other executable code. When another object file references a function from a shared object, the link editor automatically creates a procedure linkage table entry for the referenced symbol.
3	`section`	*Symbol* is associated with a section. These types of symbols are primarily used for relocation.
4	`file`	Gives the name of the source file associated with the object file.
13 15	`locproc` `hiproc`	Values reserved for processor-specific semantics.

Table 3-7 Symbol Bindings

Value	Binding	Description
0	`local`	*Symbol* is defined in the object file and not accessible in other files. Local symbols of the same name may exist in multiple files.
1	`global`	*Symbol* is either defined externally or defined in the object file and accessible in other files.
2	`weak`	*Symbol* is either defined externally or defined in the object file and accessible in other files; however, these definitions have a lower precedence than globally defined symbols.
13 15	`loproc` `hiproc`	Values reserved for processor-specific semantics.

String Tables

A *string table* is a section which contains null-terminated variable-length character sequences, or strings, in the object file; for example, symbol names and file names. The strings are referenced in the section header as indexes into the string table section.

- A string table index may refer to any byte in the section.
- Empty string table sections are permitted; however, the index referencing this section must contain zero.

A string may appear multiple times and may also be referenced multiple times. References to substrings may exist, and unreferenced strings are allowed.

3

Assembler Directives

Assembler directives, or pseudo-operations (pseudo-ops), are commands to the assembler that may or may not result in the generation of code. The different types of assembler directives are:

- Section Control Directives
- Symbol Attribute Directives
- Assignment Directives
- Data Generating Directives
- Optimizer Directives

See Appendix A, "Pseudo-Operations," for a complete description of the pseudo-ops supported by the SPARC assembler.

Section Control Directives

When a section is created, a section header is generated and entered in the ELF object file section header table. The *section control pseudo-ops* allow you to make entries in this table. Sections that can be manipulated with the section control directives are known as *user sections*. You can also use the section control directives to change the user section in which code or data is generated.

Note – The *symbol table*, *relocation table*, and *string table* sections are created implicitly. The section control pseudo-ops cannot be used to manipulate these sections.

The section control directives also create a section symbol which is associated with the location at the beginning of each created section. The section symbol has an offset value of zero.

Symbol Attribute Directives

The *symbol attribute* pseudo-ops declare the symbol type and size and whether it is local or global.

Assignment Directive

The *assignment* directive associates the value and type of expression with the symbol and creates a symbol table entry for the symbol. This directive constitutes a *definition* of the symbol and, therefore, must be the only definition of the symbol.

Data Generating Directives

The *data generating* directives are used for allocating storage and loading values.

3

Converting Files to the New Format 4

Introduction

This chapter discusses how to convert existing SunOS 4.1 SPARC assembly files to the SunOS 5.*x* SPARC assembly file format.

Conversion Instructions

- Remove the leading underscore (_) from symbol names.
  The Solaris 2.*x* SPARCompilers do not prepend a leading underscore to symbol names in the users' programs, like the SPARCompilers that ran under SunOS 4.1.
- Prefix local symbol names ,with a dot (.) .
  Local symbol names in the SunOS 5.*x* SPARC assembly language begin with a dot (.) so that they will not conflict with user programs' symbol names.
- Change the usage of the pseudo-op `.seg` to `.section`
  e.g. Change `.seg "data"` to `.section ".data"`. See Appendix A, "Pseudo-Operations," for more information.

Note – The above conversions can be automatically achieved by passing the `-T` option to the assembler.

4

Examples

Figure 4-1 shows how to convert an existing 4.1 file to the new format. The lines that are different in the new format are marked with change bars.

Example 4.x File

```
    .seg    "data1"
    .align  4
L16:
    .ascii  "hello world\n"
    .seg    "text"
    .proc   04
    .global _main

    .align  4
_main:
    !#PROLOGUE# 0
    sethi   %hi(LF12),%g1
    add     %g1,%lo(LF12),%g1
    save    %sp,%g1,%sp
    !#PROLOGUE# 1
L14:
    .seg    "text"
    set     L16,%o0
    call    _printf,1
    nop
LE12:
    ret
    restore
    .optim  "-O~Q~R~S"
    LF12 = -96
    LP12 = 96
    LST12 = 96
    LT12 = 96
```

Converted to the New Format

```
    .section    ".data1"
    .align      4
.L16:
    .ascii      "hello world\n"
    .section    ".text"
    .proc       04
    .global     main

    .align      4
main:
    !#PROLOGUE# 0
    sethi       %hi(.LF12),%g1
    add         %g1,%lo(.LF12),%g1
    save        %sp,%g1,%sp
    !#PROLOGUE# 1
.L14:
    .section    ".text"
    set         .L16,%o0
    call        printf,1
    nop
.LE12:
    ret
    restore
    .optim      "-O~Q~R~S"
    .LF12 = -96
    .LP12 = 96
    .LST12 = 96
    .LT12 = 96
```

Figure 4-1 Converting a 4.x File to the New Format

Instruction-Set Mapping 5

The tables in this chapter describe the relationship between hardware instructions of the SPARC architecture, as defined in the *SPARC Processor Architecture Manual*, and the assembly language instruction set recognized by the SunOS 5.*x* SPARC assembler.

Table Notation

Table 5-1 describes the notation used in the tables in this chapter to describe the instruction set of the assembler. The following notations are commonly suffixed to assembler mnemonics (uppercase letters refer to SPARC architecture instruction names):

5

Table 5-1 Notations Used to Describe Instruction Sets

Notations	Describes	Comment
address	reg_{rs1} + reg_{rs2} reg_{rs1} + const13 reg_{rs1} – const13 const13 + reg_{rs1} const13	Address formed from register contents, immediate constant, or both.
asi		Alternate address space identifier; an unsigned 8-bit value. It can be the result of the evaluation of a symbol expression.
const13		A signed constant which fits in 13 bits. It can be the result of the evaluation of a symbol expression.
const22		A constant which fits in 22 bits. It can be the result of the evaluation of a symbol expression.
creg	%c0 ... %c31	Coprocessor registers.
freg	%f0 ... %f31	Floating-point registers.
imm7		A signed or unsigned constant that can be represented in 7 bits (it is in the range -64 ... 127). It can be the result of the evaluation of a symbol expression.
reg	%r0 ... %r31 %g0 ... %g7 %o0 ... %o7 %l0 ... %l7 %i0 ... %i7	General purpose registers. Same as %r0 ... %r7 (Globals) Same as %r8 ... %r15 (Outs) Same as %r16 ... %r23 (Locals) Same as %r24 ... %r31 (Ins)
reg_{rd}		Destination register.
reg_{rs1}, reg_{rs2}		Source register 1, source register 2.

Table 5-1 Notations Used to Describe Instruction Sets

Notations	Describes	Comment
`reg_or_imm`	reg_{rs2} `const13`	Value from either a single register, or an immediate constant.
`regaddr`	reg_{rs1} reg_{rs1} + reg_{rs2}	Address formed with register contents only.
`Software_trap_number`	reg_{rs1} + reg_{rs2} reg_{rs1} + imm7 reg_{rs1} - imm7 uimm7 imm7 + reg_{rs1}	A value formed from register contents, immediate constant, or both. The resulting value must be in the range 0.....127, inclusive.
`uimm7`		An unsigned constant that can be represented in 7 bits (it is in the range 0 ... 127). It can be the result of the evaluation of a symbol expression.

5

Integer Instructions

The notations described in Table 5-2 are commonly suffixed to assembler mnemonics (uppercase letters for architecture instruction names).

Table 5-2 Commonly Suffixed Notations

Notation	Description
`a`	Instructions that deal with alternate space
`b`	Byte instructions
`c`	Reference to coprocessor registers
`d`	Doubleword instructions
`f`	Reference to floating-point registers
`h`	Halfword instructions
`q`	Quadword instructions
`sr`	Status register

Table 5-3 outlines the correspondence between SPARC hardware integer instructions and SPARC assembly language instructions.

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.

Note – In Table 5-3,

- Braces ({ }) indicate optional arguments.
  Braces are not literally coded.
- Brackets ([ ]) indicate indirection: the contents of the addressed memory location are being read from or written to.
  Brackets are coded literally in the assembly language.
  Note that the usage of brackets described in Chapter 2, “Assembler Syntax” differ from the usage of these brackets.
- All `Bicc` and `Bfcc` instructions described may indicate that the annul bit is to be set by appending `",a"` to the opcode mnemonic; for example,

```
"bgeu,a label"
```

Table 5-3 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}		

Table 5-3 SPARC to Assembly Language Mapping *(Continued)*

SPARC	Mnemonic	Argument List	Name	Comments
Bicc	*bn*{,*a*}	label	Branch on integer condition codes	branch never
	bne{,*a*}	label		synonym: bnz
	be{,*a*}	label		synonym: bz
	bg{,*a*}	label		
	ble{,*a*}	label		
	bge{,*a*}	label		
	bl{,*a*}	label		
	bgu{,*a*}	label		
	bleu{,*a*}	label		
	bcc{,*a*}	label		synonym: bgeu
	bcs{,*a*}	label		synonym: blu
	bpos{,*a*}	label		
	bneg{,*a*}	label		
	bvc{,*a*}	label		
	bvs{,*a*}	label		
	ba{,*a*}	label		synonym: b
CALL	call	label label{,n}	Call subprogram	*n* = # of out registers used as arguments

Table 5-3 SPARC to Assembly Language Mapping *(Continued)*

SPARC	Mnemonic	Argument List	Name	Comments
CBccc	cbn{,a} cb3{,a} cb2{,a} cb23{,a} cb1{,a} cb13{,eo} cb12{,a} cb123{,a} cb0{,a} cb03{,a} cb02{,a} cb023{,a} cb01{,a} cb013{,a} cb012{,a} cba{,a}	label label label label label label label label label label label label label label label label	Branch on coprocessor condition codes	branch never

Table 5-3 SPARC to Assembly Language Mapping *(Continued)*

SPARC	Mnemonic	Argument List	Name	Comments
FBfcc	fbn{,a} fbu{,a} fbg{,a} fbug{,a} fbl{,a} fbul{,a} fblg{,a} fbne{,a} fbe{,a} fbue{,a} fbge{,a} fbuge{,a} fble{,a} fbule{,a} fbo{,a} fba{,a}	label label label label label label label label label label label label label label label label	Branch on floating-point condition codes	branch never synonym: fbnz synonym: fbz
FLUSH	flush	address	Instruction cache flush	
JMPL	jmpl	*address*, reg_{rd}	Jump and link	

Table 5-3 SPARC to Assembly Language Mapping *(Continued)*

SPARC	Mnemonic	Argument List	Name	Comments
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*$_{rd}$		
LDFSR	ld	*[address],* %fsr	Load floating-point register	
LDDF	ldd	*[address], freg*$_{rd}$	Load double floating-point	*freg*$_{rd}$ must be even
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 5-3 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	%asr*n*$_{rs1}$, *reg*$_{rd}$		1≤*n*≤31
RDY	rd	%y, *reg*$_{rd}$		See synthetic instructions
RDPSR	rd	%psr, *reg*$_{rd}$		See synthetic instructions
RDWIM	rd	%wim, *reg*$_{rd}$		See synthetic instructions
RDTBR	rd	%tbr, *reg*$_{rd}$		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	sdiv	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	Signed divide	
SDIVcc	sdivcc	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	Signed divide and modify icc	

Table 5-3 SPARC to Assembly Language Mapping *(Continued)*

SPARC	Mnemonic	Argument List	Name	Comments
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 register	
	sethi	%hi(*value*), *reg*$_{rd}$		See synthetic instructions
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	*reg*$_{rd}$, [address]	Store byte	Synonyms: stub, stsb
STH	sth	*reg*$_{rd}$, [address]	Store half-word	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]*		*freg*$_{rd}$ Must be even
STFSR	st	%fsr, *[address]*	Store floating-point status register	
STDFQ	std	%fq, *[address]*	Store double floating-point	
STC	st	*creg*$_{rd}$, *[address]*	Store coprocessor	
STDC	std	*creg*$_{rd}$, *[address]*		*creg*$_{rd}$ Must be even
STCSR	st	%csr, *[address]*		
STDCQ	std	%cq, *[address]*	Store double coprocessor	

Table 5-3 SPARC to Assembly Language Mapping *(Continued)*

SPARC	Mnemonic	Argument List	Name	Comments
STBA	stba	*reg*$_{rd}$ [regaddr]asi	Store byte into alternate space	Synonyms: stuba, stsba
STHA	stha	*reg*$_{rd}$ [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*$_{rd}$		

Table 5-3 SPARC to Assembly Language Mapping *(Continued)*

SPARC	Mnemonic	Argument List	Name	Comments
`Ticc`	`tn`	`software_trap_number`	Trap on integer condition code	Trap never
	`tne`	`software_trap_number`	**Note:** Trap numbers 16-31 are reserved for the user. Currently-defined trap numbers are those defined in `/usr/include/sys/trap.h`	Synonym: `tnz`
	`te`	`software_trap_number`		Synonym: `tz`
	`tg`	`software_trap_number`		
	`tle`	`software_trap_number`		
	`tge`	`software_trap_number`		
	`tl`	`software_trap_number`		
	`tgu`	`software_trap_number`		
	`tleu`	`software_trap_number`		
	`tlu`	`software_trap_number`		Synonym: `tcs`
	`tgeu`	`software_trap_number`		Synonym: `tcc`
	`tpos`	`software_trap_number`		
	`tneg`	`software_trap_number`		
	`tvc`	`software_trap_number`		
	`tvs`	`software_trap_number`		
	`ta`	`software_trap_number`		Synonym: `t`

Table 5-3 SPARC to Assembly Language Mapping *(Continued)*

SPARC	Mnemonic	Argument List	Name	Comments
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	
UMULcc	umulcc	*reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$	Unsigned multiply and modify icc	
UNIMP	unimp	const22	Illegal instruction	
WRASR	wr	*reg_or_imm*, %asr*n*$_{rs1}$		1≤*n*≤31
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*$_{rd}$		

Floating-Point Instruction

Table 5-4 shows floating-point instructions. In 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 5-4 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

* Types of Operands are denoted by the following lower-case letters:

i	integer
s	single
d	double
q	quad

Table 5-4 Floating-Point Instructions *(Continued)*

SPARC	Mnemonic*	Argument List	Description
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*$_{rd}$	

* Types of Operands are denoted by the following lower-case letters:

i	integer
s	single
d	double
q	quad

Table 5-4 Floating-Point Instructions *(Continued)*

SPARC	Mnemonic*	Argument List	Description
FCMPs	fcmps	*freg*$_{rs1}$, *freg*$_{rs2}$	Compare
FCMPd	fcmpd	*freg*$_{rs1}$, *freg*$_{rs2}$	
FCMPq	fcmpq	*freg*$_{rs1}$, *freg*$_{rs2}$	
FCMPEs	fcmpes	*freg*$_{rs1}$, *freg*$_{rs2}$	Compare, generate exception if not ordered
FCMPEd	fcmped	*freg*$_{rs1}$, *freg*$_{rs2}$	
FCMPEq	fcmpeq	*freg*$_{rs1}$, *freg*$_{rs2}$	

* Types of Operands are denoted by the following lower-case letters:

i	integer
s	single
d	double
q	quad

Coprocessor Instructions

All *coprocessor-operate* (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 also coprocessor-dependent. Coprocessor-operate instructions are described in Table 5-5.

If the EC (PSR_enable_coprocessor) field of the processor state register (PSR) is 0, or if a coprocessor is not present, a cpop*n* instruction causes a *cp_disabled* trap.

The conditions that cause a *cp_exception* trap are coprocessor-dependent.

Table 5-5 Coprocessor-Operate Instructions

SPARC	Mnemonic	Argument List	Name	Comments
CPop1	cpop1	*opc*, *reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$	Coprocessor operation	
CPop2	cpop2	*opc*, *reg*$_{rs1}$, *reg*$_{rs2}$, *reg*$_{rd}$	Coprocessor operation	May modify ccc

5

Synthetic Instructions

Table 5-6 describes the mapping of synthetic instructions to hardware instructions.

Table 5-6 Synthetic Instruction to Hardware Instruction Mapping

Synthetic Instruction		Hardware Equivalent(s)		Comment
`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
`call`	`reg_or_imm`	`jmpl`	*reg_or_imm*, `%o7`	
`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
`cmp`	`reg, reg_or_imm`	`subcc`	reg_{rs1}, *reg_or_imm*, `%g0`	Compare

Table 5-6 Synthetic Instruction to Hardware Instruction Mapping *(Continued)*

Synthetic Instruction	Hardware Equivalent(s)	Comment
`dec` *reg_{rd}* `dec` *const13, reg_{rd}* `deccc` *reg_{rd}* `deccc` *const13, reg_{rd}*	`sub` *reg_{rd}, 1, reg_{rd}* `sub` *reg_{rd}, const13, reg_{rd}* `subcc` *reg_{rd}, 1, reg_{rd}* `subcc` *reg_{rd}, const13, reg_{rd}*	Decrement by 1 Decrement by *const13* Decrement by 1 and set `icc` Decrement by *const13* and set `icc`
`dec` *reg_{rd}* `dec` *const13, reg_{rd}* `deccc` *reg_{rd}* `deccc` *const13, reg_{rd}*	`sub` *reg_{rd}, 1, reg_{rd}* `sub` *reg_{rd}, const13, reg_{rd}* `subcc` *reg_{rd}, 1, reg_{rd}* `subcc` *reg_{rd}, const13, reg_{rd}*	Decrement by 1 Decrement by *const13* Decrement by 1 and set `icc` Decrement by *const13* and set `icc`
`inc` *reg_{rd}* `inc` *const13, reg_{rd}* `inccc` *reg_{rd}* `inccc` *const13, reg_{rd}*	`add` *reg_{rd}, 1, reg_{rd}* `add` *reg_{rd}, const13, reg_{rd}* `addcc` *reg_{rd}, 1, reg_{rd}* `addcc` *reg_{rd}, const13, reg_{rd}*	Increment by 1 Increment by *const13* Increment by 1 and set `icc` Increment by *const13* and set `icc`
`jmp` address	`jmpl` *address*, `%g0`	

Table 5-6 Synthetic Instruction to Hardware Instruction Mapping *(Continued)*

Synthetic Instruction	Hardware Equivalent(s)	Comment
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	
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_{rs1}, reg_{rd}	sub %g0, reg_{rs2}, reg_{rd}	two's complement
neg reg_{rd}	sub %g0, reg_{rd}, reg_{rd}	two's complement
restore	restore %g0, %g0, %g0	trivial *restore*
ret	jmpl %i7+8, %g0	return from subroutine
retl	jmpl %o7+8, %g0	return from leaf subroutine
save	save %g0, %g0, %g0	trivial *save* **Warning –** *trivial save* should only be used in supervisor code!

Table 5-6 Synthetic Instruction to Hardware Instruction Mapping *(Continued)*

Synthetic Instruction	Hardware Equivalent(s)	Comment
`set` *value*`,`*reg*$_{rd}$	`or` `%g0,` *value*`,` *reg*$_{rd}$	if -4096 ≤ value ≤ 4095
`set` *value*`,`*reg*$_{rd}$	`sethi` `%hi(`*value*`),` *reg*$_{rd}$	if ((value & 0x3ff) == 0)
`set` *value*`,`*reg*$_{rd}$	`sethi` `%hi(`*value*`),` *reg*$_{rd}$`;` `or` *reg*$_{rd}$`, %lo(`*value*`),` *reg*$_{rd}$	otherwise **Warning –** Do not use the `set` synthetic instruction in an instruction delay slot.
`skipz` `skipnz`	`bnz,a .+8` `bz,a .+8`	if *z* is set, ignores next instruction if *z* is not set, ignores next instruction
`tst` `reg`	`orcc` *reg*$_{rs1}$`, %g0, %g0`	test

5

Pseudo-Operations A

The pseudo-operations listed in this appendix are supported by the SPARC assembler.

Alphabetized Listing with Descriptions

.alias

Turns off the effect of the preceding `.noalias` pseudo-op. (Compiler-generated only.)

.align *boundary*

Aligns the location counter on a boundary where `((“location counter” mod boundary)==0)`; *boundary* may be any power of 2.

.ascii "*string*" [,"*string*"]*

Generates the given sequence(s) of ASCII characters.

.asciz "*string*" [,"*string*"]*

Generates the given sequence(s) of ASCII characters. This pseudo-op appends a null (zero) byte to each *string.*

A

.byte *8bitval* [,*8bitval*]*

Generates (a sequence of) initialized bytes in the current segment.

.common *symbol*, *size* [,"*sect_name*"] [, *alignment*]

Provides a tentative definition of *symbol*. *Size* bytes are allocated for the object represented by *symbol*.

- If the symbol is not defined in the input file and is declared to be *local* to the file, the symbol is allocated in *sect_name* and its location is optionally aligned to a multiple of *alignment*. If *sect_name* is not given, the symbol is allocated in the uninitialized data section (*bss*). Currently, only `".bss"` is supported for the section name.
- If the symbol is not defined in the input file and is declared to be *global*, the SPARC link editor allocates storage for the symbol, depending on the definition of *symbol_name* in other files. Global is the default binding for common symbols.
- If the symbol is defined in the input file, the definition specifies the location of the symbol and the tentative definition is overridden.

.double 0r*floatval* [,0r*floatval*]*

Generates (a sequence of) initialized double-precision floating-point values in the current segment. *floatval* is a string acceptable to `atof(3)`; that is, an optional sign followed by a non-empty string of digits with optional decimal point and optional exponent.

.empty

Suppresses assembler complaints about the next instruction presence in a delay slot when used in the delay slot of a Control Transfer Instruction (CTI).

Some instructions should not be in the delay slot of a CTI. See the S*PARC Architecture Manual* for details.

.file "*string*"

Creates a symbol table entry where *string* is the symbol name and STT_FILE is the symbol table type. *string* specifies the name of the source file associated with the object file.

.global *symbol* [, *symbol*]*
.globl *symbol* [, *symbol*]*

Declares each *symbol* in the list to be global; that is, each symbol is either defined externally or defined in the input file and accessible in other files; default bindings for the symbol are overridden.

- A global symbol definition in one file will satisfy an undefined reference to the same global symbol in another file.
- Multiple definitions of a defined global symbol is not allowed. If a defined global symbol has more than one definition, an error will occur.

Note – This pseudo-op by itself does not define the symbol.

.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).

.ident "*string*"

Generates the null terminated string in a comment section. This operation is equivalent to:

```
.pushsection ".comment"
.asciz "string"
.popsection
```

A

.local *symbol* [, *symbol*]*

Declares each *symbol* in the list to be local; that is, each symbol is defined in the input file and not accessible in other files; default bindings for the symbol are overridden. These symbols take precedence over *weak* and *global* symbols.

Since local symbols are not accessible to other files, local symbols of the same name may exist in multiple files.

Note – This pseudo-op by itself does not define the symbol.

.noalias *%reg1, %reg2*

%reg1 and *%reg2* will not alias each other (that is, point to the same destination) until a `.alias` pseudo-op is issued. (Compiler-generated only.)

.nonvolatile

Defines the end of a block of instruction. The instructions in the block may not be permuted. This pseudo-op has no effect if:

- The block of instruction has been previously terminated by a Control Transfer Instruction (CTI) or a label
- There is no preceding `.volatile` pseudo-op

.optim *"string"*

This pseudo-op changes the optimization level of a particular function. (Compiler-generated only.)

.popsection

Removes the top section from the section stack. The new section on the top of the stack becomes the current section. This pseudo-op and its corresponding `.pushsection` command allow you to switch back and forth between the named sections.

.proc *n*

Signals the beginning of a *procedure* (that is, a unit of optimization) to the peephole optimizer in the SPARC assembler; *n* specifies which registers will contain the return value upon return from the procedure. (Compiler-generated only.)

.pushsection "*sect_name*" [, *attributes*]

Moves the named section to the top of the section stack. This new top section then becomes the current section. This pseudo-op and its corresponding `.popsection` command allow you to switch back and forth between the named sections.

.quad `0r`*floatval* [, `0r`*floatval*]*

Generates (a sequence of) initialized quad-precision floating-point values in the current segment. *floatval* is a string acceptable to `atof(3)`; that is, an optional sign followed by a non-empty string of digits with optional decimal point and optional exponent.

Note – The `.quad` command currently generates quad-precision values with only double-precision significance.

.reserve *symbol, size* [, "*sect_name*" [, *alignment*]]

Defines *symbol*, and reserves *size* bytes of space for it in the *sect_name*. This operation is equivalent to:

```
    .pushsection    "sect_name"
    .align          alignment
symbol:
    .skip           size
    .popsection
```

If a section is not specified, space is reserved in the current segment.

A

.section "*section_name*" [, *attributes*]

Makes the specified section the current section.

The assembler maintains a section stack which is manipulated by the section control directives. The current section is the section that is currently on top of the stack. This pseudo-op changes the top of the section stack.

- If *section_name* does not exist, a new section with the specified name and attributes is created.
- If *section_name* is a non-reserved section, *attributes* must be included the first time it is specified by the `.section` directive.

See the sections “Predefined User Sections” and “Predefined Non-User Sections” in Chapter 3, “Extensible and Linking Format,” for a detailed description of the reserved sections. See Table 3-2 in Chapter 3, “Extensible and Linking Format,” for a detailed description of the section attribute flags.

Attributes can be:

```
#write | #alloc | #execinstr
```

.seg "*section_name*"

Note – This pseudo-op is currently supported for compatibility with existing SunOS 4.1 SPARC assembly language programs. This pseudo-op has been replaced by the `.section` pseudo-op.

Changes the current section to one of the predefined user sections. The assembler will interpret the following SunOS 4.1 SPARC assembly directive:
`.seg "text", .seg "data", .seg "data1", .seg "bss",`
to be the same as the following SunOS 5.*x* SPARC assembly directive:
`.section ".text", .section ".data", .section ".data1", .section ".bss"`.

Note – Predefined user section names are changed in SunOS 5.*x*.

.single 0r*floatval* [, 0r*floatval*]*

Generates (a sequence of) initialized single-precision floating-point values in the current segment.

Note – This operation does not align automatically.

.size *symbol, expr*

Declares the symbol size to be *expr. expr* must be an absolute expression.

.skip *n*

Increments the location counter by *n*, which allocates *n* bytes of empty space in the current segment.

.stabn <*various parameters*>

The pseudo-op is used by Solaris 2.*x* SPARCompilers only to pass debugging information to the symbolic debuggers.

.stabs <*various parameters*>

The pseudo-op is used by Solaris 2.*x* SPARCompilers only to pass debugging information to the symbolic debuggers.

.type *symbol, type*

Declares the type of symbol, where *type* can be:

```
#object
#function
#no_type
```

See Table 3-6 in Chapter 3, “Extensible and Linking Format,” for detailed information on symbols.

A

.uahalf *16bitval* [,*16bitval*]*

Generates a (sequence of) 16-bit value(s).

Note – This operation does not align automatically.

.uaword *32bitval* [,*32bitval*]*

Generates a (sequence of) 32-bit value(s).

Note – This operation does not align automatically.

.version "*string*"

Identifies the minimum assembler version necessary to assemble the input file. You can use this pseudo-op to ensure assembler-compiler compatibility. If *string* indicates a newer version of the assembler than this version of the assembler, a fatal error message is displayed and the SPARC assembler exits.

.volatile

Defines the beginning of a block of instruction. The instructions in the section may not be changed. The block of instruction should end at a `.nonvolatile` pseudo-op and should not contain any Control Transfer Instructions (CTI) or labels. The volatile block of instructions is terminated after the last instruction preceding a CTI or label.

.weak *symbol* [,*symbol*]

Declares each *symbol* in the list to be defined either externally, or in the input file and accessible to other files; default bindings of the symbol are overridden by this directive.

Note the following:

- A *weak* symbol definition in one file will satisfy an undefined reference to a global symbol of the same name in another file.

- Unresolved *weak* symbols have a default value of zero; the link editor does not resolve these symbols.
- If a *weak* symbol has the same name as a defined *global* symbol, the weak symbol is ignored and no error results.

Note – This pseudo-op does not itself define the symbol.

.word *32bitval* [,*32bitval*]*

Generates (a sequence of) initialized words in the current segment.

Note – This operation does not align automatically.

.xstabs <*various parameters*>

The pseudo-op is used by Solaris 2.*x* SPARCompilers only to pass debugging information to the symbolic debuggers.

symbol =expr

Assigns the value of *expr* to *symbol*.

A

Examples of Pseudo-Operations B

This chapter shows some examples of ways to use various pseudo-ops.

Example 1

This example shows how to use the following pseudo-ops to specify the bindings of variables in C:

`common`, `.global`, `.local`, `.weak`

The following C definitions / declarations for example:

```
int           foo1 = 1;
#pragma       weak foo2 = foo1
static        int foo3;
static        int foo4 = 2;
```

can be translated into the following assembly code:

```
.pushsection ".data"

.global      foo1          ! int foo1 = 1
.align       4
```

```
foo1:
   .word        0x1
   .type        foo1,#object   ! foo1 is of type data object,
   .size        foo1,4         ! with size = 4 bytes

   .weak        foo2           ! #pragma weak foo2 = foo1
   foo2 = foo1

   .local       foo3           ! static int foo3
   .common      foo3,4,4

   .align       4              ! static int foo4 = 2
foo4:
   .word        0x2
   .type        foo4,#object
   .size        foo4,4

   .popsection
```

Example 2

This example illustrates how to use the pseudo-op `.ident` to generate a string in the `.comment` section of the object file for identification purposes.

```
.ident"acomp: (CDS) SPARCompilers 2.0 alpha4 12 Aug 1991"
```

Example 3

The pseudo-ops illustrated in this example are `.align`, `.global`, `.type`, and `.size`.

The following C subroutine for example:

```
int sum(a, b)
     int a, b;
{
     return(a + b);
}
```

can be translated into the following assembly code:

```
      .section      ".text"


      .global       sum

      .align        4

sum:

      retl
      add           %o0,%o1,%o0     ! (a + b) is done in the
                                    ! delay slot of retl

      .type         sum,#function   ! sum is of type function
      .size         sum,.-sum       ! size of sum is the diff
                                    ! of current location
                                    ! counter and the initial
                                    ! definition of sum
```

Example 4

The pseudo-ops illustrated in this example are `.section`, `.ascii`, and `.align`. The example calls the `printf` function to output the string `"hello world"`.

```
      .section      ".data1"
      .align        4
.L16:
      .ascii        "hello world\n\0"

      .section      ".text"
      .global       main
main:
      save          %sp,-96,%sp
      set           .L16,%o0
      call          printf,1
      nop
      restore
```

Example 5

This example illustrates how to use the `.volatile` and `.nonvolatile` pseudo-ops to protect a section of handwritten asembly code from peephole optimization.

```
.volatile
t     0x24
std   %g2, [%o0]
retl
nop
.nonvolatile
```

Using the Assembler Command Line C

Assembler Command Line

Invoke the assembler command line as follows:

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

Note – The language drivers (such as *cc* and *f77*) invoke the assembler command line with the `fbe` command. You can use either the `as` or `fbe` command to invoke the assembler command line.

The `as` command translates the assembly language source files, *inputfile*, into an executable object file, *objfile*. The SPARC assembler recognizes the filename argument *hyphen* (-) as the standard input. It accepts more than one file name on the command line. The input file is the concatenation of all the specified files. If an invalid option is given or the command line contains a syntax error, the SPARC assembler prints the error (including a synopsis of the command line syntax and options) to standard error output, and then terminates.

The SPARC 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 the `-P` option.)

C

Assembler Command Line Options

-b

This new option generates extra symbol table information for the source code browser.

- If the `as` command line option *`-P`* is set, the `cpp` preprocessor also collects browser information.
- If the `as` command line option *`-m`* is set, this option is ignored as the `m4` macro processor does not generate browser data.

For more information about the SPARCworks SourceBrowser, see the *Browsing Source Code* manual.

-Dname
-Dname=def

When the `-P` option is in effect, these options are passed to the *`cpp`* preprocessor without interpretation by the `as` command; otherwise, they are ignored.

-Ipath

When the `-P` option is in effect, this option is passed to the *`cpp`* preprocessor without interpretation by the `as` command; otherwise, it is ignored.

-K PIC

This new option generates position-independent code.

Note – `-K PIC` and `-K pic` are equivalent.

-L

Saves all symbols, including temporary labels that are normally discarded to save space, in the ELF symbol table.

-m

This new option runs m4 macro preprocessing on input. The *m4* preprocessor is more powerful than the C preprocessor (invoked by the -P option), so it is more useful for complex preprocessing. See the *SunOS Reference Manual* for a detailed description of the m4 macro-processor.

-o outfile

Takes the next argument as the name of the output file to be produced. By default, the .s suffix, if present, is removed from the input file and the .o suffix is appended to form the ouput file name.

-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.

-Q{y|n}

This new option produces the “assembler version” information in the comment section of the output object file if the y option is specified; if the n option is specified, the information is suppressed.

-q

This new option causes the assembler to perform a quick assembly. When the -q option is used, the node list is not built and the assembler simply emits instructions as they are read.

Note – This option disables many error checks. It is recommended that you do *not* use this option to assemble handwritten assembly language.

-S[a|C]

Produces a disassembly of the emitted code to the standard output.

- Adding the character *a* to the option appends a comment line to each assembly code which indicates its relative address in its own section.
- Adding the character *C* to the option prevents comment lines from appearing in the output.

`-s`

This new option places all stabs in the "`.stabs`" section. By default, stabs are placed in "`stabs.excl`" sections, which are stripped out by the static linker ld during final execution. When the `-s` option is used, stabs remain in the final executable because ".stab" sections are not stripped out by the static linker ld.

`-T`

This is a migration option for SunOS 4.1 assembly files to be assembled on SunOS 5.*x* systems. With this option, the symbol names in SunOS 4.1 assembly files will be interpreted as SunOS 5.*x* symbol names. This option can be used in conjunction with the `-S` option to convert SunOS 4.1 assembly files to their corresponding SunOS 5.*x* versions.

`-Uname`

When the `-P` option is in effect, this option is passed to the *`cpp`* preprocessor without interpretation by the `as` command; otherwise, it is ignored.

`-V`

This option writes the version information on the standard error output.

Disassembling Object Code

The `dis` program is the object code disassembler for ELF. It produces an assembly language listing of the object file. For detailed information about this function, see the man page `dis(1)`.

An Example Language Program D

The following code shows an example C language program; the second example code shows the corresponding assembly code generated by SPARCompiler C 2.0.1 that runs on Solaris 2.*x*. Comments have been added to the asembly code to show the correspondence to the C code.

Figure D-1 A C Program that Computes the First *n* Fibonacci Numbers

```
/* a simple program computing the first n Fibonacci numbers */

extern unsigned * fibonacci();

#define MAX_FIB_REPRESENTABLE 49


/* compute the first n Fibonacci numbers */
unsigned * fibonacci(n)
     int n;
{
  static unsigned fib_array[MAX_FIB_REPRESENTABLE] = {0,1};
  unsigned prev_number = 0;
  unsigned curr_number = 1;
  int i;

  if (n >= MAX_FIB_REPRESENTABLE) {
    printf("Fibonacci(%d) cannot be represented in a 32 bit word\n", n);
    exit(1);
  }

  for (i = 2; i < n; i++) {
```

```
    fib_array[i] = prev_number + curr_number;
    prev_number = curr_number;
    curr_number = fib_array[i];
  }

  return(fib_array);
}


main()
{
  int n, i;
  unsigned * result;

  printf("Fibonacci(n):, please enter n:\n");
  scanf("%d", &n);

  result = fibonacci(n);
  for (i = 1; i <= n; i++)
    printf("Fibonacci (%d) is %u\n", i, *result++);
}
```

Figure D-2 Assembler Progam Output by C SPARCompiler (annotated)

```
!
!  a simple program computing the first n Fibonacci numbers,
!  showing various pseudo-operations, sparc instructions, synthetic instructions
!
!  pseudo-operations:  .align, .ascii, .file, .global, .ident, .proc, .section,
!                                                  .size, .skip, .type, .word
!  sparc instructions:    add, bg, bge, bl, ble, ld, or, restore, save, sethi, st
!  synthetic instructions: call, cmp, inc, mov, ret
!

   .file          "fibonacci.c"             ! the original source file name


   .section       ".text"                   ! text section (executable instructions)
   .proc          79                        ! subroutine fibonacci, it's return
                                            ! value will be in %i0
   .global        fibonacci                 ! fibonacci() can be referenced
                                            ! outside this file
   .align         4                         ! align the beginning of this section
                                            ! to word boundary
fibonacci:
   save           %sp,-96,%sp               ! create new stack frame and register
                                            ! window for this subroutine
/*  if (n >= MAX_FIB_REPRESENTABLE) { */
                                            ! note, C style comment strings are
```

```
                                                ! also permitted
    cmp             %i0,49                      ! n >= MAX_FIB_REPRESENTABLE ?
                                                ! note, n, the 1st parameter to
                                                ! fibonacci(), is stored in %i0 upon
                                                ! entry
    bl              .L77003
    mov             0,%i2                       ! initialization of variable
                                                ! prev_number is executed in the
                                                ! delay slot

/* printf("Fibonacci(%d) cannot be represented in a 32 bits word\n", n); */
    sethi           %hi(.L20),%o0               ! if branch not taken, call printf(),
    or              %o0,%lo(.L20),%o0           ! set up 1st, 2nd argument in %o0, %o1;
    call            printf,2                    ! the ",2" means there are 2 out
    mov             %i0,%o1                     ! registers used as arguments
/* exit(1); */
    call            exit,1
    mov             1,%o0
.L77003:                                        ! initialize variables before the loop
/* for (i = 2; i < n; i++) { */
    mov             1,%i4                       ! curr_number = 1
    mov             2,%i3                       ! i = 2
    cmp             %i3,%i0                     ! i <= n?
    bge             .L77006                     ! if not, return
    sethi           %hi(.L16+8),%o0             ! use %i5 to store &fib_array[i]
    add             %o0,%lo(.L16+8),%i5
.LY1:                                           ! loop body
/* fib_array[i] = prev_number + curr_number; */
    add             %i2,%i4,%i2                 ! fib_array[i] = prev_number+curr_number
    st              %i2,[%i5]
/* prev_number = curr_number; */
    mov             %i4,%i2                     ! prev_number = curr_number
/* curr_number = fib_array[i]; */
    ld              [%i5],%i4                   ! curr_number = fib_array[i]
    inc             %i3                         ! i++
    cmp             %i3,%i0                     ! i <= n?
    bl              .LY1                        ! if yes, repeat loop
    inc             4,%i5                       ! increment ptr to fib_array[]
.L77006:
/* return(fib_array); */
    sethi           %hi(.L16),%o0               ! return fib_array in %i0
    add             %o0,%lo(.L16),%i0
    ret
    restore                                     ! destroy stack frame and register
                                                ! window
    .type           fibonacci,#function         ! fibonacci() is of type function
    .size           fibonacci,(.-fibonacci)     ! size of function:
                                                !  current location counter minus
                                                !  beginning definition of function


    .proc           18                          ! main program
    .global         main
    .align          4
```

```
main:
    save            %sp,-104,%sp                ! create stack frame for main()
/* printf("Fibonacci(n):, please input n:\n"); */
    sethi           %hi(.L31),%o0               ! call printf, with 1st arg in %o0
    call            printf,1
    or              %o0,%lo(.L31),%o0
/* scanf("%d", &n); */
    sethi           %hi(.L33),%o0               ! call scanf, with 1st arg, in %o0
    or              %o0,%lo(.L33),%o0           ! move 2nd arg. to %o1, in delay slot
    call            scanf,2
    add             %fp,-4,%o1

/* result = fibonacci(n); */
    call            fibonacci,1
    ld              [%fp-4],%o0

                                                ! some initializations before the for-
                                                ! loop, put the variables in registers
/* for (i = 1; i <= n; i++) */
    mov             1,%i5                       ! %i5 <-- i
    mov             %o0,%i4                     ! %i4 <-- result
    sethi           %hi(.L38),%o0               ! %i2 <-- format string for printf
    add             %o0,%lo(.L38),%i2
    ld              [%fp-4],%o0                 ! test if (i <= n) ?
    cmp             %i5,%o0                     ! note, n is stored in [%fp-4]
    bg              .LE27
    nop
.LY2:                                           ! loop body
/* printf("Fibonacci (%d) is %u\n", i, *result++); */
    ld              [%i4],%o2                   ! call printf, with (*result) in %o2,
    mov             %i5,%o1                     !   i in %o1, format string in %o0
    call            printf,3
    mov             %i2,%o0
    inc             %i5                         ! i++
    ld              [%fp-4],%o0                 ! i <= n?
    cmp             %i5,%o0
    ble             .LY2
    inc             4,%i4                       ! result++
.LE27:
    ret
    restore
    .type           main,#function              ! type and size of main
    .size           main,(.-main)


    .section ".data"                            ! switch to data section
                                                ! (contains initialized data)
    .align          4
.L16:
/* static unsigned fib_array[MAX_FIB_REPRESENTABLE] = {0,1}; */
    .align          4                           ! initialization of first 2 elements
    .word           0                           ! of fib_array[]
    .align          4
    .word           1
```

```
    .skip           188
    .type           .L16,#object                ! storage allocation for the rest of
                                                ! fib_array[]


    .section ".data1"                           ! the ascii string data are entered
                                                ! into the .data1 section;
                                                ! #alloc:  memory would be allocated
                                                !    for this section during run time
                                                ! #write:  the section contains data
                                                !    that is writeable during process
                                                !    execution
    .align          4
.L20:                                           ! ascii strings used in the printf stmts
    .ascii          "Fibonacci(%d) cannot be represented in a 32 bit w"
    .ascii          "ord\n\0"
    .align          4                           ! align the next ascii string to word
                                                ! boundary
.L31:
    .ascii          "Fibonacci(n):, please enter n:\n\0"
    .align          4
.L33:
    .ascii          "%d\0"
    .align          4
.L38:
    .ascii          "Fibonacci (%d) is %u\n\0"
    .ident          "acomp: (CDS) SPARCompilers 2.0 05 Jun 1991"
                                                ! an idenitfication string produced
                                                ! by the compiler to be entered into
                                                ! the .comment section
```

D

Index

A

addresses, 25
`.alias`, 55
`.align`, 55
`as` command, 69
`.ascii`, 55
`.asciz`, 55
assembler command line, 69
assembler command line options, 70
assembler directives, 28
    types, 28
assembly language, 7
    lines, 8
    statements, 8
    syntax notation, 7
assignment directive, 29
`atof`*(3), 9*, *56*, *59*

B

binary operations, 12
`.byte`, 56

C

case distinction, 8
case distinction, in special symbols, 12
cc language driver, 69
comment lines, 8
comment lines, multiple, 8
`.common`, 56
constants, 9
    decimal, 9
    floating-point, 9
    hexadecimal, 9
    octal numeric, 9
Control Transfer Instructions (CTI), 14
converting existing object files, 31
coprocessor instruction, 49
`cp_disabled` trap, 49
`cp_exception` trap, 49
current location, 25
current section, 17

D

`-D` option, 70
data generating directives, 29
default output file, 15
`dis` program, 72
disassembling object code, 72
`.double`, 56

E

.empty, 56
.empty pseudo-operation, 14
error messages, 14
escape codes, in strings, 9
Executable and Linking Format (ELF)
    files, 2, 15
expressions, 13

F

f77 language driver, 69
fbe command, 69
features, lexical, 8
.file, 57
file syntax, 7
floating-point instructions, 47
floating-point pseudo-operations, 9

G

.global, 57
.globl, 57

H

.half, 57
hardware instructions
    SPARC architecture, 33
hardware integer
    assembly language instructions, 36
hyphen (-), 69

I

-I option, 70
.ident, 57
instruction set, used by assembler, 33
instructions
    assembly language, 36
    hardware integer, 36
integer instructions, 36
integer suffixes, 9
invoking, as command, 69

K

-K option, 70

L

-L option, 70
labeling format, 2
labels, 8
language drivers, 69
lexical features, 8
lines syntax, 8
.local, 58
location counter, 25
locations, 25

M

-m option, 71
multiple comment lines, 8
multiple files, on *as command line*, 69
multiple sections, 18
multiple strings
    in string table, 27

N

.noalias, 58
.noalias pseudo-op, 55
.nonvolatile, 58
numbers, 9
numeric labels, 8

O

-o option, 71
object file format, 2
object files
    type, 2, 15
operators, 13
.optim, 58
options

command line, 4

P

-P option, 71
percentage sign (%), 11
.popsection, 58
predefined non-user sections, 24
predefined user sections, 22
.proc, 59
pseudo-operations, 55
pseudo-ops, 2, 3
examples of, 65
.pushsection, 59

Q

-Q option, 71
-q option, 71
.quad, 59

R

references, 5
relocatable files, 2, 15
relocation tables, 25
.reserve, 59

S

-S option, 71
-s option, 72
-sb option, 70
.section, 60
section control directives, 28
section control pseudo-ops, 28
section header, 18
sections, 17
.seg, 60
.single, 61
.size, 61
.skip, 61
special floating-point values, 9
special names, floating point values, 9
special symbols, 11
.stabn, 61
.stabs, 61
statement syntax, 8
string tables, 27
strings, 9
multiple in string table, 27
multiple references in string table, 27
suggested style, 9
unreferenced in string table, 27
sub-strings in string table
references to, 27
symbol, 63
symbol attribute directives, 28
symbol names, 10
symbol tables, 25
syntax notation, 7
synthetic instructions, 50

T

-T option, 72
table notation, 33
trap numbers, reserved, 45
.type, 61

U

-U option, 72
.uahalf, 62
.uaword, 62
unary operators, 12
user sections, 28
/usr/include/sys/trap.h, 45

V

-V option, 72
.version, 62
.volatile, 62

W

.weak, 62
.word, 63

X

.xstabs, 63