SPARC Assembly Language Reference Manual

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816–1681–10
May 2002

020115@3062

Contents

Preface

The SunOS assembler that runs on the SPARC operating environment, referred to as the "SunOS SPARC" in this manual, translates source files that are in assembly language format into object files in linking format.

In the program development process, the assembler is a tool to use in producing program modules intended to exploit features of the SPARC architecture in ways that cannot be easily done using high level languages and their compilers.

Whether assembly language is chosen for the development of program modules depends on the extent to which and the ease with which the language allows the programmer to control the architectural features of the processor.

The assembly language described in this manual offers full direct access to the SPARC instruction set. The assembler may also be used in connection with SunOS 5.x macro preprocessors to achieve full macro-assembler capability. Furthermore, the assembler responds to directives that allow the programmer direct control over the contents of the relocatable object file.

This document describes the language in which the source files must be written. The nature of the machine mnemonics governs the way in which the program's executable portion is written. This document includes descriptions of the pseudo operations that allow control over the object file. This facilitates the development of programs that are easy to understand and maintain.

Before You Read This Book

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* (Version 8 and Version 9)

- *System V Application Binary Interface: SPARC™ Processor Supplement*

How This Book is Organized

This book is organized as follows:

Chapter 1 discusses features of the SunOS 5.x SPARC Assembler.

Chapter 2 describes the syntax of the SPARC assembler that takes assembly programs and produces relocatable object files for processing by the link editor.

Chapter 3 describes the relocatable ELF files that hold code and data suitable for linking with other object files.

Chapter 4 describes how to convert existing SunOS 4.1 SPARC assembly files to the SunOS 5.x assembly file format.

Chapter 5 describes the relationship between hardware instructions of the SPARC architecture and the assembly language instruction set.

Appendix A lists the pseudo-operations supported by the SPARC assembler.

Appendix B shows some examples of ways to use various pseudo-operations.

Appendix C describes the available assembler command-line options.

Appendix D describes an example C language program with comments to show correspondence between the assembly code and the C code.

Appendix E describes the SPARC-V9 instruction set and the changes due to the SPARC-V9 implementation.

Accessing Sun Documentation Online

The docs.sun.com℠ Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

SPARC Assembler for *SunOS 5.x*

1.1 Introduction

This chapter discusses features of the *SunOS 5.x* SPARC assembler. This document is distributed as part of the 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™ 4.0 users who have products that run on the *SunOS 5.x* operating system. It is included as part of the SPARCworks/SPARCompiler Floating Point and Common Tools AnswerBook, which is the on-line information retrieval system.

This document contains information from *The SPARC Architecture Manual*, Version 8. Information about Version 9 support is summarized in Appendix E.

1.2 Operating Environment

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

1.3 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.

1.3.1 Labeling Format

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

1.3.2 Object File Format

The type of object files created by the SPARC assembler are ELF (*Executable 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.

1.3.3 Pseudo-Operations

See Appendix A for a detailed description of the pseudo-operations (pseudo-ops).

1.3.4 Command Line Options

See Appendix C for a detailed description of command line options and a list of SPARC architectures.

CHAPTER 2

Assembler Syntax

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* (Version 8 and Version 9) and is intended for use on machines that use the SPARC architecture.

This chapter is organized into the following sections:

2.1 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.

2.2 Assembler File Syntax

The syntax of assembly language *files* is:

```
[line]*
```

2.2.1 Lines Syntax

The syntax of assembly language *lines* is:

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

2.2.2 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.

2.3 Lexical Features

This section describes the lexical features of the assembler syntax.

2.3.1 Case Distinction

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

2.3.2 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.

2.3.3 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`.

2.3.4 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 INFinit*y 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.”

2.3.5 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 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...*

2.3.6 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.

2.3.7 Special Symbols - Registers

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

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`	
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
```

2.3.8 Operators and Expressions

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

TABLE 2–3

Binary	Operators	Unary	Operators
+	Integer addition	+	(No effect)
–	Integer subtraction	–	2's Complement
*	Integer multiplication	~	1's Complement
/	Integer division	`%lo(address)`	Extract least significant 10 bits as computed by: (address & 0x3ff)
%	Modulo	`%hi(address)`	Extract most significant 22 bits as computed by: (address >>10)
^	Exclusive OR	`%r_disp32` `%r_disp64`	Used in Sun compiler-generated code only to instruct the assembler to generate specific relocation information for the given expression.

TABLE 2–3 *(Continued)*

Binary	Operators	Unary	Operators
<<	Left shift	%r_plt32 %r_plt64	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_disp32/64, or %r_plt32/64 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.

2.3.9 SPARC V9 Operators and Expressions

The following V9 64-bit operators and expressions in Table 2–4 ease the task of converting from V8/V8plus assembly code to V9 assembly code.

TABLE 2–4

Unary	Calculation	Operators
%hh	(address) >> 42	Extract bits 42-63 of a 64-bit word
%hm	((address) >> 32) & 0x3ff	Extract bits 32-41 of a 64-bit word
%lm	(((address) >> 10) & 0x3fffff)	Extract bits 10-31 of a 64-bit word

For example:::

```
sethi %hh (address), %l1
or %l1, %hm (address), %l1

sethi %lm (address), %12
or %12, %lo (address), %12

sllx %l1, 32, %l1
or %l1, %12, %l1
```

The V9 high 32-bit operators and expressions are identified in Table 2–5.

TABLE 2–5

Unary	Calculation	Operators
`%hix`	((((address) ^ 0xffffffffffffffff >> 10) &0x4fffff)	Invert every bit and extract bits 10-31
`%lox`	((address) & 0x3ff | 0x1c00	Extract bits 0-9 and sign extend that to 13 bits

For example:

```
%sethi %hix (address), %l1
or %l1, %lox (address), %l1
```

The V9 low 44-bit operators and expressions are identified in Table 2–6.

TABLE 2–6

Unary	Calculation	Operators
`%h44`	((address) >> 22)	Extract bits 22-43 of a 64-bit word
`%m44`	((address) >> 12) & 0x3ff	Extract bits 12-21 of a 64-bit word
l44	(address) & 0xfff	Extract bits 0-11 of a 64-bit word

For example::

```
%sethi %h44 (address), %l1
or %l1, %m44 (address), %l1
sllx %l1, 12, %l1
or %l1, %144 (address), %l1
```

2.4 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.

CHAPTER 3

Executable and Linking Format

The type of object files created by the SPARC assembler version for *SunOS 5.x* are now *Executable 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 can 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.

This chapter is organized into the following sections:

The ELF object file format consists of:

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

For more information, see the *System V Application Binary Interface: SPARC™ Processor Supplement*.

3.1 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 shows reserved object file types:

TABLE 3–1

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

3.2 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.

3.2.1 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.

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

TABLE 3–2

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

Name	Value	Description	Interpretation by	
			info	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.		
`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.

TABLE 3–3 *(Continued)*

Name	Value	Description	Interpretation by	
			info	link
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 0x7fffffff	Lower and upper bound of range reserved for processor-specific semantics.		
louser hiuser	0x80000000 0xffffffff	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.

3.2.2 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

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.

TABLE 3–4 *(Continued)*

Section Name	Description
`.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.

3.2.2.1 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 Example 3–1.

EXAMPLE 3–1 Creating an `.init` Section

```
.section ".init"

.align   4

<instructions>
```

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.

Because the whole `.init` section is treated as a single function body, it is recommented that the only code added to these sections be in the following form:.

```
call routine_name

nop
```

The called routine should be located in another section. This will prevent conflicting register and stack usage within the `.init` sections.

3.2.2.2 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 Example 3–2.

EXAMPLE 3–2 Creating an .fini Section

```
.section ".fini"

.align   4

<instructions>
```

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.

Because the whole `.fini` section is treated as a single function body, it is recommended that the only code added to these section be in the following form:.

```
call routine_name

nop
```

The called routine should be located in another section. This will prevent conflicting register and stack usage within the `.fini` sections.

3.2.3 Predefined Non-User Sections

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

TABLE 3–5

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, that is, `".rel.text"`, `".rela.text"`.

TABLE 3–5 *(Continued)*

Section Name	Description
`.shstrtab`	String table for the section header table names.
`.strtab`	Section contains the string table.
`.symtab`	Section contains a symbol table.

3.3 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, the `.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.

3.4 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.

3.5 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.

3.6 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 describe 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

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	loproc	Values reserved for processor-specific semantics.
15	hiproc	

Table 3–7 shows the symbol binding attributes.

TABLE 3–7

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	loproc	Values reserved for processor-specific semantics.
15	hiproc	

3.7 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.

3.8 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 for a complete description of the pseudo-ops supported by the SPARC assembler.

3.8.1 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.

3.8.2 Symbol Attribute Directives

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

3.8.3 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.

3.8.4 Data Generating Directives

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

CHAPTER 4

Converting Files to the New Format

4.1 Introduction

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

4.2 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 as did 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`, for example, change `.seg data` to `.section .data`. See Appendix A for more information.

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

4.3 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.

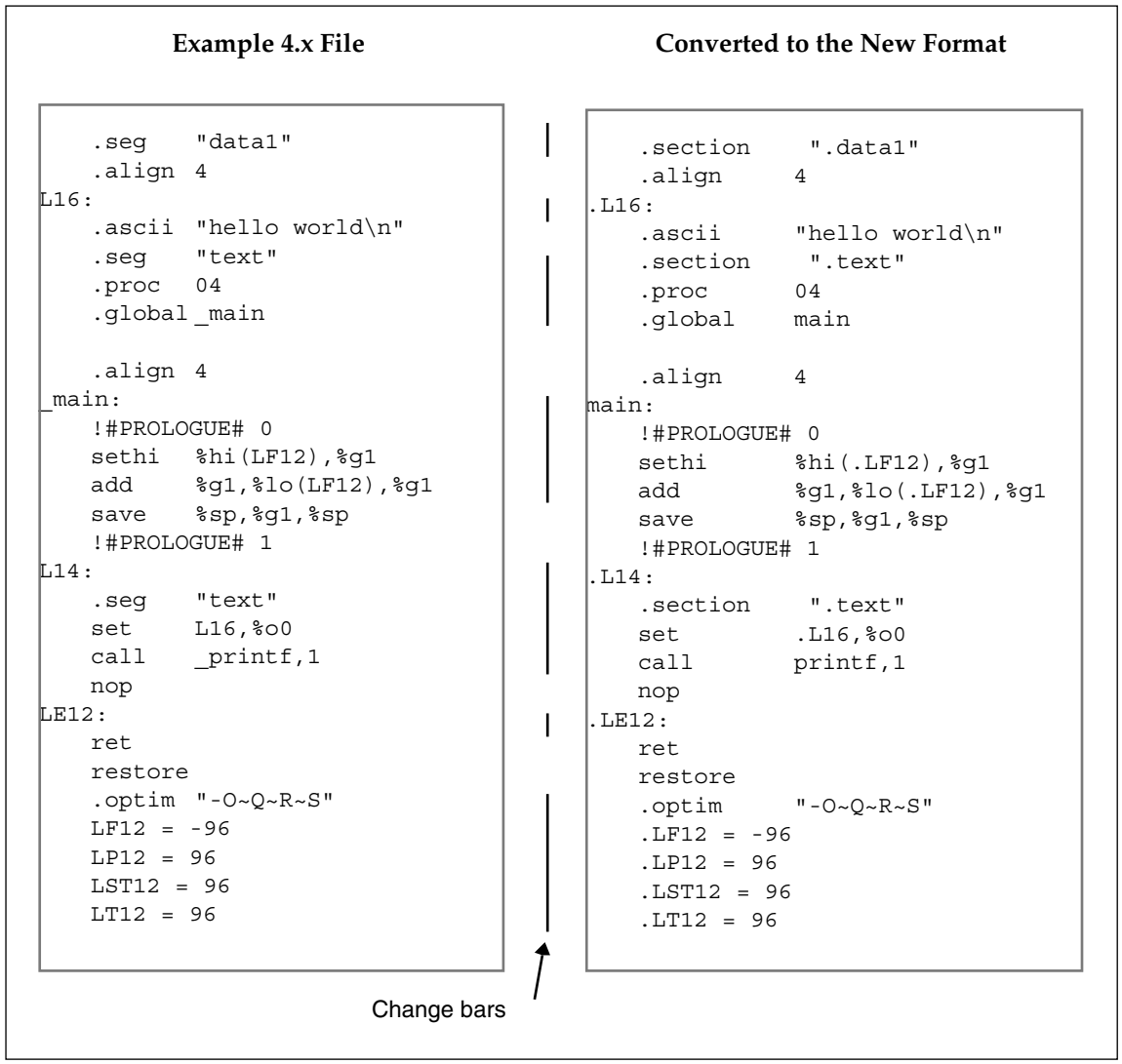

FIGURE 4–1 Converting a 4.x File to the New Format

CHAPTER 5

Instruction-Set Mapping

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

The SPARC-V9 instruction set is described in Appendix E.

5.1 Table Notation

Table 5–1 shows the table notation used 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.

TABLE 5–1

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.

TABLE 5–1 *(Continued)*

Notations	Describes	Comment
`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`	General purpose registers.
	`%g0 ... %g7`	Same as `%r0 ... %r7` (Globals)
	`%o0 ... %o7`	Same as `%r8 ... %r15` (Outs)
	`%l0 ... %l7`	Same as `%r16 ... %r23` (Locals)
	`%i0 ... %i7`	Same as `%r24 ... %r31` (Ins)
reg_{rd}		Destination register.
reg_{rs1}, reg_{rs2}		Source register 1, source register 2.
`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}	A value formed from register contents, immediate constant, or both. The resulting value must be in the range 0.....127, inclusive.
	reg_{rs1} + `imm7`	
	reg_{rs1} - `imm7`	
	`uimm7`	
	`imm7` + reg_{rs1}	
`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.

5.2 Integer Instructions

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

TABLE 5–2

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 differs 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

Opcode	Mnemonic	Argument List	Operation	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}		
ANDcc	andn	reg_{rs1}, *reg_or_imm*, reg_{rd}		
ANDNcc	andcc	reg_{rs1}, *reg_or_imm*, reg_{rd}		
BN	bn{,a}	label	Branch on integer condition codes	branch never
BNE	bne{,a}	label		synonym: `bnz`
BE	be{,a}	label		synonym: `bz`
BG	bg{,a}	label		
BLE	ble{,a}	label		
BGE	bge{,a}	label		
BI	bl{,a}	label		
BGU	bgu{,a}	label		
BLEU	bleu{,a}	label		
BCC	bcc{,a}	label		synonym: `bgeu`
BCS	bcs{,a}	label		synonym: `blu`
BPOS	bpos{,a}	label		
BNEG	bneg{,a}	label		
BVC	bvc{,a}	label		
BVS	bvs{,a}	label		
BA	ba{,a}	label		synonym: `b`
CALL	call	label	Call subprogram	

TABLE 5–3 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
CBccc	cbn{,a}	label	Branch on coprocessor condition codes	branch never
	cb3{,a}	label		
	cb2{,a}	label		
	cb23{,a}	label		
	cb1{,a}	label		
	cb13{,eo}	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		
FBN	fbn{,a}	label	Branch on floating-point condition codes	branch never
FBU	fbu{,a}	label		
FBG	fbg{,a}	label		
FBUG	fbug{,a}	label		
FBL	fbl{,a}	label		
FBUL	fbul{,a}	label		
FBLG	fblg{,a}	label		
FBNE	fbne{,a}	label		synonym: fbnz
FBE	fbe{,a}	label		synonym: fbz

TABLE 5–3 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
FBUE	fbue{,a}	label		
FBGE	fbge{,a}	label		
FBUGE	fbuge{,a}	label		
FBLE	fble{,a}	label		
FBULE	fbule{,a}	label		
FBO	fbo{,a}	label		
FBA	fba{,a}	label		
FLUSH	flush	*address*	*Instruction cache flush*	
JMPL	jmpl	*address*, reg_{rd}	Jump and link	
LDSB	ldsb	*[address]*, reg_{rd}	Load signed byte	
LDSH	ldsh	*[address]*, reg_{rd}	Load signed halfword	
LDSTUB	ldstub	*[address]*, reg_{rd}	Load-store unsigned byte	
LDUB	ldub	*[address]*, reg_{rd}	Load unsigned byte	
LDUH	lduh	*[address]*, reg_{rd}	Load unsigned halfword	
LD	ld	*[address]*, reg_{rd}	Load word	
LDD	ldd	*[address]*, reg_{rd}	Load double word	reg_{rd} *must be even*
LDF	ld	*[address]*, $freg_{rd}$		
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*	Load double coprocessor	
LDDC	ldd	*[address]*, $creg_{rd}$		
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 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
MULScc	mulscc	reg_{rs1}, *reg_or_imm*, reg_{rd}	Multiply step (and modify icc)	
NOP	nop		No operation	
OR	or	reg_{rs1}, *reg_or_imm*, reg_{rd}	Inclusive or	
ORcc	orcc	reg_{rs1}, *reg_or_imm*, reg_{rd}		
ORN	orn	reg_{rs1}, *reg_or_imm*, reg_{rd}		
ORNcc	orncc	reg_{rs1}, *reg_or_imm*, reg_{rd}		
RDASR	rd	%asrn_{rs1}, reg_{rd}		
RDY	rd	%y, reg_{rd}		See synthetic 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	
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	

TABLE 5–3 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
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]*		
STFSR	st	%fsr, *[address]*	Store floating-point status register	*freg*$_{rd}$ Must be even
STDFQ	std	%fq, *[address]*	Store double floating-point queue	
STC	st	*creg*$_{rd}$, *[address]*	Store coprocessor	*creg*$_{rd}$ *Must be even*
STDC	std	*creg*$_{rd}$, *[address]*		*creg*$_{rd}$ *Must be even*
STCSR	st	%csr, *[address]*		
STDCQ	std	%cq, *[address]*	Store double coprocessor	
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	
SWAPA	swapa	*[regaddr]asi*, *reg*$_{rd}$	with register	
Ticc	tn	software_trap_number	Trap on integer condition code	Trap never

TABLE 5–3 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
	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		Synonym: tcc
	tlu	software_trap_number		Synonym: tcc
	tgeu	software_trap_number		
	tpos	software_trap_number		
	tneg	software_trap_number		
	tvc	software_trap_number		Synonym: t
	tvs	software_trap_number		
	ta	software_trap_number		
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*, %asrn_{rs1}		

TABLE 5–3 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
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*$_{rd}$		

5.3 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

SPARC	Mnemonic[1]	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

TABLE 5–4 *(Continued)*

SPARC	Mnemonic[1]	Argument List	Description
FdTOq	fdtoq	$freg_{rs2}$, $freg_{rd}$	Convert double to quad
FqTOd	fqtod	$freg_{rs2}$, $freg_{rd}$	Convert quad to double
FqTOs	fqtos	$freg_{rs2}$, $freg_{rd}$	Convert quad to single
FMOVs	fmovs	$freg_{rs2}$, $freg_{rd}$	Move
FNEGs	fnegs	$freg_{rs2}$, $freg_{rd}$	Negate
FABSs	fabss	$freg_{rs2}$, $freg_{rd}$	Absolute value
FSQRTs	fsqrts	$freg_{rs2}$, $freg_{rd}$	Square root
FSQRTd	fsqrtd	$freg_{rs2}$, $freg_{rd}$	
FSQRTq	fsqrtq	$freg_{rs2}$, $freg_{rd}$	
FADDs	fadds	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	Add
FADDd	faddd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FADDq	faddq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FSUBs	fsubs	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	Subtract
FSUBd	fsubd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FSUBq	fsubq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FMULs	fmuls	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	Multiply
FMULd	fmuld	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FMULq	fmulq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FdMULq	fmulq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	Multiply double to quad
FsMULd	fsmuld	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	Multiply single to double
FDIVs	fdivs	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	Divide
FDIVd	fdivd	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FDIVq	fdivq	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	
FCMPs	fcmps	$freg_{rs1}$, $freg_{rs2}$	Compare
FCMPd	fcmpd	$freg_{rs1}$, $freg_{rs2}$	
FCMPq	fcmpq	$freg_{rs1}$, $freg_{rs2}$	
FCMPEs	fcmpes	$freg_{rs1}$, $freg_{rs2}$	Compare, generate exception if not ordered
FCMPEd	fcmped	$freg_{rs1}$, $freg_{rs2}$	
FCMPEq	fcmpeq	$freg_{rs1}$, $freg_{rs2}$	

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

   `i` **integer**

   `s` **single**

   `d` **double**

   `q` **quad**

5.4 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

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`

5.5 Synthetic Instructions

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

TABLE 5–6

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

TABLE 5–6 *(Continued)*

Synthetic Instruction		Hardware Equivalent(s)		Comment
`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]*	`st`	`%g0`, [*address*]	Clear halfword
`clr`	*[address]*	`st`	`%g0`, [*address*]	Clear word
`cmp`	*reg*, *reg_or_imm*	`subcc`	reg_{rs1}, *reg_or_imm*, `%g0`	Compare
`dec`	reg_{rd}	`sub`	reg_{rd}, *1*, reg_{rd}	Decrement by 1
`dec`	*const13*, reg_{rd}	`sub`	reg_{rd}, *const13*, reg_{rd}	Decrement by *const13*
`deccc`	reg_{rd}	`subcc`	reg_{rd}, *1*, reg_{rd}	Decrement by 1 and set `icc`
`deccc`	*const13*, reg_{rd}	`subcc`	reg_{rd}, *const13*, reg_{rd}	Decrement by *const13* and set `icc`
`inc`	reg_{rd}	`add`	reg_{rd}, *1*, reg_{rd}	Increment by 1
`inc`	*const13*, reg_{rd}	`add`	reg_{rd}, *const13*, reg_{rd}	Increment by *const13*
`inccc`	reg_{rd}	`addcc`	reg_{rd}, *1*, reg_{rd}	Increment by 1 and set `icc`
`inccc`	*const13*, reg_{rd}	`addcc`	reg_{rd}, *const13*, reg_{rd}	Increment by *const13* and set `icc`
`jmp`	*address*	`jmpl`	*address*, `%g0`	
`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

TABLE 5–6 *(Continued)*

Synthetic Instruction		Hardware Equivalent(s)		Comment
`neg`	*reg*$_{rd}$	`sub`	`%g0`, *reg*$_{rd}$, *reg*$_{rd}$	Two's complement
`restore`		`restore`	`%g0, %g0, %g0`	Trivial *restore*
`save`		`save`	`%g0, %g0, %g0`	Trivial *save* *trivial save* should only be used in supervisor code!
`set`	*value*,*reg*$_{rd}$	`or`	`%g0`, *value*, *reg*$_{rd}$	if -4096 ≤*value* ≤ 4095 Do not use the `set` synthetic instruction in an instruction delay slot.
`set`	*value*,*reg*$_{rd}$	`sethi`	`%hi`*(value)*, *reg*$_{rd}$	if ((*value* & 0x3ff) == 0)
`set`	*value*, *reg*$_{rd}$	`sethi` `or`	`%hi`*(value)*, *reg*$_{rd}$; reg$_{rd}$, `%lo`*(value)*, *reg*$_{rd}$	otherwise Do not use the `set` synthetic instruction in an instruction delay slot.
`skipz`		`bnz,a .+8`		if *z* is set, ignores next instruction
`skipnz`		`bz,a .+8`		if *z* is not set, ignores next instruction
`tst`	*reg*	`orcc`	*reg*$_{rs1}$, `%g0, %g0`	test

5.6 V8/V9 Natural Pseudo Instructions

Table 5–7 describes the V8/V9 natural pseudo instructions that will help increase the portability of your assembly code from V8/V8plus to V9.

TABLE 5–7

	-xarch=	
Pseudo Instructions	**V8/V8plus[1]**	**V9**
`ldn`	ld	ldx
`stn`	st	stx
`ldna`	lda	ldxa

TABLE 5–7 *(Continued)*

Pseudo Instructions	-xarch=	
	V8/V8plus[1]	V9
stna	sta	stxa
setn	set	setx
setnhi	sethi	setxhi
casn	cas	casx
slln	sll	sllx
srln	srl	srlx
sran	sra	srax
clrn	clr	clrx

1. Indicates default setting

Note – Depending on the value set for the `-xarch` option, the assembler substitutes the appropriate pseudo instruction.

APPENDIX **A**

Pseudo-Operations

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

A.1 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.*

`.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. (`.data` is not currently supported.)
- 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 *SPARC 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.
- A global psuedo-op oes not need to occur before a definition, or tentative definition, of the specified symbol.

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
```

`.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

`.nword` *64bitval* `[,` *64bitval*`]*`
If `-xarch=v8/v8plus` then assembler interprets the instruction as `.word`. If `-xarch=v9` the assembler interprets the instruction as `.xword`.

`.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.

`.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 "3.2.2 Predefined User Sections" on page 29 and "3.2.3 Predefined Non-User Sections" on page 31 in Chapter 3 for a detailed description of the reserved sections. See Table 3–2 in Chapter 3 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: to be the same as the following *SunOS 5.x* SPARC assembly directive:

```
 .seg text, .seg data, .seg data1, .seg bss,

.section .text, .section .data, .section .data1,

.section .bss.
```

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, for detailed information on symbols.

`.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.

`.xword` *64bitval* `[,` *64bitval*`]*`

Generates (a sequence of) initialized 64-bit values 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*.

APPENDIX **B**

Examples of Pseudo-Operations

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

B.1 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:

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

can be translated into the following assembly code:

EXAMPLE B–1

```
    .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
```

EXAMPLE B–1 *(Continued)*

```
    .common             foo3,4,4

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

    .popsection
```

B.2 Example 2

This example shows 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"
```

B.3 Example 3

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

The following C subroutine:

```
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
```

B.4 Example 4

The pseudo-ops shown 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
```

B.5 Example 5

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

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

APPENDIX **C**

Using the Assembler Command Line

This appendix is organized into the following secitons:

C.1 Assembler Command Line

You 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.2 Assembler Command Line Options

`-b`

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

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 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 `m4(1)` man page for more information about the `m4` macro-processor.

`-n`

Suppress all warnings while assembling.

`-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 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 option causes the assembler to perform a quick assembly. Many error-checks are not performed when `-q` is specified.

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

`-S[a|b|c|l|A|B|C|L]`
Produces a disassembly of the emitted code to the standard output. Adding each of the following characters to the `-S` option produces:

- `a` - disassembling with address
- `b` - disassembling with ".bof"
- `c` - disassembling with comments
- `l` - disassembling with line numbers

Capital letters turn the switch off for the corresponding option.

`-s`
This 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.

`-xarch=v7`
This option instructs the assembler to accept instructions defined in the SPARC version 7 (V7) architecture. The resulting object code is in ELF format.

`-xarch=v8`
This option instructs the assembler to accept instructions defined in the SPARC-V8 architecture. The resulting object code is in ELF format. The quad-precision floating-point instructions are allowed; however when the program is executed these instructions cause a hardware exception called "trap" (an illegal instruction trap). The kernel has the trap handler to emulate the quad precision floating-point arithmetic. Consequently, all quad precision arithmetic is performed by the emulator in the kernel.

`-xarch=v8a`
This option instructs the assembler to accept instructions defined in the SPARC-V8 architecture, less the fsmuld instruction. The resulting object code is in ELF format. The quad-precision floating-point instructions are allowed; however when the program is executed these instructions cause a hardware exception called "trap" (an illegal instruction trap). The kernel has the trap handler to emulate the quad precision floating-point arithmetic. Consequently, all quad precision arithmetic is performed by the emulator in the kernel. This is the default choice of the `-xarch=` options.

`-xarch=v8plus`
This option instructs the assembler to accept instructions defined in the SPARC-V9 architecture. The resulting object code is in ELF format. The quad-precision floating-point instructions are allowed; however when the program is executed these instructions cause a hardware exception called "trap" (an illegal instruction trap). The kernel has the trap handler to emulate the quad precision floating-point arithmetic. Consequently, all quad precision arithmetic is performed by the emulator in the kernel. It will not execute on a Solaris V8 system (a machine with a V8 processor). It will execute on a Solaris V8+ system. This combination is a SPARC 64-bit processor and a 32-bit OS. For more information regarding SPARC-V9 instructions, see Appendix E.

`-xarch=v8plusa`
This option instructs the assembler to accept instructions defined in the SPARC-V9 architecture, plus the instructions in the Visual Instruction Set (VIS). The resulting object code is in V8+ ELF format. It will not execute on a Solaris V8 system. It will execute on a Solaris V8+ system. For more information about VIS instructions, see the *UltraSPARC Programmer's Reference Manual* and the *UltraSPARC User's Guide*.

The quad-precision floating-point instructions are allowed; however when the program is executed these instructions cause a hardware exception called "trap" (an illegal instruction trap). The kernel has the trap handler to emulate the quad precision floating-point arithmetic. Consequently, all quad precision arithmetic is performed by the emulator in the kernel.

`-xarch=v9`

This option limits instruction set to the SPARC-V9 architecture. The resulting `.o` object files are in 64-bit ELF format and can only be linked with other object files in the same format. The resulting executable can only be run on a 64-bit SPARC processor running 64-bit Solaris 7 with the 64-bit kernel.

Note – This option is available only on Solaris 7.

`-xarch=v9a`

This option limits instruction set to the SPARC-V9 architecture, adding the Visual Instruction Set (VIS) and extensions specific to UltraSPARC processors. The resulting .o object files are in 64-bit ELF format and can only be run on a 64-bit SPARC processor running 64-bit Solaris 7 with the 64-bit kernel.

Note – This option is available only on Solaris 7.

C.3 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).

APPENDIX **D**

An Example Language Program

The following code shows an example C language program; the second example code shows the corresponding assembly code generated by SPARCompiler C 3.0.2 that runs on the *Solaris 2.x* operating environment. Comments have been added to the assembly code to show correspondence to the C code.

The following C Program computes the first *n* Fibonacci numbers:

EXAMPLE D–1 C Program Example Source

```
/* 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);
}
```

EXAMPLE D–1 C Program Example Source *(Continued)*

```
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++);
}
```

The C SPARCompiler generates the following assembler output for the Fibonacci number C source. Annotation has been added to help you understand the code.

EXAMPLE D–2 Assembler Output From C Source

```
!
!  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
```

EXAMPLE D–2 Assembler Output From C Source *(Continued)*

```
                        ! 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
```

EXAMPLE D–2 Assembler Output From C Source *(Continued)*

```
/* 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[]
```

EXAMPLE D–2 Assembler Output From C Source *(Continued)*

```
    .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
```

APPENDIX **E**

SPARC-V9 Instruction Set

This appendix describes changes made to the SPARC instruction set due to the SPARC-V9 architecture. Application software for the 32-bit SPARC-V8 (Version8) architecture can execute, unchanged, on SPARC-V9 systems.

This appendix is organized into the following sections:

E.1 SPARC-V9 Changes

The SPARC-V9 architecture differs from SPARC-V8 architecture in the following areas, expanded below: registers, alternate space access, byte order, and instruction set.

E.1.1 Registers

These registers have been deleted:

TABLE E–1

PSR	Processor State Register
TBR	Trap Base Register

TABLE E–1 *(Continued)*

WIM	Window Invalid Mask

These registers have been widened from 32 to 64 bits:

TABLE E–2

Integer registers	
All state registers	FSR, PC, nPC, and Y

Note – FSR Floating-Point State Register: fcc1, fcc2, and fcc3 (added floating-point condition code) bits are added and the register widened to 64-bits.

These SPARC-V9 registers are within a SPARC-V8 register field:

TABLE E–3

CCR	Condition Codes Register
CWP	Current Window Pointer
PIL	Processor Interrupt Level
TBA	Trap Base Address
TT[MAXTL]	Trap Type
VER	Version

These are registers that have been added.

TABLE E–4

ASI	Address Space Identifier
CANRESTORE	Restorable Windows
CANSAVE	Savable windows
CLEANWIN	Clean Windows
FPRS	Floating-point Register State
OTHERWIN	Other Windows
PSTATE	Processor State

TABLE E–4 *(Continued)*

TICK	Hardware clock tick-counter
TL	Trap Level
TNPC[MAXTL]	Trap Next Program Counter
TPC[MAXTL]	Trap Program Counter
TSTATE[MAXTL]	Trap State
WSTATE	Windows State

Also, there are sixteen additional double-precision floating-point registers, f[32] .. f[62]. These registers overlap (and are aliased with) eight additional quad-precision floating-point registers, f[32] .. f[60]

The SPARC-V9, CWP register is decremented during a RESTORE instruction, and incremented during a SAVE instruction. This is the opposite of PSR.CWP's behavior in SPARC-V8. This change has no effect on nonprivileged instructions.

E.1.2 Alternate Space Access

Load- and store-alternate instructions to one-half of the alternate spaces can now be included in user code. In SPARC-V9, loads and stores to ASIs 00_{16} .. $7f_{16}$ are privileged; those to ASIs 80_{16} .. FF_{16} are nonprivileged. In SPARC-V8, access to alternate address spaces is privileged.

E.1.3 Byte Order

SPARC-V9 supports both little- and big-endian byte orders for data accesses only; instruction accesses are always performed using big-endian byte order. In SPARC-V8, all data and instruction accesses are performed in big-endian byte order.

E.2 SPARC-V9 Instruction Set Changes

Application software written for the SPARC-V8 processor runs unchanged on a SPARC-V9 processor.

E.2.1 Extended Instruction Definitions to Support the 64-bit Model

TABLE E–5

FCMP, FCMPE	Floating-Point Compare—can set any of the four floating-point condition codes.
LDFSR, STFSR	Load/Store FSR- only affect low-order 32 bits of FSR
LDUW, LDUWA	Same as LD, LDA in SPARC-V8
RDASR/WRASR	Read/Write State Registers - access additional registers
SAVE/RESTORE	
SETHI	
SRA, SRL, SLL, Shifts	Split into 32-bit and 64-bit versions
Tcc	(was Ticc) Operates with either the 32-bit integer condition codes (icc), or the 64-bit integer condition codes (xcc)

All other arithmetic operations operate on 64-bit operands and produce 64-bit results.

E.2.2 Added Instructions to Support 64 bits

TABLE E–6

F[sdq]TOx	Convert floating point to 64-bit word
FxTO[sdq]	Convert 64-bit word to floating point
FMOV[dq]	Floating-Point Move, double and quad
FNEG[dq]	Floating-point Negate, double and quad
FABS[dq]	Floating-point Absolute Value, double and quad
LDDFA, STDFA, LDFA, STFA	Alternate address space forms of LDDF, STDF, LDF, and STF
LDSW	Load a signed word
LDSWA	Load a signed word from an alternate space
LDX	Load an extended word
LDXA	Load an extended word from an alternate space
LDXFSR	Load all 64 bits of the FSR register

TABLE E–6 *(Continued)*

STX	Store an extended word
STXA	Store an extended word into an alternate space
STXFSR	Store all 64 bits if the FSR register

E.2.3 Added Instructions to Support High-Performance System Implementation

TABLE E–7

BPcc	Branch on integer condition code with prediction
BPr	Branch on integer register contents with prediction
CASA, CASXA	Compare and Swap from an alternate space
FBPfcc	Branch on floating-point condition code with prediction
`FLUSHW`	Flush windows
FMOVcc	Move floating-point register if condition code is satisfied
FMOVr	Move floating-point register if integer register satisfies condition
LDQF(A), STQF(A)	Load/Store Quad Floating-point (in an alternate space)
MOVcc	Move integer register if condition code is satisfied
MOVr	Move integer register if register contents satisfy condition
MULX	Generic 64-bit multiply
POPC	Population count
PREFETCH, PREFETCHA	Prefetch Data
SDIVX, UDIVX	Signed and Unsigned 64-bit divide

E.2.4 Deleted Instructions

TABLE E–8

Coprocessor loads and stores	
RDTBR and WRTBR	TBR no longer exists. It is replaced by TBA, which can be read/written with RDPR/WRPR instructions

TABLE E–8 *(Continued)*

RDWIM and WRWIM	WIM no longer exists. WIM has been replaced by several register-window registers
REPSR and WRPSR	PSR no longer exists. It has been replaced by several separate registers that are read/written with other instructions
RETT	Return from trap (replace by DONE/RETRY)
STDFQ	Store Double from Floating-point Queue (replaced by the RDPR FQ instruction

E.2.5 Miscellaneous Instruction Changes

TABLE E–9

IMPDEPn	(Changed) Implementation-dependent instructions (replace SPARC-V8 CPop instructions)
MEMBAR	(Added) Memory barrier (memory synchronization support)

E.3 SPARC-V9 Instruction Set Mapping

TABLE E–10

Opcode	Mnemonic	Argument List	Operation	Comments
BPA	ba{,a} {,pt|,pn}	%icc or %xcc, label	(Branch on cc with prediction) Branch always	1
BPN	bn{,a} {,pt|,pn}	%icc or %xcc, label	Branch never	0
BPNE	bne{,a} {,pt|,pn}	%icc or %xcc, label	Branch on not equal	not Z
BPE	be{,a} {,pt|,pn}	%icc or %xcc, label	Branch on equal	Z
BPG	bg{,a} {,pt|,pn}	%icc or %xcc, label	Branch on greater	not (Z or (N xor V))

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
BPLE	ble{,a} {,pt|,pn}	%icc or %xcc, label	Branch on less or equal	Z or (N xor V)
BPGE	bge{,a} {,pt|,pn}	%icc or %xcc, label	Branch on greater or equal	not (N xor V)
BPL	bl{,a} {,pt|,pn}	%icc or %xcc, label	Branch on less	N xor V
BPGU	bgu{,a} {,pt|,pn}	%icc or %xcc, label	Branch on greater unsigned	not (C or Z)
BPLEU	bleu{,a} {,pt|,pn}	%icc or %xcc, label	Branch on less or equal unsigned	C or Z
BPCC	bcc{,a} {,pt|,pn}	%icc or %xcc, label	Branch on carry clear (greater than or equal, unsigned)	not C
BPCS	bcs{,a} {,pt|,pn}	%icc or %xcc, label	Branch on carry set (less than, unsigned)	C
BPPOS	bpos{,a} {,pt|,pn}	%icc or %xcc, label	Branch on positive	not N
BPNEG	bneg{,a} {,pt|,pn}	%icc or %xcc, label	Branch on negative	N
BPVC	bvc{,a} {,pt|,pn}	%icc or %xcc, label	Branch on overflow clear	not V
BPVS	bvs{,a} {,pt|,pn}	%icc or %xcc, label	Branch on overflow set	V
BRZ	brz{,a} {,pt|,pn}	reg_{rs1}, label	Branch on register zero	Z
BRLEZ	brlez{,a} {,pt|,pn}	reg_{rs1}, label	Branch on register less than or equal to zero	N or Z
BRLZ	brlz{,a} {,pt|,pn}	reg_{rs1}, label	Branch on register less than zero	N

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
BRNZ	brnz{,a} {,pt|,pn}	reg_{rs1}, label	Branch on register not zero	not Z
BRGZ	brgz{,a} {,pt|,pn}	reg_{rs1}, label	Branch on register greater than zero	not (N or Z)
BRGEZ	brgez{,a} {,pt|,pn}	reg_{rs1}, label	Branch on register greater than or equal to zero	not N
CASA	casa casa	[reg_{rs1}]*imm_asi*,reg_{rs2},reg_{rd} [reg_{rs1}]%asi,reg_{rs2},reg_{rd}	Compare and swap word from alternate space	
CASXA	casxa casxa	[reg_{rs1}]*imm_asi*,reg_{rs2},reg_{rd} [reg_{rs1}]%asi,reg_{rs2},reg_{rd}	Compare and swap extended from alternate space	
FBPA	fba{,a} {,pt|,pn}	%fcc*n*, label	(Branch on cc with prediction) Branch never	1
FBPN	fbn{,a} {,pt|,pn}	%fcc*n*, label	Branch always	0
FBPU	fbu{,a} {,pt|,pn}	%fcc*n*, label	Branch on unordered	U
FBPG	fbg{,a} {,pt|,pn}	%fcc*n*, label	Branch on greater	G
FBPUG	fbug{,a} {,pt|,pn}	%fcc*n*, label	Branch on unordered or greater	G or U
FBPL	fbl{,a} {,pt|,pn}	%fcc*n*, label	Branch on less	L
FBPUL	fbul{,a} {,pt|,pn}	%fcc*n*, label	Branch on unordered or less	L or U
FBPLG	fblg{,a} {,pt|,pn}	%fcc*n*, label	Branch on less or greater	L or G
FBPNE	fbne{,a} {,pt|,pn}	%fcc*n*, label	Branch on not equal	L or G or U

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
FBPE	fbe{,a} {,pt|,pn}	%fcc*n*, label	Branch on equal	E
FBPUE	fbue{,a} {,pt|,pn}	%fcc*n*, label	Branch on unordered or equal	E or U
FBPGE	fbge{,a} {,pt|,pn}	%fcc*n*, label	Branch on greater or equal	E or G
FBPUGE	fbuge{,a} {,pt|,pn}	%fcc*n*, label	Branch on unordered or greater or equal	E or G or U
FBPLE	fble{,a} {,pt|,pn}	%fcc*n*, label	Branch on less or equal	E or L
FBPULE	fbule{,a} {,pt|,pn}	%fcc*n*, label	Branch on unordered or less or equal	E or L or u
FBPO	fbo{,a} {,pt|,pn}	%fcc*n*, label	Branch on ordered	E or L or G
FLUSHW	flushw		Flush register windows	
FMOVA	fmov {s,d,q}a	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	(Move on integer cc) Move always	1
FMOVN	fmov {s,d,q}n	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move never	0
FMOVNE	fmov {s,d,q}ne	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if not equal	not Z
FMOVE	fmov {s,d,q}e	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if equal	Z
FMOVG	fmov {s,d,q}g	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if greater	not (Z or (N xor V))
FMOVLE	fmov {s,d,q}le	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if less or equal	Z or (N xor V)
FMOVGE	fmov {s,d,q}ge	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if greater or equal	not (N xor V)

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
FMOVL	fmov {s,d,q}l	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if less	N xor V
FMOVGU	fmov {s,d,q}gu	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if greater unsigned	not (C or Z)
FMOVLEU	fmov {s,d,q}leu	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if less or equal unsigned	C or Z
FMOVCC	fmov {s,d,q}cc	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if carry clear (greater or equal, unsigned)	not C
FMOVCS	fmov {s,d,q}cs	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if carry set (less than, unsigned)	C
FMOVPOS	fmov {s,d,q}pos	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if positive	not N
FMOVNEG	fmov {s,d,q}neg	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if negative	N
FMOVVC	fmov {s,d,q}vc	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if overflow clear	not V
FMOVVS	fmov {s,d,q}vs	%icc or %xcc, $freg_{rs2}$, $freg_{rd}$	Move if overflow set	V
FMOVRZ	fmovr {s,d,q}e	reg_{rs1}, $freg_{rs2}$, $freg_{rd}$	(Move f-p register on cc) Move if register zero	
FMOVRLEZ	fmovr {s,d,q}lz	reg_{rs1}, $freg_{rs2}$, $freg_{rd}$	Move if register less than or equal zero	
FMOVRLZ	fmovr {s,d,q}lz	reg_{rs1}, $freg_{rs2}$, $freg_{rd}$	Move if register less than zero	

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
FMOVRNZ	fmovr	reg_{rs1}, $freg_{rs2}$, $freg_{rd}$	Move if register not zero	
FMOVRGZ	{s,d,q}ne	reg_{rs1}, $freg_{rs2}$, $freg_{rd}$	Move if register greater than zero	
FMOVRGEZ	fmovr	reg_{rs1}, $freg_{rs2}$, $freg_{rd}$	Move if register greater than or equal to zero	
	{s,d,q}gz			
	fmovr			
	{s,d,q}gez			
FMOVFA	fmov{s,d,q}a	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	(Move on floating-point cc)	1
FMOVFN	fmov{s,d,q}n	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move always	0
FMOVFU	fmov{s,d,q}u	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move never	U
FMOVFG	fmov{s,d,q}g	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if unordered	G
FMOVFUG	fmov{s,d,q}ug	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if greater	G or U
FMOVFL	fmov{s,d,q}l	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if unordered or greater	L
FMOVFUL	fmov{s,d,q}ul	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if less	L or U
FMOVFLG	fmov{s,d,q}lg	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if unordered or less	L or G
FMOVFNE	fmov{s,d,q}ne	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if less or greater	L or G or U
FMOVFE	fmov{s,d,q}e	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if not equal	E
FMOVFUE	fmov{s,d,q}ue	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if equal	E or U
FMOVFGE	fmov{s,d,q}ge	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if unordered or equal	E or G
FMOVFUGE	fmov{s,d,q}uge	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if greater or equal	E or G or U
FMOVFLE	fmov{s,d,q}le	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if unordered or greater or equal	E or L
FMOVFULE	fmov{s,d,q}ule	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if less or equal	E or L or u
FMOVFO	fmov{s,d,q}o	%fcc*n*, $freg_{rs2}$, $freg_{rd}$	Move if unordered or less or equal	E or L or G
			Move if ordered	
LDSW	ldsw	[*address*], reg_{rd}	Load a signed word	
LDSWA	ldsw	[*regaddr*] *imm_asi*, reg_{rd}	Load signed word from alternate space	

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
LDX	ldx	[*address*], reg_{rd}	Load extended word	
LDXA	ldxa	[*regaddr*] *imm_asi*, reg_{rd}	Load extended word from alternate space	
LDXFSR	ldxa	[*reg_plus_imm*] %asi, reg_{rd}		
	ldx	[*address*], %fsr	Load floating-point state register	
MEMBAR	membar	*membar_mask*	Memory barrier	

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
MOVA	mova	%icc or %xcc, *reg_or_imm11*, reg_{rd}	(Move integer register on cc)	1
MOVN	movn	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move always	0
MOVNE	movne	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move never	not Z
MOVE	move	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if not equal	Z
MOVG	movg	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if equal	not (Z or (N xor V))
MOVLE	movle	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if greater	Z or (N xor V)
MOVGE	movge	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if less or equal	not (N xor V)
MOVL	movl	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if greater or equal	N xor V
MOVGU	movgu	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if less	not (C or Z)
MOVLEU	movleu	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if greater unsigned	C or Z
MOVCC	movcc	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if less or equal unsigned	not C
MOVCS	movcs	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if carry clear (greater or equal, unsigned)	C
MOVPOS	movpos	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if carry set (less than, unsigned)	not N
MOVNEG	movneg	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if positive	N
MOVVC	movvc	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if negative	not V
MOVVS	movvs	%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if overflow clear	V
		%icc or %xcc, *reg_or_imm11*, reg_{rd}	Move if overflow set	
		%icc or %xcc, *reg_or_imm11*, reg_{rd}		
		%icc or %xcc, *reg_or_imm11*, reg_{rd}		
		%icc or %xcc, *reg_or_imm11*, reg_{rd}		
		%icc or %xcc, *reg_or_imm11*, reg_{rd}		
		%icc or %xcc, *reg_or_imm11*, reg_{rd}		

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
MOVFA	mova	%fccn, *reg_or_imm11*, reg_{rd}	(Move on floating-point cc)	1
MOVFN	movn	%fccn,*reg_or_imm11*, reg_{rd}	Move always	0
MOVFU	movu	%fccn, *reg_or_imm11*, reg_{rd}	Move never	U
MOVFG	movg	%fccn, *reg_or_imm11*, reg_{rd}	Move if unordered	G
MOVFUG	movug	%fccn, *reg_or_imm11*, reg_{rd}	Move if greater	G or U
MOVFL	movl	%fccn, *reg*_or_imm11, reg_{rd}	Move if unordered or greater	L
MOVFUL	movul	%fccn, *reg_or_imm11*, reg_{rd}	Move if less	L or U
MOVFLG	movlg	%fccn, *reg_or_imm11*, reg_{rd}	Move if unordered or less	L or G
MOVFNE	movne	%fccn, *reg_or_imm11*, reg_{rd}	Move if less or greater	L or G or U
MOVFE	move	%fccn, *reg_or_imm11*, reg_{rd}	Move if not equal	E E or U
MOVFUE	movue	%fccn, *reg_or_imm11*, reg_{rd}	Move if equal	E or G
MOVFGE	movge	%fccn, *reg_or_imm11*, reg_{rd}	Move if unordered or equal	E or G or U
MOVFUGE	movuge	%fccn,*reg_or_imm11*,reg_{rd}	Move if greater or equal	E or L
MOVFLE	movle	%fccn, *reg_or_imm11*, reg_{rd}	Move if unordered or greater or equal	E or L or u
MOVFULE	movule	%fccn, *reg_or_imm11*, reg_{rd}	Move if less or equal	E or L or G
MOVFO	movo	%fccn, *reg_or_imm11*, reg_{rd}	Move if unordered or less or equal	
			Move if ordered	
MOVRZ	movre	reg_{rs1}, *reg_or_imm10*,reg_{rd}	(Move register on register cc)	Z
MOVRLEZ	movrlez	reg_{rs1}, *reg_or_imm10*,reg_{rd}	Move if register zero	N or Z
MOVRLZ	movrlz	reg_{rs1}, *reg_or_imm10*,reg_{rd}	Move if register less than or equal to zero	N
MOVRNZ	movrnz	reg_{rs1}, *reg_or_imm10*,reg_{rd}	Move if register less than zero	not Z
MOVRGZ	movrgz	reg_{rs1}, *reg_or_imm10*,reg_{rd}	Move if register not zero	N nor Z
MOVRGEZ	movrgez	reg_{rs1}, *reg_or_imm10*,reg_{rd}	Move if register greater than zero	not N
			Move if register greater than or equal to zero	

TABLE E–10 *(Continued)*

Opcode	Mnemonic	Argument List	Operation	Comments
MULX	mulx	reg_{rs1}, *reg_or_imm*,reg_{rd}	(Generic 64-bit Multiply) Multiply (signed or unsigned)	See SDIVX and UDIVX
POPC	popc	*reg_or_imm*, reg_{rd}	Population count	
PREFETCH PREFETCHA	prefetch prefetcha prefetcha	*[address], prefetch_dcn [regaddr] imm_asi, prefetch_fcn [reg_plus_imm]* %asi, *prefetch_fcn*	Prefetch data Prefetch data from alternate space	See *The SPARC architecture manual, version 9*
SDIVX	sdivx	reg_{rs1}, *reg_or_imm*,reg_{rd}	(64-bit signed divide) Signed Divide	See MULX and UDIVX
STX STXA STXFSR	stx stxa stxa stx	reg_{rd}, *[address]* reg_{rd}, *[*address*] imm_asi* reg_{rd}, *[reg_plus_imm]* %asi %fsr, *[address]*	Store extended word Store extended word into alternate space Store floating-point register (all 64-bits)	
UDIVX	udivx	reg_{rs1}, *reg_or_imm*, reg_{rd}	(64-bit unsigned divide) Unsigned divide	See MULX and SDIVX

E.4 SPARC-V9 Floating-Point Instruction Set Mapping

SPARC-V9 floating-point instructions are shown in the following table.

TABLE E–11

SPARC	Mnemonic[1]	Argument List	Description
F[sdq]TOx	fstox	$freg_{rs2}$, $freg_{rd}$	Convert floating point to 64-bit integer
	fdtox	$freg_{rs2}$, $freg_{rd}$	
	fqtox	$freg_{rs2}$, $freg_{rd}$	
	fstoi	$freg_{rs2}$, $freg_{rd}$	Convert floating-point to 32-bit integer
	fdtoi	$freg_{rs2}$, $freg_{rd}$	
	fqtoi	$freg_{rs2}$, $freg_{rd}$	

TABLE E–11 *(Continued)*

SPARC	Mnemonic[1]	Argument List	Description
FxTO[sdq]	fxtos	*freg*$_{rs2}$, *freg*$_{rd}$	Convert 64-bit integer to floating point
	fxtod	*freg*$_{rs2}$, *freg*$_{rd}$	
	fxtoq	*freg*$_{rs2}$, *freg*$_{rd}$	
	fitos	*freg*$_{rs2}$, *freg*$_{rd}$	Convert 32-bit integer to floating point
	fitod	*freg*$_{rs2}$, *freg*$_{rd}$	
	fitoq	*freg*$_{rs2}$, *freg*$_{rd}$	
FMOV[dq]	fmovd	freg$_{rs2}$, freg$_{rd}$	Move double
	fmovq	freg$_{rs2}$, freg$_{rd}$	Move quad
FNEG[dq]	fnegd	freg$_{rs2}$, freg$_{rd}$	Negate double
	fnegq	freg$_{rs2}$, freg$_{rd}$	Negate quad
FABS[dq]	fabsd	freg$_{rs2}$, freg$_{rd}$	Absolute value double
	fabsq	freg$_{rs2}$, freg$_{rd}$	Absolute value quad
LDFA	lda	[*regaddr*] *imm_asi*, *freg*$_{rd}$	Load floating-point register from alternate space
LDDFA	lda	[*reg_plus_imm*] %asi, *freg*$_{rd}$	Load double floating-point register from alternate space.
LDQFA	ldda	[*regaddr*] *imm_asi*, *freg*$_{rd}$	Load quad floating-point register from alternate space
	ldda	[*reg_plus_imm*] %asi, *freg*$_{rd}$	
	ldqa	[*regaddr*] *imm_asi*, *freg*$_{rd}$	
	ldqa	[*reg_plus_imm*] %asi, *freg*$_{rd}$	
STFA	sta	*freg*$_{rd}$, [*regaddr*] *imm_asi*	Store floating-point register to alternate space
STDFA	sta	*freg*$_{rd}$, [*reg_plus_imm*] %asi	Store double floating-point register to alternate space
STQFA	stda	*freg*$_{rd}$, [*regaddr*] *imm_asi*	Store quad floating-point register to alternate space
	stda	*freg*$_{rd}$, [*reg_plus_imm*] %asi	
	stqa	*freg*$_{rd}$, [*regaddr*] *imm_asi*	
	stqa	*freg*$_{rd}$, [*reg_plus_imm*] %asi	

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

   `i` **32-bit integer**

   `x` **64-bit integer**

   `s` **single**

   `d` **double**

   `q` **quad**

E.5 SPARC-V9 Synthetic Instruction-Set Mapping

Here is a mapping of synthetic instructions to hardware equivalent instructions.

TABLE E–12

Synthetic Instruction		Hardware Equivalent(s)		Comment
`cas`	[reg_{rsl}], reg_{rs2}, reg_{rd}	`casa`	[reg_{rsl}]ASI_P, reg_{rs2}, reg_{rd}	Compare & swap (cas)
`casl`	[reg_{rsl}], reg_{rs2}, reg_{rd}	`casa`	[reg_{rsl}]ASI_P_L, reg_{rs2}, reg_{rd}	cas little-endian
`casx`	[reg_{rsl}], reg_{rs2}, reg_{rd}	`casxa`	[reg_{rsl}]ASI_P, reg_{rs2}, reg_{rd}	cas extended
`casxl`	[reg_{rsl}], reg_{rs2}, reg_{rd}	`casxa`	[reg_{rsl}]ASI_P_L, reg_{rs2}, reg_{rd}	cas little-endian, extended
`clrx`	[*address*]	`stx`	`%g0`, [*address*]	Clear extended word
`clruw`	reg_{rs1}, reg_{rd}	`srl`	reg_{rs1}, %g0, reg_{rd}	Copy and clear upper word
`clruw`	reg_{rd}	`srl`	reg_{rd}, %g0, reg_{rd}	Clear upper word
`iprefetch`	*label*	`bn`, `pt`	%xcc, *label*	Instruction prefetch,
`mov`	%y, reg_{rd}	`rd`	%y, reg_{rd}	
`mov`	%asr*n*, reg_{rd}	`rd`	%asr*n*, reg_{rd}	
`mov`	*reg_or_imm*, %asr*n*	`wr`	*%g0, reg_or_imm*, %asr*n*	
`ret`		`jmpl`	%i7+8, %g0	Return from subroutine
`retl`		`jmpl`	%o7+8, %g0	Return from leaf subroutine
`setn`	*value, r1, r2*	for `-xarch=v9` same as `setx` *value r1, r2*		
		for `-xarch=v8` same as `set` *value r2*		
`setnhi`	*value, r1, r2*	for `-xarch=v9` same as `setxhi` *value r1, r2*		
		for `-xarch=v8` same as `sethi` *value r2*		

TABLE E–12 *(Continued)*

Synthetic Instruction		Hardware Equivalent(s)		Comment
`setuw`	*value*,*reg*$_{rd}$	`sethi`	`%hi`(*value*), *reg*$_{rd}$	(value & $3FF_{16}$)==0
		`or`	`%g0`, *value*, *reg*$_{rd}$	when 0 ≤ value ≤ 4095
		`sethi`	`%hi`(*value*), *reg*$_{rd;}$	(otherwise)
		`or`	*reg*$_{rd}$, `%lo`(*value*), *reg*$_{rd}$	Do not use setuw in a DCTI delay slot.
`setsw`	*value*,*reg*$_{rd}$	`sethi`	`%hi`(*value*), *reg*$_{rd}$	value>=0 and (value & $3FF_{16}$)==0
		`or`	`%g0`, *value*, *reg*$_{rd}$	-4096 ≤ value ≤ 4095
		`sethi`	`%hi`(*value*), *reg*$_{rd}$	if (value<0) and ((value & 3FF)==0)
		`sra`	*reg*$_{rd}$, `%g0`, *reg*$_{rd}$	(otherwise, if value>=0)
		`sethi`	`%hi`(*value*), *reg*$_{rd;}$	(otherwise, if value<0)
		`or`	*reg*$_{rd}$, `%lo`(*value*), *reg*$_{rd}$	Do not use setsw in a CTI delay slot.
		`sethi`	`%hi`(*value*), *reg*$_{rd}$;	
		`or`	*reg*$_{rd}$, `%lo`(*value*), *reg*$_{rd}$	
		`sra`	*reg*$_{rd}$, `%g0`, *reg*$_{rd}$	
`setx`	*value, r1, r2*	`sethi`	%hh(*value*), r1	
		`or`	r1, %hm(*value*), r1	
		`sethi`	%lm(*value*), r2	
		`or`	r2, %lo(*value*), r2	
		`sllx`	r1, 32, r1	
		`or`	r1, r2, r2	
`setxhi`	*value r1, r2*	`sethi`	%hh(value), r1	
		`or`	r1, %hm(value), r1	
		`sethi`	%lm(value), r2	
		`sllx`	r1, 32, r1	
		`or`	r1, r2, r2	
`signx`	*reg*$_{rsl,}$ *reg*$_{rd}$	`sra`	*reg*$_{rsl,}$ %g0, *reg*$_{rd}$	Sign-extend 32-bit value to 64 bits
`signx`	*reg*$_{rd}$	`sra`	*reg*$_{rd,}$ %g0, *reg*$_{rd}$	

E.6 UltraSPARC and VIS Instruction Set Extensions

This section describes extensions that require SPARC-V9. The extensions support enhanced graphics functionality and improved memory access efficiency.

Note – SPARC-V9 instruction set extensions used in executables may not be portable to other SPARC-V9 systems.

E.6.1 Graphics Data Formats

The overhead of converting to and from floating-point arithmetic is high, so the graphics instructions are optimized for short-integer arithmetic. Image components are 8 or 16 bits. Intermediate results are 16 or 32 bits.

E.6.2 Eight-bit Format

A 32-bit word contains pixels of four unsigned 8-bit integers. The integers represent image intensity values (α , G, B, R). Support is provided for *band interleaved* images (store color components of a point), and *band sequential* images (store all values of one color component).

E.6.3 Fixed Data Formats

A 64-bit word contains four 16-bit signed fixed-point values. This is the fixed 16-bit data format.

A 64-bit word contains two 8-bit signed fixed-point values. This is the fixed 32-bit data format.

Enough precision and dynamic range (for filtering and simple image computations on pixel values) can be provided by an intermediate format of fixed data values. Pixel multiplication is used to convert from pixel data to fixed data. Pack instructions are used to convert from fixed data to pixel data (clip and truncate to an 8-bit unsigned value). The FPACKFIX instruction supports conversion from 32-bit fixed to 16-bit fixed. Rounding is done by adding one to the rounding bit position. You should use floating-point data to perform complex calculations needing more precision or dynamic range.

E.6.4 SHUTDOWN Instruction

All outstanding transactions are completed before the SHUTDOWN instruction completes.

TABLE E–13

SPARC	Mnemonic	Argument List	Description
`SHUTDOWN`	`shutdown`		shutdown to enter power down mode

E.6.5 Graphics Status Register (GSR)

You use ASR 0x13 instructions RDASR and WRASR to access the Graphics Status Register.

TABLE E–14

SPARC	Mnemonic	Argument List	Description
`RDASR`	`rdasr`	%gsr, reg_{rd}	read GSR
`WRASR`	`wrasr`	reg_{rs1}, *reg_or_imm*, %gsr	write GSR

E.6.6 Graphics Instructions

Unless otherwise specified, floating-point registers contain all instruction operands. There are 32 double-precision registers. Single-precision floating-point registers contain the pixel values, and double-precision floating-point registers contain the fixed values.

The opcode space reserved for the Implementation-Dependent Instruction1 (IMPDEP1) instructions is where the graphics instruction set is mapped.

Partitioned add/subtract instructions perform two 32-bit or four 16-bit partitioned adds or subtracts between the source operands corresponding fixed point values.

TABLE E–15

SPARC	Mnemonic	Argument List	Description
FPADD16	fpadd16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	four 16-bit add
FPADD16S	fpadd16s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	two 16-bit add
FPADD32	fpadd32	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	two 32-bit add
FPADD32S	fpadd32s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	one 32-bit add
FPSUB16	fpsub16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	four 16-bit subtract
FPSUB16S	fpsub16s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	two 16-bit subtract
FPSUB32	fpsub32	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	two 32-bit subtract
FPSUB32S	fpsub32s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	one 32-bit subtract

Pack instructions convert to a lower pixel or precision fixed format.

TABLE E–16

SPARC	Mnemonic	Argument List	Description
FPACK16	fpack16	$freg_{rs2}$, $freg_{rd}$	four 16-bit packs
FPACK32	fpack32	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	two 32-bit packs
FPACKFIX	fpackfix	$freg_{rs2}$, $freg_{rd}$	four 16-bit packs
FEXPAND	fexpand	$freg_{rs2}$, $freg_{rd}$	four 16-bit expands
FPMERGE	fpmerge	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	two 32-bit merges

Partitioned multiply instructions have the following variations.

TABLE E–17

SPARC	Mnemonic	Argument List	Description
FMUL8x16	fmul8x16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	8x16-bit partition
FMUL8x16AU	fmul8x16au	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	8x16-bit upper α partition
FMUL8x16AL	fmul8x16al	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	8x16-bit lower α partition
FMUL8SUx16	fmul8sux16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	upper 8x16-bit partition
FMUL8ULx16	fmul8ulx16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	lower unsigned 8x16-bit partition
FMULD8SUx16	fmuld8sux16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	upper 8x16-bit partition
FMULD8ULx16	fmuld8ulx16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	lower unsigned 8x16-bit partition

Alignment instructions have the following variations.

TABLE E–18

SPARC	Mnemonic	Argument List	Description
ALIGNADDRESS	alignaddr	reg_{rs1}, reg_{rs2}, reg_{rd}	find misaligned data access address
ALIGNADDRESS_LITTLE	alignaddrl	reg_{rs1}, reg_{rs2}, reg_{rd}	same as above, but little-endian
FALIGNDATA	faligndata	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	do misaligned data, data alignment

Logical operate instructions perform one of sixteen 64-bit logical operations between *rs1* and *rs2* (in the standard 64-bit version).

TABLE E–19

SPARC	Mnemonic	Argument List	Description
FZERO	fzero	$freg_{rd}$	zero fill
FZEROS	fzeros	$freg_{rd}$	zero fill, single precision
FONE	fone	$freg_{rd}$	one fill
FONES	fones	$freg_{rd}$	one fill, single precision
FSRC1	fsrc1	$freg_{rs1}$, $freg_{rd}$	copy src1
FSRC1S	fsrc1s	$freg_{rs1}$, $freg_{rd}$	copy src1, single precision
FSRC2	fsrc2	$freg_{rs2}$, $freg_{rd}$	copy src2
FSRC2S	fsrc2s	$freg_{rs2}$, $freg_{rd}$	copy src2, single precision
FNOT1	fnot1	$freg_{rs1}$, $freg_{rd}$	negate src1, 1's complement
FNOT1S	fnot1s	$freg_{rs1}$, $freg_{rd}$	same as above, single precision
FNOT2	fnot2	$freg_{rs2}$, $freg_{rd}$	negate src2, 1's complement
FNOT2S	fnot2s	$freg_{rs2}$, $freg_{rd}$	same as above, single precision
FOR	for	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical OR
FORS	fors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical OR, single precision
FNOR	fnor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical NOR
FNORS	fnors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical NOR, single precision
FAND	fand	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical AND
FANDS	fands	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical AND, single precision
FNAND	fnand	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical NAND
FNANDS	fnands	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical NAND, single precision

TABLE E–19 *(Continued)*

SPARC	Mnemonic	Argument List	Description
FXOR	fxor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical XOR
FXORS	fxors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical XOR, single precision
FXNOR	fxnor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical XNOR
FXNORS	fxnors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	logical XNOR, single precision
FORNOT1	fornot1	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	negated src1 OR src2
FORNOT1S	fornot1s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	same as above, single precision
FORNOT2	fornot2	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	src1 OR negated src2
FORNOT2S	fornot2s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	same as above, single precision
FANDNOT1	fandnot1	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	negated src1 AND src2
FANDNOT1S	fandnot1s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	same as above, single precision
FANDNOT2	fandnot2	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	src1 AND negated src2
FANDNOT2S	fandnot2s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	same as above, single precision

Pixel compare instructions compare fixed-point values in *rs1* and *rs2* (two 32 bit or four 16 bit)

TABLE E–20

SPARC	Mnemonic	Argument List	Description
FCMPGT16	fcmpgt16	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}	4 16-bit compare, set rd if src1>src2
FCMPGT32	fcmpgt32	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}	2 32-bit compare, set rd if src1>src2
FCMPLE16	fcmple16	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}	4 16-bit compare, set rd if src1≤src2
FCMPLE32	fcmple32	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}	2 32-bit compare, set rd if src1≤src2
FCMPNE16	fcmpne16	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}	4 16-bit compare, set rd if src1≠src2
FCMPNE32	fcmpne32	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}	2 32-bit compare, set rd if src1≠src2
FCMPEQ16	fcmpeq16	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}	4 16-bit compare, set rd if src1=src2
FCMPEQ32	fcmpeq32	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}	2 32-bit compare, set rd if src1=src2

Edge handling instructions handle the boundary conditions for parallel pixel scan line loops.

TABLE E–21

SPARC	Mnemonic	Argument List	Description
EDGE8	edge8	reg_{rs1}, reg_{rs2}, reg_{rd}	8 8-bit edge boundary processing
EDGE8L	edge8l	reg_{rs1}, reg_{rs2}, reg_{rd}	same as above, little-endian
EDGE16	edge16	reg_{rs1}, reg_{rs2}, reg_{rd}	4 16-bit edge boundary processing
EDGE16L	edge16l	reg_{rs1}, reg_{rs2}, reg_{rd}	same as above, little-endian
EDGE32	edge32	reg_{rs1}, reg_{rs2}, reg_{rd}	2 32-bit edge boundary processing
EDGE32L	edge32l	reg_{rs1}, reg_{rs2}, reg_{rd}	same as above, little-endian

Pixel component distance instructions are used for motion estimation in video compression algorithms.

TABLE E–22

SPARC	Mnemonic	Argument List	Description
PDIST	pdist	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	8 8-bit components, distance between

The three-dimensional array addressing instructions convert three- dimensional fixed-point addresses (in rs1) to a blocked-byte address. The result is stored in rd.

TABLE E–23

SPARC	Mnemonic	Argument List	Description
ARRAY8	array8	reg_{rs1}, reg_{rs2}, reg_{rd}	convert 8-bit 3-D address to blocked byte address
ARRAY16	array16	reg_{rs1}, reg_{rs2}, reg_{rd}	same as above, but 16-bit
ARRAY32	array32	reg_{rs1}, reg_{rs2}, reg_{rd}	same as above, but 32-bit

E.6.7 Memory Access Instructions

These memory access instructions are part of the SPARC-V9 instruction set extensions.

TABLE E–24

SPARC	imm_asi	Argument List	Description
STDFA	ASI_PST8_P	stda $freg_{rd}$, [$freg_{rs1}$] $reg_{mask,}$ *imm_asi*	eight 8-bit conditional stores to:
STDFA	ASI_PST8_S		primary address space
STDFA	ASI_PST8_PL		secondary address space
STDFA	ASI_PST8_SL		primary address space, little endian
			secondary address space, little endian
STDFA	ASI_PST16_P		four 16-bit conditional stores to:
STDFA	ASI_PST16_S		primary address space
STDFA	ASI_PST16_PL		secondary address space
STDFA	ASI_PST16_SL		primary address space, little endian
			secondary address space, little endian
STDFA	ASI_PST32_P		two 32-bit conditional stores to:
STDFA	ASI_PST32_S		primary address space
STDFA	ASI_PST32_PL		secondary address space
STDFA	ASI_PST32_SL		primary address space, little endian
			secondary address space, little endian

Note – To select a partial store instruction, use one of the partial store ASIs with the STDA instruction.

TABLE E–25

SPARC	imm_asi	Argument List	Description
LDDFA	ASI_FL8_P	ldda [*reg_addr*] *imm_asi*, $freq_{rd}$	8-bit load/store from/to:
STDFA		stda $freq_{rd}$, [*reg_addr*] *imm_asi*	primary address space
LDDFA	ASI_FL8_S	ldda [*reg_plus_imm*] %asi, $freq_{rd}$	secondary address space
STDFA		stda [*reg_plus_imm*] %asi	
LDDFA	ASI_FL8_PL		primary address space, little endian
STDFA			

TABLE E–25 *(Continued)*

SPARC	imm_asi	Argument List	Description
LDDFA	ASI_FL8_SL		secondary address space, little endian
STDFA			
LDDFA	ASI_FL16_P		16-bit load/store from/to: primary address space
STDFA			
LDDFA	ASI_FL16_S		secondary address space
STDFA			
LDDFA	ASI_FL16_PL		primary address space, little endian
STDFA			
LDDFA	ASI_FL16_SL		secondary address space, little endian
STDFA			

Note – To select a short floating-point load and store instruction, use one of the short ASIs with the LDDA and STDA instructions.

TABLE E–26

SPARC	imm_asi	Argument List	Description
LDDA	ASI_NUCLEUS_QUAD_LDD	[*reg_addr*] *imm_asi*, reg_{rd}	128-bit atomic load
LDDA	ASI_NUCLEUS_QUAD_LDD_L	[*reg_plus_imm*] %asi, reg_{rd}	128-bit atomic load, little endian
LDDFA	ASI_BLK_AIUP	ldda [*reg_addr*] *imm_asi*, $freq_{rd}$	64-byte block load/store from/to: primary address space, user privilege
STDFA		stda $freq_{rd}$, [*reg_addr*] *imm_asi*	
LDDFA	ASI_BLK_AIUS	ldda [reg_plus_imm] %asi, $freq_{rd}$	secondary address space, user privilege.
STDFA		stda $freg_{rd}$, [*reg_plus_imm*] %asi	
LDDFA	ASI_BLK_AIUPL		primary address space, user privilege, little endian
STDFA			

TABLE E–26 *(Continued)*

SPARC	imm_asi	Argument List	Description
LDDFA STDFA	ASI_BLK_AIUSL		secondary address space, user privilege little endian
LDDFA STDFA	ASI_BLK_P		primary address space
LDDFA STDFA	ASI_BLK_S		secondary address space
LDDFA STDFA	ASI_BLK_PL		primary address space, little endian
LDDFA STDFA	ASI_BLK_SL		secondary address space, little endian
LDDFA STDFA	ASI_BLK_COMMIT_P		64-byte block commit store to primary address space
LDDFA STDFA	ASI_BLK_COMMIT_S		64-byte block commit store to secondary address space

Note – To select a block load and store instruction, use one of the block transfer ASIs with the LDDA and STDA instructions.

Index

F

G

H

I

K

L

M

N

T

U

V

W

X