

SPARC Assembly Language Reference Manual

Beta

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface	7
1 SPARC Assembler for <i>SunOS 5.x</i>	11
1.1 Operating Environment	11
1.2 SPARC Assembler for SunOS 4.1 Versus <i>SunOS 5.x</i>	11
1.2.1 Labeling Format	11
1.2.2 Object File Format	12
1.2.3 Pseudo-Operations	12
1.2.4 Command Line Options	12
2 Assembler Syntax	13
2.1 Syntax Notation	13
2.2 Assembler File Syntax	14
2.2.1 Lines Syntax	14
2.2.2 Statement Syntax	14
2.3 Lexical Features	14
2.3.1 Case Distinction	14
2.3.2 Comments	14
2.3.3 Labels	15
2.3.4 Numbers	15
2.3.5 Strings	15
2.3.6 Symbol Names	16
2.3.7 Special Symbols - Registers	16
2.3.8 Operators and Expressions	18
2.3.9 SPARC V9 Operators and Expressions	19
2.4 Assembler Error Messages	20

3 Executable and Linking Format	21
3.1 ELF Header	22
3.2 Sections	23
3.2.1 Section Header	24
3.2.2 Predefined User Sections	27
3.2.3 Predefined Non-User Sections	29
3.3 Locations	30
3.4 Addresses	30
3.5 Relocation Tables	30
3.6 Symbol Tables	30
3.7 String Tables	32
3.8 Assembler Directives	32
3.8.1 Section Control Directives	33
3.8.2 Symbol Attribute Directives	33
3.8.3 Assignment Directive	33
3.8.4 Data Generating Directives	33
4 Converting Files to the New Format	35
4.1 Conversion Instructions	35
4.2 Examples	35
5 Instruction-Set Mapping	37
5.1 Table Notation	37
5.2 Integer Instructions	39
5.3 Floating-Point Instruction	46
5.4 Coprocessor Instructions	48
5.5 Synthetic Instructions	48
5.6 V8/V9 Natural Pseudo Instructions	50
A Pseudo-Operations	53
A.1 Alphabetized Listing with Descriptions	53
B Examples of Pseudo-Operations	61
B.1 Example 1	61

B.2 Example 2	62
B.3 Example 3	62
B.4 Example 4	63
B.5 Example 5	63
C Using the Assembler Command Line	65
C.1 Assembler Command Line	65
C.2 Assembler Command Line Options	66
C.3 Disassembling Object Code	69
D An Example Language Program	71
E SPARC-V9 Instruction Set	77
E.1 SPARC-V9 Changes	77
E.1.1 Registers	77
E.1.2 Alternate Space Access	79
E.1.3 Byte Order	79
E.2 SPARC-V9 Instruction Set Changes	79
E.2.1 Extended Instruction Definitions to Support the 64-Bit Model	79
E.2.2 Added Instructions to Support 64 Bits	80
E.2.3 Added Instructions to Support High-Performance System Implementation	81
E.2.4 Deleted Instructions	81
E.2.5 Miscellaneous Instruction Changes	82
E.3 SPARC-V9 Instruction Set Mapping	82
E.4 SPARC-V9 Floating-Point Instruction Set Mapping	90
E.5 SPARC-V9 Synthetic Instruction-Set Mapping	91
E.6 UltraSPARC and VIS Instruction Set Extensions	93
E.6.1 Graphics Data Formats	94
E.6.2 Eight-bit Format	94
E.6.3 Fixed Data Formats	94
E.6.4 SHUTDOWN Instruction	94
E.6.5 Graphics Status Register (GSR)	94
E.6.6 Graphics Instructions	95
E.6.7 Memory Access Instructions	99

Index 103

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, “SPARC Assembler for SunOS 5.x,”](#) discusses features of the SunOS 5.x SPARC Assembler.

[Chapter 2, “Assembler Syntax,”](#) describes the syntax of the SPARC assembler that takes assembly programs and produces relocatable object files for processing by the link editor.

[Chapter 3, “Executable and Linking Format,”](#) describes the relocatable ELF files that hold code and data suitable for linking with other object files.

[Chapter 4, “Converting Files to the New Format,”](#) describes how to convert existing SunOS 4.1 SPARC assembly files to the SunOS 5.x assembly file format.

[Chapter 5, “Instruction-Set Mapping,”](#) describes the relationship between hardware instructions of the SPARC architecture and the assembly language instruction set.

[Appendix A, “Pseudo-Operations,”](#) lists the pseudo-operations supported by the SPARC assembler.

[Appendix B, “Examples of Pseudo-Operations,”](#) shows some examples of ways to use various pseudo-operations.

[Appendix C, “Using the Assembler Command Line,”](#) describes the available assembler command-line options.

[Appendix D, “An Example Language Program,”](#) describes an example C language program with comments to show correspondence between the assembly code and the C code.

[Appendix E, “SPARC-V9 Instruction Set,”](#) describes the SPARC-V9 instruction set and the changes due to the SPARC-V9 implementation.

Documentation, Support, and Training

See the following web sites for additional resources:

- [Documentation \(http://docs.sun.com\)](http://docs.sun.com)
- [Support \(http://www.oracle.com/us/support/systems/index.html\)](http://www.oracle.com/us/support/systems/index.html)
- [Training \(http://education.oracle.com\)](http://education.oracle.com) – Click the Sun link in the left navigation bar.

Oracle Software Resources

Oracle Technology Network (<http://www.oracle.com/technetwork/index.html>) offers a range of resources related to Oracle software:

- Discuss technical problems and solutions on the [Discussion Forums](http://forums.oracle.com) (<http://forums.oracle.com>).
- Get hands-on step-by-step tutorials with [Oracle By Example](http://www.oracle.com/technetwork/tutorials/index.html) (<http://www.oracle.com/technetwork/tutorials/index.html>).
- Download [Sample Code](http://www.oracle.com/technology/sample_code/index.html) (http://www.oracle.com/technology/sample_code/index.html).

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P-1 Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% you have mail.</code>
AaBbCc123	What you type, contrasted with onscreen computer output	<code>machine_name% su</code> Password:
<i>aabbcc123</i>	Placeholder: replace with a real name or value	The command to remove a file is <i>rm filename</i> .
<i>AaBbCc123</i>	Book titles, new terms, and terms to be emphasized	Read <i>Chapter 6</i> in the <i>User's Guide</i> . <i>A cache</i> is a copy that is stored locally. Do <i>not</i> save the file. Note: Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P-2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	\$
Bash shell, Korn shell, and Bourne shell for superuser	#
C shell	machine_name%
C shell for superuser	machine_name#

SPARC Assembler for *SunOS 5.x*

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](#), “SPARC-V9 Instruction Set.”

1.1 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.

1.2 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.

1.2.1 Labeling Format

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

1.2.2 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.

1.2.3 Pseudo-Operations

See [Appendix A, “Pseudo-Operations,”](#) for a detailed description of the pseudo-operations (pseudo-ops).

1.2.4 Command Line Options

See [Appendix C, “Using the Assembler Command Line,”](#) for a detailed description of command line options and a list of SPARC architectures.

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” on page 13
- “2.2 Assembler File Syntax” on page 14
- “2.3 Lexical Features” on page 14
- “2.4 Assembler Error Messages” on page 20

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

Description: is a symbol name.

instruction

Description: 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 nb , and before its definition (forward reference) as nf .

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 INFINITY* are specified as `0r-nan` and `0r-inf`.

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

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 Codes Recognized in Strings

Escape Code	Description
\a	Alert
\b	Backspace
\f	Form feed
\n	Newline (line feed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\nnn	Octal value <i>nnn</i>
\xnn...	Hexadecimal value <i>nn...</i>

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 Special Symbol Names

Symbol Object	Name	Comment
General-purpose registers	%r0 ... %r31	
General-purpose global registers	%g0 ... %g7	Same as %r0 ... %r7
General-purpose out registers	%o0 ... %o7	Same as %r8 ... %r15
General-purpose local registers	%l0 ... %l7	Same as %r16 ... %r23
General-purpose in registers	%i0 ... %i7	Same as %r24 ... %r31
Stack-pointer register	%sp	(%sp = %o6 = %r14)
Frame-pointer register	%fp	(%fp = %i6 = %r30)
Floating-point registers	%f0 ... %f31	
Floating-point status register	%fsr	
Front of floating-point queue	%fq	
Coprocessor registers	%c0 ... %c31	
Coprocessor status register	%csr	
Coprocessor queue	%cq	
Program status register	%psr	
Trap vector base address register	%tbr	
Window invalid mask	%wim	
Y register	%y	
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 Operators Recognized in Constant Expressions

Binary	Operators	Unary	Operators
+	Integer addition	+	(No effect)
-	Integer subtraction	-	2's Complement
*	Integer multiplication	~	1's Complement
/	Integer division	%lo(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.
<<	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		

TABLE 2-3 Operators Recognized in Constant Expressions (Continued)

Binary	Operators	Unary	Operators
&	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 V9 64-bit Operators and Expressions

Unary	Calculation	Operators
<code>%hh</code>	<code>(address) >> 42</code>	Extract bits 42-63 of a 64-bit word
<code>%hm</code>	<code>((address) >> 32) & 0x3ff</code>	Extract bits 32-41 of a 64-bit word
<code>%lm</code>	<code>((address) >> 10) & 0x3ffff</code>	Extract bits 10-31 of a 64-bit word

For example:

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

```
sethi %lm (address), %l2
or %l2, %lo (address), %l2
```

```
sllx %l1, 32, %l1
or %l1, %l2, %l1
```

The V9 high 32-bit operators and expressions are identified in [Table 2-5](#).

TABLE 2-5 V9 32-bit Operators and Expressions

Unary	Calculation	Operators
<code>%hix</code>	<code>((address) ^ 0xffffffff >> 10) & 0x4ffff</code>	Invert every bit and extract bits 10-31

TABLE 2-5 V9 32-bit Operators and Expressions (Continued)

Unary	Calculation	Operators
%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 Low 44-Bit Operators and Expressions

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, %l44 (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.

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:

- “3.1 ELF Header” on page 22
- “3.2 Sections” on page 23
- “3.3 Locations” on page 30
- “3.5 Relocation Tables” on page 30
- “3.6 Symbol Tables” on page 30
- “3.4 Addresses” on page 30
- “3.7 String Tables” on page 32
- “3.8 Assembler Directives” on page 32

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`

Description: ELF header size in bytes.

`entry`

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

`flag`

Description: Processor-specific flags associated with the file.

`ident`

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

`machine`

Description: Specifies the required architecture for an individual file. A value of 2 specifies SPARC.

`phentsize`

Description: Size in bytes of entries in the program header table. All entries are the same size.

`phnum`

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

`phoff`

Description: Program header table file offset in bytes. The value of 0 indicates no program header.

`shentsize`

Description: 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`

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

shoff

Description: Section header table file offset in bytes. The value of 0 indicates no section header.

shstrndx

Description: 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

Description: Identifies the object file type. [Table 3–1](#) describes the reserved object file types.

version

Description: Identifies the object file version.

[Table 3–1](#) shows reserved object file types.

TABLE 3–1 Reserved Object File Types

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

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`

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

`addralign`

Description: 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`

Description: Size in bytes for entries in fixed-size tables such as the symbol table.

`flags`

Description: One-bit descriptions of section attributes. [Table 3–2](#) describes the section attribute flags.

TABLE 3-2 Section Attribute Flags

Flag	Default Value	Description
SHF_WRITE	0x1	Contains data that is writable during process execution.
SHF_ALLOC	0x2	Occupies memory during process execution. This attribute is <i>off</i> 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`

Description: Extra information. The interpretation of this information depends on the section type, as described in [Table 3-3](#).

`link`

Description: Section header table index link. The interpretation of this information depends on the section type, as described in [Table 3-3](#).

`name`

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

`offset`

Description: 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`

Description: 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`

Description: Categorizes the section contents and semantics. [Table 3-3](#) describes the section types.

TABLE 3-3 Section Types

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. <i>Note: Only one section of this type is allowed in a file</i>	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. <i>Note: Only one section of this type is allowed in a file</i>	0	The section header index of the symbol table to which the hash table applies.
dynamic	6	Contains dynamic linking information. <i>Note: Only one section of this type is allowed in a file</i>	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.
shlib	10	Reserved.		

TABLE 3-3 Section Types (Continued)

Name	Value	Description	Interpretation by	
			info	link
dynsym	11	Contains a symbol table with a minimal set of symbols for dynamic linking. <i>Note: Only one section of this type is allowed in a file</i>	One greater than the symbol table index of the last local symbol.	The section header index of the associated string table.
loproc	0x70000000	Lower and upper bound of range reserved for processor-specific semantics.		
hiproc	0x7fffffff			
louser	0x80000000	Lower and upper bound of range reserved for application programs. <i>Note: Section types in this range may be used by an application without conflicting with system-defined section types.</i>		
hiuser	0xffffffff			

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 User Sections In Section Control Directives

Section Name	Description
.bss	Section contains uninitialized read-write data.
.comment	Comment section.
.data & .data1	Section contains initialized read-write data.
.debug	Section contains debugging information.
.fini	Section contains runtime finalization instructions.
.init	Section contains runtime initialization instructions.

TABLE 3-4 User Sections In Section Control Directives (Continued)

Section Name	Description
<code>.rodata & .rodata1</code>	Section contains read-only data.
<code>.text</code>	Section contains executable text.
<code>.line</code>	Section contains line # info for symbolic debugging.
<code>.note</code>	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 recommended that the only code added to these sections be in the following form:

```
call routine_namenop
```

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_namenop
```

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 Sections Not In Section Control Directives

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

Description: 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

Description: 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

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

info

Description: Specifies the symbol type and binding attributes. [Table 3–6](#) and [Table 3–7](#) describe these values.

other

Description: Undefined meaning. Current value is 0.

shndx

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

TABLE 3–6 Symbol Type Attributes

Value	Type	Description
0	notype	Type not specified.
1	object	<i>Symbol</i> is associated with a data object; for example, a variable or an array.
2	func	<i>Symbol</i> 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	<i>Symbol</i> 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 Symbol Binding Attributes

Value	Binding	Description
0	local	<i>Symbol</i> 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	<i>Symbol</i> is either defined externally or defined in the object file and accessible in other files.
2	weak	<i>Symbol</i> 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, “Pseudo-Operations,”](#) 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.

Converting Files to the New Format

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

4.1 Conversion Instructions

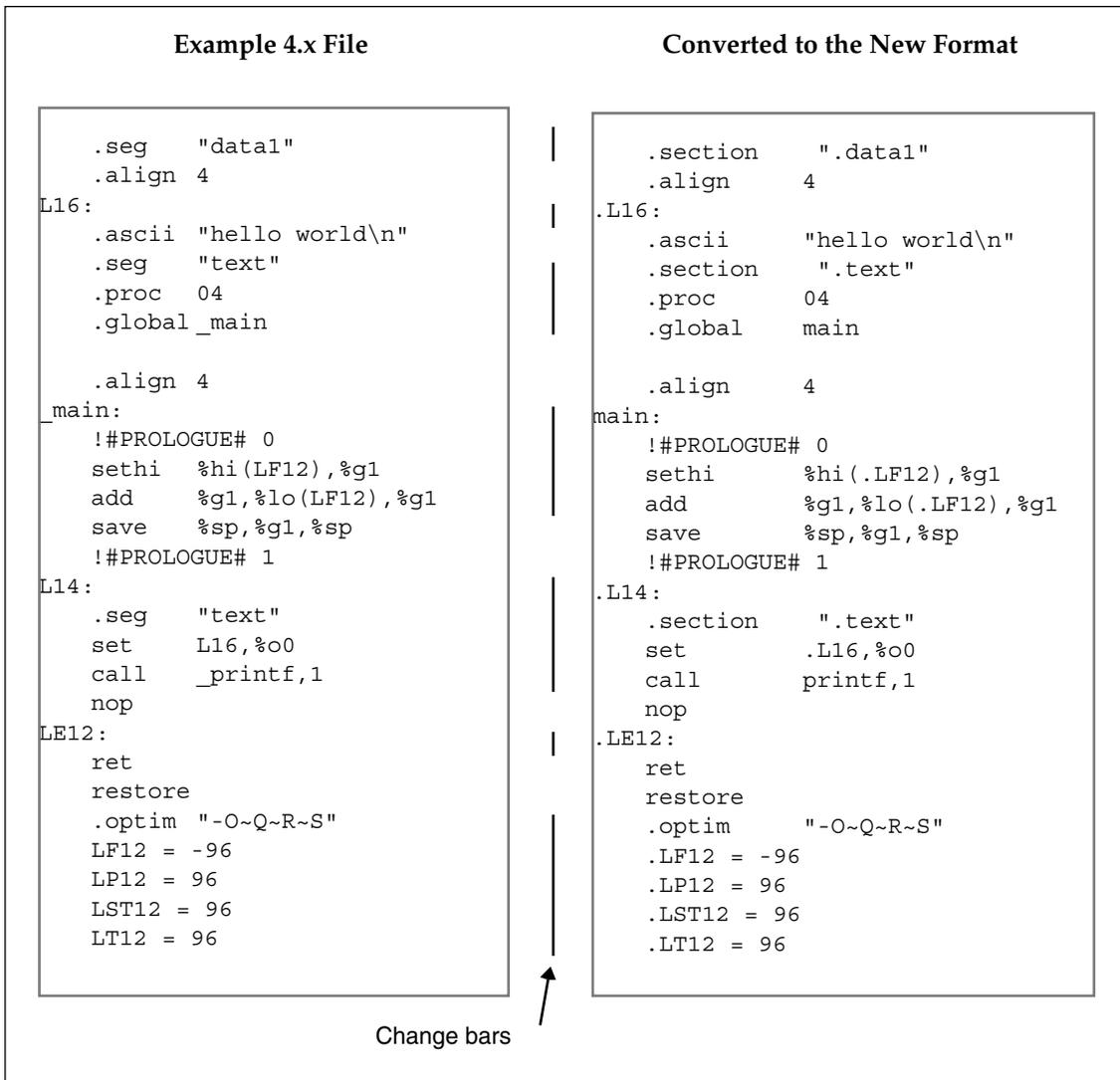
- Remove the leading underscore (`_`) from symbol names. The *Solaris 2.x* SPARCompilers do not prepend a leading underscore to symbol names in the users' programs as did the SPARCompilers that ran under SunOS 4.1.
- Prefix local symbol names with a dot (`.`). Local symbol names in the *SunOS 5.x* SPARC 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, "Pseudo-Operations,"](#) for more information.

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

4.2 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



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.

- “5.1 Table Notation” on page 37
- “5.2 Integer Instructions” on page 39
- “5.3 Floating-Point Instruction” on page 46
- “5.4 Coprocessor Instructions” on page 48
- “5.5 Synthetic Instructions” on page 48

The SPARC-V9 instruction set is described in [Appendix E, “SPARC-V9 Instruction Set.”](#)

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 Instruction Set Notations

Notations	Describes	Comment
address	$reg_{rs1} + reg_{rs2}$ $reg_{rs1} + const13$ $reg_{rs1} - const13$ $const13 + reg_{rs1}$ $const13$	Address formed from register contents, immediate constant, or both.
asi		Alternate address space identifier; an unsigned 8-bit value. It can be the result of the evaluation of a symbol expression.

TABLE 5-1 Instruction Set Notations (Continued)

Notations	Describes	Comment
const13		A signed constant which fits in 13 bits. It can be the result of the evaluation of a symbol expression.
const22		A constant which fits in 22 bits. It can be the result of the evaluation of a symbol expression.
creg	%c0 ... %c31	Coprocessor registers.
freg	%f0 ... %f31	Floating-point registers.
imm7		A signed or unsigned constant that can be represented in 7 bits (it is in the range -64 ... 127). It can be the result of the evaluation of a symbol expression.
reg	%r0 ... %r31 %g0 ... %g7 %o0 ... %o7 %l0 ... %l7 %i0 ... %i7	General purpose registers. Same as %r0 ... %r7 (Globals) Same as %r8 ... %r15 (Outs) Same as %r16 ... %r23 (Locals) Same as %r24 ... %r31 (Ins)
reg _{rd}		Destination register.
reg _{rs1} , reg _{rs2}		Source register 1, source register 2.
reg_or_imm	reg _{rs2} , const13	Value from either a single register, or an immediate constant.
regaddr	reg _{rs1} reg _{rs1} + reg _{rs2}	Address formed with register contents only.
Software_trap_number	reg _{rs1} + reg _{rs2} reg _{rs1} + imm7 reg _{rs1} - imm7 uimm7 imm7 + reg _{rs1}	A value formed from register contents, immediate constant, or both. The resulting value must be in the range 0.....127, inclusive.
uimm7		An unsigned constant that can be represented in 7 bits (it is in the range 0 ... 127). It can be the result of the evaluation of a symbol expression.

5.2 Integer Instructions

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

TABLE 5–2 Assembler Mnemonics Suffixes

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

[Table 5–3](#) outlines the correspondence between SPARC hardware integer instructions and SPARC assembly language instructions.

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

Note – In [Table 5–3](#),

- Braces ({}) indicate optional arguments.
Braces are not literally coded.
- Brackets ([]) indicate indirection: the contents of the addressed memory location are being read from or written to.
Brackets are coded literally in the assembly language. Note that the usage of brackets described in [Chapter 2, “Assembler Syntax,”](#) 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 Hardware Integer Instructions and Assembly Language Instructions

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 i cc	
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 Hardware Integer Instructions and Assembly Language Instructions (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: fbz
FBE	fbe{,a}	label			synonym: fbz

TABLE 5-3 Hardware Integer Instructions and Assembly Language Instructions (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	jmp _l	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

TABLE 5-3 Hardware Integer Instructions and Assembly Language Instructions (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
LDSTUBA	ldstuba	[regaddr]asi, reg _{rd}		
MULScc	mul _{scc}	reg _{rs1} , reg_or_imm, reg _{rd}	Multiply step (and modify i _{cc})	
NOP	nop		No operation	
OR	or	reg _{rs1} , reg_or_imm, reg _{rd}	Inclusive or	
ORcc	or _{cc}	reg _{rs1} , reg_or_imm, reg _{rd}		
ORN	orn	reg _{rs1} , reg_or_imm, reg _{rd}		
ORNcc	orn _{cc}	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	sdiv _{cc}	reg _{rs1} , reg_or_imm, reg _{rd}	Signed divide and modify i _{cc}	
SMUL	smul	reg _{rs1} , reg_or_imm, reg _{rd}	Signed multiply	
SMULcc	smul _{cc}	reg _{rs1} , reg_or_imm, reg _{rd}	Signed multiply and modify i _{cc}	
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 Hardware Integer Instructions and Assembly Language Instructions (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 i cc	
SUBX	subx	$reg_{rs1}, reg_or_imm, reg_{rd}$	Subtract with carry	
SUBXcc	subxcc	$reg_{rs1}, reg_or_imm, reg_{rd}$		
SWAP	swap	$[address], reg_{rd}$	Swap memory word with register	
SWAPA	swapa	$[regaddr]asi, reg_{rd}$		
Ticc	tn	software_trap_number	Trap on integer condition code	Trap never
	tne	software_trap_number	Note: Trap numbers 16-31 are reserved for the user. Currently-defined trap numbers are those defined in <code>/usr/include/sys/trap.h</code>	Synonym: tnz

TABLE 5-3 Hardware Integer Instructions and Assembly Language Instructions (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments	
	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	$reg_or_imm, \%asrn_{rs1}$		See synthetic instructions	
WRY	wr	$reg_{rs1}, reg_or_imm, \%y$			
WRPSR	wr	$reg_{rs1}, reg_or_imm, \%psr$			
WRWIM	wr	$reg_{rs1}, reg_or_imm, \%wim$			

TABLE 5-3 Hardware Integer Instructions and Assembly Language Instructions (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
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.

In the Mnemonic column, types of operands are denoted by the following lowercase letters: i for integer, s for single, d for double, and q for quad.

TABLE 5-4 Floating-Point Instructions

SPARC	Mnemonic	Argument List	Description
FiT0s	fitos	$freg_{rs2}, freg_{rd}$	Convert integer to single
FiT0d	fitod	$freg_{rs2}, freg_{rd}$	Convert integer to double
FiT0q	fitoq	$freg_{rs2}, freg_{rd}$	Convert integer to quad
FsT0i	fstoi	$freg_{rs2}, freg_{rd}$	Convert single to integer
FdT0i	fdtoi	$freg_{rs2}, freg_{rd}$	Convert double to integer
FqT0i	fqtoi	$freg_{rs2}, freg_{rd}$	Convert quad to integer
FsT0d	fstod	$freg_{rs2}, freg_{rd}$	Convert single to double
FsT0q	fstoq	$freg_{rs2}, freg_{rd}$	Convert single to quad
FdT0s	fdtos	$freg_{rs2}, freg_{rd}$	Convert double to single
FdT0q	fdtoq	$freg_{rs2}, freg_{rd}$	Convert double to quad
FqT0d	fqtod	$freg_{rs2}, freg_{rd}$	Convert quad to double
FqT0s	fqtos	$freg_{rs2}, freg_{rd}$	Convert quad to single
FMOVs	fmovs	$freg_{rs2}, freg_{rd}$	Move

TABLE 5-4 Floating-Point Instructions (Continued)

SPARC	Mnemonic	Argument List	Description
FNEGs	fnegs	$freq_{rs2}, freq_{rd}$	Negate
FABSs	fabss	$freq_{rs2}, freq_{rd}$	Absolute value
FSQRTs	fsqrts	$freq_{rs2}, freq_{rd}$	Square root
FSQRTd	fsqrtd	$freq_{rs2}, freq_{rd}$	
FSQRTq	fsqrtq	$freq_{rs2}, freq_{rd}$	
FADDs	fadds	$freq_{rs1}, freq_{rs2}, freq_{rd}$	Add
FADDd	faddd	$freq_{rs1}, freq_{rs2}, freq_{rd}$	
FADDq	faddq	$freq_{rs1}, freq_{rs2}, freq_{rd}$	
FSUBs	fsubs	$freq_{rs1}, freq_{rs2}, freq_{rd}$	Subtract
FSUBd	fsubd	$freq_{rs1}, freq_{rs2}, freq_{rd}$	
FSUBq	fsubq	$freq_{rs1}, freq_{rs2}, freq_{rd}$	
FMULs	fmuls	$freq_{rs1}, freq_{rs2}, freq_{rd}$	Multiply
FMULd	fmuld	$freq_{rs1}, freq_{rs2}, freq_{rd}$	
FMULq	fmulq	$freq_{rs1}, freq_{rs2}, freq_{rd}$	
FdMULq	fmulq	$freq_{rs1}, freq_{rs2}, freq_{rd}$	Multiply double to quad
FsMULd	fsmuld	$freq_{rs1}, freq_{rs2}, freq_{rd}$	Multiply single to double
FDIVs	fdivs	$freq_{rs1}, freq_{rs2}, freq_{rd}$	Divide
FDIVd	fdivd	$freq_{rs1}, freq_{rs2}, freq_{rd}$	
FDIVq	fdivq	$freq_{rs1}, freq_{rs2}, freq_{rd}$	
FCMPs	fcmps	$freq_{rs1}, freq_{rs2}$	Compare, generate exception if not ordered
FCMPd	fcmpd	$freq_{rs1}, freq_{rs2}$	
FCMPq	fcmpq	$freq_{rs1}, freq_{rs2}$	
FCMPES	fcmpes	$freq_{rs1}, freq_{rs2}$	
FCMPED	fcmped	$freq_{rs1}, freq_{rs2}$	
FCMPEq	fcmpcq	$freq_{rs1}, freq_{rs2}$	

5.4 Coprocessor Instructions

All *coprocessor-operate* (cpopn) instructions take all operands from and return all results to coprocessor registers. The data types supported by the coprocessor are coprocessor-dependent. Operand alignment is 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 instruction causes a *cp_disabled* trap.

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

TABLE 5–5 Coprocessor Instructions

SPARC	Mnemonic	Argument List	Name	Comments
CPop1	cpop1	<i>opc, reg_{rs1}, reg_{rs2}, reg_{rd}</i>	Coprocessor operation	
CPop2	cpop2	<i>opc, reg_{rs1}, reg_{rs2}, reg_{rd}</i>	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 Instructions and Hardware Instructions Mapping

Synthetic Instruction		Hardware Equivalents		Comment
btst	<i>reg_or_imm, reg_{rs1}</i>	andcc	<i>reg_{rs1}, reg_or_imm, %g0</i>	Bit test
bset	<i>reg_or_imm, reg_{rd}</i>	or	<i>reg_{rd}, reg_or_imm, reg_{rd}</i>	Bit set
bclr	<i>reg_or_imm, reg_{rd}</i>	andn	<i>reg_{rd}, reg_or_imm, reg_{rd}</i>	Bit clear
btog	<i>reg_or_imm, reg_{rd}</i>	xor	<i>reg_{rd}, reg_or_imm, reg_{rd}</i>	Bit toggle
call	<i>reg_or_imm</i>	jmp _l	<i>reg_or_imm, %o7</i>	
clr	<i>reg_{rd}</i>	or	<i>%g0, %g0, reg_{rd}</i>	Clear (zero) register
clrb	<i>[address]</i>	stb	<i>%g0, [address]</i>	Clear byte
clrh	<i>[address]</i>	st	<i>%g0, [address]</i>	Clear halfword
clr	<i>[address]</i>	st	<i>%g0, [address]</i>	Clear word
cmp	<i>reg, reg_or_imm</i>	subcc	<i>reg_{rs1}, reg_or_imm, %g0</i>	Compare
dec	<i>reg_{rd}</i>	sub	<i>reg_{rd}, 1, reg_{rd}</i>	Decrement by 1
dec	<i>const13, reg_{rd}</i>	sub	<i>reg_{rd}, const13, reg_{rd}</i>	Decrement by <i>const13</i>

TABLE 5-6 Synthetic Instructions and Hardware Instructions Mapping (Continued)

Synthetic Instruction		Hardware Equivalents		Comment
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	<i>address</i>	jmp _l	<i>address</i> , %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
neg	reg_{rd}	sub	%g0, reg_{rd}, reg_{rd}	Two's complement
restore		restore	%g0, %g0, %g0	Trivial <i>restore</i>
save		save	%g0, %g0, %g0	Trivial <i>save</i> <i>trivial save</i> should only be used in supervisor code!
set	<i>value</i> , reg_{rd}	or	%g0, <i>value</i> , reg_{rd}	if $-4096 \leq value \leq 4095$ Do not use the set synthetic instruction in an instruction delay slot.

TABLE 5-6 Synthetic Instructions and Hardware Instructions Mapping (Continued)

Synthetic Instruction		Hardware Equivalents		Comment
set	<i>value, reg_{rd}</i>	sethi	%hi(<i>value</i>), <i>reg_{rd}</i>	if ((<i>value</i> & 0x3ff) == 0)
set	<i>value, reg_{rd}</i>	sethi or	%hi(<i>value</i>), <i>reg_{rd}</i> ; <i>reg_{rd}</i> , %lo(<i>value</i>), <i>reg_{rd}</i>	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	<i>reg</i>	orcc	<i>reg_{rs1}</i> , %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 V8/V9 Natural Pseudo Instructions

Pseudo Instructions	-xarch=	
	V8/V8plus ¹	V9
ldn	ld	ldx
stn	st	stx
ldna	lda	ldxa
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.

Pseudo-Operations

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

A.1 Alphabetized Listing with Descriptions

`.alias`

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

`.align boundary`

Description: Aligns the location counter on a boundary where $((\text{"location counter"} \bmod \text{boundary}) == 0)$; *boundary* may be any power of 2.

`.ascii string [, string"]`

Description: Generates the given sequences of ASCII characters.

`.asciz string [, string]*`

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

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

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

`.common symbol, size [, sect_name] [, alignment]`

Description: 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 0rfloatval [, 0rfloatval]*

Description: 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

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

Description: Some instructions should not be in the delay slot of a CTI. See the *SPARC Architecture Manual* for details.

.file string

Description: 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]*

Description: 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 pseudo-op does 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]*

Description: 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

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

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

.local symbol [, symbol]*

Description: 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.

Description: 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

Description: *%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

Description: 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]*

Description: If `-xarch=v8/v8plus` then assembler interprets the instruction as `.word`. If `-xarch=v9` the assembler interprets the instruction as `.xword`.

.optim string

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

.popsection

Description: 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.

detailed description of the reserved sections. See [Table 3–2](#) in [Chapter 3](#), “Executable and Linking Format,” for a detailed description of the section attribute flags.

Attributes can be:

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

.seg *section_name*

Description: 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 *0rfloatval* [, *0rfloatval*]*

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

Note – This operation does not align automatically.

.size *symbol*, *expr*

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

.skip *n*

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

.stabn <various parameters>

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

.stabs <various parameters>

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

.type symbol, type

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

#object

#function

#no_type

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

.uahalf 16bitval [, 16bitval]*

Description: Generates a (sequence of) 16-bit values.

Note – This operation does not align automatically.

.uaword 32bitval [, 32bitval]*

Description: Generates a (sequence of) 32-bit values.

Note – This operation does not align automatically.

.version string

Description: 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

Description: 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]

Description: 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.

Description: 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]*`

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

Note – This operation does not align automatically.

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

Description: Generates (a sequence of) initialized 64-bit values in the current segment.

Note – This operation does not align automatically.

`.xstabs <various parameters>`

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

`symbol =expr`

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

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 Using Pseudo-ops to Specify C Variable Bindings

```
.pushsection ".data"

.global      foo1      ! int foo1 = 1
.align      4
foo1:
.word       0x1
.type       foo1,#object      ! foo1 is of type data object,
.size       foo1,4           ! with size = 4 bytes

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

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

.align      4           ! static int foo4 = 2
```

EXAMPLE B-1 Using Pseudo-ops to Specify C Variable Bindings (Continued)

```

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      "acom: (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,%00
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, [%00]
retl
nop
.nonvolatile

```


Using the Assembler Command Line

This appendix is organized into the following sections:

- “C.1 Assembler Command Line” on page 65
- “C.2 Assembler Command Line Options” on page 66
- “C.3 Disassembling Object Code” on page 69

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 *fbc* command. You can use either the *as* or *fbc* 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

Description: 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

Description: 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

Description: 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

Description: 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

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

-m

Description: 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

Description: Suppress all warnings while assembling.

-o out file

Description: 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 output file name.

-P

Description: 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}

Description: 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

Description: 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]

Description: 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

Description: 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

Description: 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

Description: 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

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

-xarch=v7

Description: 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

Description: 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

Description: 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

Description: 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, "SPARC-V9 Instruction Set."](#)

-xarch=v8plusa

Description: 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

Description: 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

Description: 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)`.

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
        ! prev_number is executed in the
        ! delay slot

```

EXAMPLE D-2 Assembler Output From C Source (Continued)

```

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

.proc     18                      ! main program
.global   main
.align    4
main:
    save   %sp,-104,%sp          ! create stack frame for main()
/* printf("Fibonacci(n):, please input n:\n"); */
    sethi   %hi(.L31),%o0        ! call printf, with 1st arg in %o0
    call    printf,1
    or      %o0,%lo(.L31),%o0
/* scanf("%d", &n); */
    sethi   %hi(.L33),%o0        ! call scanf, with 1st arg, in %o0
    or      %o0,%lo(.L33),%o0    ! move 2nd arg. to %o1, in delay slot
    call    scanf,2
    add     %fp,-4,%o1

```

EXAMPLE D-2 Assembler Output From C Source (Continued)

```

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

                                ! some initializations before the for-
                                ! loop, put the variables in registers
/* for (i = 1; i <= n; i++) */
mov     1,%i5                    ! %i5 <-- i
mov     %o0,%i4                  ! %i4 <-- result
sethi   %hi(.L38),%o0           ! %i2 <-- format string for printf
add     %o0,%lo(.L38),%i2
ld      [%fp-4],%o0              ! test if (i <= n) ?
cmp     %i5,%o0                  ! note, n is stored in [%fp-4]
bg      .LE27
nop

.LY2:                             ! loop body
/* printf("Fibonacci (%d) is %u\n", i, *result++); */
ld      [%i4],%o2                ! call printf, with (*result) in %o2,
mov     %i5,%o1                  ! i in %o1, format string in %o0
call    printf,3
mov     %i2,%o0
inc     %i5                      ! i++
ld      [%fp-4],%o0              ! i <= n?
cmp     %i5,%o0
ble     .LY2
inc     4,%i4                    ! result++
.LE27:
ret
restore
.type   main,#function           ! type and size of main
.size   main,(.-main)

.section ".data"                 ! switch to data section
                                ! (contains initialized data)
.align  4
.L16:
/* static unsigned fib_array[MAX_FIB_REPRESENTABLE] = {0,1}; */
.align  4                        ! initialization of first 2 elements
.word   0                        ! of fib_array[]
.align  4
.word   1
.skip   188
.type   .L16,#object             ! storage allocation for the rest of
                                ! fib_array[]

.section ".data1"                 ! the ascii string data are entered
                                ! into the .data1 section;
                                ! #alloc: memory would be allocated
                                ! for this section during run time
                                ! #write: the section contains data
                                ! that is writeable during process
                                ! execution
.align  4
.L20:                             ! ascii strings used in the printf stmts

```

EXAMPLE D-2 Assembler Output From C Source (Continued)

```
.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
```


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” on page 77
- “E.2 SPARC-V9 Instruction Set Changes” on page 79
- “E.3 SPARC-V9 Instruction Set Mapping” on page 82
- “E.4 SPARC-V9 Floating-Point Instruction Set Mapping” on page 90
- “E.5 SPARC-V9 Synthetic Instruction-Set Mapping” on page 91
- “E.6 UltraSPARC and VIS Instruction Set Extensions” on page 93

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 Deleted Registers

PSR	Processor State Register
TBR	Trap Base Register
WIM	Window Invalid Mask

These registers have been widened from 32 to 64 bits.

TABLE E-2 Widened Registers

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 SPARC-V9 Registers Within a SPARC-V8 Field

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 Added Registers

ASI	Address Space Identifier
CANRESTORE	Restorable Windows
CANSAVE	Savable windows
CLEANWIN	Clean Windows
FPRS	Floating-point Register State
OTHERWIN	Other Windows
PSTATE	Processor State
TICK	Hardware clock tick-counter
TL	Trap Level
TNPC[MAXTL]	Trap Next Program Counter
TPC[MAXTL]	Trap Program Counter

TABLE E-4 Added Registers *(Continued)*

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 Extended Instruction Definitions

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

TABLE E-5 Extended Instruction Definitions (Continued)

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 Added 64-Bit Instructions

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 LDDE, STDE, 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
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 Added High-Performance System Instructions

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 Deleted Instructions

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

TABLE E-8 Deleted Instructions (Continued)

STDFQ	Store Double from Floating-point Queue (replaced by the RDPR FQ instruction)
-------	--

E.2.5 Miscellaneous Instruction Changes

TABLE E-9 Changed Instructions

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 SPARC-V9 Instruction Set Mapping

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))
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

TABLE E-10 SPARC-V9 Instruction Set Mapping (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
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
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

TABLE E-10 SPARC-V9 Instruction Set Mapping (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
CASA	casa	$[reg_{rs1}]imm_asi, reg_{rs2}, reg_{rd}$	Compare and swap word from alternate space	
	casa	$[reg_{rs1}] \%asi, reg_{rs2}, reg_{rd}$		
CASXA	casxa	$[reg_{rs1}]imm_asi, reg_{rs2}, reg_{rd}$	Compare and swap extended from alternate space	
	casxa	$[reg_{rs1}] \%asi, reg_{rs2}, reg_{rd}$		
FBPA	fbpa{, a}	%fccn, label	(Branch on cc with prediction)	1
	{, pt , pn}		Branch never	
FBPN	fbpn{, a}	%fccn, label	Branch always	0
	{, pt , pn}			
FBPU	fbpu{, a}	%fccn, label	Branch on unordered	U
	{, pt , pn}			
FBPG	fbpg{, a}	%fccn, label	Branch on greater	G
	{, pt , pn}			
FBPUG	fbpug{, a}	%fccn, label	Branch on unordered or greater	G or U
	{, pt , pn}			
FBPL	fbpl{, a}	%fccn, label	Branch on less	L
	{, pt , pn}			
FBPUL	fbpul{, a}	%fccn, label	Branch on unordered or less	L or U
	{, pt , pn}			
FBPLG	fbplg{, a}	%fccn, label	Branch on less or greater	L or G
	{, pt , pn}			
FBPNE	fbpne{, a}	%fccn, label	Branch on not equal	L or G or U
	{, pt , pn}			
FBPE	fbpe{, a}	%fccn, label	Branch on equal	E
	{, pt , pn}			
FBPUE	fbpue{, a}	%fccn, label	Branch on unordered or equal	E or U
	{, pt , pn}			
FBPGE	fbpge{, a}	%fccn, label	Branch on greater or equal	E or G
	{, pt , pn}			

TABLE E-10 SPARC-V9 Instruction Set Mapping (Continued)

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

TABLE E-10 SPARC-V9 Instruction Set Mapping (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
FMOVLEU	fmov {s, d, q}leu	%icc or %xcc, reg_{rs2} , reg_{rd}	Move if less or equal unsigned	C or Z
FMOVCC	fmov {s, d, q}cc	%icc or %xcc, reg_{rs2} , reg_{rd}	Move if carry clear (greater or equal, unsigned)	not C
FMOVCS	fmov {s, d, q}cs	%icc or %xcc, reg_{rs2} , reg_{rd}	Move if carry set (less than, unsigned)	C
FMOVPOS	fmov {s, d, q}pos	%icc or %xcc, reg_{rs2} , reg_{rd}	Move if positive	not N
FMOVNEG	fmov {s, d, q}neg	%icc or %xcc, reg_{rs2} , reg_{rd}	Move if negative	N
FMOVVC	fmov {s, d, q}vc	%icc or %xcc, reg_{rs2} , reg_{rd}	Move if overflow clear	not V
FMOVVS	fmov {s, d, q}vs	%icc or %xcc, reg_{rs2} , reg_{rd}	Move if overflow set	V
FMOVZR	fmovr {s, d, q}e	reg_{rs1} , reg_{rs2} , reg_{rd}	(Move f-p register on cc) Move if register zero	
FMOVRLZ	fmovr {s, d, q}lz	reg_{rs1} , reg_{rs2} , reg_{rd}	Move if register less than or equal zero	
FMOVRLZ	fmovr {s, d, q}lz	reg_{rs1} , reg_{rs2} , reg_{rd}	Move if register less than zero	
FMOV RNZ	fmovr	reg_{rs1} , reg_{rs2} , reg_{rd}	Move if register not zero	
FMOV RGZ	{s, d, q}ne	reg_{rs1} , reg_{rs2} , reg_{rd}	Move if register greater than zero	
FMOV RGEZ	fmovr {s, d, q}gz fmovr {s, d, q}gez	reg_{rs1} , reg_{rs2} , reg_{rd}	Move if register greater than or equal to zero	

TABLE E-10 SPARC-V9 Instruction Set Mapping (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
FMOVFA	fmov{s, d, q}a	%fccn, freg _{rs2} , freg _{rd}	(Move on floating-point cc)	1
FMOVFN	fmov{s, d, q}n	%fccn, freg _{rs2} , freg _{rd}	Move always	0
FMOVFU	fmov{s, d, q}u	%fccn, freg _{rs2} , freg _{rd}	Move never	U
FMOVFG	fmov{s, d, q}g	%fccn, freg _{rs2} , freg _{rd}	Move if unordered	G
FMOVFUG	fmov{s, d, q}ug	%fccn, freg _{rs2} , freg _{rd}	Move if greater	G or U
FMOVFL	fmov{s, d, q}l	%fccn, freg _{rs2} , freg _{rd}	Move if unordered or greater	L
FMOVFUL	fmov{s, d, q}ul	%fccn, freg _{rs2} , freg _{rd}	Move if less	L or U
FMOVFLG	fmov{s, d, q}lg	%fccn, freg _{rs2} , freg _{rd}	Move if unordered or less	L or G
FMOVFNE	fmov{s, d, q}ne	%fccn, freg _{rs2} , freg _{rd}	Move if less or greater	L or G or U
FMOVFE	fmov{s, d, q}e	%fccn, freg _{rs2} , freg _{rd}	Move if not equal	E
FMOVFUE	fmov{s, d, q}ue	%fccn, freg _{rs2} , freg _{rd}	Move if equal	E or U
FMOVFGGE	fmov{s, d, q}ge	%fccn, freg _{rs2} , freg _{rd}	Move if unordered or equal	E or G
FMOVFUGE	fmov{s, d, q}uge	%fccn, freg _{rs2} , freg _{rd}	Move if greater or equal	E or G or U
FMOVFLE	fmov{s, d, q}le	%fccn, freg _{rs2} , freg _{rd}	Move if unordered or greater or equal	E or L
FMOVFULE	fmov{s, d, q}ule	%fccn, freg _{rs2} , freg _{rd}	Move if less or equal	E or L or u
FMOVFO	fmov{s, d, q}o	%fccn, freg _{rs2} , freg _{rd}	Move if unordered or less or equal Move if ordered	E or L or G
LDSW	ldsw	[address], reg _{rd}	Load a signed word	
LDSWA	ldsw	[regaddr] imm_asi, reg _{rd}	Load signed word from alternate space	
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}	Load floating-point state register	
	ldx	[address], %fsr		
MEMBAR	membar	membar_mask	Memory barrier	

TABLE E-10 SPARC-V9 Instruction Set Mapping (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
MOVA	mov _a	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	(Move integer register on cc)	1
MOVN	mov _n	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move always	0
MOVNE	mov _{ne}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move never	not Z
MOVE	mov _e	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if not equal	Z
MOVG	mov _g	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if equal	not (Z or (N xor V))
MOVLE	mov _{le}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if greater	Z or (N xor V)
MOVGE	mov _{ge}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if less or equal	not (N xor V)
MOVL	mov _l	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if greater or equal	N xor V
MOVGU	mov _{gu}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if less	not (C or Z)
MOVLEU	mov _{leu}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if greater unsigned	C or Z
MOVCC	mov _{cc}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if less or equal unsigned	not C
MOVCS	mov _{cs}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if carry clear (greater or equal, unsigned)	C
MOVPOS	mov _{pos}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if carry clear (greater or equal, unsigned)	not N
MOVNEG	mov _{neg}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if carry set (less than, unsigned)	N
MOVVC	mov _{vc}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if positive	not V
MOVVS	mov _{vs}	%icc or %xcc, <i>reg_or_imm11</i> , <i>reg_{rd}</i>	Move if negative	V
			Move if overflow clear	
			Move if overflow set	

TABLE E-10 SPARC-V9 Instruction Set Mapping (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
MOVFA	movfa	$\%fccn, reg_or_imm11, reg_{rd}$	(Move on floating-point cc)	1
MOVFN	movfn	$\%fccn, reg_or_imm11, reg_{rd}$	Move always	0
MOVFU	movfu	$\%fccn, reg_or_imm11, reg_{rd}$	Move never	U
MOVFG	movfg	$\%fccn, reg_or_imm11, reg_{rd}$	Move if unordered	G
MOVFUG	movfug	$\%fccn, reg_or_imm11, reg_{rd}$	Move if greater	G or U
MOVFL	movfl	$\%fccn, reg_or_imm11, reg_{rd}$	Move if unordered or greater	L
MOVFUL	movful	$\%fccn, reg_or_imm11, reg_{rd}$	Move if less	L or U
MOVFLG	movflg	$\%fccn, reg_or_imm11, reg_{rd}$	Move if unordered or less	L or G
MOVFNE	movfne	$\%fccn, reg_or_imm11, reg_{rd}$	Move if less or greater	L or G or U
MOVFE	movfe	$\%fccn, reg_or_imm11, reg_{rd}$	Move if not equal	E E or U
MOVFUE	movfue	$\%fccn, reg_or_imm11, reg_{rd}$	Move if equal	E or G
MOVFG	movfge	$\%fccn, reg_or_imm11, reg_{rd}$	Move if unordered or equal	E or G or U
MOVFUGE	movfuge	$\%fccn, reg_or_imm11, reg_{rd}$	Move if greater or equal	E or L
MOVFLE	movfle	$\%fccn, reg_or_imm11, reg_{rd}$	Move if unordered or greater or equal	E or L or u
MOVFULE	movfule	$\%fccn, reg_or_imm11, reg_{rd}$	Move if less or equal	E or L or G
MOVFO	movfo	$\%fccn, reg_or_imm11, reg_{rd}$	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 Move if register greater than or equal to zero	not N
MULX	mulx	$reg_{rs1}, reg_or_imm, reg_{rd}$	(Generic 64-bit Multiply) Multiply (signed or unsigned)	See SDIVX and UDIVX

TABLE E-10 SPARC-V9 Instruction Set Mapping (Continued)

Opcode	Mnemonic	Argument List	Operation	Comments
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 <i>The SPARC architecture manual, version 9</i>
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.

In the Mnemonic column, types of operands are denoted by the following lowercase letters: i for 32-bit integer, x for 64-bit integer, s for single, d for double, and q for quad.

TABLE E-11 SPARC-V9 Floating-Point Instruction Set Mapping

SPARC	Mnemonic	Argument List	Description
F[<i>sdq</i>]T0x	fstox	$freg_{rs2}, freg_{rd}$	Convert floating point to 64-bit integer
	fdtox	$freg_{rs2}, freg_{rd}$	
	fqttox	$freg_{rs2}, freg_{rd}$	
	fstoi	$freg_{rs2}, freg_{rd}$	Convert floating-point to 32-bit integer
	fdtoi	$freg_{rs2}, freg_{rd}$	
	fqttoi	$freg_{rs2}, freg_{rd}$	
FxT0[<i>sdq</i>]	fxtos	$freg_{rs2}, freg_{rd}$	Convert 64-bit integer to floating point
	fxtod	$freg_{rs2}, freg_{rd}$	
	fxtsq	$freg_{rs2}, freg_{rd}$	

TABLE E-11 SPARC-V9 Floating-Point Instruction Set Mapping (Continued)

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

E.5 SPARC-V9 Synthetic Instruction-Set Mapping

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

TABLE E-12 SPARC-V9 Synthetic Instruction-Set Mapping

Synthetic Instruction		Hardware Equivalents		Comment
cas	$[reg_{\delta rs1}, reg_{\delta rs2}, reg_{rd}]$	casa	$[reg_{\delta rs1}]ASI_P, reg_{rs2}, reg_{rd}$	Compare & swap (cas)
casl	$[reg_{\delta rs1}, reg_{\delta rs2}, reg_{rd}]$	casa	$[reg_{\delta rs1}]ASI_P_L, reg_{rs2}, reg_{rd}$	cas little-endian
casx	$[reg_{\delta rs1}, reg_{\delta rs2}, reg_{rd}]$	casxa	$[reg_{\delta rs1}]ASI_P, reg_{rs2}, reg_{rd}$	cas extended
casxl	$[reg_{\delta rs1}, reg_{\delta rs2}, reg_{rd}]$	casxa	$[reg_{\delta rs1}]ASI_P_L, reg_{rs2}, reg_{rd}$	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	<i>label</i>	bn, pt	$\%xcc, label$	Instruction prefetch,
mov	$\%y, reg_{rd}$	rd	$\%y, reg_{rd}$	
mov	$\%asrn, reg_{rd}$	rd	$\%asrn, reg_{rd}$	
mov	$reg_or_imm, \%asrn$	wr	$\%g0, reg_or_imm, \%asrn$	
ret		jmp1	$\%i7+8, \%g0$	Return from subroutine
retl		jmp1	$\%o7+8, \%g0$	Return from leaf subroutine
setn	<i>value, r1, r2</i>	for -xarch=v9 same as setx <i>value r1, r2</i> for -xarch=v8 same as set <i>value r2</i>		
setnhi	<i>value, r1, r2</i>	for -xarch=v9 same as setxhi <i>value r1, r2</i> for -xarch=v8 same as sethi <i>value r2</i>		
setuw	<i>value, reg_{rd}</i>	sethi or sethi or	$\%hi(value), reg_{rd}$ $\%g0, value, reg_{rd}$ $\%hi(value), reg_{rd};$ $reg_{rd}, \%lo(value), reg_{rd}$	($value \& 3FF_{16}$) $\neq 0$ when $0 \leq value \leq 4095$ (otherwise) Do not use setuw in a DCTI delay slot.

TABLE E-12 SPARC-V9 Synthetic Instruction-Set Mapping (Continued)

Synthetic Instruction		Hardware Equivalents		Comment
setsw	$value, reg_{rd}$	sethi or sethi sra sethi or sethi or sra	$\%hi(value), reg_{rd}$ $\%g0, value, reg_{rd}$ $\%hi(value), reg_{rd}$ $reg_{rd}, \%g0, reg_{rd}$ $\%hi(value), reg_{rd},$ $reg_{rd}, \%lo(value), reg_{rd}$ $\%hi(value), reg_{rd},$ $reg_{rd}, \%lo(value), reg_{rd}$ $reg_{rd}, \%g0, reg_{rd}$	$value \geq 0$ and $(value \& 3FF_{16}) == 0$ $-4096 \leq value \leq 4095$ if $(value < 0)$ and $((value \& 3FF) == 0)$ (otherwise, if $value \geq 0$) (otherwise, if $value < 0$) Do not use setsw in a CTI delay slot.
setx	$value, r1, r2$	sethi or sethi or sllx or	$\%hh(value), r1$ $r1, \%hm(value), r1$ $\%lm(value), r2$ $r2, \%lo(value), r2$ $r1, 32, r1$ $r1, r2, r2$	
setxhi	$value, r1, r2$	sethi or sethi sllx or	$\%hh(value), r1$ $r1, \%hm(value), r1$ $\%lm(value), r2$ $r1, 32, r1$ $r1, r2, r2$	
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 SHUTDOWN Instruction

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 Graphics Status Register (GSR)

SPARC	Mnemonic	Argument List	Description
RDASR	<code>rdasr</code>	<code>%gsr, reg_{rd}</code>	read GSR
WRASR	<code>wrasr</code>	<code>reg_{rs1}, reg_or_imm, %gsr</code>	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 Graphics Instructions

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

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

TABLE E-16 Pack Instructions

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

Partitioned multiply instructions have the following variations.

TABLE E-17 Partitioned Multiply Instructions

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

Alignment instructions have the following variations.

TABLE E-18 Alignment Instructions

SPARC	Mnemonic	Argument List	Description
ALIGNADDRESS	alignaddr	$reg_{rs1}, reg_{rs2}, reg_{rd}$	find misaligned data access address
ALIGNADDRESS_LITTLE	alignaddrll	$reg_{rs1}, reg_{rs2}, reg_{rd}$	same as above, but little-endian
FALIGNDATA	faligndata	$freq_{rs1}, freq_{rs2}, freq_{rd}$	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 Logical Operate Instructions

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
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

TABLE E-19 Logical Operate Instructions (Continued)

SPARC	Mnemonic	Argument List	Description
FANDNOT1S	fandnot1s	$freq_{rs1}, freq_{rs2}, freq_{rd}$	same as above, single precision
FANDNOT2	fandnot2	$freq_{rs1}, freq_{rs2}, freq_{rd}$	src1 AND negated src2
FANDNOT2S	fandnot2s	$freq_{rs1}, freq_{rs2}, freq_{rd}$	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 Pixel Compare Instructions

SPARC	Mnemonic	Argument List	Description
FCMPGT16	fcmpgt16	$freq_{rs1}, freq_{rs2}, reg_{rd}$	4 16-bit compare, set rd if $src1 > src2$
FCMPGT32	fcmpgt32	$freq_{rs1}, freq_{rs2}, reg_{rd}$	2 32-bit compare, set rd if $src1 > src2$
FCMPLE16	fcmple16	$freq_{rs1}, freq_{rs2}, reg_{rd}$	4 16-bit compare, set rd if $src1 \leq src2$
FCMPLE32	fcmple32	$freq_{rs1}, freq_{rs2}, reg_{rd}$	2 32-bit compare, set rd if $src1 \leq src2$
FCMPNE16	fcmpne16	$freq_{rs1}, freq_{rs2}, reg_{rd}$	4 16-bit compare, set rd if $src1 \neq src2$
FCMPNE32	fcmpne32	$freq_{rs1}, freq_{rs2}, reg_{rd}$	2 32-bit compare, set rd if $src1 \neq src2$
FCMPEQ16	fcmqeq16	$freq_{rs1}, freq_{rs2}, reg_{rd}$	4 16-bit compare, set rd if $src1 = src2$
FCMPEQ32	fcmqeq32	$freq_{rs1}, freq_{rs2}, reg_{rd}$	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 Edge Handling Instructions

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 Pixel Component Distance Instructions

SPARC	Mnemonic	Argument List	Description
PDIST	<code>pdist</code>	$freq_{rs1}, freq_{rs2}, freq_{rd}$	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 Three-Dimensional Array Addressing Instructions

SPARC	Mnemonic	Argument List	Description
ARRAY8	<code>array8</code>	$reg_{rs1}, reg_{rs2}, reg_{rd}$	convert 8-bit 3-D address to blocked byte address
ARRAY16	<code>array16</code>	$reg_{rs1}, reg_{rs2}, reg_{rd}$	same as above, but 16-bit
ARRAY32	<code>array32</code>	$reg_{rs1}, reg_{rs2}, reg_{rd}$	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 Memory Access Instructions

SPARC	<code>imm_asi</code>	Argument List	Description
STDFA	<code>ASI_PST8_P</code>	$stda\ freq_{rd}, [reg_{addr}]\ reg_{mask}, imm_asi$	eight 8-bit conditional stores to: primary address space
STDFA	<code>ASI_PST8_S</code>		secondary address space
STDFA	<code>ASI_PST8_PL</code>		primary address space, little endian
STDFA	<code>ASI_PST8_SL</code>		secondary address space, little endian
STDFA	<code>ASI_PST16_P</code>		four 16-bit conditional stores to: primary address space
STDFA	<code>ASI_PST16_S</code>		secondary address space
STDFA	<code>ASI_PST16_PL</code>		primary address space, little endian
STDFA	<code>ASI_PST16_SL</code>		secondary address space, little endian
			two 32-bit conditional stores to:

TABLE E-24 Memory Access Instructions (Continued)

SPARC	imm_asi	Argument List	Description
STDFA	ASI_PST32_P		primary address space
STDFA	ASI_PST32_S		secondary address space
STDFA	ASI_PST32_PL		primary address space, little endian
STDFA	ASI_PST32_SL		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 Partial Store Instructions

SPARC	imm_asi	Argument List	Description
LDDFA	ASI_FL8_P	<i>ldda [reg_addr] imm_asi, freg_{rd}</i>	8-bit load/store from/to: 8-bit load/store from/to:primary address space
STDFA		<i>stda freg_{rd}, [reg_addr] imm_asi</i>	
LDDFA	ASI_FL8_S	<i>ldda [reg_plus_imm] %asi, freg_{rd}</i>	secondary address space
STDFA		<i>stda [reg_plus_imm] %asi</i>	
LDDFA	ASI_FL8_PL		primary address space, little endian
STDFA			
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 Load and Store Instructions

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	$ldda [reg_addr] imm_asi,$	64-byte block load/store from/to: primary address space, user privilege
STDFA		$freg_{rd}$ $stda freg_{rd}, [reg_addr]$ imm_asi	
LDDFA	ASI_BLK_AIUS	$ldda [reg_plus_imm] \%asi,$	secondary address space, user privilege.
STDFA		$freg_{rd}$ $stda freg_{rd}, [reg_plus_imm]$ $\%asi$	
LDDFA	ASI_BLK_AIUPL		primary address space, user privilege, little endian
STDFA			
LDDFA	ASI_BLK_AIUSL		secondary address space, user privilege little endian
STDFA			
LDDFA	ASI_BLK_P		primary address space
STDFA			
LDDFA	ASI_BLK_S		secondary address space
STDFA			
LDDFA	ASI_BLK_PL		primary address space, little endian
STDFA			
LDDFA	ASI_BLK_SL		secondary address space, little endian
STDFA			
LDDFA	ASI_BLK_COMMIT_P		64-byte block commit store to primary address space
STDFA			
LDDFA	ASI_BLK_COMMIT_S		64-byte block commit store to secondary address space
STDFA			

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

Index

A

- addresses, 30
- .alias, 53
- .align, 53
- as command, 65
- .ascii, 53
- .asciz, 53
- assembler command line, 65
- assembler command line options, 66–69
- assembler directives, 32–33
 - types, 32
- assembly language, 13
 - lines, 14
 - statements, 14
 - syntax notation, 13
- assignment directive, 33
- atof, 15, 54, 56

B

- binary operations, 18
- .byte, 53
- byte order for V9, 79

C

- case distinction, 14
- case distinction, in special symbols, 18
- cc language driver, 65
- command-line options, 66–69

- comment lines, 14
- comment lines, multiple, 14
- .common, 53
- constants, 15
 - decimal, 15
 - floating-point, 15
 - hexadecimal, 15
 - octal numeric, 15
- Control Transfer Instructions (CTI), 20
- converting existing object files, 35
- coprocessor instruction, 48
- cp_disabled trap, 48
- cp_exception trap, 48
- current location, 30
- current section, 23

D

- D option, 66
- data generating directives, 33
- default output file, 21
- dis program, 69
- disassembling object code, 69
- .double, 54

E

- ELF header, 22, 23
 - ehsize, 22
 - entry, 22
 - flag, 22

ELF header (*Continued*)

- ident, 22
- machine, 22
- phentsize, 22
- phnum, 22
- phoff, 22
- shentsize, 22
- shnum, 22
- shoff, 23
- shstrndx, 23
- type, 23
- version, 23

- .empty pseudo-operation, 20
- .empty, 54
- error messages, 20
- escape codes, in strings, 15
- Executable and Linking Format (ELF) files, 12, 21
- expressions, 18–19
- expressions, SPARC-V9, 19–20

F

- f77 language driver, 65
- fbe command, 65
- features, lexical, 14
- .file, 54
- file syntax, 14
- floating-point instructions, 46–47
- floating-point pseudo-operations, 15

G

- .global, 54
- .globl, 54

H

- .half, 55
- hardware instructions, SPARC architecture, 37
- hardware integer, assembly language instructions, 39
- hyphen (-), 65

I

- I option, 66
- .ident, 55
- instruction set, used by assembler, 37
- instruction set changes (V9), 79–82
- instruction set extensions (V9), 93–102
- instructions
 - assembly language, 39
 - hardware integer, 39
- integer instructions, 39–46
- integer suffixes, 15
- invoking, as command, 65

K

- K option, 66

L

- L option, 66
- labeling format, 11
- labels, 15
- language drivers, 65
- lexical features, 14
- lines syntax, 14
- .local, 55
- location counter, 30
- locations, 30

M

- m option, 66
- multiple comment lines, 14
- multiple files, on, 65
- multiple sections, 24
- multiple strings, in string table, 32

N

- .noalias pseudo-op, 53
- .noalias, 55

.nonvolatile, 55
 numbers, 15
 numeric labels, 15

O

-o option, 67
 object file format, 12
 object files
 type, 12, 21
 operators, 18–19
 operators, SPARC-V9, 19–20
 .optim, 55
 options, command-line, 66–69

P

-P option, 67
 percentage sign (%), 16
 .popsection, 55
 predefined non-user sections, 29
 predefined user sections, 27–29
 .proc, 56
 pseudo-operations, 53
 pseudo-ops, examples of, 61
 .pushsection, 56

Q

-Q option, 67
 -q option, 67
 .quad, 56

R

references, 7
 registers, 16–18
 relocatable files, 12, 21
 relocation tables, 30
 .reserve, 56

S

-S option, 67
 -s option, 67
 -sb option, 66
 .section, 56
 section control directives, 33
 section control pseudo-ops, 33
 section header, 24, 27
 addr, 24
 addralign, 24
 entsize, 24
 flags, 24
 info, 25
 link, 25
 name, 25
 offset, 25
 size, 25
 type, 25
 sections, 23–29
 .seg, 57
 .single, 57
 .size, 57
 .skip, 57
 SPARC-V9, 77–79
 8-bit format, 94
 alternate space access, 79
 byte order, 79
 fixed data formats, 94
 floating-point instructions, 90–91
 graphics data formats, 94
 instruction set changes, 79–82
 instruction set extensions, 93–102
 instruction set mapping, 82–90
 registers, 77–79
 synthetic instruction set, 91–93
 SPARC-V9, 64-bit expressions, 19–20
 SPARC-V9, 64-bit operators, 19–20
 special floating-point values, 15
 special names, floating point values, 15
 special symbols, 16–18
 .stabn, 57
 .stabs, 57
 statement syntax, 14
 string tables, 32

strings, 15–16
 multiple in string table, 32
 multiple references in string table, 32
 suggested style, 15
 unreferenced in string table, 32
sub-strings in string table, references to, 32
symbol, 59
symbol attribute directives, 33
symbol names, 16
symbol table, 30, 31
 info, 31
 name, 31
 other, 31
 shndx, 31
 size, 31
 value, 31
symbol tables, 30–32
syntax notation, 13
synthetic instructions, 48–50

T

-T option, 67
table notation, 37–38
trap numbers, reserved, 44
.type, 58

U

-U option, 68
.uahalf, 58
.uaword, 58
unary operators, 18
user sections, 33
/usr/include/sys/trap.h, 44

V

-V option, 68
.version, 58
.volatile, 58

W

.weak, 58
.word, 59

X

-xarch=v7 option, 68
-xarch=v8 option, 68
-xarch=v8a option, 68
-xarch=v8plus option, 68
-xarch=v8plusa option, 68
.xstabs, 59