SPARC Assembly Language Reference Manual

Part No: E36858
July 2014

ORACLE

Contents

Using This Documentation

- **Overview** – The SPARC assembler translates source files that are in assembly language format into object files for linking into executables on Oracle Solaris SPARC platforms.

  The assembler is a tool for 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.

  The choice of assembly language 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 and Solaris macro preprocessors to achieve full macro-assembler capability. Furthermore, the assembler responds to directives that allow the programmer 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.
- **Audience** – This manual is intended for experienced SPARC assembly language programmers who are familiar with the SPARC architecture.
- **Required knowledge** – You should also become familiar with the following:
  - Manual pages: as(1), ld(1), cpp(1), elf(3elf), dis(1), a.out(1)
  - *Oracle SPARC Architecture 2011 Guide*
  - *SPARC Architecture Manual* (Version 8 and Version 9)
  - *System V Application Binary Interface: SPARC™ Processor Supplement*

Product Documentation Library

Late-breaking information and known issues for this product are included in the documentation library at http://www.oracle.com/pls/topic/lookup?ctx=E36784.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

♦ ♦ ♦ CHAPTER 1

SPARC Assembler Syntax

The Oracle Solaris SPARC assembler takes assembly language programs, as specified in this document, and produces relocatable object files for processing by the Solaris link editor. The assembly language described in this document corresponds to the SPARC instruction set defined in the *Oracle SPARC Architecture 2011 Guide* (OSA) and is intended for use on Oracle Solaris SPARC platforms.

This chapter is organized into the following sections:

1.1 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.

1.2 Assembler File Syntax

The syntax of assembly language *files* is:

```
[line]*
```

1.2.1 Lines Syntax

The syntax of assembly language *lines* is:

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

1.2.2 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.

1.3 Lexical Features

This section describes the lexical features of the assembler syntax.

1.3.1 Case Distinction

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

1.3.2 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.

1.3.3 Labels

A `label` is either a symbol or a single decimal digit *n* (0…99). A label is immediately followed by a *colon* (:).

Numeric labels may be defined repeatedly in an assembly file; symbolic labels may be defined only once.

A numeric label *n* is referenced after its definition (backward reference) as *n*`b`, and before its definition (forward reference) as *n*`f`.

1.3.4 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.”

1.3.5 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 1-1 , derived from ANSI C, are recognized in strings.

TABLE 1-1 Escape Codes Recognized in Strings

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

1.3.6 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.

1.3.7 Special Symbols - Registers

Special symbol names begin with a *percentage sign* (`%`) to avoid conflict with user symbols. Table 1-2 lists these special symbol names.

TABLE 1-2 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`	
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 Solaris Studio compiler-generated code.
	`%r_plt32`	Used only in Solaris Studio 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
```

1.3.8 Attributes

Attributes, in the form *#attribute*, can be used to modify certain pseudo-operations and instructions. Pseudo-ops `.global, .section, .register`, and `.type` accept specific attributes that correspond to linker attribute flags, as shown in Table 3-1.

Several instructions, such as `membar` and `prefetch`, also accept attributes. See the instruction descriptions in the Oracle SPARC Architecture Specifications for details on the attributes a given instruction supports.

1.3.9 Operators and Expressions

The operators described in Table 1-3 are recognized in constant expressions.

TABLE 1-3 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 Solaris Studio compiler-generated code only to instruct the assembler to generate specific relocation information for the given expression.

Binary	Operators	Unary	Operators
<<	Left shift	%r_plt32 %r_plt64	Used in Solaris Studio compiler-generated code only to instruct the assembler to generate specific relocation information for the given expression.
>>	Right shift		
&	Bitwise AND		
|	Bitwise OR		

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

To avoid confusion with register names or with the %hi, %lo, %r_disp32/64, or %r_plt32/64 operators, the modulo operator % must *not* be immediately followed by a letter or digit. The modulo operator is typically followed by a space or left parenthesis character.

1.3.10 SPARC V9 Operators and Expressions

The following V9 64-bit operators and expressions in Table 1-4 ease the task of converting from V8/V8plus assembly code to V9 assembly code.

TABLE 1-4 V9 64-bit Operators and Expressions

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

For example:

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

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

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

The V9 high 32-bit operators and expressions are identified in Table 1-5.

TABLE 1-5 V9 32-bit Operators and Expressions

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

For example:

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

The V9 low 44-bit operators and expressions are identified in Table 1-6 .

TABLE 1-6 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
```

1.4 Assembler Error Messages

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

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

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

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

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

♦ ♦ ♦ CHAPTER 2

Executable and Linking Format

The object files created by the Oracle Solaris SPARC assembler are *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.

The ELF object file format consists various component features, including:

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

This chapter is just a summary of the ELF features. For complete details on the ELF format, see Chapter 12, “Object File Format,” in “Oracle Solaris 11.2 Linkers and Libraries Guide ” .

2.1 Sections

A section is the smallest unit of an object that can be relocated. Use the `elfdump`(1) command to inspect the components of an object or executable file generated by the assembler.

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 component sections need to be present.

2.1.1 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 a number of fields as described in detail in `sys/elf.h` and "Sections" in "Oracle Solaris 11.2 Linkers and Libraries Guide " . However, only the following fields are of immediate interest to the assembly language programmer because they can be specified in assembler pseudo-operations (directives):

`sh_flags`

Description: One-bit descriptions of section attributes. Table 2-1 describes the some of the section attribute flags. For details and additional flags, see "Sections" in "Oracle Solaris 11.2 Linkers and Libraries Guide "

`sh_info`

Description: Extra information. The interpretation of this information depends on the section type, as described in . Table 2-2

`sh_link`

Description: Section header table index link. The interpretation of this information depends on the section type, as described in . Table 2-2

`sh_name`

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

TABLE 2-1 Section Attribute Flags

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

TABLE 2-2 Section Types Modified by Assembler Pseudo-ops

Name	Value	Description
`null`	`0`	Marks section header as inactive.
`progbits`	`1`	Contains information defined explicitly by the program.
`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.

2.1.2 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 2-3 lists some of the predefined user sections that can be named in the section

control directives. For details and additional information, see “Special Sections” in “Oracle Solaris 11.2 Linkers and Libraries Guide ”

TABLE 2-3 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.
`.rodata & .rodata1`	Section contains read-only data.
`.text`	Section contains executable text.
`.line`	Section contains line # info for symbolic debugging.
`.note`	Section contains note information.

2.1.2.1 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 2-1 .

EXAMPLE 2-1 Creating an `.init` Section

```
.section ".init"

.align   4

<instructions>
```

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

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

```
call routine_name
```

```
nop
```

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

2.1.2.2 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 2-2 .

EXAMPLE 2-2 Creating an .fini Section

```
.section ".fini"

.align   4

<instructions>
```

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

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

```
call routine_name

nop
```

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

2.1.3 Predefined Non-User Sections

Table 2-4 lists sections that are predefined and not under user control. Therefore, these section names are reserved by the assembler and should be avoided.

TABLE 2-4 Reserved Sections

Section Name	Description
`".dynamic"`	Section contains dynamic linking information.

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

2.1.4 Symbol Tables

A *symbol table* contains information to locate and relocate symbolic definitions and references. The Oracle Solaris 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 Oracle Solaris linker during relocation. The section header contains the symbol table index for the first non-local symbol.

A symbol table contains the following information defined by `Elf32_Sym` and `Elf64_Sym` in `sys/elf.h` and "Symbol Table Section" in "Oracle Solaris 11.2 Linkers and Libraries Guide ":

`st_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.

`st_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.

`st_size`

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

`st_info`

Description: Specifies the symbol type and binding attributes. Table 2-5 and Table 2-6 describe these values.

`st_other`

Description: Specifies a symbol's visibility.

`st_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 2-5 Symbol Type Attributes `ELF32_ST_TYPE` and `ELF64_ST_TYPE`

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

Table 2-6 shows the symbol binding attributes.

TABLE 2-6 Symbol Binding Attributes `ELF32_ST_BIND` and `ELF64_ST_BIND`

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

Value	Binding	Description
13	loproc	Values reserved for processor-specific semantics.
15	hiproc	

2.1.5 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.

2.2 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.

2.3 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.

2.3.1 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.

2.4 Tools

Solaris provides a number of command-line tools to display, analyze, and modify the functional components of object and executable files, such as the following:

- `elfdump` — The `elfdump` utility symbolically dumps selected parts of the specified object file(s). The options allow specific portions of the file to be displayed.
- `dump` — The `dump` utility dumps selected parts of each of its object file arguments, and is best suited for use in shell scripts, while the `elfdump` command is recommended for more human-readable output.
- `/usr/sfw/bin/greadelf` — `greadelf` displays information about one or more ELF format object files. The options control what particular information to display.
- `mcs` — The `mcs` command is used to manipulate a section in an ELF object file.
- `dis` — The `dis` command produces an assembly language listing of an object file or an archive of object files. The listing includes assembly statements and an octal or hexadecimal representation of the binary that produced those statements.
- `/usr/sfw/bin/gobjdump` — `gobjdump` displays information about one or more object files. The options control what particular information to display.

♦ ♦ ♦ CHAPTER 3

Directives and Pseudo-Operations

Assembler directives are commands to the assembler in the form of *pseudo-operations*. Some directives cause the assembler to generate code or data, while others do not. The different types of assembler directives are:

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

3.1 Assembler Directives

3.1.1 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.

3.1.2 Symbol Attribute Directives

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

3.1.3 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.

3.1.4 Data Generating Directives

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

3.2 Notation

The synopses of the pseudo-operations in this appendix use the following notation:

- Pseudo-operations and literal characters are displayed in typewriter font. For example, `.popsection`
- Italics are used to denote a replaceable (variable) item explained in the description. For example, `.section` *section_name*
- Items enclosed in square brackets are optional. For example, [*item, ..., item*] denotes an optional list of *items* of arbitrary length. When shown as *item*[, *item, ..., item*], at least one instance of *item* is required. The brackets [and] are meta-characters and not part of the declaration.
- *string* denotes a string of characters enclosed in double quotes, as in "`a string of characters`".
- Items in curly brackets separated by a vertical bar denote a required option with at least two choices. For example, {`#scratch` | *symbol_name*} denotes that either `#scratch` or a symbolic name is required. The brackets { and }are meta-characters and not part of the declaration.

3.3 Alphabetized Listing with Descriptions

`.alias`

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

`.align` *boundary*

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

`.ascii` *string*`[,` *string*`, ..., ` *string*`]`

Generates the given sequences of ASCII characters.

`.asciz` *string*`[,` *string*`, ..., ` *string*`]`

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

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

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

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

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

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

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

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

`.empty`

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

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

`.exported` *symbol*`[`, *symbol, ..., symbol*`]`

Declares each symbol in the list to have *exported* linker scope. Ensures that these symbols remain global and are visible to all modules. References to them are bound at runtime.This visibility can not be demoted, or eliminated by any other symbol visibility technique. A symbol with `STB_LOCAL` binding will not have `STV_EXPORTED` visibility.

`.file` *string*

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

`.global` *symbol*`[`, *symbol, ..., symbol*`]`

(Spelling `.globl` is also accepted.) 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.

`.group` *group*`,` *section*`, #comdat`

Adds *section* to a COMDAT *group*. Refer to the Oracle Solaris Linker and Libraries Guide for information about COMDAT.

`.half` *16bitval*`[, ` *16bitval*`, ..., ` *16bitval*`]`

Generates (a sequence of) initialized halfwords in the current segment. The location counter must already be aligned on a halfword boundary (use `.align 2`).

`.hidden` *symbol*`[`, *symbol, ..., symbol*`]`

Declares each symbol in the list to have *hidden* linker scoping. All references to one of these listed symbols within a dynamic module bind to the definition within that module. These symbols are not visible outside the module, and are given linker scope `STV_HIDDEN`.

`.ident` *string*

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

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

`.internal` *symbol*`[`, *symbol, ..., symbol*`]`

Same as `.hidden`

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

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

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

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

`.noalias %`*reg1*`, %`*reg2*

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

`.nonvolatile`

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

- The block of instruction has been previously terminated by a Control Transfer Instruction (CTI) or a label

- There is no preceding `.volatile` pseudo-op

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

If assembling with `-m32`, the assembler interprets the instruction as `.word`. If `-m64` the assembler interprets the instruction as `.xword`.

`.popsection`

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

`.proc` *n*

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

`.protected` *symbol*`[,` *symbol, ..., symbol*`]`

Declares each symbol in the list to have *protected* linker scoping and visible to all external objects. References to these symbols from within the object are bound at link-edit, thus preventing runtime interposition. This visibility scope can be demoted, or eliminated by other symbol visibility techniques. This scope definition has the same affect as a symbol with `STV_PROTECTED` visibility.

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

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

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

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

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

`.register %g{2|3|6|7}, {#scratch|`*symbol_name*`}`

With SPARC-V9, the four registers `%g2, %g3, %g6, %g7`, should not be used unless explicitly declared on a `.register` pseudo-op. When assembling under `-m64`, the SPARC assembler will issue an error message if it detects the use of `%g2` or `%g3` registers without a `.register` declaration. A `.register` declaration is not required for `%g6` or `%g7`, but its appearance does invoke checking for proper use of these registers.

Specify the `#scratch` option when the register is used as a scratch register:

```
.register %g3, #scratch
```

Or, declare the global register with a symbolic name, as in:

```
.register %g2, xyz
```

A `.register` declaration must appear before the first use of the register. Linking objects containing conflicting register will cause the linker to issue error messages.

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

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

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

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

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

Makes the specified section the current section.

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

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

See the sections “2.1.2 Predefined User Sections” on page 21 and “2.1.3 Predefined Non-User Sections” on page 23 for a detailed description of the reserved sections. SeeTable 2-1 for a list of the section attribute flags.

Attributes can be:

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

`.seg` *section_name*

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 Oracle Solaris 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 Oracle Solaris .

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

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

Note - This operation does not align automatically.

`.size` *symbol, expr*

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

`.skip` *n*

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

`.stabn` *various_parameters*

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

`.stabs` *various_parameters*

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

`.symbolic` *symbol*[, *symbol*, ..., *symbol*]

Same as `.protected`

`.tls_common` *symbol*, *size*[, *sect_name*][, *alignment*]

Similar to `.common`, provides a tentative definition of *symbol*. *Size* bytes are allocated in thread local storage (TLS) for the object represented by *symbol*. See `.common` for details.

`.type` *symbol*[, *symbol*, ..., *symbol*], *type*[, *visibility*]

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

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

and where *visibility* can be one of:

```
#hidden  #protected  #eliminate  #singleton  #exported  #internal
```

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

Generates a (sequence of) 16-bit values.

Note - This operation does not align automatically.

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

Generates a (sequence of) 32-bit values.

Note - This operation does not align automatically.

`.version` *string*

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

`.volatile`

Defines the beginning of a block of instructions in the section that may not be changed. *This directive is obsolete and no longer has any affect.*

`.weak` *symbol*`[`, *symbol, ..., symbol*`]`

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

Note the following:

- A *weak* symbol definition in one file will satisfy an undefined reference to a global symbol of the same name in another file.
- Unresolved *weak* symbols have a default value of zero; the link editor does not resolve these symbols.
- If a *weak* symbol has the same name as a defined *global* symbol, the weak symbol is ignored and no error results.

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

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

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

Note - This operation does not align automatically.

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

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

Note - This operation does not align automatically.

`.xstabs` *various_parameters*

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

symbol `=`*expr*

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

3.4 Pseudo-Op Attributes

Pseudo-ops `.global, .section, .register`, and `.type` accept specific attributes that correspond to linker attribute flags, as shown in the following table.

Example: `.type sum, #function`

TABLE 3-1 Pseudo-op Linker Attributes

Attribute	Linker Symbol	Accepting Pseudo-op
#alloc	SHF_ALLOC	.section
#annotate	SHT_SUNW_ANNOTATE	.section
#comdat	SHT_SUNW_COMDAT	.group
#eliminate	STV_ELIMINATE	.type
#exclude	SHF_EXCLUDE	.section
#execinstr	SHF_EXECINSTR	.section
#exported	STV_EXPORTED	.type
#fini_array	SHT_FINI_ARRAY	.section
#function	STT_FUNC	.type
#group	SHF_GROUP	.section
#hidden	STV_HIDDEN	.type
#init_array	SHT_INIT_ARRAY	.section
#internal	STV_INTERNAL	.type
#linkafter	SHN_AFTER	.section
#linkbefore	SHN_BEFORE	.section
#linkorder	SHF_LINK_ORDER	.section
#nobits	SHT_NOBITS	.section
#no_type	STT_NOTYPE	.type
#object	STT_OBJECT	.type
#ordered	SHF_ORDERED	.section
#preinit_array	SHT_PREINIT_ARRAY	.section
#progbits	SHT_PROGBITS	.section
#protected	STV_PROTECTED	.type

Attribute	Linker Symbol	Accepting Pseudo-op
#scratch	*no linker flag*	.register
#singleton	STV_SINGLETON	.type
#symbolic	STV_PROTECTED	.type
#tls	SHF_TLS	.section
#tls_object	STT_TLS	.type
#visible	STV_DEFAULT	.type
#write	SHF_WRITE	.section

See the "Oracle Solaris 11.2 Linkers and Libraries Guide " for details.

3.5 Pseudo-Op Examples

3.5.1 Example 1: Binding to C Variables

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 3-1 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
  foo4:
.word  0x2
.type  foo4,#object
.size  foo4,4

.popsection
```

3.5.2 Example 2: Generating Ident Strings

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   "

myprog

:

This is an example of an ident string

"
```

3.5.3 Example 3: Data Alignment, Size, Scope, and Type

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
```

3.5.4 Example 4: “Hello World”

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

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

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

♦ ♦ ♦ CHAPTER 4

Creating Data in Assembler

This chapter gives examples of creating various data types using assembler pseudo-ops.

4.1 Examples

Here are some examples of writing declarations and definitions for various kinds of data types.

EXAMPLE 4-1 Examples

The following demonstrates the use of the `.word, .half, .byte, .xword, nword,` and `.asciiz` pseudo-ops, along with `.align, .skip, .global`, and `.local`, to define data in `.data, .rodata,` and `.bss` sections.

```
! --------.data-----------------
        ! the .data section is used for normal read/write data
        .section        ".data"

        ! iii is a global integer (word), "iii"
        .global iii
        .align  4
iii:
        .word   12345678


        ! sss is a global short (half)
        .global sss
        .align  2
sss:
        .half   12345


        ! ccc is a static (local) char (byte)
        .local  ccc
        .align  1
ccc:
        .byte   12


        ! lll is a a global long long (xword)
        .global lll
```

```
        .align  8
lll:
        .xword  1234567812345678


        ! aaa is a global char string
        .global aaa
        .align  1
aaa:
        .asciiz "a string"


        ! sss is a global pointer to a string (absolute addressing)
        .global sss
        .align  8
sss:
        .nword  aaa


        ! --------.rodata-------------------
        ! the .rodata section is used for read-only data
        .section        ".rodata"

        ! jjj is a global read-only integer (word)
        .global jjj
        .align  4
jjj:
        .word   12345678


        ! ---------.bss---------------------
        ! the .bss section is used for data allocated (as zeroes) at run-time
        ! data in this section does not occupy space in the ELF file
        .section        ".bss"

        ! kkk is a global "bss" integer allocated at run-time
        .global kkk
        .align  4
kkk:
        .skip   4
```

♦ ♦ ♦ CHAPTER 5

SPARC Code Models

There are two SPARC code models, *absolute* and *position independent*, and two address space sizes, *32–bit* and *64–bit*. This chapter describes how the different code models use different methods for creating an address.

5.1 Basics

When compiling a simple C program like:

```
int sum = 0;
void add(int a)
{
   sum += a;
}
```

the kind of code used to access the variable *sum* is different for different code models. All the code models need a way to create the address for *sum* so that the address can be used to load or store the value. Different code models use different methods for creating an address.

There are two basic kinds of code models, *absolute* and *position independent*. Absolute code models create an address that is a constant, so that the variable cannot be moved without changing the code. Position independent code models create an address that is more movable and can be decided at run-time. Position independent code (PIC) is recommended and sometimes required for creating shared objects. The dynamic linker (`ld.so`) determines the location of the shared object at run-time and finalizes the position independent address. See the *Oracle Solaris Linkers and Libraries Guide* for more info on this topic.

5.2 Address Sizes

There are also two different sizes for code models, 32–bit and 64–bit., which results in three code models for 32–bit code, and five code models for 64–bit code.

For 32–bit code, there are the following address modes:

- 32–bit absolute

- 13–bit PIC
- 32–bit PIC

For 64–bit code, there are the following address modes:

- 32–bit absolute
- 44–bit absolute
- 64–bit absolute
- 13–bit PIC
- 32–bit PIC

5.2.1 32–Bit Absolute

An example of 32–bit absolute assembly code for the function `add()` shown earlier looks like this:

```
add:
        sethi   %hi(sum),%o4
        ld      [%o4+%lo(sum)],%o5
        add     %o5,%o0,%o3
        retl
        st      %o3,[%o4+%lo(sum)]
```

It takes two instructions to form the address of `sum`. The `%hi()` operator tells the assembler to create a *R_SPARC_HI22* relocation symbol `sum`, and the `%lo(sum)` operator creates a *R_SPARC_LO10* relocation on the symbol `sum`.

5.2.2 64–Bit Absolute

The 64–bit absolute code model for `add()` might look like this:

```
add:
        sethi   %hh(sum),%o5
        sethi   %lm(sum),%o2
        or      %o5,%hm(sum),%o4
        sllx    %o4,32,%o3
        or      %o3,%o2,%o1
        ld      [%o1+%lo(sum)],%g5
        add     %g5,%o0,%g3
        retl
        st      %g3,[%o1+%lo(sum)]
```

Here it takes 6 instruction to form address of `sum`. The operators act as follows:

`%hh(sum)` ⇒ *R_SPARC_HH22* relocation
`%hm(sum)` ⇒ *R_SPARC_HM10*

`%lm(sum)` ⇒ *R_SPARC_LM22*
`%lo(sum)` ⇒ *R_SPARC_LO10*

5.2.3 44–Bit Absolute

The 44–bit absolute code model for `add()` might look like the following:

```
add:
        sethi   %h44(sum),%o5
        or      %o5,%m44(sum),%o4
        sllx    %o4,12,%o2
        ld      [%o2+%l44(sum)],%o3
        add     %o3,%o0,%o1
        retl
        st      %o1,[%o2+%l44(sum)]
```

It takes 4 instructions to form the 44 bits of address for `sum`. The operators act like as follows:

`%h44(sum)` ⇒ *R_SPARC_H44* relocation
`%m44(sum)` ⇒ *R_SPARC_M44*
`%l44(sum)` ⇒ *R_SPARC_L44*

5.2.4 64–Bit with 13–Bit PIC

The 64-bit with 13-bit PIC code for `add()` might look like the following:

```
add:
.L900000106:
        rd      %pc,%o3
        sethi   %pc22(_GLOBAL_OFFSET_TABLE_-(.L900000106-.)),%g1
        add     %g1,%pc10(_GLOBAL_OFFSET_TABLE_-(.L900000106-.)),%g1
        add     %g1,%o3,%o3
        ldx     [%o3+%got13(sum)],%o1
        ld      [%o1],%o2
        add     %o2,%o0,%g5
        retl    ! Result =
        st      %g5,[%o1]
```

The address of `sum` is formed in two parts. The first four instructions form the address of the global offset table (GOT). Then a 13-bit offset into the GOT is used to load the address of `sum`. The dynamic linker puts the correct address for `sum` into the GOT at run-time.

The operators act as follows:

`%pc22(...)` ⇒ *R_SPARC_PC22* relocation
`%pc13(...)` ⇒ *R_SPARC_PC13*
`%got13(sum)` ⇒ *R_SPARC_GOT13*

The 32-bit with 13-bit PIC code for `add()` is similar to the above 64–bit with 13–bit PIC, but the `ldx` used for 64-bit code is changed to `ld` for 32-bit code.

5.2.5 64–Bit With 32–Bit PIC

The 64-bit with 32-bit PIC code for `add()` might look as follows:

```
add:
        .L900000106:
        rd      %pc,%o1
        sethi   %pc22(_GLOBAL_OFFSET_TABLE_-(.L900000106-.)),%g1
        sethi   %got22(sum),%o3
        add     %g1,%pc10(_GLOBAL_OFFSET_TABLE_-(.L900000106-.)),%g1
        xor     %o3,%got10(sum),%o2
        add     %g1,%o1,%o1
        ldx     [%o1+%o2],%g4,%gdop(sum)
        ld      [%g4],%g5
        add     %g5,%o0,%g3
        retl    ! Result =
        st      %g3,[%g4]
```

Again, the address of `sum` is formed in two parts. The first part forms the address of the global offset table (GOT). Then a 32-bit offset into the GOT is used to load the address of `sum`.

The operators act as follows:

`%pc22(...)` ⇒ *R_SPARC_PC22* relocation
`%pc13(...)` ⇒ *R_SPARC_PC13*
`%got22(sum)` ⇒ *R_SPARC_GOT22*
`%got10(sum)` ⇒ *R_SPARC_GOT10*
`%gdop(sum)` ⇒ *R_SPARC_GOTDATA_OP*

Similarly, the 32-bit code with 32-bit PIC would use just `ld` instead of `ldx` to load the address of `sum` from the GOT.

5.3 Global Object Table (GOT) Code Models

On SPARC processors, position independent code (PIC) uses a global object table (GOT) for loading addresses, so that the addresses can be determined at run-time by the dynamic linker.

For some data items, a faster access is possible by using the *GOTdata* relocations.

There are two different code models for GOTdata, *plain GOTdata*, and *GOTdata_op*.

Here is the code for the GOTdata_op code model.

```
add:
.L900000105:
        rd      %pc,%o1
        sethi   %pc22(_GLOBAL_OFFSET_TABLE_-(.L900000105-.)),%g1
        add     %g1,%pc10(_GLOBAL_OFFSET_TABLE_-(.L900000105-.)),%g1
        add     %g1,%o1,%o1
        sethi   %gdop_hix22(sum),%o3
        xor     %o3,%gdop_lox10(sum),%o2
        ldx     [%o1+%o2],%g4,%gdop(sum)
        ld      [%g4],%g5
        add     %g5,%o0,%g3
        retl
        st      %g3,[%g4]
        .type   add,#function
        .size   add,(.-add)
```

There are four instructions to form a pointer to the GOT into register `%o1`. Then it takes two more instructions to form the offset of the address of sum in the GOT (in `%o2`), and another instruction to add those to the address of the GOT to form the address of the GOT slot for `sum`, and load that address into `%g4`.

At this point, the code looks much like the pic32 code model.

The operators act like this:

```
%gdop_hix22(sum) --> R_SPARC_GOTDATA_OP_HIX22
%gdop_lox10(sum) --> R_SPARC_GOTDATA_OP_LOX10
%gdop(sum)       --> R_SPARC_GOTDATA_OP
```

The difference for the GOTdata_op code model is that the static linker, `ld`, can re-write the code sequence to avoid one load from the GOT. This can happen if the distance from the beginning of the GOT to the location of `sum` is less than 2 GB away.

So, then the load:

```
ldx     [%o1+%o2],%g4,%gdop(sum)
```

is re-written into an `add`:

```
 add     %o1,%o2,%g4
```

Here is the code for the plain GOTdata code model.

```
add:
.L900000105:
        rd      %pc,%o1
        sethi   %pc22(_GLOBAL_OFFSET_TABLE_-(.L900000105-.)),%g1
        add     %g1,%pc10(_GLOBAL_OFFSET_TABLE_-(.L900000105-.)),%g1
        add     %g1,%o1,%o1
        sethi   %gd_hix22(sum),%o3
        xor     %o3,%gd_lox10(sum),%o2
        ld      [%o1+%o2],%g5
        add     %g5,%o0,%g3
        retl
        st      %g3,[%o1+%o2]
```

```
        .type   add,#function
        .size   add,(.-add)
```

There are four instructions to form a pointer to the GOT into register `%o1`. Then it takes two more instructions to form the offset of the address of sum in the GOT (in `%o2`). Now the sum of `%o1` and `%o2` can be used directly to load the value of `sum`.

The operators act like this:

```
%gd_hix22(sum) --> R_SPARC_GOTDATA_HIX22
%gd_lox10(sum) --> R_SPARC_GOTDATA_LOX10
```

The plain GOTdata code is simpler than the GOTdata_op code, but it requires that the data be within 2 GB of the start of the GOT, otherwise a static link-time error will result.

5.4 Thread Local Storage (TLS) Code Models

The local executable code model uses the fastest code sequence, but it can only be used for code within an executable accessing a variable within that executable.

The initial executable code model allows code in the executable to access TLS variables in the shared objects to which the executable has been statically linked. The IE code is somewhat slower than the LE code.

The local dynamic code model allows code in a shared object to access TLS variables of its own. The LD code is usually somewhat slower than the IE code.

The general dynamic code model allows code from anywhere to access TLS variables anywhere. So, the executable could access a TLS variable in a dynamically linked shared object, for example. The GD code is the slowest.

Note that on Solaris, the `%g7` register is used by the OS to point to thread data, and the TLS variables are sometimes accessed through this register. The program should not modify `%g7` or unpredicatable results will happen.

5.4.1 Local Executable Code Model

An example of the code for the local executable TLS code model is shown below:

```
add:
        sethi   %tle_hix22(sum),%o3
        xor     %o3,%tle_lox10(sum),%o2
        ld      [%g7+%o2],%o1
        add     %o1,%o0,%g4
        retl
        st      %g4,[%g7+%o2]
```

It takes two instructions to form the address of `sum`. The operators act as follows:

`%tle_hix22(sum)` ⇒ *R_SPARC_TLS_LE_HIX22* relocation
`%tle_lox10(sum)` ⇒ *R_SPARC_TLS_LE_LOX10*

5.4.2 Initial Executable Code Model

An example of the code for the initial executable TLS code model is shown below:

```
add:
        rd      %pc,%o1
        sethi   %pc22(_GLOBAL_OFFSET_TABLE_-(.L900000106-.)),%g1
        add     %g1,%pc10(_GLOBAL_OFFSET_TABLE_-(.L900000106-.)),%g1
        add     %g1,%o1,%o1
        sethi   %tie_hi22(sum),%o3
        add     %o3,%tie_lo10(sum),%o2
        ldx     [%o1+%o2],%g5,%tie_ldx(sum)
        add     %g7,%g5,%g3,%tie_add(sum)
        ld      [%g3],%g4
        add     %g4,%o0,%g2
        retl
        st      %g2,[%g3]
```

Here it takes four instructions to form a pointer to the GOT into register `%o1`, followed by two instruction to form the offset of the address of `sum` in the GOT.

The operators act as follows:

`%tie_hi22(sum)` ⇒ *R_SPARC_TLS_IE_HI22* relocation
`%tie_lo10(sum)` ⇒ *R_SPARC_TLS_IE_LO10*
`%tie_ldx(sum)` ⇒ *R_SPARC_TLS_IE_LDX*
`%tie_add(sum)` ⇒ *R_SPARC_TLS_IE_ADD*

5.4.3 Local Dynamic TLS Code Model

An example of the local dynamic TLS code model is shown below:

```
add:
        save    %sp,-176,%sp
.L900000107:
        rd      %pc,%i3
        sethi   %pc22(_GLOBAL_OFFSET_TABLE_-(.L900000107-.)),%g1
        add     %g1,%pc10(_GLOBAL_OFFSET_TABLE_-(.L900000107-.)),%g1
        add     %g1,%i3,%i3
        sethi   %tldm_hi22(sum),%i2
        add     %i2,%tldm_lo10(sum),%i1
        add     %i3,%i1,%o0,%tldm_add(sum)
        call    __tls_get_addr,%tldm_call(sum)
```

```
        nop
        sethi   %tldo_hix22(sum),%l7
        xor     %l7,%tldo_lox10(sum),%l6
        add     %o0,%l6,%l4,%tldo_add(sum)
        ld      [%l4],%l5
        add     %l5,%i0,%l3
        st      %l3,[%l4]
        ret
        restore %g0,%g0,%g0
```

Notice that we could not have a leaf routine because of the `call` instruction.

There are four instructions to form a pointer to the GOT into register `%i3`. Then it takes two more instructions to form the offset of the address of `sum` in the GOT, and another instruction to add those to the address of the GOT to form the address of the GOT slot for `sum`. This address is passed to function `__tls_get_addr` that returns the address for local module's TLS data in register `%o0`. Three more instructions form the offset of `sum` in the module data and to add that to the module data address. Note that the module data address can be reused to access multiple TLS variables.

The operators act as follows:

`%tldm_hi22(sum)` ⇒ *R_SPARC_TLS_LDM_HI22* relocation
`%tldm_lo10(sum)` ⇒ *R_SPARC_TLS_LDM_LO10*
`%tldm_add(sum)` ⇒ *R_SPARC_TLS_LDM_ADD*
`%tldm_call(sum)` ⇒ *R_SPARC_TLS_LDM_CALL*
`%tldo_hix22(sum)` ⇒ *R_SPARC_TLS_LDO_HIX22* relocation
`%tldo_lox10(sum)` ⇒ *R_SPARC_TLS_LDO_LOX10*
`%tldo_add(sum)` ⇒ *R_SPARC_TLS_LDO_ADD*

5.4.4 General Dynamic TLS Code Model

An example of the general dynamic TLS code model is shown below:

```
add:
        save    %sp,-176,%sp
.L900000107:
        rd      %pc,%i3
        sethi   %pc22(_GLOBAL_OFFSET_TABLE_-(.L900000107-.)),%g1
        add     %g1,%pc10(_GLOBAL_OFFSET_TABLE_-(.L900000107-.)),%g1
        add     %g1,%i3,%i3
        sethi   %tgd_hi22(sum),%i2
        add     %i2,%tgd_lo10(sum),%i1
        add     %i3,%i1,%o0,%tgd_add(sum)
        call    __tls_get_addr,%tgd_call(sum)
        nop
        ld      [%o0],%l7
        add     %l7,%i0,%l6
        st      %l6,[%o0]
```

```
        ret
        restore %g0,%g0,%g0
```

Notice that we could not have a leaf routine because of the `call` instruction.

There are four instructions to form a pointer to the GOT into register `%i3`, followed by two more instruction to form the offset of the address of `sum` in the GOT, and another instruction to add those to the address of the GOT to form the address of the GOT slot for `sum`. This address is passed to function `__tls_get_addr` that returns the address for `sum` in register `%o0`.

The operators act as follows:

`%tgd_hi22(sum)` ⇒ *R_SPARC_TLS_GD_HI22* relocation
`%tgd_lo10(sum)` ⇒ *R_SPARC_TLS_GD_LO10*
`%tgd_add(sum)` ⇒ *R_SPARC_TLS_GD_ADD*
`%tgd_call(sum)` ⇒ *R_SPARC_TLS_GD_CALL*

♦ ♦ ♦ CHAPTER 6

Writing Functions — The SPARC ABI

This chapter outlines the basic design of an assembly language function that can be called from a C program. In order for an assembly language program to interoperate with a C program or the C library functions or the operating system calls, certain conventions about register usage, stack usage, parameter passing, and returning values must be followed. These agreed-to conventions are referred to as the Application Binary Interface, the ABI.

6.1 Anatomy of a C Function

A good place to start is with a simple C function:

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

Compiling with the C compiler's –S option generates the assembler code:

```
demo% cc -O add.c -S
demo% cat add.s

! --------BEGIN PROLOG ------------
        .section        ".text",#alloc,#execinstr,#progbits
        .file   "add.c"

        .section        ".bss",#alloc,#write,#nobits

Bbss.bss:

        .section        ".data",#alloc,#write,#progbits

Ddata.data:

        .section        ".rodata",#alloc,#progbits
!
! CONSTANT POOL
!

Drodata.rodata:
```

```
        .section        ".text",#alloc,#execinstr,#progbits
/* 000000          0 */         .align  4
/* 000000            */         .skip   16
/* 0x0010            */         .align  4
! FILE add.c

!    1                !int add(int a, int b)
!    2                !{

!
! SUBROUTINE add
!
! OFFSET    SOURCE LINE LABEL   INSTRUCTION

                        .global add

! ---END PROLOG --- BEGIN BODY -------

                        add:

!    3                !    return a + b;



                        .L900000105:
/* 000000          3 */         retl    ! Result =  %o0
/* 0x0004            */         add     %o0,%o1,%o0
!
!  -------END BODY ------ BEGIN EPILOG -------
!
/* 0x0008          0 */         .type   add,#function
/* 0x0008          0 */         .size   add,(.-add)


                        .L900000106:

        .section        ".text",#alloc,#execinstr,#progbits


                        .L900000107:

        .section        ".annotate",#progbits
/* 000000          0 */         .asciz  "anotate"
/* 0x0008          0 */         .half   6,0
/* 0x000c          0 */         .word   28
/* 0x0010          0 */         .half   0,8
/* 0x0014          0 */         .word   (.L900000107-0x18)
/* 0x0018          0 */         .word   24
/* 0x001c          0 */         .half   1,12
/* 0x0020          0 */         .word   .L900000105
/* 0x0024          0 */         .word   (.L900000106-.L900000105)
/* 0x0028          0 */         .word   1577472

! ----------------END EPILOG-------------
```

The purpose of the prolog is to put the body of the code into the correct context for the assembler to create an object file of executable code. The prolog creates the section where to

code should go and defines the proper alignment. It also declares the entry point as *global* so that it can be called from another object.

The body of the function has a label for the function name and the instruction code for the function.

The epilog section declares the type and size of the function for compatibility with other tools.

6.2 Register Usage

The input registers `%i0` through `%i7`, and the local registers `%l0` through `%l7` are *callee saves registers.*In other words, to use them in an assembly function called from C code, they should be saved first and restored before returning. These registers are normally saved in the callee by executing a save instruction to change the register windows, and restored with the restore instruction or the return instruction at the end of the function. If you use the input or local registers within an assembly language function and call a C function, there is no need to save those registers before the call.

The output registers `%o0` to `%o7` are caller saves registers and are also used for parameter passing and return values. To use the output registers in an assembly language function, there is no need to save them before using them. If calling a C function from assembly language, any useful value in an output register should be saved before the call is made.

The global registers `%g0-%g7` are more complicated. The `%g0` register is always zero. The `%g6` and `%g7` are always reserved for the operating system, so assembly code should not modify them. The other global registers, `%g1-%g5`, are caller saves, and are usable by applications code. But note that `%g1` and `%g5` may be used in the program linkage table (PLT) or other interposition code, and thus cannot be used to pass parameters from caller to callee.

6.3 Parameter Passing

On SPARC processors, arguments to C functions are passed as-if they were in a parameter array. The array elements are called *slots*. The slots are numbered from zero. For 32-bit code, the array has 32-bit elements (slots), and for 64-bit code the array has 64-bit elements (slots). Successive parameters to a routine are passed in successive slots of the parameter array.

There is space allocated on the stack for all the slots necessary for a function's parameters, but some slots may be promoted to registers and thus the stack location may contain no value.

For 32-bit code, the parameter array starts at `%fp+68` (after a save instruction, `%sp+68` before), and the stack and frame pointers are aligned on a 64-bit (8 Byte) boundary. For 64-bit code, the parameter array starts at `%fp+BIAS+128` (after a save instruction), and the stack and frame

pointers are aligned on a 128-bit (16 Byte) boundary. For 64-bit code, `BIAS` is 2047. Parameters that are passed in registers also have a (unused) memory location corresponding to their slot(s).

Integer data types smaller then a slot are passed in the lower part of that slot. For 64-bit code, where a smaller integer type (`int`, `short`, `char`) parameter is passed in a register, the caller must sign extend or zero extend the value to the full 64-bit slot width. Similarly, in 64-bit code where a smaller integer type is returned in a register, the callee must sign extend or zero extend the value to the full 64-bit slot width. In 32-bit code, there is no requirement for sign or zero extensions, and only the lower bits of the values should be used.

Data types that are larger than the slot size are passed in multiple slots. For 32-bit code, `double` and `long long` data types are passed in 2 slots, and they are not aligned, but packed next to the previous parameter slot. For 32-bit code, values longer than a `double` are passed like passing a structure by value. For 64-bit code, `double` and `long long` data types occupy just one slot, but `long double` and `double complex` data occupy two slots, and these slots are aligned (slot number modulo 2 equals 0), skipping a slot if necessary for alignment.

The first six slots of the parameter array are always passed in registers. For 32-bit code, these slots always go into the lower 32-bits of registers `%o0` to `%o5`, regardless of whether they are integer or floating-point values. For 64-bit code, these 6 slots go into the full 64-bits of registers `%o0` to `%o5` if they are integer types. The `float`, `double`, `long double` types are passed in the double registers `%d0` to `%d10`, corresponding to slots 0 to 5.

The `float complex, double complex`, and `long double complex` data types are passed as though there were just two parameters of their base type. The imaginary types are passed the same as the plain `float` types. For 64-bit code, `float, double`, and `long double` data in slots 6-31 are passed in registers `%d12` to `%d62`.

Structure, union, or array parameters passed by value are passed by making a copy on the stack and passing a pointer to the copy. The details of this kind of parameter passing are complicated and beyond the scope of this manual. Similarly, for functions returning a structure, union, or array by value, the caller allocates space for the return value and passes a pointer for that area to the callee. The callee puts there returned value into the designated area before returning. The details of this kind of return value code are also complicated and beyond the scope of this manual.

6.4 Functions Returning Values

Functions that return an integer value return it in `%o0` or `%o0` and `%o1`. For 32-bit code, `long long` data are returned with the upper 32-bits in `%o0` and the lower 32-bits in `%o1`, treating `%o0` and `%o1` as if they were 32-bit registers.

If the function returning a value has executed a save instruction, then the return value would normally be put into `%i0` or `%i1` just before executing a restore or return instruction, which changes register windows and puts the values into `%o0` and `%o1`.

Functions that return a floating-point or complex value return it in some subset of `%f0, %f1, %d0, %d2, %d4,` and `%d6`. So a float value is returned in `%f0`, and a double value is returned in `%d0` (or equivalently in `%f0` and `%f1`).

Structure and array values returned by value are more complicated and beyond the scope of this manual. Registers `%o0-%o5` and `%f0-%f31` may be used as temporaries.

6.4.1 Limitations for 32–Bit Code

The global registers and the output registers can be used to hold 64-bit integer values, but the input registers and the local registers can only be used to hold 32-bit values in the lower half of the register. This is because the register save area for the input and local registers does not have enough room to store the full 64-bits and only the lower 32-bits are saved. The input and local registers may be saved and later reloaded at any point in time by a trap to handle an interrupt.

6.4.2 Limitations for Both 32–Bit and 64–Bit Code

There is a minimum stack size for any routine, and certain areas of the stack that cannot be used.

The minimum stack frame size is just large enough to hold the register save area plus the required 6 slots for parameter passing. For 32-bit code, the minimum stack frame size is 92 bytes. This is normally allocated with a "save `%sp, 92, %sp`" instruction. For 64-bit code, the minimum stack frame size is 176 bytes. This is normally allocated with a "`save %sp, 176, %sp`" instruction.

The stack area below where `%sp` points is volatile and might be overwritten at any point in time (for example. by an interrupt). Do not store any useful data there, instead, change the `%sp` downward first and store above the `%sp`.

6.4.3 Additional Information

For more information about the SPARC application binary interface (ABI), see the following documents available at sparc.org:

- SPARC Compliance Definition 2.3 (32–bit specification)
- SPARC Compliance Definition 2.4 (64–bit specification)

Refer also to the SPARC V9 Architecture Manual (PDF) descriptions of the instruction set, registers, and other details.

♦ ♦ ♦ CHAPTER 7

Assembler Inline Functions and `__asm` Code

This chapter discusses how to use the C or C++ compiler to create inline functions and `__asm` assembler code. Inline templates and the C/C++ `__asm` statement provide a way to insert assembler code into a C or C++ program. The assembler code is processed by the compiler's code generator, and not the SPARC assembler. However, the syntax recognized by the compilers is similar to the SPARC Assembler syntax. This chapter describes how inline templates and `__asm` statements can be used effectively.

7.1 Inline Function Templates in C and C++

The following are examples where inline templates are particularly useful:

- Hand-coded mutex locks using atomic instructions.
- Machine-level access for a hardware device or to access certain hardware registers.
- Precise implementation of algorithms that can be implemented optimally using hand-coding that the compiler is unable to replicate.

Inline templates appear as normal function calls in the C/C++ source code. When the source code program cc -O prog.c code.il and the file containing the inline template defining the function are compiled together, the compiler will insert the code from the inline template in place of the function call in the code generated from the C/C++ source code.

7.1.1 Compiling C/C++ with Inline Templates

Inline template files have .il file extension. Compile inline templates along with the source file that calls them. The code is inlined by the code-generator stage of compilation.

```
cc -O prog.c code.il
```

The example above will compile prog.c and inline the code from code.il wherever the function defined by code.il is called in prog.c.

7.1.2 Layout of Code in Inline Templates

A single inline template file can define more than one inline templates. Each template definition starts with a declaration, and ends with an end statement:

```
.inline identifier
  ...assembler code...
.end
```

identifier is the name of the template function. Multiple template definitions with the same name can appear in the file, but the compiler will use only the first definition.

Since the template code will be inlined directly, without a call, into the code generated by the compiler, there is no need for a return instruction.

The template requires a prototype declaration in C/C++ source code to ensure that the compiler assigns correct types for all the parameters and recognizes the template name as a function.

For example, the following prototype declaration defines the template function:

```
void do_nothing();
```

And the associated template definition of this function might look like the following:

```
/* The do_nothing() template does nothing*/
.inline do_nothing,0
  nop
end
```

The inline template definition would appear in a separate .il file and would be compiled along with the source code file containing the call.

7.1.3 Guidelines for Coding Inline Templates

SPARC inline assembly code can use only integer registers `%o0` to `%o5` and floating point registers `%f0` to `%f31` for temporary values. These registers are referred to as the *caller-saved* registers. Other registers should not be used. Calls can be made to other routines from the inline template, but these calls are subject to the same constraint.

The compiler will handle most of the SPARC instruction set. If the template utilises only those instructions that the compiler normally generates it will be early inlined (see “7.1.4 Late and Early Inlining” on page 65), and the code will be scheduled optimally. However, if the template utilises instructions that the compiler accepts but does not typically generate (such as VIS instructions or atomics), the code might be late inlined. Consequently, the code might not be optimally scheduled by the compiler, resulting in a possible performance loss.

7.1.3.1 Parameter Passing

Passing parameters between the C/C++ caller program and the assembly language template code must obey the parameter passing rules defined by the target architecture, which are different for 32-bit and 64-bit code. Parameter passing is described by the SPARC ABI. See the SPARC International Technical Documents page. SCD 2.3 describes Version 8 (32-bit code) and SCD 2.4.1 describes Version 9 (64-bit code).

Entering the template code, arguments will be passed in `%o0` to `%o5` and will continue on the stack. For 32-bit code, the offset is `[%sp+0x5c]` and `%sp` is guaranteed to be 64-byte aligned; for 64-bit code, the offset is `[%sp+0x8af]`. (For 64-bit code, the stack bias is `%sp+2047`, which is aligned on a 16-byte boundary.)

For example (function prototype in C followed by assembler template equivalent):

```
int add_up(int v1,int v2, int v3, int v4, int v5, int v6, int v7);

/*Add up 7 integer parameters; last one will be passed on stack*/
.inline add_up,28
  add %o0,%o1,%o0
  ld [%sp+0x5c],%o1
  add %o2,%o3,%o2
  add %o4,%o5,%o4
  add %o0,%o1,%o0
  add %o2,%o4,%o2
  add %o0,%o2,%o0
.end
```

The same example for 64-bit code, but note that when a 32-bit int register is passed on the stack, the full 64 bits of the register are saved:

```
int add_up(int v1,int v2, int v3, int v4, int v5, int v6, int v7);

/*Add up 7 integer parameters; last one will be passed on stack*/
.inline add_up,28
  add %o0,%o1,%o0
  ldx [%sp+0x8af],%o1
  add %o2,%o3,%o2
  add %o4,%o5,%o4
  add %o0,%o1,%o0
  add %o2,%o4,%o2
  add %o0,%o2,%o0
.end
```

For 32-bit floating point, values will be passed in the integer registers. For 64-bit code, they will be passed in the floating point registers.

32-bit floating-point passing by value example:

```
double sum_val(double a, double b);

/*sum of two doubles by value*/
```

```
.inline sum_val,16
  st   %o0,[%sp+0x48]
  st   %o1,[%sp+0x4c]
  ldd  [%sp+0x48],%f0
  st   %o2,[%sp+0x48]
  st   %o3,[%sp+0x4c]
  ldd  [%sp+0x48],%f2
  faddd %f0,%f2,%f0
.end
```

64-bit floating-point passing by value example:

```
double sum(double a, double b);

/*sum of two doubles 64-bit calling convention*/
.inline sum,16
  faddd %f0,%f2,%f0
.end
```

Values passed in memory, single-precision floating point values, and integers are guaranteed to be 4-byte aligned. Double-precision floating point values will be 8-byte aligned if their offset in the parameters is a multiple of 8-bytes.

Integer return values are passed in `%o0`. Floating point return values are passed in `%f0`/`%f1` (single-precision values in `%f0`, double-precision values in the register pair `%f0,%f1`).

For 32-bit code, there are two ways of passing the floating point registers. The first way is to pass them by value, and the second is to pass them by reference. Either way, the compiler will do its best to optimize out the load and store instructions. It is often more successful at doing this if the floating point parameters are passed by reference.

Here is an example of 32-bit by reference parameter passing:

```
double sum_ref(double *a, double *b);

/*sum of two doubles by reference*/
.inline sum_ref,16
  ldd [%o0],%f0
  ldd [%o1],%f2
  faddd %f0,%f2,%f0
.end
```

7.1.3.2 Stack Space

Sometimes, it is necessary to store variables to the stack in order to load them back later; this is the case for moving between the int and fp registers. The best way of doing this is to use the space already set aside for parameters that are passed into the function.

For example, in the 32-bit floating-point passing by value code shown above, the location `%sp +0x48` is 8-byte aligned (`%sp` is 8-byte aligned), and it corresponds to the place where the second

and third 4-byte integer parameters would be stored if they were passed on the stack. (Note that the first parameter would be stored at a non-8-byte boundary.)

7.1.3.3 Branches and Calls

Branching and calls within template code is allowed. Every branch or call must be followed by a nop instruction to fill the branch delay slot. It is possible to put instructions in the delay slot of branches, which can be useful if you wish to use the processor support for annulled instructions, but doing so will cause the code to be late-inlined (described in Late and Early Inlining) and may result in sub-optimal performance.

Call instructions must have an extra last argument that indicates the number of registers used to pass arguments in the call parameters. In general, you should avoid inlining call instructions.

The destinations of branches must be indicated with a number, and the branch instructions should use this number to indicate the appropriate destination together with an `f` for a forward branch or a `b` for a backward branch.

Here is an example of using branches in an inline template:

```
int is_true(int i);
/*return whether true*/
.inline is_true,4
   cmp  %o0,%g0
   bne  1f
   nop
   mov  1,%o0
   ba   2f
   nop
1:
   mov  0,%o0
2:
.end
```

7.1.4 Late and Early Inlining

The code generator of the compiler processes template inlining. There are two opportunities for inlining: before and after optimization. If the inline template is complicated, the compiler may choose to do the inlining after optimization (late inlining), which means that the code will more or less appear exactly as it appears in the template. Otherwise, the code is inlined before optimization (early inlining) and will be merged and optimized with the rest of the code around the call site.

Early inlining leads to better performance. Things that will cause late inlining are:

- Use of instructions that the compiler cannot generate
- Instructions in the delay slots of branches
- Call instructions

View the compiler commentary generated with -g to see if a routine is late inlined. The following example shows a template that fails early inlining because it uses the frame pointer (%fp) rather than the stack pointer (%sp).

```
.inline sum_val,16
  st   %o0,[%fp+0x48]
  st   %o1,[%fp+0x4c]
  ldd  [%fp+0x48],%f0
  st   %o2,[%fp+0x48]
  st   %o3,[%fp+0x4c]
  ldd  [%fp+0x48],%f2
  faddd %f0,%f2,%f0
.end
```

The compiler will still inline the code, but it is unable to early inline the code and the code will not participate in the compiler's optimization.

The following example compiles a 32-bit executable with compiler commentary information and displays it using the Oracle Solaris Studio `er_src` command. The debug information is stored in the `.o` files by default, so it is necessary to keep these files available.

```
% cc -g -O inline32.il driver32.c
% er_src a.out main
Source file: /home/AUser/code/inline/driver32.c
Object file: /home/AUser/code/inline/driver32.o
Load Object: a.out

     1. #include <stdio.h>
     2.
     3. void do_nothing();
     4. int add_up(int v1,int v2, int v3, int v4, int v5, int v6, int v7);
     5. double sum_val(double a, double b);
     6. double sum_ref(double *a, double *b);
     7. int is_true(int i);
     8.
     9.
     10. void main()
     11. {
     12.   double a=3.11,b=7.22;
     13.   do_nothing();
     14.   printf("add_up  %i\n",add_up(1,2,3,4,5,6,7));

   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
     15.   printf("sum_val %f\n",sum_val(a,b));
     16.   printf("sum_ref %f\n",sum_ref(&a,&b));
     17.   printf("is_true 0=%i,1=%i\n", is_true(0),is_true(1));
     18. }
```

Use the Solaris Studio `er_src` command to examine the compiler commentary for a particular file. It takes two parameters: the name of the executable and the name of the function to

examine. In this case, the template that cannot be early inlined is `sum_val`. Each time the compiler comes across the `%fp` register, it inserts a debug message, so you can tell that there are six instances of references to `%fp` in the template.

7.1.5 Compiler Calling Convention

The calling convention differs for each architecture. You can see this by examining the assembler code generated by the compiler for a simple test function.

The following example is compiled for a 32-bit platform:

```
% more fptest.c

double sum(double d1,double d2, double d3, double d4)
{
  return d1 + d2 + d3 + d4;
  }

% cc -O -xarch=sparc -m32 -S fptest.c
% more fptest.s
....
                        .global sum
                       sum:
/* 000000          2 */         st      %o0,[%sp+68]
/* 0x0004            */         st      %o2,[%sp+76]
/* 0x0008            */         st      %o1,[%sp+72]
/* 0x000c            */         st      %o3,[%sp+80]
/* 0x0010            */         st      %o4,[%sp+84]
/* 0x0014            */         st      %o5,[%sp+88]

!    3               !  return d1 + d2 + d3 + d4;

/* 0x0018          3 */         ld      [%sp+68],%f2
/* 0x001c            */         ld      [%sp+72],%f3
/* 0x0020            */         ld      [%sp+76],%f10
/* 0x0024            */         ld      [%sp+80],%f11
/* 0x0028            */         ld      [%sp+84],%f4
/* 0x002c            */         faddd   %f2,%f10,%f12
/* 0x0030            */         ld      [%sp+88],%f5
/* 0x0034            */         ld      [%sp+92],%f6
/* 0x0038            */         ld      [%sp+96],%f7
/* 0x003c            */         faddd   %f12,%f4,%f14
/* 0x0040            */         retl    ! Result =  %f0
/* 0x0044            */         faddd   %f14,%f6,%f0
....
```

In the example code, you can see that the first three floating-point parameters are passed in `%o0-%o5`, and the fourth is passed on the stack at locations `%sp+92` and `%sp+96`. Note that this location is 4-byte aligned, so it is not possible to use a single floating point load double instruction to load it.

Here is an example for 64-bit code.

```
% more inttest.c
long sum(long v1,long v2, long v3, long v4, long v5, long v6, long v7)
{
return v1 + v2 + v3 + v4 + v5 + v6 + v7;
}

% cc -O -xarch=sparc -m64 -S inttest.c
% more inttest.s...
/* 000000          2 */         ldx     [%sp+2223],%g2
/* 0x0004          3 */         add     %o0,%o1,%g1
/* 0x0008            */         add     %o3,%o2,%g3
/* 0x000c            */         add     %g3,%g1,%g4
/* 0x0010            */         add     %o5,%o4,%g5
/* 0x0014            */         add     %g5,%g4,%o1
/* 0x0018            */         retl    ! Result =  %o0
/* 0x001c            */         add     %o1,%g2,%o0
...
```

In the code above, you can see that the first action is to load the seventh integer parameter from the stack.

7.1.6 Improving Efficiency of Inlined Functions

In the following example, when we examine the code the compiler generated we see a number of unnecessary loads and stores when all the data could be held in registers.

Calling C program:

```
int lzd(int);

int a;
int c=0;

int main()
{
  for(a=0; a<1000; a++)
  {
    c=lzd(c);
  }
  return 0;
}
```

The program is intended to use the Leading Zero Detect (LZD) instruction on the SPARC T4 to do a count of the number of leading zero bits in an integer register. The inline template `lzd.il` might look like this:

```
.inline lzd
  lzd %o0,%o0
.end
```

Compiling the code with optimization gives the resulting code:

```
% cc -O -xtarget=T4 -S lzd.c lzd.il
```

```
% more lzd.s
...
                        .L77000018:
/* 0x001c         11 */         lzd     %o0,%o0
/* 0x0020          9 */         ld      [%i1],%i3
/* 0x0024         11 */         st      %o0,[%i2]
/* 0x0028          9 */         add     %i3,1,%i0
/* 0x002c            */         cmp     %i0,999
/* 0x0030            */         ble,pt  %icc,.L77000018
/* 0x0034            */         st      %i0,[%i1]
...
```

Clearly everything could be held in registers, but the compiler is adding unnecessary loads and stores because it sees the inline template as a call to a function and must load and save registers around a function call it knows nothing about.

But we can insert a `#pragma` directive to tell the compiler that the routine `lzd()` has no side effects - meaning that it does not read or write to memory:

```
#pragma no_side_effect(routine_name)
```

and it needs to be placed after the declaration of the function. The new C code might look like:

```
int lzd(int);
#pragma no_side_effect(lzd)

int a;
int c=0;

int main()
{
  for(a=0; a<1000; a++)
  {
    c=lzd(c);
  }
  return 0;
}
```

Now the generated assembler code for the loop looks much neater:

```
/* 0x0014         10 */         add     %i1,1,%i1

!   11                !  {
!   12                !    c=lzd(c);

/* 0x0018         12 */         lzd     %o0,%o0
/* 0x001c         10 */         cmp     %i1,999
/* 0x0020            */         ble,pt  %icc,.L77000018
/* 0x0024            */         nop
```

7.1.7 Inline Templates in C++

To prevent linker errors, calls to inline template functions in C++ must be enclosed in an `extern "C"` declaration. For example:

```
extern "C"
  {
    void nothing();
  }

int main()
{
  nothing();
}
```

Inline template function:

```
.inline nothing
  nop
.end
```

7.1.7.1 C++ Inline Templates and Exceptions

In C++, `#pragma no_side_effect` cannot be combined with exceptions. But we know that the code cannot produce exceptions. The compiler might be able to produce even better code by adding the `throw()`keyword to the template declaration:

```
extern "C"
  {
    int mytemplate(int) throw();
    #pragma no_side_effect(mytemplate)
  }
```

7.2 Using __asm Statements in C and C++

The Oracle Solaris Studio C and C++ compilers support the `__ asm` statement:

```
__asm(string);

__asm{
...block of instructions...
}
```

The string may be a single assembler instruction, or a block of instructions, as in the following examples:

The statement

```
__asm("lzd ccx %o0");
```

does something.

The block of instructions

```
__asm{
```

```
  ldd ccx %f0
  ldd ccy %f1
  fadd %f0 %f1 %f0
  st ccz %f0
}
```

The Oracle Solaris Studio C and C++ compilers also support the GCC Extended ASM Statement syntax. See the GCC compiler documentation at gcc.gnu.org for details.

♦ ♦ ♦ APPENDIX A

Using the Assembler Command Line

This appendix is organized into the following sections:

A.1 Assembler Command Line

You invoke the assembler command line as follows:

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

Note - The Oracle Solaris Studio C, C++, and Fortran compilers (`cc`(1), `CC`(1), and `f95`(1)) invoke the assembler with the `fbe` command. You can use either the `as` or `fbe` command on a Oracle Solaris SPARC platform to invoke the SPARC assembler. (Note that the `as` or `fbe` command will invoke the x86 assembler on a Solaris x86 platform.)

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.)

A.2 Assembler Command Line Options

`-Dname -Dname=`*def*

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

`-hwcap={1|0}`

Enable (`-hwcap=1`) or suppress (`-hwcap=0`) the generation of the Hardware Capabilities section. Default is to generate the section.

`-I`*path*

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.

`-i`

Ignore line number information from the preprocessor.

`-L`

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

`-m`

This option runs `m4` macro preprocessing on input. The `m4` preprocessor is more useful for complex preprocessing than the C preprocessor invoked by the `-P` option. See the `m4(1)` man page for more information about the `m4` macro-processor.

`-m64|-m32`

Select the 64–bit (`-m64`) or 32–bit (`-m32`) memory model. With –`m64`, 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 OS. –`m32` is the default.

`-n`

Suppress all warnings while assembling.

`-o` *outfile*

Write the output of the assembler to *outfile*. By default, if –`o` is not specified, the output file name is the same as the input file name with `.s` replaced with `.o`.

`-P`

Run `cpp`(1), the C preprocessor, on the files being assembled. The preprocessor is run separately on each input file, not on their concatenation. The preprocessor output is passed to the assembler.

`-Q{y|n}`

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

`-S[a|b|c|l|A|B|C|L]`

Produces a disassembly of the emitted code to the standard output. Adding each of the following characters to the `-S` option produces:

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

Capital letters turn the switch off for the corresponding option.

`-s`

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

`-U`*name*

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.

`–ul`

By default, undefined symbols are marked as *global*. With –`ul`, they are marked as *local*.

`-V`

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

`-xarch=`*isa*

isa specifies the target architecture instruction set (ISA). This option limits the instructions accepted by the assembler to the instructions of the specified instruction set architecture. The assembler will issue an error when encountering an instruction that is not part of the specified *isa*.

Use the `-m64` or `-m32` option to specify the intended memory model, 64-bit or 32-bit respectively. The `-xarch` flag no longer indicates the memory model.

Note: The assembler and linker will mark `.o` files and executables that require a particular instruction set architecture (ISA) so that the executable will not be loaded at runtime if the running system does not support that particular ISA. If you compile and link in separate steps, make sure to specify the same *isa* value for `-xarch` in both steps.

isa **value**	**Meaning**
`generic`	Equivalent to `–xarch=sparc`
`sparc`	Limit the instruction set to SPARC V9 without the VIS (Visual Instruction Set) and without other implementation-specific extensions.
`sparcvis`	Limit the instruction set to SPARC V9 plus the VIS version 1.0 and the UltraSPARC extensions.
`sparcvis2`	Limit the instruction set to SPARC V9 and the UltraSPARC extensions, plus the VIS version 2.0 and the UltraSPARC III extensions.
`sparcvis3`	Limit the instruction set to SPARC V9 and the Ultra SPARC extensions, plus the VIS version 3.0 and the UltraSPARC III extensions, plus the fused multiply-add instructions.
`sparcfmaf`	Limit the instruction set to SPARC V9 and the Ultra SPARC extensions, plus the VIS version 2.0 and the UltraSPARC III extensions, and the SPARC64 VI extensions for floating-point multiply-add.
`sparcima`	Limit the instruction set to the SPARC IMA version of SPARC V9 and the Ultra SPARC extensions, plus the VIS version 2.0 and the UltraSPARC III extensions, the SPARC64 VI extensions for floating-point multiply-add, and the SPARC64 VII instructions for integer multiply-add.
`sparc4`	Limit the instruction set to the SPARC4 version of SPARC V9 and the Ultra SPARC extensions, plus the VIS version 3.0 and the UltraSPARC III extensions, the SPARC64 VI extensions for fused floating-point multiply-add, and the SPARC64 VII instructions for integer multiply-add, and SPARC4 instructions.
`sparcace`	Limit the instruction set to the SPARCACE version of the SPARC-V9 ISA and includes the following instruction sets ■ SPARC-V9

isa **value**	**Meaning**
	■ UltraSPARC extensions, including the Visual Instruction Set(VIS) version 1.0 ■ UltraSPARC-III extensions, including the Visual Instruction Set(VIS) version 2.0 ■ SPARC64 VI extensions for floating-point multiply-add ■ SPARC64 VII extensions for integer multiply-add, and SPARCACE instructions
`sparcaceplus`	Limit the instruction set to the SPARCACEPLUS version of the SPARC-V9 ISA and includes the following instruction sets ■ SPARC-V9 ■ UltraSPARC extensions, including the Visual Instruction Set(VIS) version 1.0 ■ UltraSPARC-III extensions, including the Visual Instruction Set(VIS) version 2.0 ■ SPARC64 VI extensions for floating-point multiply-add ■ SPARC64 VII extensions for integer multiply-add, SPARCACE, and SPARCACEPLUS instructions
`v9`	Equivalent to `–m64 –xarch=sparc`.
`v9a`	Equivalent to `–m64 –xarch=sparcvis`
`v9b`	Equivalent to `–m64 –xarch=sparcvis2`

`-xF`

Generates additional information for use by the Oracle Solaris Studio performance analyzer. If the input file does not contain any debugging directives, the assembler will generate default stabs needed by the analyzer. See also the `dbx`(1) man page.

`-Y{c|m},`*path*

Specify the path to locate the version of `cpp` (`-Yc,`*path*) or `m4` (`-Ym,`*path*) to use.

A.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 `dis`(1) man page.

♦ ♦ ♦ APPENDIX B

A Sample Assembler Program

The following code takes a sample C language program and generates the corresponding assembly code using the Oracle Solaris Studio C compiler running on the Solaris 11 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 B-1 C Program Example Source

```
#include <stdio.h>
#include <stdlib.h>

/* 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);
}

int 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);

  return 0;
}
```

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

EXAMPLE B-2 Assembler Output From C Source

```
    .section    ".text",#alloc,#execinstr
    .file   "fib.c"

    .section    ".data",#alloc,#write       ! open a data section
                            ! #alloc - memory will be allocated for this section at runtime
                            ! #write - section contains data that is writeable during process
 execution
Ddata.data:
    .align  4                       ! align the beginning of this section to a 4-byte boundary

.L18:
    .skip   4                       ! skip 4 bytes, which initializes fib_array[0]=0
    .word   1                       ! write the 4-byte value '1', initializes fib_array[1]=1
    .skip   188                     ! skip 188 bytes, which initializes the remainder of fib_array[]
 to 0
    .type   .L18,#object                ! set the type of .L17 (fib_array) to be an object

Drodata.rodata:
    .section    ".rodata1",#alloc           ! open a read-only data section.
    .align  4

!
! CONSTANT POOL
!

.L21:
    .ascii  "Fibonacci(%d) cannot be represented in a 32 bit word\n\000"    ! ascii string for
 printf
    .align  4                                   ! align the next ascii string to a 4-byte boundary

.L34:
    .ascii  "Fibonacci(n):, please enter n:\n\000"
    .align  4

.L35:
    .ascii  "%d\000"
    .align  4
```

```
.L40:
    .ascii  "Fibonacci (%d) is %u\n\000"

    .section    ".text",#alloc,#execinstr       ! open a text section
/* 000000      0 */     .align  4
/* 000000        */     .skip   16
/* 0x0010        */     .align  4
! FILE fib.c

!    1        !#include <stdio.h>
!    2        !#include <stdlib.h>
!    4        !/* a simple program computing the first n Fibonacci numbers */
!    6        !extern unsigned * fibonacci();
!    8        !#define MAX_FIB_REPRESENTABLE 49
!   10        !/* compute the first n Fibonacci numbers */
!   11        !unsigned * fibonacci(n)
!   12        !     int n;
!   13        !{

!
! SUBROUTINE fibonacci
!
! OFFSET    SOURCE LINE LABEL   INSTRUCTION

            .global fibonacci           ! create a symbol with global scope

            fibonacci:

            .L900000112:
/* 000000     13 */     save    %sp,-96,%sp     ! create a new stack frame and
                            ! register window for this subroutine

!   14        !  static unsigned fib_array[MAX_FIB_REPRESENTABLE] = {0,1};
!   15        !  unsigned prev_number = 0;
!   16        !  unsigned curr_number = 1;
!   17        !  int i;
!   19        !  if (n >= MAX_FIB_REPRESENTABLE) {

/* 0x0004     19 */     cmp     %i0,49      ! cmp is a synthetic instr, equivalent to
                            ! subcc %i0,49,%g0
/* 0x0008        */     bge,pn  %icc,.L77000033 ! branch %i0 (n) on gt 49 to .L77000033 ;
!                                               predict not taken
/* 0x000c     24 */     cmp     %i0,2       ! delay slot instr. Note that although
                            ! this instr is conceptually executed before the branch, it does
                            ! not influence the condition codes as seen by the branch

!   20        !    printf("Fibonacci(%d) cannot be represented in a 32 bit word\n", n);
!   21        !    exit(1);
!   22        !  }
!   24        !  for (i = 2; i < n; i++) {

            .L77000052:
/* 0x0010     24 */     ble,pn  %icc,.L77000043 ! branch on n less equal to 2 ;  predict not taken
/* 0x0014        */     mov     2,%l4       ! delay slot instr. %l4 = i = 2

            .L77000061:
/* 0x0018     24 */     add     %i0,-1,%l5          ! %l5 = %i0 (n) - 1
```

```
/* 0x001c     16 */     mov     1,%i4           ! %i4 (curr_number) = 1
/* 0x0020     15 */     mov     0,%i3           ! %i3 (prev_number) = 0
/* 0x0024        */     sethi   %hi(.L18),%i1       ! set the high 22-bits of %i1 to the address
 of .L18
                                ! (fib_array)
            .L900000109:
/* 0x0028     15 */     add     %i1,%lo(.L18),%i0       ! complete the formation of the address of
 fib_array

!   25        !    fib_array[i] = prev_number + curr_number;

/* 0x002c     25 */     add     %i3,%i4,%l7         ! %i7 = %i3 (prev_number) + %i4 (curr_number)
/* 0x0030     15 */     add     %i0,8,%l6           ! %l6 = &fib_array[i]

            .L900000110:
                                ! beginning of the loop body
/* 0x0034     24 */     add     %l4,1,%l4           ! increment i by 1

!   26        !    prev_number = curr_number;

/* 0x0038     26 */     mov     %i4,%i3         ! %i3 (prev_number) = %i4 (curr_number)
/* 0x003c     25 */     st      %l7,[%l6]           ! store %l7 into fib_array[i]

!   27        !    curr_number = fib_array[i];

/* 0x0040     27 */     mov     %l7,%i4         ! %i4 (curr_number) = %l7 (fib_array[i])
/* 0x0044     24 */     add     %l6,4,%l6       ! increase %l6 by 4 bytes, so that it now contains
 &fib_array[i+1]
/* 0x0048        */     cmp     %l4,%l5         ! i <= (n - 1)
/* 0x004c        */     ble,pt  %icc,.L900000110    ! if yes (predict taken), goto beginning of loop
/* 0x0050     25 */     add     %i3,%i4,%l7     ! delay slot instr. %i7 = %i3 (prev_number) + %l4
 (curr_number)
                                ! end of loop body

!   28        !  }
!   30        !  return(fib_array);

                                ! Body of if (n >= MAX_FIB_REPRESENTABLE) {}
            .L77000043:
/* 0x0054     30 */     sethi   %hi(.L18),%i5       ! set the high 22-bits of %l4 to the address
 of .L18
                                ! (fib_array)
/* 0x0058     24 */     ret                 ! synthetic instr.  equivalent to jmpl %i7+8, %g0
/* 0x005c        */     restore %i5,%lo(.L18),%o0       ! delay slot instr. restore the caller's
 window.
                                ! the subroutine return value is in %o0

            .L77000033:
/* 0x0060     20 */     sethi   %hi(.L21),%i2       ! set the high 22-bits of %i2 to the address
 of .L21
                                ! (string to be passed to printf)
/* 0x0064     16 */     mov     1,%i4           ! ** note that the instrs marked "**" are
 unnecessary. These instrs
                                ! perform the same function as those earlier in the program. They
 are created
                                ! by the compiler as it is not aware that ( exit(1) ) will terminate
 the
                                ! program.
```

```
/* 0x0068     20 */     add     %i2,%lo(.L21),%o0   ! add high and low bits
!                                 to complete formation of address of .L21
/* 0x006c     15 */     mov     0,%i3           ! **
/* 0x0070     20 */     call    printf  ! params =  %o0 %o1 ! Call printf with args  %o0 and %o1
/* 0x0074        */     mov     %i0,%o1
/* 0x0078     21 */     call    exit    ! params =  %o0     ! Call exit whose 1st arg is %o0
/* 0x007c        */     mov     1,%o0           ! **
/* 0x0080        */     add     %i0,-1,%l5          ! **
/* 0x0084     24 */     mov     2,%l4           ! **
/* 0x0088        */     ba      .L900000109         ! **
/* 0x008c     15 */     sethi   %hi(.L18),%i1       ! **

/* 0x0090      0 */     .type   fibonacci,#function     ! set the type of fibonacci to be a function
/* 0x0090      0 */     .size   fibonacci,(.-fibonacci) ! set the size of the function
                                 ! size of function:
                                 !   current location counter minus beginning definition of function


            .L900000113:

    .section    ".text",#alloc,#execinstr
/* 000000      0 */     .align  4

!   31        !}
!   33        !int main()
!   34        !{

!
! SUBROUTINE main
!
! OFFSET    SOURCE LINE LABEL   INSTRUCTION

            .global main

            main:

            .L900000210:
/* 000000     34 */     save    %sp,-104,%sp
/* 0x0004      0 */     sethi   %hi(.L34),%i5
/* 0x0008      0 */     add     %i5,%lo(.L34),%i1

!   35        !  int n, i;
!   36        !  unsigned * result;
!   38        !  printf("Fibonacci(n):, please enter n:\n");

/* 0x000c     38 */     call    printf      ! params =  %o0
/* 0x0010        */     mov     %i1,%o0

!   39        !  scanf("%d", &n);

/* 0x0014     39 */     add     %i1,32,%o0      ! %o0 = %i1+32 (&.L35)
/* 0x0018        */     call    scanf       ! params =  %o0 %o1
/* 0x001c        */     add     %fp,-4,%o1      ! %o1 = %fp-4 (&n)

!   41        !  result = fibonacci(n);

/* 0x0020     41 */     call    fibonacci       ! params =  %o0 ! Result =  %o0. On return from the
                            ! routine, %o0 = &fib_array
```

```
/* 0x0024        */     ld      [%fp-4],%o0     ! delay slot instr. load the value at %fp-4 (n) into
 %o0
/* 0x0028        */     ld      [%fp-4],%i4     ! load the value at %fp-4 (n) into %i4

!   42        !  for (i = 1; i <= n; i++)

/* 0x002c     42 */     cmp     %i4,1       ! n < 1 ?
/* 0x0030        */     bl,pn   %icc,.L77000075 ! if yes, branch to end of main()
/* 0x0034     41 */     mov     %o0,%i2     ! %i2 (result) = %o0 ( result of fibonacci() )

!   43        !    printf("Fibonacci (%d) is %u\n", i, *result++);

            .L77000082:
/* 0x0038     42 */     mov     1,%i3       ! i = 1

            .L900000207:
                         ! beginning of loop body
/* 0x003c     43 */     ld      [%i2],%o2       ! %o2 (3rd arg) = value at &result
/* 0x0040        */     add     %i1,36,%o0      ! %o0 (1st arg) = %i1+36 (&.L40)
/* 0x0044        */     add     %i2,4,%i2       ! increment &result by 4, result now points to the
                         ! next value in fib_array[]
/* 0x0048        */     call    printf  ! params =  %o0 %o1 %o2 ! Result =
/* 0x004c        */     mov     %i3,%o1     ! %o1 (2nd arg) = %i3 (i)
/* 0x0050        */     ld      [%fp-4],%i0     ! %i0 = value at %fp-4 (n)
/* 0x0054     42 */     add     %i3,1,%i3       ! increment i by 1
/* 0x0058        */     cmp     %i3,%i0     ! i <= n ?
/* 0x005c        */     ble,pt  %icc,.L900000207! if yes, goto beginning of loop body
/* 0x0060        */     nop             ! end of the loop body

            .L77000075:
/* 0x0064     42 */     ret
/* 0x0068        */     restore %g0,0,%o0
/* 0x006c      0 */     .type   main,#function
/* 0x006c      0 */     .size   main,(.-main)
```

♦ ♦ ♦ APPENDIX C

SPARC Instruction Sets and Mnemonics

This appendix provides information about the SPARC instruction sets, operation codes, and mnemonics accepted by the assembler.

The full SPARC instruction set is detailed in the *Oracle SPARC Architecture 2011 Guide*, referred to in this appendix as OSA.

C.1 Natural Instructions

Some of the synthetic SPARC instructions were extended to include the *natural* operations, which are interpreted differently for `-m32` or `-m64` assembly.

TABLE C-1 Natural Instructions

Natural Instruction	`–m32` Interpretation	`–m64` Interpretation
`ldn`	`ld`	`ldx`
`stn`	`st`	`stx`
`ldna`	`lda`	`ldax`
`stna`	`sta`	`stxa`
`casn`	`cas`	`casx`
`slln`	`sll`	`sllx`
`srln`	`srl`	`srlx`
`sran`	`sra`	`srax`
`clrn`	`clr`	`clrx`
`setn`	`set`	`setx`
`setnhi`	`sethi`	`setxhi`

C.1.1 Natural Register, Natural Word

In addition to natural instructions, there is also the natural register `%ncc` and the natural word pseudo-op `.nword`, which are interpreted differently depending on the assembly mode.

TABLE C-2 Natural Register and Word

Register and Word	`–m32` Interpretation	`–m64` Interpretation
`%ncc` register	`%icc`	`%xcc`
`.nword` pseudo-op	`.word`	`.xword`

Index